Inside: Racket C API

Version 8.1

Matthew Flatt

May 4, 2021

The Racket runtime system is responsible for the implementation of primitive datatypes such as numbers and strings, the macro expansion and compilation of Racket from source, the allocation and reclamation of memory used during evaluation, and the scheduling of concurrent threads and parallel tasks.

This manual describes the C interface of Racket's runtime system, which varies depending on the implementation of Racket (see §19.2 "Racket Virtual Machine Implementations"): the CS implementation of Racket has one interface, while the BC (3m and CGC) implementation of Racket has another.

The C interface is relevant to some degree when interacting with foreign libraries as described in *The Racket Foreign Interface*. Even though interactions with foreign code are constructed in pure Racket using the `ffi/unsafe` module, many details of representations, memory management, and concurrency are described here. This manual also describes embedding the Racket run-time system in larger programs and extending Racket directly with C-implemented libraries.

Contents

Index 173

Part I

Inside Racket CS

The Racket CS API is a small extension of the Chez Scheme C API as described in *The Chez Scheme User's Guide*.

1 Overview (CS)

The Racket CS runtime system is implemented by a wrapper around the Chez Scheme kernel. The wrapper implements additional glue to the operating system (e.g., for I/O and networking) and provides entry points into the Racket layer's evaluator.

1.1 “S” versus “Racket”

In the C API for Racket CS, names that start with `S` are from the Chez Scheme layer, while names that start with `racket_` are from the Racket wrapper.

1.2 Racket CS Memory Management

Racket values may be moved or garbage collected any time that `racket_...` functions are used to run Racket code. Do not retain a reference to any Racket value across such a call. This requirement contrasts with the BC implementation of Racket, which provide a way for C code to more directly cooperate with the memory manager.

API functions that start with `S` do not collect or move objects unless noted otherwise, so references to Racket values across such calls is safe.

1.3 Racket CS and Places

Each Racket place corresponds to a Chez Scheme thread, which also corresponds to an OS-implemented thread. Chez Scheme threads share a global allocation space, so GC-managed objects can be safely be communicated from one place to another. Beware, however, that Chez Scheme threads are unsafe; any synchronization needed to safely share a value across places is must be implemented explicitly. Racket-level functions for places will only share values across places when they can be safely used in both places.

In an embedding application, the OS thread that originally calls `racket_boot` is the OS thread of the original place.

1.4 Racket CS and Threads

Racket implements threads for Racket programs without aid from the operating system or Chez Scheme's threads, so that Racket threads are cooperative from the perspective of C code. Stand-alone Racket uses a few private OS-implemented threads for background tasks, but these OS-implemented threads are never exposed by the Racket API.

Racket can co-exist with additional OS-implemented threads, but care must be taken when calling `S` functions, and additional OS or Chez Scheme threads must not call any `racket_` function. For other OS threads to call `S` functions, the thread must be first activated as a Chez Scheme thread using `Sactivate_thread`.

1.5 Racket CS Integers

The C type `iptr` is defined by Racket CS headers to be an integer type that is big enough to hold a pointer value. In other words, it is an alias for `intptr_t`. The `uptr` type is the unsigned variant.

2 Embedding into a Program (CS)

To embed Racket CS in a program, follow these steps:

- Locate or build the Racket CS library.

  On Unix, the library is `"libracketcs.a"`. Building from source and installing places the libraries into the installation's `"lib"` directory.

  On Windows, link to `"libracketcsx.dll"` (where *x* represents the version number). At run time, either `"libracketcsx.dll"` must be moved to a location in the standard DLL search path, or your embedding application must "delayload" link the DLLs and explicitly load them before use. (`"Racket.exe"` uses the latter strategy.)

  On Mac OS, besides `"libracketcs.a"` for static linking, a dynamic library is provided by the `"Racket"` framework, which is typically installed in `"lib"` sub-directory of the installation. Supply `-framework Racket` to `gcc` when linking, along with `-F` and a path to the `"lib"` directory. At run time, either `"Racket.framework"` must be moved to a location in the standard framework search path, or your embedding executable must provide a specific path to the framework (possibly an executable-relative path using the Mach-O `@executable_path` prefix). When targeting the Hardened Runtime, you must enable the "Allow Unsigned Executable Memory" entitlement, otherwise you will run into "out of memory" errors when calling `racket_boot`.

- For each C file that uses Racket library functions, `#include` the files `"chezscheme.h"` and `"racketcs.h"`.

  The `"chezscheme.h"` and `"racketcs.h"` files are distributed with the Racket software in the installation's `"include"` directory. Building and installing from source also places this file in the installation's `"include"` directory.

- In your program, call `racket_boot`. The `racket_boot` function takes a pointer to a `racket_boot_arguments_t` for configuring the Racket instance. After zeroing out the `racket_boot_arguments_t` value (typicially with `memset`), only the following fields are required to be set:

  - `exec_file` — a path to be reported by `(find-system-path 'exec-file)`, usually `argv[0]` for the `argv` received by your program's `main`.
  - `boot1_path` — a path to `"petite.boot"`. Use a path that includes at least one directory separator.
  - `boot2_path` — a path to `"scheme.boot"` (with a separator).
  - `boot3_path` — a path to `"racket.boot"` (which a separator).

  The `"petite.boot"`, `"scheme.boot"`, and `"racket.boot"` files are distributed with the Racket software in the installation's `"lib"` directory for Windows, and they are distributed within the `"Racket"` framework on Mac OS X; they must be built from source on Unix. These files can be combined into a single file—or even embedded into the executable—as long as the `boot1_offset`, `boot2_offset`, and `boot3_offset` fields of `racket_boot_arguments_t` are set to identify the starting offset of each boot image in the file.

  See §30 "Embedding Files in Executable Sections" for advice on embedding files like `"petite.boot"` in an executable.

- Configure the main thread's namespace by adding module declarations. The initial namespace contains declarations only for a few primitive modules, such as `'#%kernel`, and no bindings are imported into the top-level environment.

  To embed a module like `racket/base` (along with all its dependencies), use `raco ctool --c-mods` ⟨*dest*⟩, which generates a C file ⟨*dest*⟩ that contains modules in compiled form as encapsulated in a static array. The generated C file defines a `declare_modules` function that takes no arguments and installs the modules into the environment, and it adjusts the module name resolver to access the embedded declarations. If embedded modules refer to runtime files that need to be carried along, supply `--runtime` to `raco ctool --c-mods` to collect the runtime files into a directory; see §12.2 "Embedding Modules via C" for more information.

  Alternatively, set fields like `collects_dir`, `config_dir`, and/or `argv` in the `racket_boot_arguments_t` passed to `racket_boot` to locate collections/packages and initialize the namespace the same way as when running the `racket` executable.

  On Windows, `raco ctool --c-mods` ⟨*dest*⟩ `--runtime` ⟨*dest-dir*⟩ includes in ⟨*dest-dir*⟩ optional DLLs that are referenced by the Racket library to support `bytes-open-converter`. Set `dll_dir` in `racket_boot_arguments_t` to register ⟨*dest-dir*⟩ so that those DLLs can be found at run time.

  Instead of using `--c-mods` with `raco ctool`, you can use `--mods`, embed the file content (see §30 "Embedding Files in Executable Sections") and load the content with `racket_embedded_load_file_region`.

- Access Racket through `racket_dynamic_require`, `racket_eval`, and/or other functions described in this manual.

  If the embedding program configures built-in parameters in a way that should be considered part of the default configuration, then call the `seal` function provided by the primitive `#%boot` module afterward. The snapshot of parameter values taken by `seal` is used for certain privileged operations, such as installing a PLaneT package.

- Compile the program and link it with the Racket libraries.

Racket values may be moved or garbage collected any time that `racket_...` functions are used to run Racket code. Do not retain a reference to any Racket value across such a call.

For example, the following is a simple embedding program that runs a module `"run.rkt"`, assuming that `"run.c"` is created as

```
raco ctool --c-mods run.c "run.rkt"
```

to generate `"run.c"`, which encapsulates the compiled form of `"run.rkt"` and all of its transitive imports (so that they need not be found separately a run time). Copies of "petite.boot", "scheme.boot", and "racket.boot" must be in the current directory on startup.

"main.c"

```
#include <string.h>
#include "chezscheme.h"
#include "racketcs.h"

#include "run.c"

int main(int argc, char *argv[])
{
  racket_boot_arguments_t ba;

  memset(&ba, 0, sizeof(ba));

  ba.boot1_path = "./petite.boot";
  ba.boot2_path = "./scheme.boot";
  ba.boot3_path = "./racket.boot";

  ba.exec_file = argv[0];

  racket_boot(&ba);

  declare_modules();

  ptr mod = Scons(Sstring_to_symbol("quote"),
                  Scons(Sstring_to_symbol("run"),
                        Snil));

  racket_dynamic_require(mod, Sfalse);

  return 0;
}
```

As another example, the following is a simple embedding program that evaluates all expressions provided on the command line and displays the results, then runs a `read`-`eval`-`print` loop, all using `racket/base`. Run

```
raco ctool --c-mods base.c ++lib racket/base
```

to generate "base.c", which encapsulates `racket/base` and all of its transitive imports.

"main.c"

```
#include <string.h>
#include "chezscheme.h"
```

```
#include "racketcs.h"

#include "base.c"

static ptr to_bytevector(char *s);

int main(int argc, char *argv[])
{
  racket_boot_arguments_t ba;

  memset(&ba, 0, sizeof(ba));

  ba.boot1_path = "./petite.boot";
  ba.boot2_path = "./scheme.boot";
  ba.boot3_path = "./racket.boot";

  ba.exec_file = argv[0];

  racket_boot(&ba);

  declare_modules();

  racket_namespace_require(Sstring_to_symbol("racket/base"));

  {
    int i;
    for (i = 1; i < argc; i++) {
      ptr e = to_bytevector(argv[i]);
      e = Scons(Sstring_to_symbol("open-input-bytes"),
                Scons(e, Snil));
      e = Scons(Sstring_to_symbol("read"), Scons(e, Snil));
      e = Scons(Sstring_to_symbol("eval"), Scons(e, Snil));
      e = Scons(Sstring_to_symbol("println"), Scons(e, Snil));

      racket_eval(e);
    }
  }

  {
    ptr rbase_sym = Sstring_to_symbol("racket/base");
    ptr repl_sym = Sstring_to_symbol("read-eval-print-loop");

    racket_apply(Scar(racket_dynamic_require(rbase_sym,
                                             repl_sym)),
                 Snil);
  }
```

```
  return 0;
}

static ptr to_bytevector(char *s)
{
  iptr len = strlen(s);
  ptr bv = Smake_bytevector(len, 0);
  memcpy(Sbytevector_data(bv), s, len);
  return bv;
}
```

If modules embedded in the executable need to access runtime files (via `racket/runtime-path` forms), supply the `--runtime` flag to `raco ctool`, specifying a directory where the runtime files are to be gathered. The modules in the generated `".c"` file will then refer to the files in that directory.

3 Values and Types (CS)

A Racket value is represented by a pointer-sized value. The low bits of the value indicate the encoding that it uses. For example, two (on 32-bit platform) or three (on 64-bit platforms) low bits indicates a fixnum encoding, while a one low bit and zero second-lowest bit indicates a pair whose address in memory is specified by the other bits.

The C type for a Racket value is `ptr`. For most Racket types, a constructor is provided for creating values of the type. For example, `Scons` takes two `ptr` values and returns the `cons` of the values as a new `ptr` value. In addition to providing constructors, Racket defines several global constant Racket values, such as `Strue` for `#t`.

3.1 Global Constants

There are six global constants:

- `Strue` — `#t`
- `Sfalse` — `#f`
- `Snil` — `null`
- `Seof_object` — `eof-object`
- `Svoid` — `(void)`

3.2 Value Functions

Many of these functions are actually macros.

```
int Sfixnump(ptr v)
int Scharp(ptr v)
int Snullp(ptr v)
int Seof_objectp(ptr v)
int Sbooleanp(ptr v)
int Spairp(ptr v)
int Ssymbolp(ptr v)
int Sprocedurep(ptr v)
int Sflonump(ptr v)
int Svectorp(ptr v)
int Sfxvectorp(ptr v)
int Sbytevectorp(ptr v)
int Sstringp(ptr v)
```

```
int Sbignump(ptr v)
int Sboxp(ptr v)
int Sinexactnump(ptr v)
int Sexactnump(ptr v)
int Sratnump(ptr v)
int Srecordp(ptr v)
```

Predicates to recognize different kinds of Racket values, such as fixnums, characters, the empty list, etc. The `Srecordp` predicate recognizes structures, but some built-in Racket datatypes are also implemented as records.

```
ptr Sfixnum(int i)
```

Returns a Racket integer value, where *i* must fit in a fixnum.

```
ptr Sinteger(iptr i)
ptr Sunsigned(uptr i)
ptr Sinteger32(int i)
ptr Sunsigned32(unsigned int i)
ptr Sinteger64(long i)
ptr Sunsigned64(unsigned long i)
```

Returns an integer value for different conversions from C, where the result is allocated as a bignum if necessary to hold the value.

```
iptr Sfixnum_value(ptr v)
```

Converts a Racket fixnum to a C integer.

```
iptr Sinteger_value(ptr v)
uptr Sunsigned_value(ptr v)
int Sinteger32_value(ptr v)
long Sunsigned32_value(ptr v)
long Sinteger64_value(ptr v)
unsigned long Sunsigned64_value(ptr v)
```

Converts a Racket integer (possibly a bignum) to a C integer, assuming that the integer fits in the return type.

```
ptr Sflonum(double f)
```

Returns a Racket flonum value.

```
double Sflonum_value(ptr v)
```

Converts a Racket flonum value to a C floating-point number.

```
ptr Schar(int ch)
```

Returns a Racket character value. The *ch* value must be a legal Unicode code point (and not a surrogate, for example). All characters are represented by constant values.

```
ptr Schar_value(ptr ch)
```

Returns the Unicode code point for the Racket character *ch*.

```
ptr Sboolean(int bool)
```

Returns `Strue` or `Sfalse`.

```
ptr Scons(ptr car,
          ptr cdr)
```

Makes a `cons` pair.

```
ptr Scar(ptr pr)
ptr Scdr(ptr pr)
```

Extracts the `car` or `cdr` of a pair.

```
ptr Sstring_to_symbol(const char* str)
```

Returns the interned symbol whose name matches *str*.

```
ptr Ssymbol_to_string(ptr sym)
```

Returns the Racket immutable string value for the Racket symbol *sym*.

```
ptr Smake_string(iptr len,
                 int ch)
ptr Smake_uninitialized_string(iptr len)
```

Allocates a fresh Racket mutable string with *len* characters. The content of the string is either all *ch*s when *ch* is provided or unspecified otherwise.

```
ptr Sstring(const char* str)
ptr Sstring_of_length(const char* str,
                      iptr len)
ptr Sstring_utf8(const char* str,
                 iptr len)
```

Allocates a fresh Racket mutable string with the content of *str*. If *len* is not provided, *str* must be nul-terminated. In the case of `Sstring_utf8`, *str* is decoded as UTF-8, otherwise it is decided as Latin-1.

```
uptr Sstring_length(ptr str)
```

Returns the length of the string *str*.

```
ptr Sstring_ref(ptr str,
                uptr i)
```

Returns the *i*th Racket character of the string *str*.

```
int Sstring_set(ptr str,
                uptr i,
                ptr ch)
```

Installs *ch* as the *i*th Racket character of the string *str*.

```
ptr Smake_vector(iptr len,
                 ptr v)
```

Allocates a fresh mutable vector of length *len* and with *v* initially in every slot.

```
uptr Svector_length(ptr vec)
```

Returns the length of the vector *vec*.

```
ptr Svector_ref(ptr vec,
                uptr i)
```

Returns the *i*th element of the vector *vec*.

```
void Svector_set(ptr vec,
                 uptr i,
                 ptr v)
```

Installs *v* as the *i*th element of the vector *vec*.

```
ptr Smake_fxvector(iptr len,
                   ptr v)
```

Allocates a fresh mutable fxvector of length *len* and with *v* initially in every slot.

```
uptr Sfxvector_length(ptr vec)
```

Returns the length of the fxvector *vec*.

```
iptr Sfxvector_ref(ptr vec,
                   uptr i)
```

Returns the *i*th fixnum of the fxvector *vec*.

```
void Sfxvector_set(ptr vec,
                   uptr i,
                   ptr v)
```

Installs the fixnum *v* as the *i*th element of the fxvector *vec*.

```
ptr Smake_bytevector(iptr len,
                     int byte)
```

Allocates a fresh mutable byte string of length *len* and with *byte* initially in every slot.

```
uptr Sbytevector_length(ptr bstr)
```

Returns the length of the byte string *bstr*.

```
int Sbytevector_u8_ref(ptr bstr,
                       uptr i)
```

Returns the *i*th byte of the byte string *bstr*.

```
int Sbytevector_u8_set(ptr bstr,
                       uptr i,
                       int byte)
```

Installs *byte* as the *i*th byte of the byte string *bstr*.

```
char* Sbytevector_data(ptr vec)
```

Returns a pointer to the start of the bytes for the byte string *bstr*.

```
ptr Sbox(ptr v)
```

Allocates a fresh mutable box containing *v*.

```
ptr Sunbox(ptr bx)
```

Extract the content of the box *bx*.

```
ptr Sset_box(ptr bx,
             ptr v)
```

Installs *v* as the content of the box *bx*.

```
ptr Srecord_type(ptr rec)
ptr Srecord_type_parent(ptr rtd)
uptr Srecord_type_size(ptr rtd)
int Srecord_type_uniformp(ptr rtd)
ptr Srecord_uniform_ref(ptr rec,
                        iptr i)
```

Accesses record information, where Racket structures are implemented as records. The `Srecord_type` returns a value representing a record's type (so, a structure type). Given a record type, `Srecord_type_parent` returns its supertype or `Sfalse`, `Srecord_type_size` returns the allocation size of a record in bytes, and `Srecord_type_uniformp` indicates whether all of the record fields are Scheme values — which is always true for a Racket structure. When a record has all Scheme-valued fields, the allocation size is the number of fields plus one times the size of a pointer in bytes.

When a record has all Scheme fields (which is the case for all Racket structures), `Srecord_uniform_ref` accesses a field value in the same way as `unsafe-struct*-ref`.

```
void* racket_cpointer_address(ptr cptr)
void* racket_cpointer_base_address(ptr cptr)
iptr racket_cpointer_offset(ptr cptr)
```

Extracts an address and offset from a C-pointer object in the sense of `cpointer?`, but only for values using the predefined representation that is not a byte string, `#f`, or implemented by a new structure type with `prop:cpointer`.

The result of `racket_cpointer_address` is the same as `racket_cpointer_base_address` plus `racket_cpointer_offset`, where `racket_cpointer_offset` is non-zero for C-pointer values created by `ptr-add`.

4 Calling Procedures (CS)

As an entry point into Racket, C programs should normally call Racket procedures by using `racket_apply`, which calls the procedure in the initial Racket thread of the main Racket place. Chez Scheme entry points such as `Scall0` and `Scall` directly call a procedure outside of any Racket thread, which will not work correctly with Racket facilities such as threads, parameters, continuations, or continuation marks.

The functions in this section are meant to be used as an entry point to Racket, but not as a *re-entry* point. When Racket calls a C function that in turn calls back into Racket, the best approach is to use the FFI (see *The Racket Foreign Interface*) so that the C call recieves a Racket callback that is wrapped as a plain C callback. That way, the FFI can handle the details of boundary crossings between Racket and C.

```
ptr racket_apply(ptr proc,
                 ptr arg_list)
```

Applies the Racket procedure *proc* to the list of arguments *arg_list*. The procedure is called in the original Racket thread of the main Racket place. Applying *proc* must not raise an exception or otherwise escape from the call to *proc*.

The result is a list of result values, where a single result from *proc* causes `racket_apply` to return a list of length one.

Other Racket threads can run during the call to *proc*. At the point that *proc* results, all Racket thread scheduling in the main Racket place is suspended. No garbage collections will occur, so other Racket places can block waiting for garbage collection.

```
ptr Scall0(ptr proc)
ptr Scall1(ptr proc,
           ptr arg1)
ptr Scall2(ptr proc,
           ptr arg1,
           ptr arg2)
ptr Scall3(ptr proc,
           ptr arg1,
           ptr arg2,
           ptr arg3)
```

Applies the Chez Scheme procedure *proc* to zero, one, two, or three arguments. Beware that not all Racket procedures are Chez Scheme procedures. (For example, an instance of a structure type that has `prop:procedure` is not a Chez Scheme procedure.)

The procedure is called outside of any Racket thread, and other Racket threads are not scheduled during the call to *proc*. A garbage collection may occur.

```
void Sinitframe(iptr num_args)
```

```
void Sput_arg(iptr i,
              ptr arg)
ptr Scall(ptr proc,
          iptr num_args)
```

Similar to `Scall0`, but these functions are used in sequence to apply a Chez Scheme procedure to an arbitrary number of arguments. First, `Sinitframe` is called with the number of arguments. Then, each argument is installed with `Sput_arg`, where the *i* argument indicates the argumenrt position and *arg* is the argument value. Finally, `Scall` is called with the procedure and the number of arguments (which must match the number provided to `Sinitframe`).

5 Starting and Declaring Initial Modules (CS)

As sketched in §2 "Embedding into a Program (CS)", and embedded instance of Racket CS is started with `racket_boot`. Functions such as `racket_embedded_load_bytes` help to initialize a Racket namespace with already-compiled modules.

For functions and struct fields that contain a path in `char*` form, the path is treated as UTF-8 encoded on Windows.

5.1 Boot and Configuration

`void racket_boot(racket_boot_arguments_t*` *boot_args*`)`

Initializes a Racket CS instance. A main thread is created and then suspended, waiting for further evaluation via `racket_apply`, `racket_eval`, and similar functions.

A `racket_boot_arguments_t` struct contains fields to specify how `racket_boot` should initialize a Racket instance. New fields may be added in the future, but in that case, a `0` or `NULL` value for a field will imply backward-compatible default.

Fields in `racket_boot_arguments_t`:

- `const char *boot1_path` — a path to a file containing a Chez Scheme image file with base functionality. Normally, the file is called `"petite.boot"`. The path should contain a directory separator, otherwise Chez Scheme will consult its own search path.
- `long boot1_offset` — an offset into `boot1_path` to read for the first boot image, which allows boot images to be combined with other data in a single file. The image as distributed is self-terminating, so no size or ending offset is needed.
- `long boot1_len` — an optional length in bytes for the first boot image, which is used as a hint for loading the boot file if non-zero. If this hint is provided, it must be at least as large as the boot image bytes, and it must be no longer than the file size after the boot image offset.
- `const char *boot2_path` — like `boot1_path`, but for the image that contains compiler functionality, normally called `"scheme.boot"`.
- `long boot2_offset` — an offset into `boot2_path` to read for the second boot image.
- `long boot2_len` — `boot1_len`, an optional length in bytes for the second boot image.
- `const char *boot3_path` — like `boot1_path`, but for the image that contains Racket functionality, normally called `"racket.boot"`.

- `long boot3_offset` — `boot1_len`, an offset into `boot2_path` to read for the third boot image.
- `long boot3_len` — an optional length in bytes for the third boot image.
- `int argc` and `char ** argv` — command-line arguments to be processed the same as for a stand-alone `racket` invocation. If *argv* is `NULL`, the command line `-n` is used, which loads boot files without taking any further action.
- `const char * exec_file` — a path to use for `(system-type 'exec-file)`, usually `argv[0]` using the `argv` delivered to a program's `main`. This field must not be `NULL`.
- `const char * run_file` — a path to use for `(system-type 'run-file)`. If the field is `NULL`, the value of `exec_file` is used.
- `const char * collects_dir` — a path to use as the main `"collects"` directory for locating library collections. If this field holds `NULL` or `""`, then the library-collection search path is initialized as empty.
- `const char * config_dir` — a path to used as an `"etc"` directory that holds configuration information, including information about installed packages. If the value if `NULL`, `"etc"` is used.
- `wchar_t * dll_dir` — a path used to find DLLs, such as `iconv` support. Note that this path uses wide characters, not a UTF-8 byte encoding.
- `int cs_compiled_subdir` — A true value indicates that the `use-compiled-file-paths` parameter should be initialized to have a platform-specific subdirectory of `"compiled"`, which is used for a Racket CS installation that overlays a Racket BC installation.

5.2 Loading Racket Modules

```
void racket_embedded_load_bytes(const char* code,
                                uptr len,
                                int as_predefined)
void racket_embedded_load_file(const char* path,
                               int as_predefined)
void racket_embedded_load_file_region(const char* path,
                                      uptr start,
                                      uptr end,
                                      int as_predefined)
```

These functions evaluate Racket code, either in memory as *code* or loaded from *path*, in the initial Racket thread. The intent is that the code is already compiled. Normally, also, the contains module declarations. The `raco ctool --c-mods` and `raco ctool --mods` commands generate code suitable for loading with these functions, and `--c-mods` mode generates C code that calls `racket_embedded_load_bytes`.

If *as_predefined* is true, then the code is loaded during the creation of any new Racket place in the new place, so that modules declared by the code are loaded in the new place, too.

These functions are not meant to be called in C code that was called from Racket. See also §4 “Calling Procedures (CS)” for a discussion of *entry* points versus *re-entry* points.

6 Evaluation and Running Modules (CS)

The `racket_apply` function provides basic evaluation support, but `racket_eval`, `racket_dynamic_require`, and `racket_namespace_require` provide higher-level support for the most common evaluation tasks to initialize a Racket instance.

```
ptr racket_eval(ptr s_expr)
```

Evaluates *s_expr* in the initial Racket thread using its current namespace, the same as calling `eval`. The *s_expr* can be an S-expression constructed with pairs, symbols, etc., or it can be a syntax object.

Use `racket_namespace_require` to initialize a namespace, or use `racket_dynamic_require` to access functionality without going through a top-level namespace. Although those functions are the same as using `namespace-require` and `dynamic-require`, they work without having those identifiers bound in a namespace already.

This function and others in this section are not meant to be called in C code that was called from Racket. See also §4 "Calling Procedures (CS)" for a discussion of *entry* points versus *re-entry* points.

```
ptr racket_dynamic_require(ptr module_path,
                           ptr sym_or_false)
```

The same as calling `dynamic-require` in the initial Racket thread using its current namespace. See also `racket_eval`.

```
ptr racket_namespace_require(ptr module_path)
```

The same as calling `namespace-require` in the initial Racket thread using its current namespace. See also `racket_eval`.

```
ptr racket_primitive(const char* name)
```

Accesses a primitive function in the same sense as `vm-primitive` from `ffi/unsafe/vm`.

7 Managing OS-Level Threads (CS)

Chez Scheme functionality can only be accessed from OS-level threads that are known to the Chez Scheme runtime system. Otherwise, there's a race condition between such an access and a garbage collection that is triggered by other threads.

A thread not created by Chez Scheme can be made known to the runtime system by activating it with `Sactivate_thread`. As long as a thread is active by not running Chez Scheme code, the thread prevents garbage collection in all other running threads. Deactivate a thread using `Sdeactivate_thread`. When a deactivated thread

```
int Sactivate_thread()
```

Activates the current OS-level thread. An already-activated thread can be activated again, but each activating must be balanced by a decativation. The result is `0` if the thread was previously activated `1` otherwise.

```
void Sdeactivate_thread()
```

Deactivates the current OS-level thread—or, at least, balances on activation, making the thread deactive if there are no remaining activations to balance with deactivation.

```
int Sdestroy_thread()
```

Releases any Chez Scheme resources associated with the current OS thread, which must have been previously activated by which must not be activated still.

Part II

Inside Racket BC (3m and CGC)

The Racket BC API was originally designed for a tight integration with C code. As a result, the BC API is considerably larger than the Racket CS API.

8 Overview (BC)

The Racket BC runtime system is implemented in C and provides the compiler from source to bytecode format, the JIT compiler from bytecode to machine code, I/O functionality, threads, and memory management.

8.1 “Scheme” versus “Racket”

The old name for Racket was “PLT Scheme,” and the core compiler and run-time system used to be called “MzScheme.” The old names are entrenched in Racket internals, to the point that most C bindings defined in this manual start with `scheme_`. In principle, they all should be renamed to start `racket_`.

8.2 CGC versus 3m

Before mixing any C code with Racket BC, first decide whether to use the **3m** variant of Racket, the **CGC** variant of Racket, or both:

- **3m** : the main variant of Racket BC, which uses *precise* garbage collection and requires explicit registration of pointer roots and allocation shapes. The precise garbage collector may move its objects in memory during a collection.
- **CGC** : the original variant of Racket BC, where memory management depends on a *conservative* garbage collector. The conservative garbage collector can automatically find references to managed values from C local variables and (on some platforms) static variables, and it does not move allocated objects.

At the C level, working with CGC can be much easier than working with 3m, but overall system performance is typically better with 3m.

8.3 Embedding and Extending Racket

The Racket run-time system can be embedded into a larger program; see §9 “Embedding into a Program (BC)” for more information. As an alternative to embedding, the `racket` executable can also be run in a subprocess, and that choice may be better for many purposes. On Windows, MzCom provides another option.

The Racket run-time system can be extended with new C-implemented functions. Historically, writing an extension could provide performance benefits relative to writing pure Racket code, but Racket performance has improved to the point that performance benefits of writing

C code (if any) are usually too small to justify the maintenance effort. For calling functions that are provided by a C-implemented library, meanwhile, using with foreign-function interface within Racket is a better choice than writing an extension of Racket to call the library.

8.4 Racket BC and Places

Each Racket place corresponds to a separate OS-implemented thread. Each place has its own memory manager. Pointers to GC-managed memory cannot be communicated from one place to another, because such pointers in one place are invisible to the memory manager of another place.

When place support is enabled, static variables at the C level generally cannot hold pointers to GC-managed memory, since the static variable may be used from multiple places. For some OSes, a static variable can be made thread-local, in which case it has a different address in each OS thread, and each different address can be registered with the GC for a given place.

In an embedding application, the OS thread that originally calls `scheme_basic_env` is the OS thread of the original place. When `scheme_basic_env` is called a second time to reset the interpreter, it can be called in an OS thread that is different from the original call to `scheme_basic_env`. Thereafter, the new thread is the OS thread for the original place.

8.5 Racket BC and Threads

Racket implements threads for Racket programs without aid from the operating system, so that Racket threads are cooperative from the perspective of C code. Stand-alone Racket may uses a few private OS-implemented threads for background tasks, but these OS-implemented threads are never exposed by the Racket API.

Racket can co-exist with additional OS-implemented threads, but the additional OS threads must not call any `scheme_` function. Only the OS thread representing a particular place can call `scheme_` functions. (This restriction is stronger than saying all calls for a given place must be serialized across threads. Racket relies on properties of specific threads to avoid stack overflow and garbage collection.) In an embedding application, for the original place, only the OS thread used to call `scheme_basic_env` can call `scheme_` functions. For any other place, only the OS thread that is created by Racket for the place can be used to call `scheme_` functions.

See §17 “Threads (BC)” for more information about threads, including the possible effects of Racket’s thread implementation on extension and embedding C code.

8.6 Racket BC, Unicode, Characters, and Strings

A character in Racket is a Unicode code point. In C, a character value has type `mzchar`, which is an alias for `unsigned` — which is, in turn, 4 bytes for a properly compiled Racket. Thus, a `mzchar*` string is effectively a UCS-4 string.

Only a few Racket functions use `mzchar*`. Instead, most functions accept `char*` strings. When such byte strings are to be used as a character strings, they are interpreted as UTF-8 encodings. A plain ASCII string is always acceptable in such cases, since the UTF-8 encoding of an ASCII string is itself.

See also §11.3 "Strings" and §20 "String Encodings (BC)".

8.7 Racket BC Integers

Racket expects to be compiled in a mode where `short` is a 16-bit integer, `int` is a 32-bit integer, and `intptr_t` has the same number of bits as `void*`. The `long` type can match either `int` or `intptr_t`, depending on the platform. The `mzlonglong` type has 64 bits for compilers that support a 64-bit integer type, otherwise it is the same as `intptr_t`; thus, `mzlonglong` tends to match `long long`. The `umzlonglong` type is the unsigned version of `mzlonglong`.

9 Embedding into a Program (BC)

The Racket run-time system can be embedded into a larger program. The embedding process for Racket CGC or Racket 3m (see §8.2 “CGC versus 3m”) is essentially the same, but the process for Racket 3m is most easily understood as a variant of the process for Racket CGC (even though Racket 3m is the standard variant of Racket).

9.1 CGC Embedding

To embed Racket CGC in a program, follow these steps:

- Locate or build the Racket CGC libraries. Since the standard distribution provides 3m libraries, only, you will most likely have to build from source.

  On Unix, the libraries are `"libracket.a"`, `"librktio.a"`, and `"libmzgc.a"` (or `"libracket.so"`, `"librrktio.so"`, and `"libmzgc.so"` for a dynamic-library build, with `"libracket.la"`, `"librktio.la"`, and `"libmzgc.la"` files for use with `libtool`). Building from source and installing places the libraries into the installation’s `"lib"` directory. Be sure to build the CGC variant, since the default is 3m.

  On Windows, stub libraries for use with Microsoft tools are `"libracketx.lib"` and `"libmzgcx.lib"` (where *x* represents the version number) are in a compiler-specific directory in `"racket\lib"`. These libraries identify the bindings that are provided by `"libracketx.dll"` and `"libmzgcx.dll"` — which are typically installed in `"racket\lib"`. When linking with Cygwin, link to `"libracketx.dll"` and `"libmzgcx.dll"` directly. At run time, either `"libracketx.dll"` and `"libmzgcx.dll"` must be moved to a location in the standard DLL search path, or your embedding application must “delayload” link the DLLs and explicitly load them before use. (`"Racket.exe"` and `"GRacket.exe"` use the latter strategy.)

  On Mac OS, dynamic libraries are provided by the `"Racket"` framework, which is typically installed in `"lib"` sub-directory of the installation. Supply `-framework Racket` to `gcc` when linking, along with `-F` and a path to the `"lib"` directory. Beware that CGC and 3m libraries are installed as different versions within a single framework, and installation marks one version or the other as the default (by setting symbolic links); install only CGC to simplify accessing the CGC version within the framework. At run time, either `"Racket.framework"` must be moved to a location in the standard framework search path, or your embedding executable must provide a specific path to the framework (possibly an executable-relative path using the Mach-O `@executable_path` prefix).

- For each C file that uses Racket library functions, `#include` the file `"scheme.h"`.

  The C preprocessor symbol `SCHEME_DIRECT_EMBEDDED` is defined as `1` when `"scheme.h"` is `#include`d, or as `0` when `"escheme.h"` is `#include`d.

The `"scheme.h"` file is distributed with the Racket software in the installation's `"include"` directory. Building and installing from source also places this file in the installation's `"include"` directory.

- Start your main program through the `scheme_main_setup` (or `scheme_main_stack_setup`) trampoline, and put all uses of Racket functions inside the function passed to `scheme_main_setup`. The `scheme_main_setup` function registers the current C stack location with the memory manager. It also creates the initial namespace `Scheme_Env*` by calling `scheme_basic_env` and passing the result to the function provided to `scheme_main_setup`. (The `scheme_main_stack_setup` trampoline registers the C stack with the memory manager without creating a namespace.)

  On Windows, when support for parallelism is enabled in the Racket build (as is the default), then before calling `scheme_main_setup`, your embedding application must first call `scheme_register_tls_space`:

  ```
  scheme_register_tls_space(&tls_space, 0);
  ```

  where `tls_space` is declared as a thread-local pointer variable in the main executable (i.e., not in a dynamically linked DLL):

  ```
  static __declspec(thread) void *tls_space;
  ```

  Changed in version 6.3: Calling `scheme_register_tls_space` is required on all Windows variants, although the call may be a no-op, depending on how Racket is built.

- Configure the namespace by adding module declarations. The initial namespace contains declarations only for a few primitive modules, such as `'#%kernel`, and no bindings are imported into the top-level environment.

  To embed a module like `racket/base` (along with all its dependencies), use `raco ctool --c-mods` ⟨*dest*⟩, which generates a C file ⟨*dest*⟩ that contains modules in bytecode form as encapsulated in a static array. The generated C file defines a `declare_modules` function that takes a `Scheme_Env*`, installs the modules into the environment, and it adjusts the module name resolver to access the embedded declarations. If embedded modules refer to runtime files that need to be carried along, supply `--runtime` to `raco ctool --c-mods` to collect the runtime files into a directory; see §12.2 "Embedding Modules via C" for more information.

  Alternatively, use `scheme_set_collects_path` and `scheme_init_collection_paths` to configure and install a path for finding modules at run time.

  On Windows, `raco ctool --c-mods` ⟨*dest*⟩ `--runtime` ⟨*dest-dir*⟩ includes in ⟨*dest-dir*⟩ optional DLLs that are referenced by the Racket library to support extflonums and `bytes-open-converter`. Call `scheme_set_dll_path` to register ⟨*dest-dir*⟩ so that those DLLs can be found at run time.

- Access Racket through `scheme_dynamic_require`, `scheme_load`, `scheme_eval`, and/or other functions described in this manual.

If the embedding program configures built-in parameters in a way that should be considered part of the default configuration, then call `scheme_seal_parameters` afterward. The snapshot of parameter values taken by `scheme_seal_parameters` is used for certain privileged operations, such as installing a PLaneT package.

- Compile the program and link it with the Racket libraries.

With Racket CGC, Racket values are garbage collected using a conservative garbage collector, so pointers to Racket objects can be kept in registers, stack variables, or structures allocated with `scheme_malloc`. In an embedding application on some platforms, static variables are also automatically registered as roots for garbage collection (but see notes below specific to Mac OS and Windows).

For example, the following is a simple embedding program that runs a module `"run.rkt"`, assuming that `"run.c"` is created as

```
raco ctool --c-mods run.c "run.rkt"
```

to generate `"run.c"`, which encapsulates the compiled form of `"run.rkt"` and all of its transitive imports (so that they need not be found separately a run time).

`"main.c"`

```
#include "scheme.h"
#include "run.c"

static int run(Scheme_Env *e, int argc, char *argv[])
{
  Scheme_Object *a[2];

  /* Declare embedded modules in "run.c": */
  declare_modules(e);

  a[0] = scheme_make_pair(scheme_intern_symbol("quote"),
                          scheme_make_pair(scheme_intern_symbol("run"),
                                           scheme_make_null()));
  a[1] = scheme_false;

  scheme_dynamic_require(2, a);

  return 0;
}

int main(int argc, char *argv[])
{
  return scheme_main_setup(1, run, argc, argv);
}
```

As another example, the following is a simple embedding program that evaluates all expressions provided on the command line and displays the results, then runs a `read-eval-print` loop, all using `racket/base`. Run

```
raco ctool --c-mods base.c ++lib racket/base
```

to generate `"base.c"`, which encapsulates `racket/base` and all of its transitive imports.

`"main.c"`

```
#include "scheme.h"
#include "base.c"

static int run(Scheme_Env *e, int argc, char *argv[])
{
  Scheme_Object *curout;
  int i;
  Scheme_Thread *th;
  mz_jmp_buf * volatile save, fresh;

  /* Declare embedded modules in "base.c": */
  declare_modules(e);

  scheme_namespace_require(scheme_intern_symbol("racket/base"));

  curout = scheme_get_param(scheme_current_config(),
                            MZCONFIG_OUTPUT_PORT);

  th = scheme_get_current_thread();

  for (i = 1; i < argc; i++) {
    save = th->error_buf;
    th->error_buf = &fresh;
    if (scheme_setjmp(*th->error_buf)) {
      th->error_buf = save;
      return -1; /* There was an error */
    } else {
      Scheme_Object *v, *a[2];
      v = scheme_eval_string(argv[i], e);
      scheme_display(v, curout);
      scheme_display(scheme_make_char('\n'), curout);
      /* read-eval-print loop, uses initial Scheme_Env: */
      a[0] = scheme_intern_symbol("racket/base");
      a[1] = scheme_intern_symbol("read-eval-print-loop");
      scheme_apply(scheme_dynamic_require(2, a), 0, NULL);
      th->error_buf = save;
    }
```

```
  }
  return 0;
}

int main(int argc, char *argv[])
{
  return scheme_main_setup(1, run, argc, argv);
}
```

If modules embedded in the executable need to access runtime files (via `racket/runtime-path` forms), supply the `--runtime` flag to `raco ctool`, specifying a directory where the runtime files are to be gathered. The modules in the generated `".c"` file will then refer to the files in that directory; the directory is normally specified relative to the executable, but the embedding application must call `scheme_set_exec_cmd` to set the executable path (typically `argv[0]`) before declaring modules.

On Mac OS, or on Windows when Racket is compiled to a DLL using Cygwin, the garbage collector cannot find static variables automatically. In that case, `scheme_main_setup` must be called with a non-zero first argument.

On Windows (for any other build mode), the garbage collector finds static variables in an embedding program by examining all memory pages. This strategy fails if a program contains multiple Windows threads; a page may get unmapped by a thread while the collector is examining the page, causing the collector to crash. To avoid this problem, call `scheme_main_setup` with a non-zero first argument.

When an embedding application calls `scheme_main_setup` with a non-zero first argument, it must register each of its static variables with `MZ_REGISTER_STATIC` if the variable can contain a GCable pointer. For example, if `curout` above is made `static`, then `MZ_REGISTER_STATIC(curout)` should be inserted before the call to `scheme_get_param`.

When building an embedded Racket CGC to use SenoraGC (SGC) instead of the default collector, `scheme_main_setup` must be called with a non-zero first argument. See §12 “Memory Allocation (BC)” for more information.

9.2 3m Embedding

Racket 3m can be embedded mostly the same as Racket, as long as the embedding program cooperates with the precise garbage collector as described in §12.1 “Cooperating with 3m”.

In either your source in the in compiler command line, `#define MZ_PRECISE_GC` before including `"scheme.h"`. When using `raco ctool` with the `--cc` and `--3m` flags, `MZ_PRECISE_GC` is automatically defined.

In addition, some library details are different:

- On Unix, the libraries are just `"libracket3m.a"` and `"librrktio.a"` (or `"libracket3m.so"` and `"librktio.so"` for a dynamic-library build, with `"libracket3m.la"` and `"librktio.la"` for use with `libtool`). There is no separate library for 3m analogous to CGC's `"libmzgc.a"`.
- On Windows, the stub library for use with Microsoft tools is `"libracket3mx.lib"` (where *x* represents the version number). This library identifies the bindings that are provided by `"libracket3mx.dll"`. There is no separate library for 3m analogous to CGC's `"libmzgcx.lib"`.
- On Mac OS, 3m dynamic libraries are provided by the `"Racket"` framework, just as for CGC, but as a version suffixed with `"_3m"`.

For Racket 3m, an embedding application must call `scheme_main_setup` with a non-zero first argument.

The simple embedding programs from the previous section can be processed by `raco ctool --xform`, then compiled and linked with Racket 3m. Alternately, the source code can be extended to work with either CGC or 3m depending on whether `MZ_PRECISE_GC` is defined on the compiler's command line:

`"main.c"`

```
#include "scheme.h"
#include "run.c"

static int run(Scheme_Env *e, int argc, char *argv[])
{
  Scheme_Object *l = NULL;
  Scheme_Object *a[2] = { NULL, NULL };

  MZ_GC_DECL_REG(5);
  MZ_GC_VAR_IN_REG(0, e);
  MZ_GC_VAR_IN_REG(1, l);
  MZ_GC_ARRAY_VAR_IN_REG(2, a, 2);

  MZ_GC_REG();

  declare_modules(e);

  l = scheme_make_null();
  l = scheme_make_pair(scheme_intern_symbol("run"), l);
  l = scheme_make_pair(scheme_intern_symbol("quote"), l);

  a[0] = l;
  a[1] = scheme_false;
```

```
  scheme_dynamic_require(2, a);

  MZ_GC_UNREG();

  return 0;
}

int main(int argc, char *argv[])
{
  return scheme_main_setup(1, run, argc, argv);
}
```

"main.c"

```
#include "scheme.h"
#include "base.c"

static int run(Scheme_Env *e, int argc, char *argv[])
{
  Scheme_Object *curout = NULL, *v = NULL, *a[2] = {NULL, NULL};
  Scheme_Config *config = NULL;
  int i;
  Scheme_Thread *th = NULL;
  mz_jmp_buf * volatile save = NULL, fresh;

  MZ_GC_DECL_REG(9);
  MZ_GC_VAR_IN_REG(0, e);
  MZ_GC_VAR_IN_REG(1, curout);
  MZ_GC_VAR_IN_REG(2, save);
  MZ_GC_VAR_IN_REG(3, config);
  MZ_GC_VAR_IN_REG(4, v);
  MZ_GC_VAR_IN_REG(5, th);
  MZ_GC_ARRAY_VAR_IN_REG(6, a, 2);

  MZ_GC_REG();

  declare_modules(e);

  v = scheme_intern_symbol("racket/base");
  scheme_namespace_require(v);

  config = scheme_current_config();
  curout = scheme_get_param(config, MZCONFIG_OUTPUT_PORT);

  th = scheme_get_current_thread();

  for (i = 1; i < argc; i++) {
```

```
      save = th->error_buf;
      th->error_buf = &fresh;
      if (scheme_setjmp(*th->error_buf)) {
        th->error_buf = save;
        return -1; /* There was an error */
      } else {
        v = scheme_eval_string(argv[i], e);
        scheme_display(v, curout);
        v = scheme_make_char('\n');
        scheme_display(v, curout);
        /* read-eval-print loop, uses initial Scheme_Env: */
        a[0] = scheme_intern_symbol("racket/base");
        a[1] = scheme_intern_symbol("read-eval-print-loop");
        v = scheme_dynamic_require(2, a);
        scheme_apply(v, 0, NULL);
        th->error_buf = save;
      }
    }

    MZ_GC_UNREG();

    return 0;
  }

  int main(int argc, char *argv[])
  {
    return scheme_main_setup(1, run, argc, argv);
  }
```

Strictly speaking, the `config` and `v` variables above need not be registered with the garbage collector, since their values are not needed across function calls that allocate. The code is much easier to maintain, however, when all local variables are registered and when all temporary values are put into variables.

9.3 Flags and Hooks

The following flags and hooks are available when Racket is embedded:

- `scheme_exit` — This pointer can be set to a function that takes an integer argument and returns `void`; the function will be used as the default exit handler. The default is `NULL`.
- `scheme_make_stdin`, `scheme_make_stdout`, `scheme_make_stderr`, — These pointers can be set to a function that takes no arguments and returns a Racket port

`Scheme_Object *` to be used as the starting standard input, output, and/or error port. The defaults are `NULL`. Setting the initial error port is particularly important for seeing unexpected error messages if `stderr` output goes nowhere.

- `scheme_console_output` — This pointer can be set to a function that takes a string and a `intptr_t` string length; the function will be called to display internal Racket warnings and messages that possibly contain non-terminating nuls. The default is *NULL*.
- `scheme_check_for_break` — This points to a function of no arguments that returns an integer. It is used as the default user-break polling procedure in the main thread. A non-zero return value indicates a user break, and each time the function returns a non-zero value, it counts as a new break signal (though the break signal may be ignored if a previous signal is still pending). The default is `NULL`.
- `scheme_case_sensitive` — If this flag is set to a non-zero value before `scheme_basic_env` is called, then Racket will not ignore capitalization for symbols and global variable names. The value of this flag should not change once it is set. The default is zero.
- `scheme_allow_set_undefined` — This flag determines the initial value of `compile-allow-set!-undefined`. The default is zero.
- `scheme_console_printf` — This function pointer was left for backward compatibility. The default builds a string and calls `scheme_console_output`.

```
void scheme_set_collects_path(Scheme_Object* path)
```

Sets the path to be returned by `(find-system-path 'collects-dir)`.

```
void scheme_set_addon_path(Scheme_Object* path)
```

Sets the path to be returned by `(find-system-path 'addon-dir)`.

```
void scheme_set_exec_cmd(const char* path)
```

Sets the path to be returned by `(find-system-path 'exec-file)`.

```
 void
scheme_init_collection_paths_post(Scheme_Env* env,
                                  Scheme_Object* pre_extra_paths,
                                  Scheme_Object* post_extra_paths)
```

Initializes the `current-library-collection-paths` parameter using `find-library-collection-paths`. The *pre_extra_paths* and *post_extra-paths* arguments are propagated to `find-library-collection-paths`.

The function calls `scheme_seal_parameters` automatically.

```
void scheme_init_collection_paths(Scheme_Env* env,
                                  Scheme_Object* pre_extra_paths)
```

Like `scheme_init_collection_paths_post`, but with `null` as the last argument.

```
void scheme_set_dll_path(wchar_t* path)
```

On Windows only, sets the path used to find optional DLLs that are used by the runtime system: `"longdouble.dll"` and one of `"iconv.dll"`, `"libiconv.dll"`, or `"libiconv-2.dll"`. The given *path* should be an absolute path.

```
void scheme_seal_parameters()
```

Takes a snapshot of the current values of built-in parameters. These values are used for privileged actions, such as installing a PLaneT package.

10 Writing Racket Extensions (BC)

As noted in §8.3 "Embedding and Extending Racket", writing Racket code and using the foreign-function interface is usually a better option than writing an extension to Racket, but Racket also supports C-implemented extensions that plug more directly into the run-time system. (Racket CS does not have a similar extension interface.)

The process of creating an extension for Racket 3m or Racket CGC (see §8.2 "CGC versus 3m") is essentially the same, but the process for 3m is most easily understood as a variant of the process for CGC.

10.1 CGC Extensions

To write a C/C++-based extension for Racket CGC, follow these steps:

- For each C/C++ file that uses Racket library functions, `#include` the file `"escheme.h"`.

  This file is distributed with the Racket software in an `"include"` directory, but if `raco ctool` is used to compile, this path is found automatically.

- Define the C function `scheme_initialize`, which takes a `Scheme_Env*` namespace (see §13 "Namespaces and Modules (BC)") and returns a `Scheme_Object*` Racket value.

  This initialization function can install new global primitive procedures or other values into the namespace, or it can simply return a Racket value. The initialization function is called when the extension is loaded with `load-extension` the first time in a given place; the return value from `scheme_initialize` is used as the return value for `load-extension`. The namespace provided to `scheme_initialize` is the current namespace when `load-extension` is called.

- Define the C function `scheme_reload`, which has the same arguments and return type as `scheme_initialize`.

  This function is called if `load-extension` is called a second time (or more times) for an extension in a given place. Like `scheme_initialize`, the return value from this function is the return value for `load-extension`.

- Define the C function `scheme_module_name`, which takes no arguments and returns a `Scheme_Object*` value, either a symbol or `scheme_false`.

  The function should return a symbol when the effect of calling `scheme_initialize` and `scheme_reload` is only to declare a module with the returned name. This function is called when the extension is loaded to satisfy a `require` declaration.

  The `scheme_module_name` function may be called before `scheme_initialize` and `scheme_reload`, after those functions, or both before and after, depending on how the extension is loaded and re-loaded.

- Compile the extension C/C++ files to create platform-specific object files.

  The `raco ctool` compiler, which is distributed with Racket, compiles plain C files when the `--cc` flag is specified. More precisely, `raco ctool` does not compile the files itself, but it locates a C compiler on the system and launches it with the appropriate compilation flags. If the platform is a relatively standard Unix system, a Windows system with either Microsoft's C compiler or `gcc` in the path, or a Mac OS system with Apple's developer tools installed, then using `raco ctool` is typically easier than working with the C compiler directly. Use the `--cgc` flag to indicate that the build is for use with Racket CGC.

- Link the extension C/C++ files with `"mzdyn.o"` (Unix, Mac OS) or `"mzdyn.obj"` (Windows) to create a shared object. The resulting shared object should use the extension `".so"` (Unix), `".dll"` (Windows), or `".dylib"` (Mac OS).

  The `"mzdyn"` object file is distributed in the installation's `"lib"` directory. For Windows, the object file is in a compiler-specific sub-directory of `"racket\lib"`.

  The `raco ctool` compiler links object files into an extension when the `--ld` flag is specified, automatically locating `"mzdyn"`. Again, use the `--cgc` flag with `raco ctool`.

- Load the shared object within Racket using `(load-extension path)`, where `path` is the name of the extension file generated in the previous step.

  Alternately, if the extension defines a module (i.e., `scheme_module_name` returns a symbol), then place the shared object in a special directory with a special name, so that it is detected by the module loader when `require` is used. The special directory is a platform-specific path that can be obtained by evaluating `(build-path "compiled" "native" (system-library-subpath))`; see `load/use-compiled` for more information. For example, if the shared object's name is `"example_rkt.dll"`, then `(require "example.rkt")` will be redirected to `"example_rkt.dll"` if the latter is placed in the sub-directory `(build-path "compiled" "native" (system-library-subpath))` and if `"example.rkt"` does not exist or has an earlier timestamp.

  Note that `(load-extension path)` within a `module` does *not* introduce the extension's definitions into the module, because `load-extension` is a run-time operation. To introduce an extension's bindings into a module, make sure that the extension defines a module, put the extension in the platform-specific location as described above, and use `require`.

IMPORTANT: With Racket CGC, Racket values are garbage collected using a conservative garbage collector, so pointers to Racket objects can be kept in registers, stack variables, or structures allocated with `scheme_malloc`. However, static variables that contain pointers to collectable memory must be registered using `scheme_register_extension_global` (see §12 "Memory Allocation (BC)"); even then, such static variables must be thread-local (in the OS-thread sense) to work with multiple places (see §8.4 "Racket BC and Places").

As an example, the following C code defines an extension that returns `"hello world"` when it is loaded:

```
#include "escheme.h"
Scheme_Object *scheme_initialize(Scheme_Env *env) {
  return scheme_make_utf8_string("hello world");
}
Scheme_Object *scheme_reload(Scheme_Env *env) {
  return scheme_initialize(env); /* Nothing special for reload */
}
Scheme_Object *scheme_module_name() {
  return scheme_false;
}
```

Assuming that this code is in the file `"hw.c"`, the extension is compiled on Unix with the following two commands:

```
raco ctool --cgc --cc hw.c
```

```
raco ctool --cgc --ld hw.so hw.o
```

(Note that the `--cgc`, `--cc`, and `--ld` flags are each prefixed by two dashes, not one.)

The `"collects/mzscheme/examples"` directory in the Racket distribution contains additional examples.

10.2 3m Extensions

To build an extension to work with Racket 3m, the CGC instructions must be extended as follows:

- Adjust code to cooperate with the garbage collector as described in §12.1 "Cooperating with 3m". Using `raco ctool` with the `--xform` might convert your code to implement part of the conversion, as described in §12.1.3 "Local Pointers and `raco ctool --xform`".
- In either your source in the in compiler command line, `#define MZ_PRECISE_GC` before including `"escheme.h"`. When using `raco ctool` with the `--cc` and `--3m` flags, `MZ_PRECISE_GC` is automatically defined.
- Link with `"mzdyn3m.o"` (Unix, Mac OS) or `"mzdyn3m.obj"` (Windows) to create a shared object. When using `raco ctool`, use the `--ld` and `--3m` flags to link to these libraries.

For a relatively simple extension `"hw.c"`, the extension is compiled on Unix for 3m with the following three commands:

```
raco ctool --xform hw.c

raco ctool --3m --cc hw.3m.c

raco ctool --3m --ld hw.so hw_3m.o
```

Some examples in `"collects/mzscheme/examples"` work with Racket 3m in this way. A few examples are manually instrumented, in which case the `--xform` step should be skipped.

10.3 Declaring a Module in an Extension

To create an extension that behaves as a module, return a symbol from `scheme_module_name`, and have `scheme_initialize` and `scheme_reload` declare a module using `scheme_primitive_module`.

For example, the following extension implements a module named `hi` that exports a binding `greeting`:

```
#include "escheme.h"

Scheme_Object *scheme_initialize(Scheme_Env *env) {
  Scheme_Env *mod_env;
  mod_env = scheme_primitive_module(scheme_intern_symbol("hi"),
                                    env);
  scheme_add_global("greeting",
                    scheme_make_utf8_string("hello"),
                    mod_env);
  scheme_finish_primitive_module(mod_env);
  return scheme_void;
}

Scheme_Object *scheme_reload(Scheme_Env *env) {
  return scheme_initialize(env); /* Nothing special for reload */
}

Scheme_Object *scheme_module_name() {
  return scheme_intern_symbol("hi");
}
```

This extension could be compiled for 3m on i386 Linux, for example, using the following sequence of `mzc` commands:

```
raco ctool --xform hi.c

raco ctool --3m --cc hi.3m.c
```

```
  mkdir -p compiled/native/i386-linux/3m
```

```
  raco ctool --3m --ld compiled/native/i386-linux/3m/hi_rkt.so
hi_3m.o
```

The resulting module can be loaded with

```
  (require "hi.rkt")
```

11 Values and Types (BC)

A Racket value is represented by a pointer-sized value. The low bit is a mark bit: a 1 in the low bit indicates an immediate integer, a 0 indicates a (word-aligned) pointer.

A pointer Racket value references a structure that begins with a `Scheme_Object` substructure, which in turn starts with a tag that has the C type `Scheme_Type`. The rest of the structure, following the `Scheme_Object` header, is type-dependent. Racket's C interface gives Racket values the type `Scheme_Object*`. (The "object" here does not refer to objects in the sense of the `racket/class` library.)

Examples of `Scheme_Type` values include `scheme_pair_type` and `scheme_symbol_type`. Some of these are implemented as instances of `Scheme_Simple_Object`, which is defined in `"scheme.h"`, but extension or embedding code should never access this structure directly. Instead, the code should use macros, such as `SCHEME_CAR`, that provide access to the data of common Racket types.

For most Racket types, a constructor is provided for creating values of the type. For example, `scheme_make_pair` takes two `Scheme_Object*` values and returns the `cons` of the values.

The macro `SCHEME_TYPE` takes a `Scheme_Object *` and returns the type of the object. This macro performs the tag-bit check, and returns `scheme_integer_type` when the value is an immediate integer; otherwise, `SCHEME_TYPE` follows the pointer to get the type tag. Macros are provided to test for common Racket types; for example, `SCHEME_PAIRP` returns `1` if the value is a cons cell, `0` otherwise.

In addition to providing constructors, Racket defines six global constant Racket values: `scheme_true`, `scheme_false`, `scheme_null`, `scheme_eof`, `scheme_void`, and `scheme_undefined`. Each of these has a type tag, but each is normally recognized via its constant address.

An extension or embedding application can create new a primitive data type by calling `scheme_make_type`, which returns a fresh `Scheme_Type` value. To create a collectable instance of this type, allocate memory for the instance with `scheme_malloc_atomic`. From Racket's perspective, the main constraint on the data format of such an instance is that the first `sizeof(Scheme_Object)` bytes must correspond to a `Scheme_Object` record; furthermore, the first `sizeof(Scheme_Type)` bytes must contain the value returned by `scheme_make_type`. Extensions with modest needs can use `scheme_make_cptr`, instead of creating an entirely new type.

Racket values should never be allocated on the stack, and they should never contain pointers to values on the stack. Besides the problem of restricting the value's lifetime to that of the stack frame, allocating values on the stack creates problems for continuations and threads, both of which copy into and out of the stack.

11.1 Standard Types

The following are the `Scheme_Type` values for the standard types:

- `scheme_bool_type` — the constants `scheme_true` and `scheme_false` are the only values of this type; use `SCHEME_FALSEP` to recognize `scheme_false` and use `SCHEME_TRUEP` to recognize anything except `scheme_false`; test for this type with `SCHEME_BOOLP`
- `scheme_char_type` — `SCHEME_CHAR_VAL` extracts the character (of type `mzchar`); test for this type with `SCHEME_CHARP`
- `scheme_integer_type` — fixnum integers, which are identified via the tag bit rather than following a pointer to this `Scheme_Type` value; `SCHEME_INT_VAL` extracts the integer to an `intptr_t`; test for this type with `SCHEME_INTP`
- `scheme_double_type` — flonum inexact numbers; `SCHEME_FLOAT_VAL` or `SCHEME_DBL_VAL` extracts the floating-point value; test for this type with `SCHEME_DBLP`
- `scheme_float_type` — single-precision flonum inexact numbers, when specifically enabled when compiling Racket; `SCHEME_FLOAT_VAL` or `SCHEME_FLT_VAL` extracts the floating-point value; test for this type with `SCHEME_FLTP`
- `scheme_bignum_type` — test for this type with `SCHEME_BIGNUMP`
- `scheme_rational_type` — test for this type with `SCHEME_RATIONALP`
- `scheme_complex_type` — test for this type with `SCHEME_COMPLEXP`
- `scheme_char_string_type` — `SCHEME_CHAR_STR_VAL` extracts the string as a `mzchar*`; the string is always nul-terminated, but may also contain embedded nul characters, and the Racket string is modified if this string is modified; `SCHEME_CHAR_STRLEN_VAL` extracts the string length (in characters, not counting the nul terminator); test for this type with `SCHEME_CHAR_STRINGP`
- `scheme_byte_string_type` — `SCHEME_BYTE_STR_VAL` extracts the string as a `char*`; the string is always nul-terminated, but may also contain embedded nul characters, and the Racket string is modified if this string is modified; `SCHEME_BYTE_STRLEN_VAL` extracts the string length (in bytes, not counting the nul terminator); test for this type with `SCHEME_BYTE_STRINGP`
- `scheme_path_type` — `SCHEME_PATH_VAL` extracts the path as a `char*`; the string is always nul-terminated; `SCHEME_PATH_LEN` extracts the path length (in bytes, not counting the nul terminator); test for this type with `SCHEME_PATHP`
- `scheme_symbol_type` — `SCHEME_SYM_VAL` extracts the symbol's string as a `char*` UTF-8 encoding (do not modify this string); `SCHEME_SYM_LEN` extracts the number of bytes in the symbol name (not counting the nul terminator); test for this type

with `SCHEME_SYMBOLP`; 3m: see §12.1 "Cooperating with 3m" for a caution about `SCHEME_SYM_VAL`

- `scheme_keyword_type` — `SCHEME_KEYWORD_VAL` extracts the keyword's string (without the leading hash colon) as a `char*` UTF-8 encoding (do not modify this string); `SCHEME_KEYWORD_LEN` extracts the number of bytes in the keyword name (not counting the nul terminator); test for this type with `SCHEME_KEYWORDP`; 3m: see §12.1 "Cooperating with 3m" for a caution about `SCHEME_KEYWORD_VAL`

- `scheme_box_type` — `SCHEME_BOX_VAL` extracts/sets the boxed value; test for this type with `SCHEME_BOXP`

- `scheme_pair_type` — `SCHEME_CAR` extracts/sets the `car` and `SCHEME_CDR` extracts/sets the `cdr`; test for this type with `SCHEME_PAIRP`

- `scheme_mutable_pair_type` — `SCHEME_MCAR` extracts/sets the `mcar` and `SCHEME_MCDR` extracts/sets the `mcdr`; test for this type with `SCHEME_MPAIRP`

- `scheme_vector_type` — `SCHEME_VEC_SIZE` extracts the length and `SCHEME_VEC_ELS` extracts the array of Racket values (the Racket vector is modified when this array is modified); test for this type with `SCHEME_VECTORP`; 3m: see §12.1 "Cooperating with 3m" for a caution about `SCHEME_VEC_ELS`

- `scheme_flvector_type` — `SCHEME_FLVEC_SIZE` extracts the length and `SCHEME_FLVEC_ELS` extracts the array of `double`s; test for this type with `SCHEME_FLVECTORP`; 3m: see §12.1 "Cooperating with 3m" for a caution about `SCHEME_FLVEC_ELS`

- `scheme_fxvector_type` — uses the same representation as `scheme_vector_type`, so use `SCHEME_VEC_SIZE` for the length and `SCHEME_VEC_ELS` for the array of Racket fixnum values; test for this type with `SCHEME_FXVECTORP`; 3m: see §12.1 "Cooperating with 3m" for a caution about `SCHEME_VEC_ELS`

- `scheme_structure_type` — structure instances; test for this type with `SCHEME_STRUCTP`

- `scheme_struct_type_type` — structure types; test for this type with `SCHEME_STRUCT_TYPEP`

- `scheme_struct_property_type` — structure type properties

- `scheme_input_port_type` — `SCHEME_INPORT_VAL` extracts/sets the user data pointer; test for just this type with `SCHEME_INPORTP`, but use `SCHEME_INPUT_PORTP` to recognize all input ports (including structures with the `prop:input-port` property), and use `scheme_input_port_record` to extract a `scheme_input_port_type` value from a general input port

- `scheme_output_port_type` — `SCHEME_OUTPORT_VAL` extracts/sets the user data pointer; test for just this type with `SCHEME_OUTPORTP`, but use `SCHEME_OUTPUT_PORTP` to recognize all output ports (including structures with the `prop:output-port` property), and use `scheme_output_port_record` to extract a `scheme_output_port_type` value from a general input port

- `scheme_thread_type` — thread descriptors; test for this type with `SCHEME_THREADP`
- `scheme_sema_type` — semaphores; test for this type with `SCHEME_SEMAP`
- `scheme_hash_table_type` — test for this type with `SCHEME_HASHTP`
- `scheme_hash_tree_type` — test for this type with `SCHEME_HASHTRP`
- `scheme_bucket_table_type` — test for this type with `SCHEME_BUCKTP`
- `scheme_weak_box_type` — test for this type with `SCHEME_WEAKP`; `SCHEME_WEAK_PTR` extracts the contained object, or `NULL` after the content is collected; do not set the content of a weak box
- `scheme_namespace_type` — namespaces; test for this type with `SCHEME_NAMESPACEP`
- `scheme_cpointer_type` — `#<void>` pointer with a type-describing `Scheme_Object`; `SCHEME_CPTR_VAL` extracts the pointer and `SCHEME_CPTR_TYPE` extracts the type tag object; test for this type with `SCHEME_CPTRP`. The tag is used when printing such objects when it's a symbol, a byte string, a string, or a pair holding one of these in its car.

The following are the procedure types:

- `scheme_prim_type` — a primitive procedure, possibly with data elements
- `scheme_closed_prim_type` — an old-style primitive procedure with a data pointer
- `scheme_compiled_closure_type` — a Racket procedure
- `scheme_cont_type` — a continuation
- `scheme_escaping_cont_type` — an escape continuation
- `scheme_case_closure_type` — a `case-lambda` procedure
- `scheme_native_closure_type` — a procedure with native code generated by the just-in-time compiler

The predicate `SCHEME_PROCP` returns 1 for all procedure types and 0 for anything else.

The following are additional number predicates:

- `SCHEME_NUMBERP` — all numerical types

- `SCHEME_REALP` — all non-complex numerical types
- `SCHEME_EXACT_INTEGERP` — fixnums and bignums
- `SCHEME_EXACT_REALP` — fixnums, bignums, and rationals
- `SCHEME_FLOATP` — both single-precision (when enabled) and double-precision flonums

11.2 Global Constants

There are six global constants:

- `scheme_null` — test for this value with `SCHEME_NULLP`
- `scheme_eof` — test for this value with `SCHEME_EOFP`
- `scheme_true`
- `scheme_false` — test for this value with `SCHEME_FALSEP`; test *against* it with `SCHEME_TRUEP`
- `scheme_void` — test for this value with `SCHEME_VOIDP`
- `scheme_undefined`

In some embedding contexts, the function forms `scheme_make_null`, etc., must be used, instead.

11.3 Strings

As noted in §8.6 “Racket BC, Unicode, Characters, and Strings”, a Racket character is a Unicode code point represented by a `mzchar` value, and character strings are `mzchar` arrays. Racket also supplies byte strings, which are `char` arrays.

For a character string *s*, `SCHEME_CHAR_STR_VAL(`*s*`)` produces a pointer to `mzchar`s, not `char`s. Convert a character string to its UTF-8 encoding as byte string with `scheme_char_string_to_byte_string`. For a byte string *bs*, `SCHEME_BYTE_STR_VAL(`*bs*`)` produces a pointer to `char`s. The function `scheme_byte_string_to_char_string` decodes a byte string as UTF-8 and produces a character string. The functions `scheme_char_string_to_byte_string_locale` and `scheme_byte_string_to_char_string_locale` are similar, but they use the current locale’s encoding instead of UTF-8.

For more fine-grained control over UTF-8 encoding, use the `scheme_utf8_decode` and `scheme_utf8_encode` functions, which are described in §20 “String Encodings (BC)”.

11.4 Value Functions

`Scheme_Object* scheme_make_null()`

Returns `scheme_null`.

`Scheme_Object* scheme_make_eof()`

Returns `scheme_eof`.

`Scheme_Object* scheme_make_true()`

Returns `scheme_true`.

`Scheme_Object* scheme_make_false()`

Returns `scheme_false`.

`Scheme_Object* scheme_make_void()`

Returns `scheme_void`.

`Scheme_Object* scheme_make_char(mzchar` *ch*`)`

Returns the character value. The *ch* value must be a legal Unicode code point (and not a surrogate, for example). The first 256 characters are represented by constant Racket values, and others are allocated.

`Scheme_Object* scheme_make_char_or_null(mzchar` *ch*`)`

Like `scheme_make_char`, but the result is `NULL` if *ch* is not a legal Unicode code point.

`Scheme_Object* scheme_make_character(mzchar` *ch*`)`

Returns the character value. This is a macro that directly accesses the array of constant characters when *ch* is less than 256.

`Scheme_Object* scheme_make_ascii_character(mzchar` *ch*`)`

Returns the character value, assuming that *ch* is less than 256. (This is a macro.)

`Scheme_Object* scheme_make_integer(intptr_t` *i*`)`

Returns the integer value; *i* must fit in a fixnum. (This is a macro.)

`Scheme_Object* scheme_make_integer_value(intptr_t` *i*`)`

Returns the integer value. If *i* does not fit in a fixnum, a bignum is returned.

`Scheme_Object*`
`scheme_make_integer_value_from_unsigned(uintptr_t` *i*`)`

Like `scheme_make_integer_value`, but for unsigned integers.

```
 Scheme_Object*
scheme_make_integer_value_from_long_long(mzlonglong i)
```

Like `scheme_make_integer_value`, but for `mzlonglong` values (see §8.7 “Racket BC Integers”).

```
 Scheme_Object*
scheme_make_integer_value_from_unsigned_long_long(umzlonglong i)
```

Like `scheme_make_integer_value_from_long_long`, but for unsigned integers.

```
 Scheme_Object*
scheme_make_integer_value_from_long_halves(uintptr_t hi,
                                           uintptr_t lo)
```

Creates an integer given the high and low `intptr_t`s of a signed integer. Note that on 64-bit platforms where `long long` is the same as `intptr_t`, the resulting integer has 128 bits. (See also §8.7 “Racket BC Integers”.)

```
 Scheme_Object*
scheme_make_integer_value_from_unsigned_long_halves(uintptr_t hi,
                                                    uintptr_t lo)
```

Creates an integer given the high and low `intptr_t`s of an unsigned integer. Note that on 64-bit platforms where `long long` is the same as `intptr_t`, the resulting integer has 128 bits.

```
int scheme_get_int_val(Scheme_Object* o,
                       intptr_t* i)
```

Extracts the integer value. Unlike the `SCHEME_INT_VAL` macro, this procedure will extract an integer that fits in a `intptr_t` from a Racket bignum. If *o* fits in a `intptr_t`, the extracted integer is placed in **i* and 1 is returned; otherwise, 0 is returned and **i* is unmodified.

```
int scheme_get_unsigned_int_val(Scheme_Object* o,
                                uintptr_t* i)
```

Like `scheme_get_int_val`, but for unsigned integers.

```
int scheme_get_long_long_val(Scheme_Object* o,
                             mzlonglong* i)
```

Like `scheme_get_int_val`, but for `mzlonglong` values (see §8.7 “Racket BC Integers”).

```
int scheme_get_unsigned_long_long_val(Scheme_Object* o,
                                      umzlonglong* i)
```

Like `scheme_get_int_val`, but for unsigned `mzlonglong` values (see §8.7 “Racket BC

Integers”).

```
Scheme_Object* scheme_make_double(double d)
```

Creates a new floating-point value.

```
Scheme_Object* scheme_make_float(float d)
```

Creates a new single-precision floating-point value. The procedure is available only when Racket is compiled with single-precision numbers enabled.

```
double scheme_real_to_double(Scheme_Object* o)
```

Converts a Racket real number to a double-precision floating-point value.

```
Scheme_Object* scheme_make_pair(Scheme_Object* carv,
                                Scheme_Object* cdrv)
```

Makes a `cons` pair.

```
Scheme_Object* scheme_make_byte_string(char* bytes)
```

Makes a Racket byte string from a nul-terminated C string. The *bytes* string is copied.

```
 Scheme_Object*
scheme_make_byte_string_without_copying(char* bytes)
```

Like `scheme_make_byte_string`, but the string is not copied.

```
Scheme_Object* scheme_make_sized_byte_string(char* bytes,
                                             intptr_t len,
                                             int copy)
```

Makes a byte string value with size *len*. A copy of *bytes* is made if *copy* is not 0. The string *bytes* should contain *len* bytes; *bytes* can contain the nul byte at any position, and need not be nul-terminated if *copy* is non-zero. However, if *len* is negative, then the nul-terminated length of *bytes* is used for the length, and if *copy* is zero, then *bytes* must be nul-terminated.

```
Scheme_Object* scheme_make_sized_offset_byte_string(char* bytes,
                                                    intptr_t d,
                                                    intptr_t len,
                                                    int copy)
```

Like `scheme_make_sized_byte_string`, except the *len* characters start from position *d* in *bytes*. If *d* is non-zero, then *copy* must be non-zero.

```
Scheme_Object* scheme_alloc_byte_string(intptr_t size,
                                        char fill)
```

Allocates a new Racket byte string.

```
Scheme_Object* scheme_append_byte_string(Scheme_Object* a,
                                         Scheme_Object* b)
```

Creates a new byte string by appending the two given byte strings.

```
Scheme_Object* scheme_make_locale_string(char* bytes)
```

Makes a Racket string from a nul-terminated byte string that is a locale-specific encoding of a character string; a new string is allocated during decoding. The “locale in the name of this function thus refers to *bytes*, and not the resulting string (which is internally stored as UCS-4).

```
Scheme_Object* scheme_make_utf8_string(char* bytes)
```

Makes a Racket string from a nul-terminated byte string that is a UTF-8 encoding. A new string is allocated during decoding. The “utf8” in the name of this function thus refers to *bytes*, and not the resulting string (which is internally stored as UCS-4).

```
Scheme_Object* scheme_make_sized_utf8_string(char* bytes,
                                             intptr_t len)
```

Makes a string value, based on *len* UTF-8-encoding bytes (so the resulting string is *len* characters or less). The string *bytes* should contain at least *len* bytes; *bytes* can contain the nul byte at any position, and need not be null-terminated. However, if *len* is negative, then the nul-terminated length of *bytes* is used for the length.

```
Scheme_Object* scheme_make_sized_offset_utf8_string(char* bytes,
                                                    intptr_t d,
                                                    intptr_t len)
```

Like `scheme_make_sized_char_string`, except the *len* characters start from position *d* in *bytes*.

```
Scheme_Object* scheme_make_char_string(mzchar* chars)
```

Makes a Racket string from a nul-terminated UCS-4 string. The *chars* string is copied.

```
 Scheme_Object*
scheme_make_char_string_without_copying(mzchar* chars)
```

Like `scheme_make_char_string`, but the string is not copied.

```
Scheme_Object* scheme_make_sized_char_string(mzchar* chars,
                                             intptr_t len,
                                             int copy)
```

Makes a string value with size *len*. A copy of *chars* is made if *copy* is not 0. The string *chars* should contain *len* characters; *chars* can contain the nul character at any position, and need not be nul-terminated if *copy* is non-zero. However, if *len* is negative, then the nul-terminated length of *chars* is used for the length, and if *copy* is zero, then the *chars* must be nul-terminated.

```
 Scheme_Object*
scheme_make_sized_offset_char_string(mzchar* chars,
                                     intptr_t d,
                                     intptr_t len,
                                     int copy)
```

Like `scheme_make_sized_char_string`, except the *len* characters start from position *d* in *chars*. If *d* is non-zero, then *copy* must be non-zero.

```
Scheme_Object* scheme_alloc_char_string(intptr_t size,
                                        mzchar fill)
```

Allocates a new Racket string.

```
Scheme_Object* scheme_append_char_string(Scheme_Object* a,
                                         Scheme_Object* b)
```

Creates a new string by appending the two given strings.

```
 Scheme_Object*
scheme_char_string_to_byte_string(Scheme_Object* s)
```

Converts a Racket character string into a Racket byte string via UTF-8.

```
 Scheme_Object*
scheme_byte_string_to_char_string(Scheme_Object* s)
```

Converts a Racket byte string into a Racket character string via UTF-8.

```
 Scheme_Object*
scheme_char_string_to_byte_string_locale(Scheme_Object* s)
```

Converts a Racket character string into a Racket byte string via the locale's encoding.

```
 Scheme_Object*
scheme_byte_string_to_char_string_locale(Scheme_Object* s)
```

Converts a Racket byte string into a Racket character string via the locale's encoding.

```
Scheme_Object* scheme_intern_symbol(char* name)
```

Finds (or creates) the symbol matching the given nul-terminated, ASCII string (not UTF-8). The case of *name* is (non-destructively) normalized before interning if `scheme_case_sensitive` is 0.

```
Scheme_Object* scheme_intern_exact_symbol(char* name,
                                          int len)
```

Creates or finds a symbol given the symbol's length in UTF-8-encoding bytes. The case of *name* is not normalized.

```
Scheme_Object* scheme_intern_exact_char_symbol(mzchar* name,
                                               int len)
```

Like `scheme_intern_exact_symbol`, but given a character array instead of a UTF-8-encoding byte array.

```
Scheme_Object* scheme_make_symbol(char* name)
```

Creates an uninterned symbol from a nul-terminated, UTF-8-encoding string. The case is not normalized.

```
Scheme_Object* scheme_make_exact_symbol(char* name,
                                        int len)
```

Creates an uninterned symbol given the symbol's length in UTF-8-encoded bytes.

```
Scheme_Object* scheme_intern_exact_keyword(char* name,
                                           int len)
```

Creates or finds a keyword given the keywords length in UTF-8-encoding bytes. The case of *name* is not normalized, and it should not include the leading hash and colon of the keyword's printed form.

```
Scheme_Object* scheme_intern_exact_char_keyword(mzchar* name,
                                                int len)
```

Like `scheme_intern_exact_keyword`, but given a character array instead of a UTF-8-encoding byte array.

```
Scheme_Object* scheme_make_vector(intptr_t size,
                                  Scheme_Object* fill)
```

Allocates a new vector.

```
Scheme_Double_Vector* scheme_alloc_flvector(intptr_t size)
```

Allocates an uninitialized flvector. The result type is effectively an alias for `Scheme_Object*`.

```
Scheme_Vector* scheme_alloc_fxvector(intptr_t size)
```

Allocates an uninitialized fxvector. The result type is effectively an alias for `Scheme_Object*`.

```
Scheme_Object* scheme_box(Scheme_Object* v)
```

Creates a new box containing the value *v*.

```
Scheme_Object* scheme_make_weak_box(Scheme_Object* v)
```

Creates a new weak box containing the value *v*.

```
Scheme_Type scheme_make_type(char* name)
```

Creates a new type (not a Racket value). The type tag is valid across all places.

```
Scheme_Object* scheme_make_cptr(void* ptr,
                                const Scheme_Object* typetag)
```

Creates a C-pointer object that encapsulates *ptr* and uses *typetag* to identify the type of the pointer. The `SCHEME_CPTRP` macro recognizes objects created by `scheme_make_cptr`. The `SCHEME_CPTR_VAL` macro extracts the original *ptr* from the Racket object, and `SCHEME_CPTR_TYPE` extracts the type tag. The `SCHEME_CPTR_OFFSETVAL` macro returns `0` for the result Racket object.

The *ptr* can refer to either memory managed by the garbage collector or by some other memory manager. Beware, however, of retaining a *ptr* that refers to memory released by another memory manager, since the enclosing memory range might later become managed by the garbage collector (in which case *ptr* might become an invalid pointer that can crash the garbage collector).

```
 Scheme_Object*
scheme_make_external_cptr(void* ptr,
                          const Scheme_Object* typetag)
```

Like `scheme_make_cptr`, but *ptr* is never treated as referencing memory managed by the garbage collector.

```
 Scheme_Object*
scheme_make_offset_cptr(void* ptr,
                        intptr_t offset,
                        const Scheme_Object* typetag)
```

Creates a C-pointer object that encapsulates both *ptr* and *offset*. The `SCHEME_CPTR_OFFSETVAL` macro returns *offset* for the result Racket object (and the macro be used to change the offset, since it also works on objects with no offset).

The *ptr* can refer to either memory managed by the garbage collector or by some other memory manager; see also `scheme_make_cptr`.

```
 Scheme_Object*
scheme_make_offset_external_cptr(void* ptr,
                                 intptr_t offset,
                                 const Scheme_Object* typetag)
```

Like `scheme_make_offset_cptr`, but *ptr* is never treated as referencing memory managed by the garbage collector.

```
void scheme_set_type_printer(Scheme_Type type,
                             Scheme_Type_Printer printer)
```

Installs a printer to be used for printing (or writing or displaying) values that have the type tag *type*.

The type of *printer* is defined as follows:

```
typedef void (*Scheme_Type_Printer)(Scheme_Object *v, int dis,
                                    Scheme_Print_Params *pp);
```

Such a printer must print a representation of the value using `scheme_print_bytes` and `scheme_print_string`. The first argument to the printer, *v*, is the value to be printed. The second argument indicates whether *v* is printed via `write` or `display`. The last argument is to be passed on to `scheme_print_bytes` or `scheme_print_string` to identify the printing context.

```
void scheme_print_bytes(Scheme_Print_Params* pp,
                        const char* str,
                        int offset,
                        int len)
```

Writes the content of *str* — starting from *offset* and running *len* bytes — into a printing context determined by *pp*. This function is for use by a printer that is installed with `scheme_set_type_printer`.

```
void scheme_print_string(Scheme_Print_Params* pp,
                         const mzchar* str,
                         int offset,
                         int len)
```

Writes the content of *str* — starting from *offset* and running *len* characters — into a printing context determined by *pp*. This function is for use by a printer that is installed with `scheme_set_type_printer`.

```
void scheme_set_type_equality(Scheme_Type type,
                              Scheme_Equal_Proc equalp,
                              Scheme_Primary_Hash_Proc hash1,
                              Scheme_Secondary_Hash_Proc hash2)
```

Installs an equality predicate and associated hash functions for values that have the type tag *type*. The *equalp* predicate is only applied to values that both have tag *type*.

The type of *equalp*, *hash1*, and *hash2* are defined as follows:

```
typedef int (*Scheme_Equal_Proc)(Scheme_Object* obj1,
                                 Scheme_Object* obj2,
                                 void* cycle_data);
typedef intptr_t (*Scheme_Primary_Hash_Proc)(Scheme_Object* obj,
                                         intptr_t base,
                                         void* cycle_data);
typedef intptr_t (*Scheme_Secondary_Hash_Proc)(Scheme_Object* obj,
                                          void* cycle_data);
```

The two hash functions are use to generate primary and secondary keys for double hashing in an `equal?`-based hash table. The result of the primary-key function should depend on both *obj* and *base*.

The *cycle_data* argument in each case allows checking and hashing on cyclic values. It is intended for use in recursive checking or hashing via `scheme_recur_equal`, `scheme_recur_equal_hash_key`, and `scheme_recur_equal_hash_key`. That is, do not call plain `scheme_equal`, `scheme_equal_hash_key`, or `scheme_equal_hash_key` for recursive checking or hashing on sub-elements of the given value(s).

12 Memory Allocation (BC)

Racket uses both `malloc` and allocation functions provided by a garbage collector. Foreign-function and embedding/extension C code may use either allocation method, keeping in mind that pointers to garbage-collectable blocks in `malloc`ed memory are invisible (i.e., such pointers will not prevent the block from being garbage-collected).

Racket CGC uses a conservative garbage collector. This garbage collector normally only recognizes pointers to the beginning of allocated objects. Thus, a pointer into the middle of a GC-allocated string will normally not keep the string from being collected. The exception to this rule is that pointers saved on the stack or in registers may point to the middle of a collectable object. Thus, it is safe to loop over an array by incrementing a local pointer variable.

Racket 3m uses a precise garbage collector that moves objects during collection, in which case the C code must be instrumented to expose local pointer bindings to the collector, and to provide tracing procedures for (tagged) records containing pointers. This instrumentation is described further in §12.1 “Cooperating with 3m”.

The basic collector allocation functions are:

- `scheme_malloc` — Allocates collectable memory that may contain pointers to collectable objects; for 3m, the memory must be an array of pointers (though not necessarily to collectable objects). The newly allocated memory is initially zeroed.
- `scheme_malloc_atomic` — Allocates collectable memory that does not contain pointers to collectable objects. If the memory does contain pointers, they are invisible to the collector and will not prevent an object from being collected. Newly allocated atomic memory is not necessarily zeroed.

  Atomic memory is used for strings or other blocks of memory which do not contain pointers. Atomic memory can also be used to store intentionally-hidden pointers.
- `scheme_malloc_tagged` — Allocates collectable memory that contains a mixture of pointers and atomic data. With the conservative collector, this function is the same as `scheme_malloc`, but on 3m, the type tag stored at the start of the block is used to determine the size and shape of the object for future garbage collection (as described in §12.1 “Cooperating with 3m”).
- `scheme_malloc_allow_interior` — Allocates an array of pointers with special treatment by 3m: the array is never moved by the garbage collector, references are allowed into the middle of the block, and even-valued pointers to the middle of the block prevent the block from being collected. (Beware that the memory manager treats any odd-valued pointer as a fixnum, even if it refers to the middle of a block that allows interior pointers.) Use this procedure sparingly, because small, non-moving objects are handled less efficiently than movable objects by the 3m collector. This procedure is the same as `scheme_malloc` with the conservative collector, but in the

that case, having *only* a pointer into the interior will not prevent the array from being collected.

- `scheme_malloc_atomic_allow_interior` — Like `scheme_malloc_allow_interior` for memory that does not contain pointers.
- `scheme_malloc_uncollectable` — Allocates uncollectable memory that may contain pointers to collectable objects. There is no way to free the memory. The newly allocated memory is initially zeroed. This function is not available in 3m.

If a Racket extension stores Racket pointers in a global or static variable, then that variable must be registered with `scheme_register_extension_global`; this makes the pointer visible to the garbage collector. Registered variables need not contain a collectable pointer at all times (even with 3m, but the variable must contain some pointer, possibly uncollectable, at all times). Beware that static or global variables that are not thread-specific (in the OS sense of "thread") generally do not work with multiple places.

Registration is needed for the global and static variables of an embedding program on most platforms, and registration is needed on all platforms if the program calls `scheme_main_setup` or `scheme_set_stack_base` with a non-zero first or second (respectively) argument. Global and static variables containing collectable pointers must be registered with `scheme_register_static`. The `MZ_REGISTER_STATIC` macro takes any variable name and registers it with `scheme_register_static`. The `scheme_register_static` function can be safely called even when it's not needed, but it must not be called multiple times for a single memory address. When using `scheme_set_stack_base` and when places are enabled, then `scheme_register_static` or `MZ_REGISTER_STATIC` normally should be used only after `scheme_basic_env`, since `scheme_basic_env` changes the allocation space as explained in §12.1.5 "Places and Garbage Collector Instances".

Collectable memory can be temporarily locked from collection by using the reference-counting function `scheme_dont_gc_ptr`. On 3m, such locking does not prevent the object from being moved.

Garbage collection can occur during any call into Racket or its allocator, on anytime that Racket has control, except during functions that are documented otherwise. The predicate and accessor macros listed in §11.1 "Standard Types" never trigger a collection.

As described in §12.1.5 "Places and Garbage Collector Instances", different places manage allocation separately. Movable memory should not be communicated from one place to another, since the source place might move the memory before it is used in the destination place. Furthermore, allocated memory that contains pointers must not be written in a place other than the one where it is allocated, due to the place-specific implementation of a write barrier for generational garbage collection. No write barrier is used for memory that is allocated by `scheme_malloc_atomic_allow_interior` to contain no pointers.

12.1 Cooperating with 3m

To allow 3m's precise collector to detect and update pointers during garbage collection, all pointer values must be registered with the collector, at least during the times that a collection may occur. The content of a word registered as a pointer must contain either `NULL`, a pointer to the start of a collectable object, a pointer into an object allocated by `scheme_malloc_allow_interior`, a pointer to an object currently allocated by another memory manager (and therefore not into a block that is currently managed by the collector), or a pointer to an odd-numbered address (e.g., a Racket fixnum).

Pointers are registered in three different ways:

- Pointers in static variables should be registered with `scheme_register_static` or `MZ_REGISTER_STATIC`.
- Pointers in allocated memory are registered automatically when they are in an array allocated with `scheme_malloc`, etc. When a pointer resides in an object allocated with `scheme_malloc_tagged`, etc.∼the tag at the start of the object identifiers the object's size and shape. Handling of tags is described in §12.1.1 "Tagged Objects".
- Local pointers (i.e., pointers on the stack or in registers) must be registered through the `MZ_GC_DECL_REG`, etc. macros that are described in §12.1.2 "Local Pointers".

A pointer must never refer to the interior of an allocated object (when a garbage collection is possible), unless the object was allocated with `scheme_malloc_allow_interior`. For this reason, pointer arithmetic must usually be avoided, unless the variable holding the generated pointer is `NULL`ed before a collection.

IMPORTANT: The `SCHEME_SYM_VAL`, `SCHEME_KEYWORD_VAL`, `SCHEME_VEC_ELS`, and `SCHEME_PRIM_CLOSURE_ELS` macros produce pointers into the middle of their respective objects, so the results of these macros must not be held during the time that a collection can occur. Incorrectly retaining such a pointer can lead to a crash.

Tagged Objects

As explained in §11 "Values and Types (BC)", the `scheme_make_type` function can be used to obtain a new tag for a new type of object. These new types are in relatively short supply for 3m; the maximum tag is 512, and Racket itself uses nearly 300.

After allocating a new tag in 3m (and before creating instances of the tag), a *size procedure*, a *mark procedure*, and a *fixup procedure* must be installed for the tag using `GC_register_traversers`. A type tag and its associated GC procedures apply to all places, even though specific allocated objects are confined to a particular place.

A size procedure simply takes a pointer to an object with the tag and returns its size in words (not bytes). The `gcBYTES_TO_WORDS` macro converts a byte count to a word count.

A mark procedure is used to trace references among objects. The procedure takes a pointer to an object, and it should apply the `gcMARK` macro to every pointer within the object. The mark procedure should return the same result as the size procedure.

A fixup procedure is potentially used to update references to objects that have moved, although the mark procedure may have moved objects and updated references already. The fixup procedure takes a pointer to an object, and it should apply the `gcFIXUP` macro to every pointer within the object. The fixup procedure should return the same result as the size procedure.

Depending on the collector's implementation, the `gcMARK` and/or `gcFIXUP` macros may take take the address of their arguments, and the fixup procedure might not be used. For example, the collector may only use the mark procedure and not actually move the object. Or it may use mark to move objects at the same time. To dereference an object pointer during a mark or fixup procedure, use `GC_resolve` to convert a potentially old address to the location where the object has been moved. To dereference an object pointer during a fixup procedure, use `GC_fixup_self` to convert the address passed to the procedure to refer to the potentially moved object.

When allocating a tagged object in 3m, the tag must be installed immediately after the object is allocated—or, at least, before the next possible collection.

Local Pointers

The 3m collector needs to know the address of every local or temporary pointer within a function call at any point when a collection can be triggered. Beware that nested function calls can hide temporary pointers; for example, in

```
scheme_make_pair(scheme_make_pair(scheme_true, scheme_false),
                 scheme_make_pair(scheme_false, scheme_true))
```

the result from one `scheme_make_pair` call is on the stack or in a register during the other call to `scheme_make_pair`; this pointer must be exposed to the garbage collection and made subject to update. Simply changing the code to

```
tmp = scheme_make_pair(scheme_true, scheme_false);
scheme_make_pair(tmp,
                 scheme_make_pair(scheme_false, scheme_true))
```

does not expose all pointers, since `tmp` must be evaluated before the second call to `scheme_make_pair`. In general, the above code must be converted to the form

```
tmp1 = scheme_make_pair(scheme_true, scheme_false);
```

```
  tmp2 = scheme_make_pair(scheme_true, scheme_false);
  scheme_make_pair(tmp1, tmp2);
```

and this is converted form must be instrumented to register `tmp1` and `tmp2`. The final result might be

```
  {
    Scheme_Object *tmp1 = NULL, *tmp2 = NULL, *result;
    MZ_GC_DECL_REG(2);

    MZ_GC_VAR_IN_REG(0, tmp1);
    MZ_GC_VAR_IN_REG(1, tmp2);
    MZ_GC_REG();

    tmp1 = scheme_make_pair(scheme_true, scheme_false);
    tmp2 = scheme_make_pair(scheme_true, scheme_false);
    result = scheme_make_pair(tmp1, tmp2);

    MZ_GC_UNREG();

    return result;
  }
```

Notice that `result` is not registered above. The `MZ_GC_UNREG` macro cannot trigger a garbage collection, so the `result` variable is never live during a potential collection. Note also that `tmp1` and `tmp2` are initialized with `NULL`, so that they always contain a pointer whenever a collection is possible.

The `MZ_GC_DECL_REG` macro expands to a local-variable declaration to hold information for the garbage collector. The argument is the number of slots to provide for registration. Registering a simple pointer requires a single slot, whereas registering an array of pointers requires three slots. For example, to register a pointer `tmp` and an array of 10 `char*`s:

```
  {
    Scheme_Object *tmp1 = NULL;
    char *a[10];
    int i;
    MZ_GC_DECL_REG(4);

    MZ_GC_ARRAY_VAR_IN_REG(0, a, 10);
    MZ_GC_VAR_IN_REG(3, tmp1);
    /* Clear a before a potential GC: */
    for (i = 0; i < 10; i++) a[i] = NULL;
    ...
    f(a);
    ...
```

```
}
```

The `MZ_GC_ARRAY_VAR_IN_REG` macro registers a local array given a starting slot, the array variable, and an array size. The `MZ_GC_VAR_IN_REG` macro takes a slot and simple pointer variable. A local variable or array must not be registered multiple times.

In the above example, the first argument to `MZ_GC_VAR_IN_REG` is `3` because the information for `a` uses the first three slots. Even if `a` is not used after the call to `f`, `a` must be registered with the collector during the entire call to `f`, because `f` presumably uses `a` until it returns.

The name used for a variable need not be immediate. Structure members can be supplied as well:

```
{
  struct { void *s; int v; void *t; } x = {NULL, 0, NULL};
  MZ_GC_DECL_REG(2);

  MZ_GC_VAR_IN_REG(0, x.s);
  MZ_GC_VAR_IN_REG(0, x.t);
  ...
}
```

In general, the only constraint on the second argument to `MZ_GC_VAR_IN_REG` or `MZ_GC_ARRAY_VAR_IN_REG` is that `&` must produce the relevant address, and that address must be on the stack.

Pointer information is not actually registered with the collector until the `MZ_GC_REG` macro is used. The `MZ_GC_UNREG` macro de-registers the information. Each call to `MZ_GC_REG` must be balanced by one call to `MZ_GC_UNREG`.

Pointer information need not be initialized with `MZ_GC_VAR_IN_REG` and `MZ_GC_ARRAY_VAR_IN_REG` before calling `MZ_GC_REG`, and the set of registered pointers can change at any time—as long as all relevant pointers are registered when a collection might occur. The following example recycles slots and completely de-registers information when no pointers are relevant. The example also illustrates how `MZ_GC_UNREG` is not needed when control escapes from the function, such as when `scheme_signal_error` escapes.

```
{
  Scheme_Object *tmp1 = NULL, *tmp2 = NULL;
  mzchar *a, *b;
  MZ_GC_DECL_REG(2);

  MZ_GC_VAR_IN_REG(0, tmp1);
  MZ_GC_VAR_IN_REG(1, tmp2);

  tmp1 = scheme_make_utf8_string("foo");
  MZ_GC_REG();
```

```
    tmp2 = scheme_make_utf8_string("bar");
    tmp1 = scheme_append_char_string(tmp1, tmp2);

    if (SCHEME_FALSEP(tmp1))
      scheme_signal_error("shouldn't happen!");

    a = SCHEME_CHAR_VAL(tmp1);

    MZ_GC_VAR_IN_REG(0, a);

    tmp2 = scheme_make_pair(scheme_read_bignum(a, 0, 10), tmp2);

    MZ_GC_UNREG();

    if (SCHEME_INTP(tmp2)) {
      return 0;
    }

    MZ_GC_REG();
    tmp1 = scheme_make_pair(scheme_read_bignum(a, 0, 8), tmp2);
    MZ_GC_UNREG();

    return tmp1;
  }
```

A `MZ_GC_DECL_REG` can be used in a nested block to hold declarations for the block's variables. In that case, the nested `MZ_GC_DECL_REG` must have its own `MZ_GC_REG` and `MZ_GC_UNREG` calls.

```
  {
    Scheme_Object *accum = NULL;
    MZ_GC_DECL_REG(1);
    MZ_GC_VAR_IN_REG(0, accum);
    MZ_GC_REG();

    accum = scheme_make_pair(scheme_true, scheme_null);
    {
      Scheme_Object *tmp = NULL;
      MZ_GC_DECL_REG(1);
      MZ_GC_VAR_IN_REG(0, tmp);
      MZ_GC_REG();

      tmp = scheme_make_pair(scheme_true, scheme_false);
      accum = scheme_make_pair(tmp, accum);

      MZ_GC_UNREG();
```

```
  }
  accum = scheme_make_pair(scheme_true, accum);

  MZ_GC_UNREG();
  return accum;
}
```

Variables declared in a local block can also be registered together with variables from an enclosing block, but the local-block variable must be unregistered before it goes out of scope. The `MZ_GC_NO_VAR_IN_REG` macro can be used to unregister a variable or to initialize a slot as having no variable.

```
{
  Scheme_Object *accum = NULL;
  MZ_GC_DECL_REG(2);
  MZ_GC_VAR_IN_REG(0, accum);
  MZ_GC_NO_VAR_IN_REG(1);
  MZ_GC_REG();

  accum = scheme_make_pair(scheme_true, scheme_null);
  {
    Scheme_Object *tmp = NULL;
    MZ_GC_VAR_IN_REG(1, tmp);

    tmp = scheme_make_pair(scheme_true, scheme_false);
    accum = scheme_make_pair(tmp, accum);

    MZ_GC_NO_VAR_IN_REG(1);
  }
  accum = scheme_make_pair(scheme_true, accum);

  MZ_GC_UNREG();
  return accum;
}
```

The `MZ_GC_` macros all expand to nothing when `MZ_PRECISE_GC` is not defined, so the macros can be placed into code to be compiled for both conservative and precise collection.

The `MZ_GC_REG` and `MZ_GC_UNREG` macros must never be used in an OS thread other than Racket's thread.

Local Pointers and `raco ctool --xform`

When `raco ctool` is run with the `--xform` flag and a source C program, it produces a C program that is instrumented in the way described in the previous section (but with a slightly different set of macros). For each input file `"`*`name`*`.c"`, the transformed output is `"`*`name`*`.3m.c"`.

The `--xform` mode for `raco ctool` does not change allocation calls, nor does it generate size, mark, or fixup procedures. It merely converts the code to register local pointers.

Furthermore, the `--xform` mode for `raco ctool` does not handle all of C. It's ability to rearrange compound expressions is particularly limited, because `--xform` merely converts expression text heuristically instead of parsing C. A future version of the tool will correct such problems. For now, `raco ctool` in `--xform` mode attempts to provide reasonable error messages when it is unable to convert a program, but beware that it can miss cases. To an even more limited degree, `--xform` can work on C++ code. Inspect the output of `--xform` mode to ensure that your code is correctly instrumented.

Some specific limitations:

- The body of a `for`, `while`, or `do` loop must be surrounded with curly braces. (A conversion error is normally reported, otherwise.)
- Function calls may not appear on the right-hand side of an assignment within a declaration block. (A conversion error is normally reported if such an assignment is discovered.)
- Multiple function calls in `... ? ... : ...` cannot be lifted. (A conversion error is normally reported, otherwise.)
- In an assignment, the left-hand side must be a local or static variable, not a field selection, pointer dereference, etc. (A conversion error is normally reported, otherwise.)
- The conversion assumes that all function calls use an immediate name for a function, as opposed to a compound expression as in `s->f()`. The function name need not be a top-level function name, but it must be bound either as an argument or local variable with the form *type id*; the syntax *ret_type* `(*`*id*`)(...)` is not recognized, so bind the function type to a simple name with `typedef`, first: `typedef` *ret_type* `(*`*type*`)(...); ....` *type id*.
- Arrays and structs must be passed by address, only.
- GC-triggering code must not appear in system headers.
- Pointer-comparison expressions are not handled correctly when either of the compared expressions includes a function call. For example, `a() == b()` is not converted correctly when `a` and `b` produce pointer values.
- Passing the address of a local pointer to a function works only when the pointer variable remains live after the function call.
- A `return;` form can get converted to `{` *stmt*`; return; };`, which can break an `if (...) return; else ...` pattern.

- Local instances of union types are generally not supported.
- Pointer arithmetic cannot be converted away, and is instead reported as an error.

Guiding `raco ctool --xform`

The following macros can be used (with care!) to navigate `--xform` around code that it cannot handle:

- `XFORM_START_SKIP` and `XFORM_END_SKIP`: code between these two statements is ignored by the transform tool, except to tokenize it.

  Example:

  ```
  int foo(int c, ...) {
    int r = 0;
    XFORM_START_SKIP;
    {
      /* va plays strange tricks that confuse xform */
      va_list args;
      va_start(args, c);
      while (c--) {
        r += va_arg(args, int);
      }
    }
    XFORM_END_SKIP;
    return r;
  }
  ```

  These macros can also be used at the top level, outside of any function. Since they have to be terminated by a semi-colon, however, top-level uses usually must be wrapped with `#ifdef MZ_PRECISE_GC` and `#endif`; a semi-colon by itself at the top level is not legal in C.

- `XFORM_SKIP_PROC`: annotate a function so that its body is skipped in the same way as bracketing it with `XFORM_START_SKIP` and `XFORM_END_SKIP`.

  Example:

  ```
  int foo(int c, ...) XFORM_SKIP_PROC {
  }
  ```

- `XFORM_HIDE_EXPR`: a macro that takes wraps an expression to disable processing of the expression.

  Example:

  ```
  int foo(int c, ...) {
    int r = 0;
    {
  ```

```
    /* va plays strange tricks that confuse xform */
    XFORM_CAN_IGNORE va_list args; /* See below */
    XFORM_HIDE_EXPR(va_start(args, c));
    while (c--) {
      r += XFORM_HIDE_EXPR(va_arg(args, int));
    }
  }
  return r;
}
```

- `XFORM_CAN_IGNORE`: a macro that acts like a type modifier (must appear first) to indicate that a declared variable can be treated as atomic. See above for an example.
- `XFORM_START_SUSPEND` and `XFORM_END_SUSPEND`: for use at the top level (outside of any function definition), and similar to `XFORM_START_SKIP` and `XFORM_END_SKIP` in that function and class bodies are not transformed. Type and prototype information is still collected for use by later transformations, however. These forms must be terminated by a semi-colon.
- `XFORM_START_TRUST_ARITH` and `XFORM_END_TRUST_ARITH`: for use at the top level (outside of any function definition) to disable warnings about pointer arithmetic. Use only when you're absolutely certain that the garbage collector cannot be pointers offset into the middle of a collectable object. These forms must be terminated by a semi-colon.
- `XFORM_TRUST_PLUS`: a replacement for `+` that does not trigger pointer-arithmetic warnings. Use with care.
- `XFORM_TRUST_MINUS`: a replacement for `-` that does not trigger pointer-arithmetic warnings. Use with care.

Places and Garbage Collector Instances

When places are enabled, then a single process can have multiple instances of the garbage collector in the same process. Each place allocates using its own collector, and no place is allowed to hold a reference to memory that is allocated by another place. In addition, a *master* garbage collector instance holds values that are shared among places; different places can refer to memory that is allocated by the master garbage collector, but the master still cannot reference memory allocated by place-specific garbage collectors.

Calling `scheme_main_stack_setup` creates the master garbage collector, and allocation uses that collector until `scheme_basic_env` returns, at which point the initial place's garbage collector is in effect. Using `scheme_register_static` or `MZ_REGISTER_STATIC` before calling `scheme_basic_env` registers an address that should be used to hold only values allocated before `scheme_basic_env` is called. More typically, `scheme_register_static` and `MZ_REGISTER_STATIC` are used only after `scheme_basic_env` returns. Using `scheme_main_setup` calls `scheme_basic_env` automatically, in which case there is no opportunity to use `scheme_register_static` or `MZ_REGISTER_STATIC` too early.

12.2 Memory Functions

```
void* scheme_malloc(size_t n)
```

Allocates *n* bytes of collectable memory, initially filled with zeros. The allocated object is treated as an array of pointers.

```
void* scheme_malloc_atomic(size_t n)
```

Allocates *n* bytes of collectable memory containing no pointers visible to the garbage collector. The object is *not* initialized to zeros.

```
void* scheme_malloc_uncollectable(size_t n)
```

Non-3m, only. Allocates *n* bytes of uncollectable memory.

```
void* scheme_malloc_eternal(size_t n)
```

Allocates uncollectable atomic memory. This function is equivalent to `malloc`, except that the memory cannot be freed.

```
void* scheme_calloc(size_t num,
                    size_t size)
```

Allocates *num* * *size* bytes of memory using `scheme_malloc`.

```
void* scheme_malloc_tagged(size_t n)
```

Like `scheme_malloc`, but in 3m, the type tag determines how the garbage collector traverses the object; see §12 “Memory Allocation (BC)”.

```
void* scheme_malloc_allow_interior(size_t n)
```

Like `scheme_malloc`, but in 3m, the object never moves, and pointers are allowed to reference the middle of the object; see §12 “Memory Allocation (BC)”.

```
void* scheme_malloc_atomic_allow_interior(size_t n)
```

Like `scheme_malloc_atomic`, but in 3m, the object never moves, and pointers are allowed to reference the middle of the object; see §12 “Memory Allocation (BC)”.

```
void* scheme_malloc_stubborn(size_t n)
```

An obsolete variant of `scheme_malloc`, where `scheme_end_stubborn_change` can be called on the allocated pointer when no further changes will be made to the allocated memory. Stubborn allocation is potentially useful as a hint for generational collection, but the hint is normally ignored and unlikely to be used more in future version.

```
void* scheme_end_stubborn_change(void* p)
```

Declares the end of changes to the memory at *p* as allocated via `scheme_malloc_stubborn`.

```
char* scheme_strdup(char* str)
```

Copies the null-terminated string *str*; the copy is collectable.

```
char* scheme_strdup_eternal(char* str)
```

Copies the null-terminated string *str*; the copy will never be freed.

```
void* scheme_malloc_fail_ok(void *(*)(size_t) mallocf,
                            size_t size)
```

Attempts to allocate *size* bytes using *mallocf*. If the allocation fails, the `exn:fail:out-of-memory` exception is raised.

```
void** scheme_malloc_immobile_box(void* p)
```

Allocates memory that is not garbage-collected and that does not move (even with 3m), but whose first word contains a pointer to a collectable object. The box is initialized with *p*, but the value can be changed at any time. An immobile box must be explicitly freed using `scheme_free_immobile_box`.

```
void scheme_free_immobile_box(void** b)
```

Frees an immobile box allocated with `scheme_malloc_immobile_box`.

```
void* scheme_malloc_code(intptr_t size)
```

Allocates non-collectable memory to hold executable machine code. Use this function instead of `malloc` to ensure that the allocated memory has “execute” permissions. Use `scheme_free_code` to free memory allocated by this function.

```
void scheme_free_code(void* p)
```

Frees memory allocated with `scheme_malloc_code`.

```
void scheme_register_extension_global(void* ptr,
                                      intptr_t size)
```

Registers an extension’s global variable that can contain Racket pointers (for the current place). The address of the global is given in *ptr*, and its size in bytes in *size*.

In addition to global variables, this function can be used to register any permanent memory that the collector would otherwise treat as atomic. A garbage collection can occur during the registration.

```
int scheme_main_setup(int no_auto_statics,
                      Scheme_Env_Main main,
                      int argc,
                      char** argv)
```

Initializes the GC stack base, creates the initial namespace by calling `scheme_basic_env`, and then calls *main* with the namespace, *argc*, and *argv*. (The *argc* and *argv* are just passed on to *main*, and are not inspected in any way.)

The `Scheme_Env_Main` type is defined as follows:

```
typedef int (*Scheme_Env_Main)(Scheme_Env *env,
                               int argc, char **argv);
```

The result of *main* is the result of `scheme_main_setup`.

If *no_auto_statics* is non-zero, then static variables must be explicitly registered with the garbage collector; see §12 "Memory Allocation (BC)" for more information.

```
int scheme_main_stack_setup(int no_auto_statics,
                            Scheme_Nested_Main main,
                            void* data)
```

A more primitive variant of `scheme_main_setup` that initializes the GC stack base but does not create the initial namespace (so an embedding application can perform other operations that involve garbage-collected data before creating a namespace).

The *data* argument is passed through to *main*, where the `Scheme_Nested_Main` type is defined as follows:

```
typedef int (*Scheme_Nested_Main)(void *data);
```

```
void scheme_set_stack_base(void* stack_addr,
                           int no_auto_statics)
```

Overrides the GC's auto-determined stack base, and/or disables the GC's automatic traversal of global and static variables. If *stack_addr* is `NULL`, the stack base determined by the GC is used. Otherwise, it should be the "deepest" memory address on the stack where a collectable pointer might be stored. This function should be called only once, and before any other `scheme_` function is called, but only with CGC and when future and places are disabled. The function never triggers a garbage collection.

Example:

```
int main(int argc, char **argv) {
   int dummy;
   scheme_set_stack_base(&dummy, 0);
   real_main(argc, argv); /* calls scheme_basic_env(), etc. */
}
```

On 3m, the above code does not quite work, because *stack_addr* must be the beginning or end of a local-frame registration. Worse, in CGC or 3m, if `real_main` is declared `static`, the compiler may inline it and place variables containing collectable values deeper in the stack than `dummy`. To avoid these problems, use `scheme_main_setup` or `scheme_main_stack_setup`, instead.

The above code also may not work when future and/or places are enabled in Racket, because `scheme_set_stack_base` does not initialize Racket's thread-local variables. Again, use `scheme_main_setup` or `scheme_main_stack_setup` to avoid the problem.

```
void scheme_set_stack_bounds(void* stack_addr,
                             void* stack_end,
                             int no_auto_statics)
```

Like `scheme_set_stack_base`, except for the extra *stack_end* argument. If *stack_end* is non-`NULL`, then it corresponds to a point of C-stack growth after which Racket should attempt to handle stack overflow. The *stack_end* argument should not correspond to the actual stack end, since detecting stack overflow may take a few frames, and since handling stack overflow requires a few frames.

If *stack_end* is `NULL`, then the stack end is computed automatically: the stack size assumed to be the limit reported by `getrlimit` on Unix and Mac OS, or it is assumed to be the stack reservation of the executable (or 1 MB if parsing the executable fails) on Windows; if this size is greater than 8 MB, then 8 MB is assumed, instead; the size is decremented by 50000 bytes (64-bit Windows: 100000 bytes) to cover a large margin of error; finally, the size is subtracted from (for stacks that grow down) or added to (for stacks that grow up) the stack base in *stack_addr* or the automatically computed stack base. Note that the 50000-byte margin of error is assumed to cover the difference between the actual stack start and the reported stack base, in addition to the margin needed for detecting and handling stack overflow.

```
void scheme_register_tls_space(void* ptr,
                               int tls_index)
```

For Windows, registers *ptr* as the address of a thread-local pointer variable that is declared in the main executable. The variable's storage will be used to implement thread-local storage within the Racket run-time. See §9 "Embedding into a Program (BC)".

The *tls_index* argument must be `0`. It is currently ignored, but a future version may use the argument to allow declaration of the thread-local variable in a dynamically linked DLL.

Changed in version 6.3: Changed from available only on 32-bit Windows to available on all Windows variants.

```
void scheme_register_static(void* ptr,
                            intptr_t size)
```

Like `scheme_register_extension_global`, for use in embedding applications in situations where the collector does not automatically find static variables (i.e., when `scheme_set_stack_base` has been called with a non-zero second argument).

The macro `MZ_REGISTER_STATIC` can be used directly on a static variable. It expands to a comment if statics need not be registered, and a call to `scheme_register_static` (with the address of the static variable) otherwise.

```
void scheme_weak_reference(void** p)
```

Registers the pointer **p* as a weak pointer; when no other (non-weak) pointers reference the same memory as **p* references, then **p* will be set to `NULL` by the garbage collector. The value in **p* may change, but the pointer remains weak with respect to the value of **p* at the time *p* was registered.

This function is not available in 3m.

```
void scheme_weak_reference_indirect(void** p,
                                    void* v)
```

Like `scheme_weak_reference`, but **p* is set to `NULL` (regardless of its prior value) when there are no references to *v*.

This function is not available in 3m.

```
void scheme_register_finalizer(void* p,
                               fnl_proc f,
                               void* data,
                               fnl_proc* oldf,
                               void** olddata)
```

Registers a callback function to be invoked when the memory *p* would otherwise be garbage-collected, and when no "will"-like finalizers are registered for *p*.

The `fnl_proc` type is not actually defined, but it is equivalent to

```
typedef void (*fnl_proc)(void *p, void *data)
```

The *f* argument is the callback function; when it is called, it will be passed the value *p* and the data pointer *data*; *data* can be anything — it is only passed on to the callback function. If *oldf* and *olddata* are not `NULL`, then **oldf* and **olddata* are filled with the old callback information (*f* and *data* will override this old callback).

To remove a registered finalizer, pass `NULL` for *f* and *data*.

Note: registering a callback not only keeps *p* from collection until the callback is invoked, but it also keeps *data* reachable until the callback is invoked.

```
void scheme_add_finalizer(void* p,
                          fnl_proc f,
                          void* data)
```

Adds a finalizer to a chain of primitive finalizers. This chain is separate from the single finalizer installed with `scheme_register_finalizer`; all finalizers in the chain are called immediately after a finalizer that is installed with `scheme_register_finalizer`.

See `scheme_register_finalizer`, above, for information about the arguments.

To remove an added finalizer, use `scheme_subtract_finalizer`.

```
void scheme_add_scheme_finalizer(void* p,
                                 fnl_proc f,
                                 void* data)
```

Installs a "will"-like finalizer, similar to `will-register`. Will-like finalizers are called one at a time, requiring the collector to prove that a value has become inaccessible again before calling the next will-like finalizer. Finalizers registered with `scheme_register_finalizer` or `scheme_add_finalizer` are not called until all will-like finalizers have been exhausted.

See `scheme_register_finalizer`, above, for information about the arguments.

There is currently no facility to remove a will-like finalizer.

```
void scheme_add_finalizer_once(void* p,
                               fnl_proc f,
                               void* data)
```

Like `scheme_add_finalizer`, but if the combination *f* and *data* is already registered as a (non-"will"-like) finalizer for *p*, it is not added a second time.

```
void scheme_add_scheme_finalizer_once(void* p,
                                      fnl_proc f,
                                      void* data)
```

Like `scheme_add_scheme_finalizer`, but if the combination of *f* and *data* is already registered as a "will"-like finalizer for *p*, it is not added a second time.

```
void scheme_subtract_finalizer(void* p,
                               fnl_proc f,
                               void* data)
```

Removes a finalizer that was installed with `scheme_add_finalizer`.

```
void scheme_remove_all_finalization(void* p)
```

Removes all finalization ("will"-like or not) for *p*, including wills added in Scheme with `will-register` and finalizers used by custodians.

```
void scheme_dont_gc_ptr(void* p)
```

Keeps the collectable block *p* from garbage collection. Use this procedure when a reference to *p* is be stored somewhere inaccessible to the collector. Once the reference is no longer used from the inaccessible region, de-register the lock with `scheme_gc_ptr_ok`. A garbage collection can occur during the registration.

This function keeps a reference count on the pointers it registers, so two calls to `scheme_dont_gc_ptr` for the same *p* should be balanced with two calls to `scheme_gc_ptr_ok`.

```
void scheme_gc_ptr_ok(void* p)
```

See `scheme_dont_gc_ptr`.

```
void scheme_collect_garbage()
```

Forces an immediate garbage-collection.

```
void scheme_enable_garbage_collection(int on)
```

Garbage collection is enabled only when an internal counter is `0`. Calling `scheme_enable_garbage_collection` with a false value increments the counter, and calling `scheme_enable_garbage_collection` with a true value decrements the counter.

When the `PLTDISABLEGC` environment variable is set, then `racket` initializes the internal counter to 1 to initially disable garbage collection.

```
void GC_register_traversers(short tag,
                            Size_Proc s,
                            Mark_Proc m,
                            Fixup_Proc f,
                            int is_const_size,
                            int is_atomic)
```

3m only. Registers a size, mark, and fixup procedure for a given type tag; see §12.1.1 "Tagged Objects" for more information.

Each of the three procedures takes a pointer and returns an integer:

```
typedef int (*Size_Proc)(void *obj);
typedef int (*Mark_Proc)(void *obj);
typedef int (*Fixup_Proc)(void *obj);
```

If the result of the size procedure is a constant, then pass a non-zero value for *is_const_size*. If the mark and fixup procedures are no-ops, then pass a non-zero value for *is_atomic*.

`void* GC_resolve(void* p)`

3m only. Can be called by a size, mark, or fixup procedure that is registered with `GC_register_traversers`. It returns the current address of an object *p* that might have been moved already. This translation is necessary, for example, if the size or structure of an object depends on the content of an object it references. For example, the size of a class instance usually depends on a field count that is stored in the class. A fixup procedure should call this function on a reference *before* fixing it.

`void* GC_fixup_self(void* p)`

3m only. Can be called by a fixup procedure that is registered with `GC_register_traversers`. It returns the final address of *p*, which must be the pointer passed to the fixup procedure. The `GC_resolve` function would produce the same result, but `GC_fixup_self` may be more efficient. For some implementations of the memory manager, the result is the same as *p*, either because objects are not moved or because the object is moved before it is fixed. With other implementations, an object might be moved after the fixup process, and the result is the location that the object will have after garbage collection finished.

```
void scheme_register_type_gc_shape(short type,
                                   intptr_t* shape)
```

Like `GC_register_traversers`, but using a set of predefined functions that interpret *shape* to traverse a value. The *shape* array is a sequence of commands terminated with `SCHEME_GC_SHAPE_TERM`, where each command has a single argument.

Commands:

- `#define SCHEME_GC_SHAPE_TERM 0` — the terminator command, which has no argument.
- `#define SCHEME_GC_SHAPE_PTR_OFFSET 1` — specifies that a object tagged with *type* has a pointer to be made visible to the garbage collector, where the command argument is the offset from the beginning of the object.
- `#define SCHEME_GC_SHAPE_ADD_SIZE 2` — specifies the allocated size of an object tagged with *type*, where the command argument is an amount to add to an accumulated size; currently, size information is not used, but it may be needed with future implementations of the garbage collector.

To improve forward compatibility, any other command is assumed to take a single argument and is ignored.

A GC-shape registration is place-specific, even though `scheme_make_type` creates a type tag that spans places. If a traversal is already installed for `type` in the current place, the old traversal specification is replaced. The `scheme_register_type_gc_shape` function keeps its own copy of the array *shape*, so the array need not be retained.

Added in version 6.4.0.10.

```
Scheme_Object* scheme_add_gc_callback(Scheme_Object* pre_desc,
                                      Scheme_Object* post_desc)
```

The same as `unsafe-add-collect-callbacks` from `ffi/unsafe/collect-callback`.

```
void scheme_remove_gc_callback(Scheme_Object* key)
```

The same as `unsafe-remove-collect-callbacks`, removes garbage-collection callbacks installed with `scheme_add_gc_callback`.

13 Namespaces and Modules (BC)

A Racket namespace (a top-level environment) is represented by a value of type `Scheme_Env*` — which is also a Racket value, castable to `Scheme_Object*`. Calling `scheme_basic_env` returns a namespace that includes all of Racket's standard global procedures and syntax.

The `scheme_basic_env` function must be called once by an embedding program, before any other Racket function is called (except `scheme_make_param`), but `scheme_main_setup` automatically calls `scheme_basic_env`. The returned namespace is the initial current namespace for the main Racket thread. Racket extensions cannot call `scheme_basic_env`.

The current thread's current namespace is available from `scheme_get_env`, given the current parameterization (see §18 "Parameterizations (BC)"): `scheme_get_env(scheme_config)`.

New values can be added as globals in a namespace using `scheme_add_global`. The `scheme_lookup_global` function takes a Racket symbol and returns the global value for that name, or `NULL` if the symbol is undefined.

A module's set of top-level bindings is implemented using the same machinery as a namespace. Use `scheme_primitive_module` to create a new `Scheme_Env*` that represents a primitive module. The name provided to `scheme_primitive_module` is subject to change through the `current-module-declare-name` parameter (which is normally set by the module name resolver when auto-loading module files). After installing variables into the module with `scheme_add_global`, etc., call `scheme_finish_primitive_module` on the `Scheme_Env*` value to make the module declaration available. All defined variables are exported from the primitive module.

The Racket `#%variable-reference` form produces a value that is opaque to Racket code. Use `SCHEME_PTR_VAL` on the result of `#%variable-reference` to obtain the same kind of value as returned by `scheme_global_bucket` (i.e., a bucket containing the variable's value, or `NULL` if the variable is not yet defined).

```
void scheme_add_global(char* name,
                       Scheme_Object* val,
                       Scheme_Env* env)
```

Adds a value to the table of globals for the namespace *env*, where *name* is a null-terminated string. (The string's case will be normalized in the same way as for interning a symbol.)

```
void scheme_add_global_symbol(Scheme_Object* name,
                              Scheme_Object* val,
                              Scheme_Env* env)
```

Adds a value to the table of globals by symbol name instead of string name.

```
Scheme_Object* scheme_lookup_global(Scheme_Object* symbol,
                                    Scheme_Env* env)
```

Given a global variable name (as a symbol) in *sym*, returns the current value.

```
Scheme_Bucket* scheme_global_bucket(Scheme_Object* symbol,
                                    Scheme_Env* env)
```

Given a global variable name (as a symbol) in *sym*, returns the bucket where the value is stored. When the value in this bucket is `NULL`, then the global variable is undefined.

The `Scheme_Bucket` structure is defined as:

```
typedef struct Scheme_Bucket {
  Scheme_Object so; /* so.type = scheme_variable_type */
  void *key;
  void *val;
} Scheme_Bucket;
```

```
Scheme_Bucket* scheme_module_bucket(Scheme_Object* mod,
                                    Scheme_Object* symbol,
                                    int pos,
                                    Scheme_Env* env)
```

Like `scheme_global_bucket`, but finds a variable in a module. The *mod* and *symbol* arguments are as for `dynamic-require` in Racket. The *pos* argument should be `-1` always. The *env* argument represents the namespace in which the module is declared.

```
void scheme_set_global_bucket(char* procname,
                              Scheme_Bucket* var,
                              Scheme_Object* val,
                              int set_undef)
```

Changes the value of a global variable. The *procname* argument is used to report errors (in case the global variable is constant, not yet bound, or bound as syntax). If *set_undef* is not 1, then the global variable must already have a binding. (For example, `set!` cannot set unbound variables, while `define` can.)

```
Scheme_Object* scheme_builtin_value(const char* name)
```

Gets the binding of a name as it would be defined in the initial namespace.

```
Scheme_Env* scheme_get_env(Scheme_Config* config)
```

Returns the current namespace for the given parameterization (see §18 "Parameterizations (BC)"). The current thread's current parameterization is available as `scheme_config`.

```
Scheme_Env* scheme_primitive_module(Scheme_Object* name,
                                    Scheme_Env* for_env)
```

Prepares a new primitive module whose name is the symbol *name* (or an alternative that is active via `current-module-declare-name`). The module will be declared within the namespace *for_env*. The result is a `Scheme_Env *` value that can be used with `scheme_add_global`, etc., but it represents a module instead of a namespace. The module is not fully declared until `scheme_finish_primitive_module` is called, at which point all variables defined in the module become exported.

```
void scheme_finish_primitive_module(Scheme_Env* env)
```

Finalizes a primitive module and makes it available for use within the module's namespace.

14 Procedures (BC)

A *primitive procedure* is a Racket-callable procedure that is implemented in C. Primitive procedures are created in Racket with the function `scheme_make_prim_w_arity`, which takes a C function pointer, the name of the primitive, and information about the number of Racket arguments that it takes; it returns a Racket procedure value.

The C function implementing the procedure must take two arguments: an integer that specifies the number of arguments passed to the procedure, and an array of `Scheme_Object*` arguments. The number of arguments passed to the function will be checked using the arity information. (The arity information provided to `scheme_make_prim_w_arity` is also used for the Racket `arity` procedure.) The procedure implementation is not allowed to mutate the input array of arguments; as an exception, the procedure can mutate the array if it is the same as the result of `scheme_current_argument_stack`. The procedure may mutate the arguments themselves when appropriate (e.g., a fill in a vector argument).

The function `scheme_make_prim_closure_w_arity` is similar to `scheme_make_prim_w_arity`, but it takes an additional count and `Scheme_Object*` array that is copied into the created procedure; the procedure is passed back to the C function when the closure is invoked. In this way, closure-like data from the C world can be associated with the primitive procedure.

The function `scheme_make_closed_prim_w_arity` is similar to `scheme_make_prim_closure_w_arity`, but it uses an older calling convention for passing closure data.

To work well with Scheme threads, a C function that performs substantial or unbounded work should occasionally call `SCHEME_USE_FUEL`; see §17.2 "Allowing Thread Switches" for details.

```
Scheme_Object* scheme_make_prim_w_arity(Scheme_Prim* prim,
                                         char* name,
                                         int mina,
                                         int maxa)
```

Creates a primitive procedure value, given the C function pointer *prim*. The form of *prim* is defined by:

```
typedef Scheme_Object *(Scheme_Prim)(int argc,
                                     Scheme_Object **argv);
```

The value *mina* should be the minimum number of arguments that must be supplied to the procedure. The value *maxa* should be the maximum number of arguments that can be supplied to the procedure, or -1 if the procedure can take arbitrarily many arguments. The *mina* and *maxa* values are used for automatically checking the argument count before the primitive is invoked, and also for the Racket `arity` procedure. The *name* argument is used to report application arity errors at run-time.

```
Scheme_Object* scheme_make_folding_prim(Scheme_Prim* prim,
                                        char* name,
                                        int mina,
                                        int maxa,
                                        short folding)
```

Like `scheme_make_prim_w_arity`, but if *folding* is non-zero, the compiler assumes that an application of the procedure to constant values can be folded to a constant. For example, `+`, `zero?`, and `string-length` are folding primitives, but `display` and `cons` are not.

```
Scheme_Object* scheme_make_prim(Scheme_Prim* prim)
```

Same as `scheme_make_prim_w_arity`, but the arity is (0, -1) and the name "UNKNOWN" is assumed. This function is provided for backward compatibility only.

```
 Scheme_Object*
scheme_make_prim_closure_w_arity(Scheme_Prim_Closure_Proc* prim,
                                 int c,
                                 Scheme_Object** vals,
                                 char* name,
                                 int mina,
                                 int maxa)
```

Creates a primitive procedure value that includes the *c* values in *vals*; when the C function *prim* is invoked, the generated primitive is passed as the last parameter. The form of *prim* is defined by:

```
typedef
Scheme_Object *(Scheme_Prim_Closure_Proc)(int argc,
                                          Scheme_Object **argv,
                                          Scheme_Object *prim);
```

The macro `SCHEME_PRIM_CLOSURE_ELS` takes a primitive-closure object and returns an array with the same length and content as *vals*. (3m: see §12.1 "Cooperating with 3m" for a caution about `SCHEME_PRIM_CLOSURE_ELS`.)

```
 Scheme_Object*
scheme_make_closed_prim_w_arity(Scheme_Closed_Prim* prim,
                                void* data,
                                char* name,
                                int mina,
                                int maxa)
```

Creates an old-style primitive procedure value; when the C function *prim* is invoked, *data* is passed as the first parameter. The form of *prim* is defined by:

```
typedef
Scheme_Object *(Scheme_Closed_Prim)(void *data, int argc,
                                    Scheme_Object **argv);
```

`Scheme_Object* scheme_make_closed_prim(Scheme_Closed_Prim*` *prim*`,`
`void*` *data*`)`

Creates a closed primitive procedure value without arity information. This function is provided for backward compatibility only.

`Scheme_Object** scheme_current_argument_stack()`

Returns a pointer to an internal stack for argument passing. When the argument array passed to a procedure corresponds to the current argument stack address, the procedure is allowed to modify the array. In particular, it might clear out pointers in the argument array to allow the arguments to be reclaimed by the memory manager (if they are not otherwise accessible).

15 Evaluation (BC)

A Racket S-expression is evaluated by calling `scheme_eval`. This function takes an S-expression (as a `Scheme_Object*`) and a namespace and returns the value of the expression in that namespace.

The function `scheme_apply` takes a `Scheme_Object*` that is a procedure, the number of arguments to pass to the procedure, and an array of `Scheme_Object *` arguments. The return value is the result of the application. There is also a function `scheme_apply_to_list`, which takes a procedure and a list (constructed with `scheme_make_pair`) and performs the Racket `apply` operation.

The `scheme_eval` function actually calls `scheme_compile` followed by `scheme_eval_compiled`.

15.1 Top-level Evaluation Functions

The functions `scheme_eval`, `scheme_apply`, etc., are *top-level evaluation functions*. Continuation invocations are confined to jumps within a top-level evaluation (i.e., a continuation barrier is installed by these functions).

The functions `_scheme_eval_compiled`, `_scheme_apply`, etc. (with a leading underscore) provide the same functionality without starting a new top-level evaluation; these functions should only be used within new primitive procedures. Since these functions allow full continuation hops, calls to non-top-level evaluation functions can return zero or multiple times.

Currently, escape continuations and primitive error escapes can jump out of all evaluation and application functions. For more information, see §16 “Exceptions and Escape Continuations (BC)”.

15.2 Tail Evaluation

All of Racket’s built-in functions and syntax support proper tail-recursion. When a new primitive procedure or syntax is added to Racket, special care must be taken to ensure that tail recursion is handled properly. Specifically, when the final return value of a function is the result of an application, then `scheme_tail_apply` should be used instead of `scheme_apply`. When `scheme_tail_apply` is called, it postpones the procedure application until control returns to the Racket evaluation loop.

For example, consider the following implementation of a `thunk-or` primitive, which takes any number of thunks and performs `or` on the results of the thunks, evaluating only as many thunks as necessary.

```
static Scheme_Object *
thunk_or (int argc, Scheme_Object **argv)
{
  int i;
  Scheme_Object *v;

  if (!argc)
    return scheme_false;

  for (i = 0; i < argc - 1; i++)
    if (SCHEME_FALSEP((v = _scheme_apply(argv[i], 0, NULL))))
      return v;

  return scheme_tail_apply(argv[argc - 1], 0, NULL);
}
```

This `thunk-or` properly implements tail-recursion: if the final thunk is applied, then the result of `thunk-or` is the result of that application, so `scheme_tail_apply` is used for the final application.

15.3 Multiple Values

A primitive procedure can return multiple values by returning the result of calling `scheme_values`. The functions `scheme_eval_compiled_multi`, `scheme_apply_multi`, `_scheme_eval_compiled_multi`, and `_scheme_apply_multi` potentially return multiple values; all other evaluation and applications procedures return a single value or raise an exception.

Multiple return values are represented by the `scheme_multiple_values` "value." This quasi-value has the type `Scheme_Object*`, but it is not a pointer or a fixnum. When the result of an evaluation or application is `scheme_multiple_values`, the number of actual values can be obtained as `scheme_multiple_count`, and the array of `Scheme_Object*` values as `scheme_multiple_array`. (Both of those identifiers are actually macros.)

A garbage collection must not occur between the return of a `scheme_multiple_values` "value" and the receipt of the values through `scheme_multiple_count` `scheme_multiple_array`. Furthermore, if `scheme_multiple_array` is to be used across a potential garbage collection, then it must be specifically received by calling `scheme_detach_multiple_array`; otherwise, a garbage collection or further evaluation may change the content of the array. Otherwise, if any application or evaluation procedure is called, the `scheme_multiple_count` and `scheme_multiple_array` variables may be modified (but the array previously referenced by `scheme_multiple_array` is never re-used if `scheme_detach_multiple_array` is called).

The `scheme_multiple_count` and `scheme_multiple_array` variables only contain meaningful values when `scheme_multiple_values` is returned.

15.4 Evaluation Functions

```
Scheme_Object* scheme_eval(Scheme_Object* expr,
                           Scheme_Env* env)
```

Evaluates the (uncompiled) S-expression *expr* in the namespace *env*.

```
Scheme_Object* scheme_eval_compiled(Scheme_Object* obj,
                                    Scheme_Env* env)
```

Evaluates the compiled expression *obj*, which was previously returned from `scheme_compile`, first linking to the namespace *env*.

```
Scheme_Object* scheme_eval_compiled_multi(Scheme_Object* obj,
                                          Scheme_Env* env)
```

Evaluates the compiled expression *obj*, possibly returning multiple values (see §15.3 "Multiple Values").

```
Scheme_Object* _scheme_eval_compiled(Scheme_Object* obj,
                                     Scheme_Env* env)
```

Non-top-level version of `scheme_eval_compiled`. (See §15.1 "Top-level Evaluation Functions".)

```
Scheme_Object* _scheme_eval_compiled_multi(Scheme_Object* obj,
                                           Scheme_Env* env)
```

Non-top-level version of `scheme_eval_compiled_multi`. (See §15.1 "Top-level Evaluation Functions".)

```
Scheme_Env* scheme_basic_env()
```

Creates the main namespace for an embedded Racket. This procedure must be called before other Racket library function (except `scheme_make_param`). Extensions to Racket cannot call this function.

If it is called more than once, this function resets all threads (replacing the main thread), parameters, ports, namespaces, and finalizations.

```
Scheme_Object* scheme_make_namespace(int argc,
                                     Scheme_Object** argv)
```

Creates and returns a new namespace. This values can be cast to `Scheme_Env *`. It can also be installed in a parameterization using `scheme_set_param` with `MZCONFIG_ENV`.

When Racket is embedded in an application, create the initial namespace with `scheme_basic_env` before calling this procedure to create new namespaces.

```
Scheme_Object* scheme_apply(Scheme_Object* f,
                            int c,
                            Scheme_Object** args)
```

Applies the procedure *f* to the given arguments.

Beware that the procedure can mutate *args* if it is the same as the result of `scheme_current_argument_stack`.

If *c* is `0`, then *args* can be `NULL`.

```
Scheme_Object* scheme_apply_multi(Scheme_Object* f,
                                  int c,
                                  Scheme_Object** args)
```

Applies the procedure *f* to the given arguments, possibly returning multiple values (see §15.3 “Multiple Values”).

```
Scheme_Object* _scheme_apply(Scheme_Object* f,
                             int c,
                             Scheme_Object** args)
```

Non-top-level version of `scheme_apply`. (See §15.1 “Top-level Evaluation Functions”.)

```
Scheme_Object* _scheme_apply_multi(Scheme_Object* f,
                                   int c,
                                   Scheme_Object** args)
```

Non-top-level version of `scheme_apply_multi`. (See §15.1 “Top-level Evaluation Functions”.)

```
Scheme_Object* scheme_apply_to_list(Scheme_Object* f,
                                    Scheme_Object* args)
```

Applies the procedure *f* to the list of arguments in *args*.

```
Scheme_Object* scheme_eval_string(char* str,
                                  Scheme_Env* env)
```

Reads a single S-expression from *str* and evaluates it in the given namespace; the expression must return a single value, otherwise an exception is raised. The *str* argument is parsed as a UTF-8-encoded string of Unicode characters (so plain ASCII is fine).

```
Scheme_Object* scheme_eval_string_multi(char* str,
                                        Scheme_Env* env)
```

Like `scheme_eval_string`, but returns `scheme_multiple_values` when the expression returns multiple values.

```
Scheme_Object* scheme_eval_string_all(char* str,
                                      Scheme_Env* env,
                                      int all)
```

Like `scheme_eval_string`, but if *all* is not `0`, then expressions are read and evaluated from *str* until the end of the string is reached.

```
Scheme_Object* scheme_tail_apply(Scheme_Object* f,
                                 int n,
                                 Scheme_Object** args)
```

Applies the procedure as a tail-call. Actually, this function just registers the given application to be invoked when control returns to the evaluation loop. (Hence, this function is only useful within a primitive procedure that is returning to its caller.)

```
Scheme_Object* scheme_tail_apply_no_copy(Scheme_Object* f,
                                         int n,
                                         Scheme_Object** args)
```

Like `scheme_tail_apply`, but the array *args* is not copied. Use this only when *args* has infinite extent and will not be used again, or when *args* will certainly not be used again until the called procedure has returned.

```
Scheme_Object* scheme_tail_apply_to_list(Scheme_Object* f,
                                         Scheme_Object* l)
```

Applies the procedure as a tail-call.

```
Scheme_Object* scheme_compile(Scheme_Object* form,
                              Scheme_Env* env,
                              int writable)
```

Compiles the S-expression *form* in the given namespace. The returned value can be used with `scheme_eval_compiled` et al. Provide a non-zero value for *writable* if the resulting compiled object will be marshalled via `write` instead of evaluated.

```
Scheme_Object* scheme_expand(Scheme_Object* form,
                             Scheme_Env* env)
```

Expands all macros in the S-expression *form* using the given namespace.

```
Scheme_Object* scheme_values(int n,
                             Scheme_Object** args)
```

Returns the given values together as multiple return values. Unless *n* is `1`, the result will always be `scheme_multiple_values`.

`void scheme_detach_multiple_array(Scheme_Object** args)`

Called to receive multiple-value results; see §15.3 “Multiple Values”.

16 Exceptions and Escape Continuations (BC)

When Racket encounters an error, it raises an exception. The default exception handler invokes the error display handler and then the error escape handler. The default error escape handler escapes via a *primitive error escape*, which is implemented by calling `scheme_longjmp(*scheme_current_thread->error_buf)`.

An embedding program should install a fresh buffer into `scheme_current_thread->error_buf` and call `scheme_setjmp(*scheme_current_thread->error_buf)` before any top-level entry into Racket evaluation to catch primitive error escapes. When the new buffer goes out of scope, restore the original in `scheme_current_thread->error_buf`. The macro `scheme_error_buf` is a shorthand for `*scheme_current_thread->error_buf`.

```
mz_jmp_buf * volatile save, fresh;
...
save = scheme_current_thread->error_buf;
scheme_current_thread->error_buf = &fresh;
if (scheme_setjmp(scheme_error_buf)) {
  /* There was an error */
  ...
} else {
  v = scheme_eval_string(s, env);
}
scheme_current_thread->error_buf = save;
...
```

3m: when `scheme_setjmp` is used, the enclosing context must provide a local-variable registration record via `MZ_GC_DECL_REG`. Use `MZ_GC_DECL_REG(0)` if the context has no local variables to register. Unfortunately, when using `--xform` with `raco ctool` instead of `MZ_GC_DECL_REG`, etc., you may need to declare a dummy pointer and use it after `scheme_setjmp` to ensure that a local-variable registration is generated.

New primitive procedures can raise a generic exception by calling `scheme_signal_error`. The arguments for `scheme_signal_error` are roughly the same as for the standard C function `printf`. A specific primitive exception can be raised by calling `scheme_raise_exn`.

Full continuations are implemented in Racket by copying the C stack and using `scheme_setjmp` and `scheme_longjmp`. As long a C/C++ application invokes Racket evaluation through the top-level evaluation functions (`scheme_eval`, `scheme_apply`, etc., as opposed to `_scheme_apply`, `_scheme_eval_compiled`, etc.), the code is protected against any unusual behavior from Racket evaluations (such as returning twice from a function) because continuation invocations are confined to jumps within a single top-level evaluation. However, escape continuation jumps are still allowed; as explained in the following subsection, special care must be taken in extension that is sensitive to escapes.

16.1 Temporarily Catching Error Escapes

When implementing new primitive procedure, it is sometimes useful to catch and handle errors that occur in evaluating subexpressions. One way to do this is the following: save `scheme_current_thread->error_buf` to a temporary variable, set `scheme_current_thread->error_buf` to the address of a stack-allocated `mz_jmp_buf`, invoke `scheme_setjmp(scheme_error_buf)`, perform the function's work, and then restore `scheme_current_thread->error_buf` before returning a value. (3m: A stack-allocated `mz_jmp_buf` instance need not be registered with the garbage collector, and a heap-allocated `mz_jmp_buf` should be allocated as atomic.)

However, beware that a prompt abort or the invocation of an escaping continuation looks like a primitive error escape. In that case, the special indicator flag `scheme_jumping_to_continuation` is non-zero (instead of its normal zero value); this situation is only visible when implementing a new primitive procedure. When `scheme_jumping_to_continuation` is non-zero, honor the escape request by chaining to the previously saved error buffer; otherwise, call `scheme_clear_escape`.

```
mz_jmp_buf * volatile save, fresh;
save = scheme_current_thread->error_buf;
scheme_current_thread->error_buf = &fresh;
if (scheme_setjmp(scheme_error_buf)) {
  /* There was an error or continuation invocation */
  if (scheme_jumping_to_continuation) {
    /* It was a continuation jump */
    scheme_longjmp(*save, 1);
    /* To block the jump, instead: scheme_clear_escape(); */
  } else {
    /* It was a primitive error escape */
  }
} else {
  scheme_eval_string("x", scheme_env);
}
scheme_current_thread->error_buf = save;
```

This solution works fine as long as the procedure implementation only calls top-level evaluation functions (`scheme_eval`, `scheme_apply`, etc., as opposed to `_scheme_apply`, `_scheme_eval_compiled`, etc.). Otherwise, use `scheme_dynamic_wind` to protect your code against full continuation jumps in the same way that `dynamic-wind` is used in Racket.

The above solution simply traps the escape; it doesn't report the reason that the escape occurred. To catch exceptions and obtain information about the exception, the simplest route is to mix Racket code with C-implemented thunks. The code below can be used to catch exceptions in a variety of situations. It implements the function `_apply_catch_exceptions`, which catches exceptions during the application of a thunk. (This code is in `"collects/mzscheme/examples/catch.c"` in the distribution.)

```
static Scheme_Object *exn_catching_apply, *exn_p, *exn_message;

static void init_exn_catching_apply()
{
  if (!exn_catching_apply) {
    char *e =
      "(lambda (thunk) "
        "(with-handlers ([void (lambda (exn) (cons #f exn))]) "
          "(cons #t (thunk))))";
    /* make sure we have a namespace with the standard bindings: */
    Scheme_Env *env = (Scheme_Env *)scheme_make_namespace(0, NULL);

    scheme_register_extension_global(&exn_catching_apply,
                                     sizeof(Scheme_Object *));
    scheme_register_extension_global(&exn_p,
                                     sizeof(Scheme_Object *));
    scheme_register_extension_global(&exn_message,
                                     sizeof(Scheme_Object *));

    exn_catching_apply = scheme_eval_string(e, env);
    exn_p = scheme_lookup_global(scheme_intern_symbol("exn?"),
env);
    exn_message
      = scheme_lookup_global(scheme_intern_symbol("exn-message"),
                             env);
  }
}

/* This function applies a thunk, returning the Racket value if
   there's no exception, otherwise returning NULL and setting *exn
   to the raised value (usually an exn structure). */
Scheme_Object *_apply_thunk_catch_exceptions(Scheme_Object *f,
                                             Scheme_Object **exn)
{
  Scheme_Object *v;

  init_exn_catching_apply();

  v = _scheme_apply(exn_catching_apply, 1, &f);
  /* v is a pair: (cons #t value) or (cons #f exn) */

  if (SCHEME_TRUEP(SCHEME_CAR(v)))
    return SCHEME_CDR(v);
  else {
    *exn = SCHEME_CDR(v);
    return NULL;
```

```
  }
}

Scheme_Object *extract_exn_message(Scheme_Object *v)
{
  init_exn_catching_apply();

  if (SCHEME_TRUEP(_scheme_apply(exn_p, 1, &v)))
    return _scheme_apply(exn_message, 1, &v);
  else
    return NULL; /* Not an exn structure */
}
```

In the following example, the above code is used to catch exceptions that occur during while evaluating source code from a string.

```
static Scheme_Object *do_eval(void *s, int noargc,
                              Scheme_Object **noargv)
{
  return scheme_eval_string((char *)s,
                            scheme_get_env(scheme_config));
}

static Scheme_Object *eval_string_or_get_exn_message(char *s)
{
  Scheme_Object *v, *exn;

  v = scheme_make_closed_prim(do_eval, s);
  v = _apply_thunk_catch_exceptions(v, &exn);
  /* Got a value? */
  if (v)
    return v;

  v = extract_exn_message(exn);
  /* Got an exn? */
  if (v)
    return v;

  /* `raise' was called on some arbitrary value */
  return exn;
}
```

16.2 Enabling and Disabling Breaks

When embedding Racket, asynchronous break exceptions are disabled by default. Call `scheme_set_can_break` (which is the same as calling the Racket function `break-enabled`) to enable or disable breaks. To enable or disable breaks during the dynamic extent of another evaluation (where you would use `call-with-break-parameterization` in Racket), use `scheme_push_break_enable` before and `scheme_pop_break_enable` after, instead.

16.3 Exception Functions

```
void scheme_signal_error(char* msg,
                         ...)
```

Raises a generic primitive exception. The parameters are roughly as for `printf`, but with the following format directives:

- `%c` : a Unicode character (of type `mzchar`)
- `%d` : an `int`
- `%o` : an `int` formatted in octal
- `%gd` : a `long` integer
- `%gx` : a `long` integer formatted in hexadecimal
- `%ld` : an `intptr_t` integer
- `%lx` : an `intptr_t` integer formatted in hexadecimal
- `%f` : a floating-point `double`
- `%s` : a nul-terminated `char` string
- `%5` : a nul-terminated `mzchar` string
- `%S` : a Racket symbol (a `Scheme_Object*`)
- `%t` : a `char` string with a `intptr_t` size (two arguments), possibly containing a non-terminating nul byte, and possibly without a nul-terminator
- `%u` : a `mzchar` string with a `intptr_t` size (two arguments), possibly containing a non-terminating nul character, and possibly without a nul-terminator
- `%T` : a Racket string (a `Scheme_Object*`)
- `%q` : a string, truncated to 253 characters, with ellipses printed if the string is truncated

- `%Q` : a Racket string (a `Scheme_Object*`), truncated to 253 characters, with ellipses printed if the string is truncated
- `%V` : a Racket value (a `Scheme_Object*`), truncated according to the current error print width.
- `%D` : a Racket value (a `Scheme_Object*`), to `display`.
- `%@` : a Racket value (a `Scheme_Object*`), that is a list whose printed elements are spliced into the result.
- `%e` : an `errno` value, to be printed as a text message.
- `%E` : a platform-specific error value, to be printed as a text message.
- `%Z` : a potential platform-specific error value and a `char` string; if the string is non-`NULL`, then the error value is ignored, otherwise the error value is used as for `%E`.
- `%%` : a percent sign
- `%_` : a pointer to ignore
- `%-` : an `int` to ignore

The arguments following the format string must include no more than 25 strings and Racket values, 25 integers, and 25 floating-point numbers. (This restriction simplifies the implementation with precise garbage collection.)

```
void scheme_raise_exn(int exnid,
                      ...)
```

Raises a specific primitive exception. The *exnid* argument specifies the exception to be raised. If an instance of that exception has *n* fields, then the next *n*-2 arguments are values for those fields (skipping the `message` and `debug-info` fields). The remaining arguments start with an error string and proceed roughly as for `printf`; see `scheme_signal_error` above for more details.

Exception ids are `#define`d using the same names as in Racket, but prefixed with "MZ", all letters are capitalized, and all ":'s', "-"s, and "/"s are replaced with underscores. For example, `MZEXN_FAIL_FILESYSTEM` is the exception id for a filesystem exception.

```
void scheme_wrong_count(char* name,
                        int minc,
                        int maxc,
                        int argc,
                        Scheme_Object** argv)
```

This function is automatically invoked when the wrong number of arguments are given to a primitive procedure. It signals that the wrong number of parameters was received and escapes (like `scheme_signal_error`). The *name* argument is the name of the procedure that was given the wrong number of arguments; *minc* is the minimum number of expected arguments; *maxc* is the maximum number of expected arguments, or -1 if there is no maximum; *argc* and *argv* contain all of the received arguments.

```
void scheme_wrong_contract(char* name,
                           char* contract,
                           int which,
                           int argc,
                           Scheme_Object** argv)
```

Signals that an argument was received that does not satisfy a contract and escapes (like `scheme_signal_error`). The *name* argument is the name of the procedure that was given the wrong argument; *expected* is the contract; *which* is the offending argument in the *argv* array; *argc* and *argv* contain all of the received arguments. If the original *argc* and *argv* are not available, provide -1 for *which* and a pointer to the bad value in *argv*, in which case the magnitude (but not sign) of *argc* is ignored. Negate *argc* if the exception corresponds to a result contract instead of an argument contract.

```
void scheme_wrong_type(char* name,
                       char* expected,
                       int which,
                       int argc,
                       Scheme_Object** argv)
```

Signals that an argument of the wrong type was received and escapes. Use `scheme_wrong_contract`, instead.

The arguments are the same as for `scheme_wrong_contract`, except that *expected* is the name of the expected type.

```
void scheme_wrong_return_arity(char* name,
                               int expected,
                               int got,
                               Scheme_Object** argv,
                               const char* detail)
```

Signals that the wrong number of values were returned to a multiple-values context. The *expected* argument indicates how many values were expected, *got* indicates the number received, and *argv* are the received values. The *detail* string can be `NULL` or it can contain a `printf`-style string (with additional arguments) to describe the context of the error; see `scheme_signal_error` above for more details about the `printf`-style string.

```
void scheme_unbound_global(char* name)
```

Signals an unbound-variable error, where *name* is the name of the variable.

```
void scheme_contract_error(const char* name,
                           const char* msg,
                           ...)
```

Raises a contract-violation exception. The *msg* string is static, instead of a format string. After *msg*, any number of triples can be provided to add fields (each on its own line) to the error message; each triple is a string for the field name, a `0` or `1` to indicate whether the field value is a literal string or a Racket value, and either a literal string or a Racket value. The sequence of field triples must be terminated with `NULL`.

```
char* scheme_make_provided_string(Scheme_Object* o,
                                  int count,
                                  int* len)
```

Converts a Racket value into a string for the purposes of reporting an error message. The *count* argument specifies how many Racket values total will appear in the error message (so the string for this value can be scaled appropriately). If *len* is not `NULL`, it is filled with the length of the returned string.

```
char* scheme_make_arg_lines_string(char* s,
                                   int which,
                                   int argc,
                                   Scheme_Object** argv,
                                   intptr_t* len)
```

Converts an array of Racket values into a byte string, skipping the array element indicated by *which* if *which* is not `-1`. This function is used to format the "other" arguments to a function when one argument is bad (thus giving the user more information about the state of the program when the error occurred). If *len* is not `NULL`, it is filled with the length of the returned string.

If the arguments are shown on multiple lines, then the result string starts with a newline character and each line is indented by three spaces. Otherwise, the result string starts with a space. If the result would contain no arguments, it contains `[none]`, instead.

```
char* scheme_make_args_string(char* s,
                              int which,
                              int argc,
                              Scheme_Object** argv,
                              intptr_t* len)
```

Like `scheme_make_arg_lines_string`, but for old-style messages where the arguments are always shown within a single line. The result does not include a leading space.

```
void scheme_check_proc_arity(char* where,
                             int a,
                             int which,
                             int argc,
                             Scheme_Object** argv)
```

Checks the *which*th argument in *argv* to make sure it is a procedure that can take *a* arguments. If there is an error, the *where*, *which*, *argc*, and *argv* arguments are passed on to `scheme_wrong_type`. As in `scheme_wrong_type`, *which* can be -1, in which case `*`*argv* is checked.

```
Scheme_Object* scheme_dynamic_wind(Pre_Post_Proc pre,
                                   Action_Proc action,
                                   Pre_Post_Proc post,
                                   Action_Proc jmp_handler,
                                   void* data)
```

Evaluates calls the function *action* to get a value for the `scheme_dynamic_wind` call. The `Pre_Post_Proc` and `Action_Proc` types are not actually defined; instead the types are inlined as if they were defined as follows:

```
typedef void (*Pre_Post_Proc)(void *data);
typedef Scheme_Object* (*Action_Proc)(void *data);
```

The functions *pre* and *post* are invoked when jumping into and out of *action*, respectively.

The function *jmp_handler* is called when an error is signaled (or an escaping continuation is invoked) during the call to *action*; if *jmp_handler* returns `NULL`, then the error is passed on to the next error handler, otherwise the return value is used as the return value for the `scheme_dynamic_wind` call.

The pointer *data* can be anything; it is passed along in calls to *action*, *pre*, *post*, and *jmp_handler*.

```
void scheme_clear_escape()
```

Clears the "jumping to escape continuation" flag associated with a thread. Call this function when blocking escape continuation hops (see the first example in §16.1 "Temporarily Catching Error Escapes").

```
void scheme_set_can_break(int on)
```

Enables or disables breaks in the same way as calling `break-enabled`.

```
void scheme_push_break_enable(Scheme_Cont_Frame_Data* cframe,
                              int on,
                              int pre_check)
```

Use this function with `scheme_pop_break_enable` to enable or disable breaks in the same way as `call-with-break-parameterization`; this function writes to *cframe* to initialize it, and `scheme_pop_break_enable` reads from *cframe*. If *pre_check* is non-zero and breaks are currently enabled, any pending break exception is raised.

```
void scheme_pop_break_enable(Scheme_Cont_Frame_Data* cframe,
                             int post_check)
```

Use this function with `scheme_push_break_enable`. If *post_check* is non-zero and breaks are enabled after restoring the previous state, then any pending break exception is raised.

```
 Scheme_Object*
scheme_current_continuation_marks(Scheme_Object* prompt_tag)
```

Like `current-continuation-marks`. Passing `NULL` as *prompt_tag* is the same as providing the default continuation prompt tag.

```
void scheme_warning(char* msg,
                    ...)
```

Writes a warning message. The parameters are roughly as for `printf`; see `scheme_signal_error` above for more details.

Normally, Racket's logging facilities should be used instead of this function.

17 Threads (BC)

The initializer function `scheme_basic_env` creates the main Racket thread; all other threads are created through calls to `scheme_thread`.

Information about each internal Racket thread is kept in a `Scheme_Thread` structure. A pointer to the current thread's structure is available as `scheme_current_thread` or from `scheme_get_current_thread`. A `Scheme_Thread` structure includes the following fields:

- `error_buf` — the `mz_jmp_buf` value used to escape from errors. The `error_buf` value of the current thread is available as `scheme_error_buf`.
- `cjs.jumping_to_continuation` — a flag that distinguishes escaping-continuation invocations from error escapes. The `cjs.jumping_to_continuation` value of the current thread is available as `scheme_jumping_to_continuation`.
- `init_config` — the thread's initial parameterization. See also §18 "Parameterizations (BC)".
- `cell_values` — The thread's values for thread cells (see also §18 "Parameterizations (BC)").
- `next` — The next thread in the linked list of threads; this is `NULL` for the main thread.

The list of all scheduled threads is kept in a linked list; `scheme_first_thread` points to the first thread in the list. The last thread in the list is always the main thread.

17.1 Integration with Threads

Racket's threads can break external C code under two circumstances:

- *Pointers to stack-based values can be communicated between threads.* For example, if thread A stores a pointer to a stack-based variable in a global variable, if thread B uses the pointer in the global variable, it may point to data that is not currently on the stack.
- *C functions that can invoke Racket (and also be invoked by Racket) depend on strict function-call nesting.* For example, suppose a function F uses an internal stack, pushing items on to the stack on entry and popping the same items on exit. Suppose also that F invokes Racket to evaluate an expression. If the evaluation of this expression invokes F again in a new thread, but then returns to the first thread before completing the second F, then F's internal stack will be corrupted.

If either of these circumstances occurs, Racket will probably crash.

17.2 Allowing Thread Switches

C code that performs substantial or unbounded work should occasionally call `SCHEME_USE_FUEL`—actually a macro—which allows Racket to swap in another Racket thread to run, and to check for breaks on the current thread. In particular, if breaks are enabled, then `SCHEME_USE_FUEL` may trigger an exception.

The macro consumes an integer argument. On most platforms, where thread scheduling is based on timer interrupts, the argument is ignored. On some platforms, however, the integer represents the amount of "fuel" that has been consumed since the last call to `SCHEME_USE_FUEL`. For example, the implementation of `vector->list` consumes a unit of fuel for each created cons cell:

```
Scheme_Object *scheme_vector_to_list(Scheme_Object *vec)
{
  int i;
  Scheme_Object *pair = scheme_null;

  i = SCHEME_VEC_SIZE(vec);

  for (; i--; ) {
    SCHEME_USE_FUEL(1);
    pair = scheme_make_pair(SCHEME_VEC_ELS(vec)[i], pair);
  }

  return pair;
}
```

The `SCHEME_USE_FUEL` macro expands to a C block, not an expression.

17.3 Blocking the Current Thread

Embedding or extension code sometimes needs to block, but blocking should allow other Racket threads to execute. To allow other threads to run, block using `scheme_block_until`. This procedure takes two functions: a polling function that tests whether the blocking operation can be completed, and a prepare-to-sleep function that sets bits in `fd_sets` when Racket decides to sleep (because all Racket threads are blocked). On Windows, an "`fd_set`" can also accommodate OS-level semaphores or other handles via `scheme_add_fd_handle`.

Since the functions passed to `scheme_block_until` are called by the Racket thread scheduler, they must never raise exceptions, call `scheme_apply`, or trigger the evaluation of Racket code in any way. The `scheme_block_until` function itself may call the current exception handler, however, in reaction to a break (if breaks are enabled).

When a blocking operation is associated with an object, then the object might make sense as an argument to `sync`. To extend the set of objects accepted by `sync`, either register polling and sleeping functions with `scheme_add_evt`, or register a semaphore accessor with `scheme_add_evt_through_sema`.

The `scheme_signal_received` function can be called to wake up Racket when it is sleeping. In particular, calling `scheme_signal_received` ensures that Racket will poll all blocking synchronizations soon afterward. Furthermore, `scheme_signal_received` can be called from any OS-level thread. Thus, when no adequate prepare-to-sleep function can be implemented for `scheme_block_until` in terms of file descriptors or Windows handles, calling `scheme_signal_received` when the poll result changes will ensure that a poll is issued.

17.4 Threads in Embedded Racket with Event Loops

When Racket is embedded in an application with an event-based model (i.e., the execution of Racket code in the main thread is repeatedly triggered by external events until the application exits) special hooks must be set to ensure that non-main threads execute correctly. For example, during the execution in the main thread, a new thread may be created; the new thread may still be running when the main thread returns to the event loop, and it may be arbitrarily long before the main thread continues from the event loop. Under such circumstances, the embedding program must explicitly allow Racket to execute the non-main threads; this can be done by periodically calling the function `scheme_check_threads`.

Thread-checking only needs to be performed when non-main threads exist (or when there are active callback triggers). The embedding application can set the global function pointer `scheme_notify_multithread` to a function that takes an integer parameter and returns `void`. This function is be called with 1 when thread-checking becomes necessary, and then with 0 when thread checking is no longer necessary. An embedding program can use this information to prevent unnecessary `scheme_check_threads` polling.

The below code illustrates how GRacket formerly set up `scheme_check_threads` polling using the wxWindows `wxTimer` class. (Any regular event-loop-based callback is appropriate.) The `scheme_notify_multithread` pointer is set to `MrEdInstallThreadTimer`. (GRacket no longer work this way, however.)

```
class MrEdThreadTimer : public wxTimer
{
 public:
  void Notify(void); /* callback when timer expires */
};

static int threads_go;
static MrEdThreadTimer *theThreadTimer;
#define THREAD_WAIT_TIME 40
```

```
void MrEdThreadTimer::Notify()
{
  if (threads_go)
    Start(THREAD_WAIT_TIME, TRUE);

  scheme_check_threads();
}

static void MrEdInstallThreadTimer(int on)
{
  if (!theThreadTimer)
    theThreadTimer = new MrEdThreadTimer;

  if (on)
    theThreadTimer->Start(THREAD_WAIT_TIME, TRUE);
  else
    theThreadTimer->Stop();

  threads_go = on;
  if (on)
    do_this_time = 1;
}
```

An alternate architecture, which GRacket now uses, is to send the main thread into a loop, which blocks until an event is ready to handle. Racket automatically takes care of running all threads, and it does so efficiently because the main thread blocks on a file descriptor, as explained in §17.3 "Blocking the Current Thread".

Callbacks for Blocked Threads

Racket threads are sometimes blocked on file descriptors, such as an input file or the X event socket. Blocked non-main threads do not block the main thread, and therefore do not affect the event loop, so `scheme_check_threads` is sufficient to implement this case correctly. However, it is wasteful to poll these descriptors with `scheme_check_threads` when nothing else is happening in the application and when a lower-level poll on the file descriptors can be installed. If the global function pointer `scheme_wakeup_on_input` is set, then this case is handled more efficiently by turning off thread checking and issuing a "wakeup" request on the blocking file descriptors through `scheme_wakeup_on_input`.

A `scheme_wakeup_on_input` procedure takes a pointer to an array of three `fd_sets` (use `MZ_FD_SET` instead of `FD_SET`, etc.) and returns `void`. The `scheme_wakeup_on_input` function does not sleep immediately; it just sets up callbacks on the specified file descriptors. When input is ready on any of those file descriptors, the callbacks are removed and `scheme_wake_up` is called.

For example, the X Windows version of GRacket formerly set `scheme_wakeup_on_input` to this `MrEdNeedWakeup`:

```
static XtInputId *scheme_cb_ids = NULL;
static int num_cbs;

static void MrEdNeedWakeup(void *fds)
{
  int limit, count, i, p;
  fd_set *rd, *wr, *ex;

  rd = (fd_set *)fds;
  wr = ((fd_set *)fds) + 1;
  ex = ((fd_set *)fds) + 2;

  limit = getdtablesize();

  /* See if we need to do any work, really: */
  count = 0;
  for (i = 0; i < limit; i++) {
    if (MZ_FD_ISSET(i, rd))
      count++;
    if (MZ_FD_ISSET(i, wr))
      count++;
    if (MZ_FD_ISSET(i, ex))
      count++;
  }

  if (!count)
    return;

  /* Remove old callbacks: */
  if (scheme_cb_ids)
    for (i = 0; i < num_cbs; i++)
      notify_set_input_func((Notify_client)NULL, (Notify_func)NULL,
                            scheme_cb_ids[i]);

  num_cbs = count;
  scheme_cb_ids = new int[num_cbs];

  /* Install callbacks */
  p = 0;
  for (i = 0; i < limit; i++) {
    if (MZ_FD_ISSET(i, rd))
```

```
        scheme_cb_ids[p++] = XtAppAddInput(wxAPP_CONTEXT, i,
                                           (XtPointer *)XtInputRead-
Mask,
                                           (XtInputCallbackProc)MrEdWakeUp,
NULL);
      if (MZ_FD_ISSET(i, wr))
        scheme_cb_ids[p++] = XtAppAddInput(wxAPP_CONTEXT, i,
                                           (XtPointer *)XtIn-
putWriteMask,
                                           (XtInputCallbackProc)MrEdWakeUp,
NULL);
      if (MZ_FD_ISSET(i, ex))
        scheme_cb_ids[p++] = XtAppAddInput(wxAPP_CONTEXT, i,
                                           (XtPointer *)XtInputEx-
ceptMask,
                                           (XtInputCallbackProc)MrEdWakeUp,
                                           NULL);
    }
  }

  /* callback function when input/exception is detected: */
  Bool MrEdWakeUp(XtPointer, int *, XtInputId *)
  {
    int i;

    if (scheme_cb_ids) {
      /* Remove all callbacks: */
      for (i = 0; i < num_cbs; i++)
       XtRemoveInput(scheme_cb_ids[i]);

      scheme_cb_ids = NULL;

      /* ``wake up'' */
      scheme_wake_up();
    }

    return FALSE;
  }
```

17.5 Sleeping by Embedded Racket

When all Racket threads are blocked, Racket must “sleep” for a certain number of seconds or until external input appears on some file descriptor. Generally, sleeping should block the main event loop of the entire application. However, the way in which sleeping is performed may depend on the embedding application. The global function pointer `scheme_sleep` can be set by an embedding application to implement a blocking sleep, although Racket implements this function for you.

A `scheme_sleep` function takes two arguments: a `float` and a `void*`. The latter is really points to an array of three "`fd_set`" records (one for read, one for write, and one for exceptions); these records are described further below. If the `float` argument is non-zero, then the `scheme_sleep` function blocks for the specified number of seconds, at most. The `scheme_sleep` function should block until there is input one of the file descriptors specified in the "`fd_set`," indefinitely if the `float` argument is zero.

The second argument to `scheme_sleep` is conceptually an array of three `fd_set` records, but always use `scheme_get_fdset` to get anything other than the zeroth element of this array, and manipulate each "`fd_set`" with `MZ_FD_SET`, `MZ_FD_CLR`, etc. instead of `FD_SET`, `FD_CLR`, etc.

The following function `mzsleep` is an appropriate `scheme_sleep` function for most any Unix or Windows application. (This is approximately the built-in sleep used by Racket.)

```
void mzsleep(float v, void *fds)
{
  if (v) {
    sleep(v);
  } else {
    int limit;
    fd_set *rd, *wr, *ex;

# ifdef WIN32
    limit = 0;
# else
    limit = getdtablesize();
# endif

    rd = (fd_set *)fds;
    wr = (fd_set *)scheme_get_fdset(fds, 1);
    ex = (fd_set *)scheme_get_fdset(fds, 2);

    select(limit, rd, wr, ex, NULL);
  }
}
```

17.6 Thread Functions

`Scheme_Thread* scheme_get_current_thread()`

Returns the currently executing thread. The result is equivalent to `scheme_current_thread`, but the function form must be used in some embedding contexts.

```
Scheme_Object* scheme_thread(Scheme_Object* thunk)
```

Creates a new thread, just like `thread`.

```
 Scheme_Object*
scheme_thread_w_details(Scheme_Object* thunk,
                        Scheme_Config* config,
                        Scheme_Thread_Cell_Table* cells,
                        Scheme_Custodian* cust,
                        int suspend_to_kill)
```

Like `scheme_thread`, except that the created thread belongs to *cust* instead of the current custodian, it uses the given *config* for its initial configuration, it uses *cells* for its thread-cell table, and if *suspend_to_kill* is non-zero, then the thread is merely suspended when it would otherwise be killed (through either `kill-thread` or `custodian-shutdown-all`).

The *config* argument is typically obtained through `scheme_current_config` or `scheme_extend_config`. A *config* is immutable, so different threads can safely use the same value. The *cells* argument should be obtained from `scheme_inherit_cells`; it is mutable, and a particular cell table should be used by only one thread.

```
Scheme_Object* scheme_make_sema(intptr_t v)
```

Creates a new semaphore.

```
void scheme_post_sema(Scheme_Object* sema)
```

Posts to *sema*.

```
int scheme_wait_sema(Scheme_Object* sema,
                     int try)
```

Waits on *sema*. If *try* is not 0, the wait can fail and 0 is returned for failure, otherwise 1 is returned.

```
void scheme_thread_block(float sleep_time)
```

Allows the current thread to be swapped out in favor of other threads. If *sleep_time* positive, then the current thread will sleep for at least *sleep_time* seconds.

After calling this function, a program should almost always call `scheme_making_progress` next. The exception is when `scheme_thread_block` is called in a polling loop that performs no work that affects the progress of other threads. In that case, `scheme_making_progress` should be called immediately after exiting the loop.

See also `scheme_block_until`, and see also the `SCHEME_USE_FUEL` macro in §17.2 "Allowing Thread Switches".

```
void scheme_thread_block_enable_break(float sleep_time,
                                      int break_on)
```

Like `scheme_thread_block`, but breaks are enabled while blocking if *break_on* is true.

```
void scheme_swap_thread(Scheme_Thread* thread)
```

Swaps out the current thread in favor of *thread*.

```
void scheme_break_thread(Scheme_Thread* thread)
```

Sends a break signal to the given thread.

```
int scheme_break_waiting(Scheme_Thread* thread)
```

Returns `1` if a break from `break-thread` or `scheme_break_thread` has occurred in the specified thread but has not yet been handled.

```
int scheme_block_until(Scheme_Ready_Fun f,
                       Scheme_Needs_Wakeup_Fun fdf,
                       Scheme_Object* data,
                       float sleep)
```

The `Scheme_Ready_Fun` and `Scheme_Needs_Wakeup_Fun` types are defined as follows:

```
typedef int (*Scheme_Ready_Fun)(Scheme_Object *data);
typedef void (*Scheme_Needs_Wakeup_Fun)(Scheme_Object *data,
                                        void *fds);
```

Blocks the current thread until *f* with *data* returns a true value. The *f* function is called periodically—at least once per potential swap-in of the blocked thread—and it may be called multiple times even after it returns a true value. If *f* with *data* ever returns a true value, it must continue to return a true value until `scheme_block_until` returns. The argument to *f* is the same *data* as provided to `scheme_block_until`, and *data* is ignored otherwise. (The *data* argument is not actually required to be a `Scheme_Object*` value, because it is only used by *f* and *fdf*.)

If Racket decides to sleep, then the *fdf* function is called to sets bits in *fds*, conceptually an array of three `fd_set`s: one or reading, one for writing, and one for exceptions. Use `scheme_get_fdset` to get elements of this array, and manipulate an "`fd_set`" with `MZ_FD_SET` instead of `FD_SET`, etc. On Windows, an "`fd_set`" can also accommodate OS-level semaphores or other handles via `scheme_add_fd_handle`.

The *fdf* argument can be `NULL`, which implies that the thread becomes unblocked (i.e., *ready* changes its result to true) only through Racket actions, and never through external processes (e.g., through a socket or OS-level semaphore)—with the exception that `scheme_signal_received` may be called to indicate an external change.

If *sleep* is a positive number, then `scheme_block_until` polls *f* at least every *sleep* seconds, but `scheme_block_until` does not return until *f* returns a true value. The call to `scheme_block_until` can return before *sleep* seconds if *f* returns a true value.

The return value from `scheme_block_until` is the return value of its most recent call to *f*, which enables *f* to return some information to the `scheme_block_until` caller.

See §17.3 "Blocking the Current Thread" for information about restrictions on the *f* and *fdf* functions.

```
int scheme_block_until_enable_break(Scheme_Ready_Fun f,
                                    Scheme_Needs_Wakeup_Fun fdf,
                                    Scheme_Object* data,
                                    float sleep,
                                    int break_on)
```

Like `scheme_block_until`, but breaks are enabled while blocking if *break_on* is true.

```
int scheme_block_until_unless(Scheme_Ready_Fun f,
                              Scheme_Needs_Wakeup_Fun fdf,
                              Scheme_Object* data,
                              float sleep,
                              Scheme_Object* unless_evt,
                              int break_on)
```

Like `scheme_block_until_enable_break`, but the function returns if *unless_evt* becomes ready, where *unless_evt* is a port progress event implemented by `scheme_progress_evt_via_get`. See `scheme_make_input_port` for more information.

```
void scheme_signal_received()
```

Indicates that an external event may have caused the result of a synchronization poll to have a different result. Unlike most other Racket functions, this one can be called from any OS-level thread, and it wakes up if the Racket thread if it is sleeping.

```
void scheme_check_threads()
```

This function is periodically called by the embedding program to give background processes time to execute. See §17.4 "Threads in Embedded Racket with Event Loops" for more information.

As long as some threads are ready, this functions returns only after one thread quantum, at least.

```
void scheme_wake_up()
```

This function is called by the embedding program when there is input on an external file descriptor. See §17.5 “Sleeping by Embedded Racket” for more information.

```
void* scheme_get_fdset(void* fds,
                       int pos)
```

Extracts an “`fd_set`” from an array passed to `scheme_sleep`, a callback for `scheme_block_until`, or an input port callback for `scheme_make_input_port`.

```
void scheme_add_fd_handle(void* h,
                          void* fds,
                          int repost)
```

Adds an OS-level semaphore (Windows) or other waitable handle (Windows) to the “`fd_set`” *fds*. When Racket performs a “`select`” to sleep on *fds*, it also waits on the given semaphore or handle. This feature makes it possible for Racket to sleep until it is awakened by an external process.

Racket does not attempt to deallocate the given semaphore or handle, and the “`select`” call using *fds* may be unblocked due to some other file descriptor or handle in *fds*. If *repost* is a true value, then *h* must be an OS-level semaphore, and if the “`select`” unblocks due to a post on *h*, then *h* is reposted; this allows clients to treat *fds*-installed semaphores uniformly, whether or not a post on the semaphore was consumed by “`select`”.

The `scheme_add_fd_handle` function is useful for implementing the second procedure passed to `scheme_wait_until`, or for implementing a custom input port.

On Unix and Mac OS, this function has no effect.

```
void scheme_add_fd_eventmask(void* fds,
                             int mask)
```

Adds an OS-level event type (Windows) to the set of types in the “`fd_set`” *fds*. When Racket performs a “`select`” to sleep on *fds*, it also waits on events of them specified type. This feature makes it possible for Racket to sleep until it is awakened by an external process.

The event mask is only used when some handle is installed with `scheme_add_fd_handle`. This awkward restriction may force you to create a dummy semaphore that is never posted.

On Unix, and Mac OS, this function has no effect.

```
void scheme_add_evt(Scheme_Type type,
                    Scheme_Ready_Fun ready,
                    Scheme_Needs_Wakeup_Fun wakeup,
                    Scheme_Wait_Filter_Fun filter,
                    int can_redirect)
```

The argument types are defined as follows:

```
typedef int (*Scheme_Ready_Fun)(Scheme_Object *data);
typedef void (*Scheme_Needs_Wakeup_Fun)(Scheme_Object *data,
                                        void *fds);
typedef int (*Scheme_Wait_Filter_Fun)(Scheme_Object *data);
```

Extends the set of waitable objects for `sync` to those with the type tag *type*. If *filter* is non-`NULL`, it constrains the new waitable set to those objects for which *filter* returns a non-zero value.

The *ready* and *wakeup* functions are used in the same way was the arguments to `scheme_block_until`.

The *can_redirect* argument should be `0`.

```
void scheme_add_evt_through_sema(Scheme_Type type,
                                 Scheme_Wait_Sema_Fun getsema,
                                 Scheme_Wait_Filter_Fun filter)
```

Like `scheme_add_evt`, but for objects where waiting is based on a semaphore. Instead of *ready* and *wakeup* functions, the *getsema* function extracts a semaphore for a given object:

```
typedef
Scheme_Object *(*Scheme_Wait_Sema_Fun)(Scheme_Object *data,
                                       int *repost);
```

If a successful wait should leave the semaphore waited, then *getsema* should set **repost* to `0`. Otherwise, the given semaphore will be re-posted after a successful wait. A *getsema* function should almost always set **repost* to `1`.

```
void scheme_making_progress()
```

Notifies the scheduler that the current thread is not simply calling `scheme_thread_block` in a loop, but that it is actually making progress.

```
int scheme_tls_allocate()
```

Allocates a thread local storage index to be used with `scheme_tls_set` and `scheme_tls_get`.

```
void scheme_tls_set(int index,
                    void* v)
```

Stores a thread-specific value using an index allocated with `scheme_tls_allocate`.

```
void* scheme_tls_get(int index)
```

Retrieves a thread-specific value installed with `scheme_tls_set`. If no thread-specific value is available for the given index, `NULL` is returned.

```
Scheme_Object* scheme_call_enable_break(Scheme_Prim* prim,
                                        int argc,
                                        Scheme_Object** argv)
```

Calls *prim* with the given *argc* and *argv* with breaks enabled. The *prim* function can block, in which case it might be interrupted by a break. The *prim* function should not block, yield, or check for breaks after it succeeds, where "succeeds" depends on the operation. For example, `tcp-accept/enable-break` is implemented by wrapping this function around the implementation of `tcp-accept`; the `tcp-accept` implementation does not block or yield after it accepts a connection.

```
 Scheme_Object*
scheme_make_thread_cell(Scheme_Object* def_val,
                        int preserved,
                        Scheme_Object* cell,
                        Scheme_Thread_Cell_Table* cells,
                        Scheme_Object* cell,
                        Scheme_Thread_Cell_Table* cells,
                        Scheme_Object* v)
```

Prevents Racket thread swaps until `scheme_end_atomic` or `scheme_end_atomic_no_swap` is called. Start-atomic and end-atomic pairs can be nested.

```
void scheme_end_atomic()
```

Ends an atomic region with respect to Racket threads. The current thread may be swapped out immediately (i.e., the call to `scheme_end_atomic` is assumed to be a safe point for thread swaps).

```
void scheme_end_atomic_no_swap()
```

Ends an atomic region with respect to Racket threads, and also prevents an immediate thread swap. (In other words, no Racket thread swaps will occur until a future safe point.)

```
void scheme_add_swap_callback(Scheme_Closure_Func f,
                              Scheme_Object* data)
```

Registers a callback to be invoked just after a Racket thread is swapped in. The *data* is provided back to *f* when it is called, where `Closure_Func` is defined as follows:

```
typedef Scheme_Object *(*Scheme_Closure_Func)(Scheme_Object *);
```

```
void scheme_add_swap_out_callback(Scheme_Closure_Func f,
                                  Scheme_Object* data)
```

Like `scheme_add_swap_callback`, but registers a callback to be invoked just before a Racket thread is swapped out.

18 Parameterizations (BC)

A *parameterization* is a set of parameter values. Each thread has its own initial parameterization, which is extended functionally and superseded by parameterizations that are attached to a particular continuation mark.

Parameterization information is stored in a `Scheme_Config` record. For the currently executing thread, `scheme_current_config` returns the current parameterization.

To obtain parameter values, a `Scheme_Config` is combined with the current threads `Scheme_Thread_Cell_Table`, as stored in the thread record's `cell_values` field.

Parameter values for built-in parameters are obtained and modified (for the current thread) using `scheme_get_param` and `scheme_set_param`. Each parameter is stored as a `Scheme_Object *` value, and the built-in parameters are accessed through the following indices:

- `MZCONFIG_ENV` — `current-namespace` (use `scheme_get_env`)
- `MZCONFIG_INPUT_PORT` — `current-input-port`
- `MZCONFIG_OUTPUT_PORT` — `current-output-port`
- `MZCONFIG_ERROR_PORT` — `current-error-port`
- `MZCONFIG_ERROR_DISPLAY_HANDLER` — `error-display-handler`
- `MZCONFIG_ERROR_PRINT_VALUE_HANDLER` — `error-value->string-handler`
- `MZCONFIG_EXIT_HANDLER` — `exit-handler`
- `MZCONFIG_INIT_EXN_HANDLER` — `uncaught-exception-handler`
- `MZCONFIG_EVAL_HANDLER` — `current-eval`
- `MZCONFIG_LOAD_HANDLER` — `current-load`
- `MZCONFIG_PRINT_HANDLER` — `current-print`
- `MZCONFIG_PROMPT_READ_HANDLER` — `current-prompt-read`
- `MZCONFIG_CAN_READ_GRAPH` — `read-accept-graph`
- `MZCONFIG_CAN_READ_COMPILED` — `read-accept-compiled`
- `MZCONFIG_CAN_READ_BOX` — `read-accept-box`
- `MZCONFIG_CAN_READ_PIPE_QUOTE` — `read-accept-bar-quote`
- `MZCONFIG_CAN_READ_DOT` — `read-accept-dot`

- `MZCONFIG_CAN_READ_INFIX_DOT` — `read-accept-infix-dot`
- `MZCONFIG_CAN_READ_QUASI` — `read-accept-quasiquote`
- `MZCONFIG_CAN_READ_READER` — `read-accept-reader`
- `MZCONFIG_CAN_READ_LANG` — `read-accept-lang`
- `MZCONFIG_READ_DECIMAL_INEXACT` — `read-decimal-as-inexact`
- `MZCONFIG_READ_CDOT` — `read-cdot`
- `MZCONFIG_PRINT_GRAPH` — `print-graph`
- `MZCONFIG_PRINT_STRUCT` — `print-struct`
- `MZCONFIG_PRINT_BOX` — `print-box`
- `MZCONFIG_CASE_SENS` — `read-case-sensitive`
- `MZCONFIG_SQUARE_BRACKETS_ARE_PARENS` — `read-square-brackets-as-parens`
- `MZCONFIG_CURLY_BRACES_ARE_PARENS` — `read-curly-braces-as-parens`
- `MZCONFIG_SQUARE_BRACKETS_ARE_TAGGED` — `read-square-brackets-with-tag`
- `MZCONFIG_CURLY_BRACES_ARE_TAGGED` — `read-curly-braces-with-tag`
- `MZCONFIG_ERROR_PRINT_WIDTH` — `error-print-width`
- `MZCONFIG_ALLOW_SET_UNDEFINED` — `allow-compile-set!-undefined`
- `MZCONFIG_CUSTODIAN` — `current-custodian`
- `MZCONFIG_USE_COMPILED_KIND` — `use-compiled-file-paths`
- `MZCONFIG_LOAD_DIRECTORY` — `current-load-relative-directory`
- `MZCONFIG_COLLECTION_PATHS` — `current-library-collection-paths`
- `MZCONFIG_PORT_PRINT_HANDLER` — `global-port-print-handler`
- `MZCONFIG_LOAD_EXTENSION_HANDLER` — `current-load-extension`

To get or set a parameter value for a thread other than the current one, use `scheme_get_thread_param` and `scheme_set_thread_param`, each of which takes a `Scheme_Thread_Cell_Table` to use in resolving or setting a parameter value.

When installing a new parameter with `scheme_set_param`, no check is performed on the supplied value to ensure that it is a legal value for the parameter; this is the responsibility of the caller of `scheme_set_param`. Note that Boolean parameters should only be set to the values `#t` and `#f`.

New primitive parameter indices are created with `scheme_new_param` and implemented with `scheme_make_parameter` and `scheme_param_config`.

```
Scheme_Object* scheme_get_param(Scheme_Config* config,
                                int param_id)
```

Gets the current value (for the current thread) of the parameter specified by *param_id*.

```
Scheme_Object* scheme_set_param(Scheme_Config* config,
                                int param_id,
                                Scheme_Object* v)
```

Sets the current value (for the current thread) of the parameter specified by *param_id*.

```
 Scheme_Object*
scheme_get_thread_param(Scheme_Config* config,
                        Scheme_Thread_Cell_Table* cells,
                        int param_id)
```

Like `scheme_get_param`, but using an arbitrary thread's cell-value table.

```
 Scheme_Object*
scheme_set_thread_param(Scheme_Config* config,
                        Scheme_Thread_Cell_Table* cells,
                        int param_id,
                        Scheme_Object* v)
```

Like `scheme_set_param`, but using an arbitrary thread's cell-value table.

```
Scheme_Object* scheme_extend_config(Scheme_Config* base,
                                    int param_id,
                                    Scheme_Object* v)
```

Creates and returns a parameterization that extends *base* with a new value *v* (in all threads) for the parameter *param_id*. Use `scheme_install_config` to make this configuration active in the current thread.

```
void scheme_install_config(Scheme_Config* config)
```

Adjusts the current thread's continuation marks to make *config* the current parameterization. Typically, this function is called after `scheme_push_continuation_frame` to establish a new continuation frame, and then `scheme_pop_continuation_frame` is called later to remove the frame (and thus the parameterization).

```
 Scheme_Thread_Cell_Table*
scheme_inherit_cells(Scheme_Thread_Cell_Table* cells)
```

Creates a new thread-cell-value table, copying values for preserved thread cells from *cells*.

```
int scheme_new_param()
```

Allocates a new primitive parameter index. This function must be called *before* `scheme_basic_env`, so it is only available to embedding applications (i.e., not extensions).

```
Scheme_Object* scheme_register_parameter(Scheme_Prim* function,
                                         char* name,
                                         int exnid)
```

Use this function instead of the other primitive-constructing functions, like `scheme_make_prim`, to create a primitive parameter procedure. See also `scheme_param_config`, below. This function is only available to embedding applications (i.e., not extensions).

```
Scheme_Object* scheme_param_config(char* name,
                                   Scheme_Object* param,
                                   int argc,
                                   Scheme_Object** argv,
                                   int arity,
                                   Scheme_Prim* check,
                                   char* expected,
                                   int isbool)
```

Call this procedure in a primitive parameter procedure to implement the work of getting or setting the parameter. The *name* argument should be the parameter procedure name; it is used to report errors. The *param* argument is a fixnum corresponding to the primitive parameter index returned by `scheme_new_param`. The *argc* and *argv* arguments should be the un-touched and un-tested arguments that were passed to the primitive parameter. Argument-checking is performed within `scheme_param_config` using *arity*, *check*, *expected*, and *isbool*:

- If *arity* is non-negative, potential parameter values must be able to accept the specified number of arguments. The *check* and *expected* arguments should be `NULL`.
- If *check* is not `NULL`, it is called to check a potential parameter value. The arguments passed to *check* are always `1` and an array that contains the potential parameter value. If *isbool* is `0` and *check* returns `scheme_false`, then a type error is reported using *name* and *expected* as a type description. If *isbool* is `1`, then a type error is reported only when *check* returns `NULL` and any non-`NULL` return value is used as the actual value to be stored for the parameter.
- Otherwise, *isbool* should be 1. A potential procedure argument is then treated as a Boolean value.

This function is only available to embedding applications (i.e., not extensions).

```
Scheme_Object* scheme_param_config2(char* name,
                                    Scheme_Object* param,
                                    int argc,
                                    Scheme_Object** argv,
                                    int arity,
                                    Scheme_Prim* check,
                                    char* expected_contract,
                                    int isbool)
```

The same as `scheme_param_config`, but with *expected_contract* as a contract instead of type description.

19 Continuation Marks (BC)

A mark can be attached to the current continuation frame using `scheme_set_cont_mark`. To force the creation of a new frame (e.g., during a nested function call within your function), use `scheme_push_continuation_frame`, and then remove the frame with `scheme_pop_continuation_frame`.

```
void scheme_set_cont_mark(Scheme_Object* key,
                          Scheme_Object* val)
```

Add/sets a continuation mark in the current continuation.

```
void scheme_push_continuation_frame(Scheme_Cont_Frame_Data* data)
```

Creates a new continuation frame. The *data* record need not be initialized, and it can be allocated on the C stack. Supply *data* to `scheme_pop_continuation_frame` to remove the continuation frame.

```
void scheme_pop_continuation_frame(Scheme_Cont_Frame_Data* data)
```

Removes a continuation frame created by `scheme_push_continuation_frame`.

20 String Encodings (BC)

The `scheme_utf8_decode` function decodes a `char` array as UTF-8 into either a UCS-4 `mzchar` array or a UTF-16 `short` array. The `scheme_utf8_encode` function encodes either a UCS-4 `mzchar` array or a UTF-16 `short` array into a UTF-8 `char` array.

These functions can be used to check or measure an encoding or decoding without actually producing the result decoding or encoding, and variations of the function provide control over the handling of decoding errors.

```
int scheme_utf8_decode(const unsigned char* s,
                       int start,
                       int end,
                       mzchar* us,
                       int dstart,
                       int dend,
                       intptr_t* ipos,
                       char utf16,
                       int permissive)
```

Decodes a byte array as UTF-8 to produce either Unicode code points into *us* (when *utf16* is zero) or UTF-16 code units into *us* cast to `short*` (when *utf16* is non-zero). No nul terminator is added to *us*.

The result is non-negative when all of the given bytes are decoded, and the result is the length of the decoding (in `mzchars` or `shorts`). A `-2` result indicates an invalid encoding sequence in the given bytes (possibly because the range to decode ended mid-encoding), and a `-3` result indicates that decoding stopped because not enough room was available in the result string.

The *start* and *end* arguments specify a range of *s* to be decoded. If *end* is negative, `strlen(`*s*`)` is used as the end.

If *us* is `NULL`, then decoded bytes are not produced, but the result is valid as if decoded bytes were written. The *dstart* and *dend* arguments specify a target range in *us* (in `mzchar` or `short` units) for the decoding; a negative value for *dend* indicates that any number of bytes can be written to *us*, which is normally sensible only when *us* is `NULL` for measuring the length of the decoding.

If *ipos* is non-`NULL`, it is filled with the first undecoded index within *s*. If the function result is non-negative, then `*`*ipos* is set to the ending index (with is *end* if non-negative, `strlen(`*s*`)` otherwise). If the result is `-1` or `-2`, then `*`*ipos* effectively indicates how many bytes were decoded before decoding stopped.

If *permissive* is non-zero, it is used as the decoding of bytes that are not part of a valid UTF-8 encoding or if the input ends in the middle of an encoding. Thus, the function result can be

`-1` or `-2` only if *permissive* is `0`.

On Windows, when *utf16* is non-zero, decoding supports a natural extension of UTF-8 that can produce unpaired UTF-16 surrogates in the result.

This function does not allocate or trigger garbage collection.

```
int scheme_utf8_decode_offset_prefix(const unsigned char* s,
                                     int start,
                                     int end,
                                     mzchar* us,
                                     int dstart,
                                     int dend,
                                     intptr_t* ipos,
                                     char utf16,
                                     int permissive)
```

Like `scheme_utf8_decode`, but returns `-1` if the input ends in the middle of a UTF-8 encoding even if *permission* is non-zero.

Added in version 6.0.1.13.

```
int scheme_utf8_decode_as_prefix(const unsigned char* s,
                                 int start,
                                 int end,
                                 mzchar* us,
                                 int dstart,
                                 int dend,
                                 intptr_t* ipos,
                                 char utf16,
                                 int permissive)
```

Like `scheme_utf8_decode`, but the result is always the number of the decoded `mzchar`s or `short`s. If a decoding error is encountered, the result is still the size of the decoding up until the error.

```
int scheme_utf8_decode_all(const unsigned char* s,
                           int len,
                           mzchar* us,
                           int permissive)
```

Like `scheme_utf8_decode`, but with fewer arguments. The decoding produces UCS-4 `mzchar`s. If the buffer *us* is non-`NULL`, it is assumed to be long enough to hold the decoding (which cannot be longer than the length of the input, though it may be shorter). If *len* is negative, `strlen(`*s*`)` is used as the input length.

```
int scheme_utf8_decode_prefix(const unsigned char* s,
                              int len,
                              mzchar* us,
                              int permissive)
```

Like `scheme_utf8_decode`, but with fewer arguments. The decoding produces UCS-4 `mzchar`s. The buffer *us* **must** be non-`NULL`, and it is assumed to be long enough to hold the decoding (which cannot be longer than the length of the input, though it may be shorter). If *len* is negative, `strlen(`*s*`)` is used as the input length.

In addition to the result of `scheme_utf8_decode`, the result can be `-1` to indicate that the input ended with a partial (valid) encoding. A `-1` result is possible even when *permissive* is non-zero.

```
mzchar* scheme_utf8_decode_to_buffer(const unsigned char* s,
                                     int len,
                                     mzchar* buf,
                                     int blen)
```

Like `scheme_utf8_decode_all` with *permissive* as `0`, but if *buf* is not large enough (as indicated by *blen*) to hold the result, a new buffer is allocated. Unlike other functions, this one adds a nul terminator to the decoding result. The function result is either *buf* (if it was big enough) or a buffer allocated with `scheme_malloc_atomic`.

```
mzchar* scheme_utf8_decode_to_buffer_len(const unsigned char* s,
                                         int len,
                                         mzchar* buf,
                                         int blen,
                                         intptr_t* ulen)
```

Like `scheme_utf8_decode_to_buffer`, but the length of the result (not including the terminator) is placed into *ulen* if *ulen* is non-`NULL`.

```
int scheme_utf8_decode_count(const unsigned char* s,
                             int start,
                             int end,
                             int* state,
                             int might_continue,
                             int permissive)
```

Like `scheme_utf8_decode`, but without producing the decoded `mzchar`s, and always returning the number of decoded `mzchar`s up until a decoding error (if any). If *might_continue* is non-zero, the a partial valid encoding at the end of the input is not decoded when *permissive* is also non-zero.

If *state* is non-`NULL`, it holds information about partial encodings; it should be set to zero for an initial call, and then passed back to `scheme_utf8_decode` along with bytes that extend the given input (i.e., without any unused partial encodings). Typically, this mode makes sense only when *might_continue* and *permissive* are non-zero.

```
int scheme_utf8_encode(const mzchar* us,
                       int start,
                       int end,
                       unsigned char* s,
                       int dstart,
                       char utf16)
```

Encodes the given UCS-4 array of `mzchar`s (if *utf16* is zero) or UTF-16 array of `short`s (if *utf16* is non-zero) into *s*. The *end* argument must be no less than *start*.

The array *s* is assumed to be long enough to contain the encoding, but no encoding is written if *s* is `NULL`. The *dstart* argument indicates a starting place in *s* to hold the encoding. No nul terminator is added to *s*.

The result is the number of bytes produced for the encoding (or that would be produced if *s* was non-`NULL`). Encoding never fails.

On Windows, when *utf16* is non-zero, encoding supports unpaired surrogates the input UTF-16 code-unit sequence, in which case encoding generates a natural extension of UTF-8 that encodes unpaired surrogates.

This function does not allocate or trigger garbage collection.

```
int scheme_utf8_encode_all(const mzchar* us,
                           int len,
                           unsigned char* s)
```

Like `scheme_utf8_encode` with `0` for *start*, *len* for *end*, `0` for *dstart* and `0` for *utf16*.

```
char* scheme_utf8_encode_to_buffer(const mzchar* s,
                                   int len,
                                   char* buf,
                                   int blen)
```

Like `scheme_utf8_encode_all`, but the length of *buf* is given, and if it is not long enough to hold the encoding, a buffer is allocated. A nul terminator is added to the encoded array. The result is either *buf* or an array allocated with `scheme_malloc_atomic`.

```
char* scheme_utf8_encode_to_buffer_len(const mzchar* s,
                                       int len,
                                       char* buf,
                                       int blen,
                                       intptr_t* rlen)
```

Like `scheme_utf8_encode_to_buffer`, but the length of the resulting encoding (not including a nul terminator) is reported in *rlen* if it is non-`NULL`.

```
unsigned short* scheme_ucs4_to_utf16(const mzchar* text,
                                     int start,
                                     int end,
                                     unsigned short* buf,
                                     int bufsize,
                                     intptr_t* ulen,
                                     int term_size)
```

Converts a UCS-4 encoding (the indicated range of *text*) to a UTF-16 encoding. The *end* argument must be no less than *start*.

A result buffer is allocated if *buf* is not long enough (as indicated by *bufsize*). If *ulen* is non-`NULL`, it is filled with the length of the UTF-16 encoding. The *term_size* argument indicates a number of `shorts` to reserve at the end of the result buffer for a terminator (but no terminator is actually written).

```
mzchar* scheme_utf16_to_ucs4(const unsigned short* text,
                             int start,
                             int end,
                             mzchar* buf,
                             int bufsize,
                             intptr_t* ulen,
                             int term_size)
```

Converts a UTF-16 encoding (the indicated range of *text*) to a UCS-4 encoding. The *end* argument must be no less than *start*.

A result buffer is allocated if *buf* is not long enough (as indicated by *bufsize*). If *ulen* is non-`NULL`, it is filled with the length of the UCS-4 encoding. The *term_size* argument indicates a number of `mzchars` to reserve at the end of the result buffer for a terminator (but no terminator is actually written).

21 Bignums, Rationals, and Complex Numbers (BC)

Racket supports integers of an arbitrary magnitude; when an integer cannot be represented as a fixnum (i.e., 30 or 62 bits plus a sign bit), then it is represented by the Racket type `scheme_bignum_type`. There is no overlap in integer values represented by fixnums and bignums.

Rationals are implemented by the type `scheme_rational_type`, composed of a numerator and a denominator. The numerator and denominator will be fixnums or bignums (possibly mixed).

Complex numbers are implemented by the type `scheme_complex_type`, composed of a real and imaginary part. The real and imaginary parts will either be both flonums, both exact numbers (fixnums, bignums, and rationals can be mixed in any way), or the real part will be exact 0 and the imaginary part will be a single-precision (when enabled) or double-pecision flonum.

`int scheme_is_exact(Scheme_Object* n)`

Returns `1` if *n* is an exact number, `0` otherwise (*n* need not be a number).

`int scheme_is_inexact(Scheme_Object* n)`

Returns `1` if *n* is an inexact number, `0` otherwise (*n* need not be a number).

`Scheme_Object* scheme_make_bignum(intptr_t v)`

Creates a bignum representing the integer *v*. This can create a bignum that otherwise fits into a fixnum. This must only be used to create temporary values for use with the `bignum` functions. Final results can be normalized with `scheme_bignum_normalize`. Only normalized numbers can be used with procedures that are not specific to bignums.

`Scheme_Object* scheme_make_bignum_from_unsigned(uintptr_t v)`

Like `scheme_make_bignum`, but works on unsigned integers.

`double scheme_bignum_to_double(Scheme_Object* n)`

Converts a bignum to a floating-point number, with reasonable but unspecified accuracy.

`float scheme_bignum_to_float(Scheme_Object* n)`

If Racket is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_bignum_to_double`.

`Scheme_Object* scheme_bignum_from_double(double d)`

Creates a bignum that is close in magnitude to the floating-point number *d*. The conversion accuracy is reasonable but unspecified.

```
Scheme_Object* scheme_bignum_from_float(float f)
```

If Racket is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_bignum_from_double`.

```
char* scheme_bignum_to_string(Scheme_Object* n,
                              int radix)
```

Writes a bignum into a newly allocated byte string.

```
Scheme_Object* scheme_read_bignum(mzchar* str,
                                  int offset,
                                  int radix)
```

Reads a bignum from a `mzchar` string, starting from position *offset* in *str*. If the string does not represent an integer, then `NULL` will be returned. If the string represents a number that fits in a fixnum, then a `scheme_integer_type` object will be returned.

```
Scheme_Object* scheme_read_bignum_bytes(char* str,
                                        int offset,
                                        int radix)
```

Like `scheme_read_bignum`, but from a UTF-8-encoding byte string.

```
Scheme_Object* scheme_bignum_normalize(Scheme_Object* n)
```

If *n* fits in a fixnum, then a `scheme_integer_type` object will be returned. Otherwise, *n* is returned.

```
Scheme_Object* scheme_make_rational(Scheme_Object* n,
                                    Scheme_Object* d)
```

Creates a rational from a numerator and denominator. The *n* and *d* parameters must be fixnums or bignums (possibly mixed). The resulting will be normalized (thus, a bignum or fixnum might be returned).

```
double scheme_rational_to_double(Scheme_Object* n)
```

Converts the rational *n* to a `double`.

```
float scheme_rational_to_float(Scheme_Object* n)
```

If Racket is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_rational_to_double`.

```
Scheme_Object* scheme_rational_numerator(Scheme_Object* n)
```

Returns the numerator of the rational *n*.

```
Scheme_Object* scheme_rational_denominator(Scheme_Object* n)
```

Returns the denominator of the rational *n*.

```
Scheme_Object* scheme_rational_from_double(double d)
```

Converts the given `double` into a maximally-precise rational.

```
Scheme_Object* scheme_rational_from_float(float d)
```

If Racket is not compiled with single-precision floats, this procedure is actually a macro alias for `scheme_rational_from_double`.

```
Scheme_Object* scheme_make_complex(Scheme_Object* r,
                                   Scheme_Object* i)
```

Creates a complex number from real and imaginary parts. The *r* and *i* arguments must be fixnums, bignums, flonums, or rationals (possibly mixed). The resulting number will be normalized (thus, a real number might be returned).

```
Scheme_Object* scheme_complex_real_part(Scheme_Object* n)
```

Returns the real part of the complex number *n*.

```
Scheme_Object* scheme_complex_imaginary_part(Scheme_Object* n)
```

Returns the imaginary part of the complex number *n*.

22 Ports and the Filesystem (BC)

Ports are represented as Racket values with the types `scheme_input_port_type` and `scheme_output_port_type`. The function `scheme_read` takes an input port value and returns the next S-expression from the port. The function `scheme_write` takes an output port and a value and writes the value to the port. Other standard low-level port functions are also provided, such as `scheme_getc`.

File ports are created with `scheme_make_file_input_port` and `scheme_make_file_output_port`; these functions take a `FILE *` file pointer and return a Scheme port. Strings are read or written with `scheme_make_byte_string_input_port`, which takes a nul-terminated byte string, and `scheme_make_byte_string_output_port`, which takes no arguments. The contents of a string output port are obtained with `scheme_get_byte_string_output`.

Custom ports, with arbitrary read/write handlers, are created with `scheme_make_input_port` and `scheme_make_output_port`.

When opening a file for any reason using a name provided from Racket, use `scheme_expand_filename` to normalize the filename and resolve relative paths.

```
Scheme_Object* scheme_read(Scheme_Object* port)
```

`read`s the next S-expression from the given input port.

```
void scheme_write(Scheme_Object* obj,
                  Scheme_Object* port)
```

`write`s the Scheme value *obj* to the given output port.

```
void scheme_write_w_max(Scheme_Object* obj,
                        Scheme_Object* port,
                        int n)
```

Like `scheme_write`, but the printing is truncated to *n* bytes. (If printing is truncated, the last bytes are printed as “.”.)

```
void scheme_display(Scheme_Object* obj,
                    Scheme_Object* port)
```

`display`s the Racket value *obj* to the given output port.

```
void scheme_display_w_max(Scheme_Object* obj,
                          Scheme_Object* port,
                          int n)
```

Like `scheme_display`, but the printing is truncated to *n* bytes. (If printing is truncated, the last three bytes are printed as “.”.)

```
void scheme_write_byte_string(char* str,
                              intptr_t len,
                              Scheme_Object* port)
```

Writes *len* bytes of *str* to the given output port.

```
void scheme_write_char_string(mzchar* str,
                              intptr_t len,
                              Scheme_Object* port)
```

Writes *len* characters of *str* to the given output port.

```
intptr_t scheme_put_byte_string(const char* who,
                                Scheme_Object* port,
                                char* str,
                                intptr_t d,
                                intptr_t len,
                                int rarely_block)
```

Writes *len* bytes of *str*, starting with the *d*th character. Bytes are written to the given output port, and errors are reported as from *who*.

If *rarely_block* is `0`, the write blocks until all *len* bytes are written, possibly to an internal buffer. If *rarely_block* is `2`, the write never blocks, and written bytes are not buffered. If *rarely_block* is `1`, the write blocks only until at least one byte is written (without buffering) or until part of an internal buffer is flushed.

Supplying `0` for *len* corresponds to a buffer-flush request. If *rarely_block* is `2`, the flush request is non-blocking, and if *rarely_block* is `0`, it is blocking. (A *rarely_block* of `1` is the same as `0` in this case.)

The result is `-1` if no bytes are written from *str* and unflushed bytes remain in the internal buffer. Otherwise, the return value is the number of written characters.

```
intptr_t scheme_put_char_string(const char* who,
                                Scheme_Object* port,
                                char* str,
                                intptr_t d,
                                intptr_t len)
```

Like `scheme_put_byte_string`, but for a `mzchar` string, and without the non-blocking option.

```
char* scheme_write_to_string(Scheme_Object* obj,
                             intptr_t* len)
```

Prints the Racket value *obj* using `write` to a newly allocated string. If *len* is not `NULL`, `*`*len* is set to the length of the bytes string.

```
void scheme_write_to_string_w_max(Scheme_Object* obj,
                                  intptr_t* len,
                                  int n)
```

Like `scheme_write_to_string`, but the string is truncated to *n* bytes. (If the string is truncated, the last three bytes are “.”.)

```
char* scheme_display_to_string(Scheme_Object* obj,
                               intptr_t* len)
```

Prints the Racket value *obj* using `display` to a newly allocated string. If *len* is not `NULL`, `*`*len* is set to the length of the string.

```
void scheme_display_to_string_w_max(Scheme_Object* obj,
                                    intptr_t* len,
                                    int n)
```

Like `scheme_display_to_string`, but the string is truncated to *n* bytes. (If the string is truncated, the last three bytes are “.”.)

```
void scheme_debug_print(Scheme_Object* obj)
```

Prints the Racket value *obj* using `write` to the main thread's output port.

```
void scheme_flush_output(Scheme_Object* port)
```

If *port* is a file port, a buffered data is written to the file. Otherwise, there is no effect. *port* must be an output port.

```
int scheme_get_byte(Scheme_Object* port)
```

Get the next byte from the given input port. The result can be `EOF`.

```
int scheme_getc(Scheme_Object* port)
```

Get the next character from the given input port (by decoding bytes as UTF-8). The result can be `EOF`.

```
int scheme_peek_byte(Scheme_Object* port)
```

Peeks the next byte from the given input port. The result can be `EOF`.

```
int scheme_peekc(Scheme_Object* port)
```

Peeks the next character from the given input port (by decoding bytes as UTF-8). The result can be `EOF`.

```
int scheme_peek_byte_skip(Scheme_Object* port,
                          Scheme_Object* skip)
```

Like `scheme_peek_byte`, but with a skip count. The result can be `EOF`.

```
int scheme_peekc_skip(Scheme_Object* port,
                      Scheme_Object* skip)
```

Like `scheme_peekc`, but with a skip count. The result can be `EOF`.

```
intptr_t scheme_get_byte_string(const char* who,
                                Scheme_Object* port,
                                char* buffer,
                                int offset,
                                intptr_t size,
                                int only_avail,
                                int peek,
                                Scheme_Object* peek_skip)
```

Gets multiple bytes at once from a port, reporting errors with the name *who*. The *size* argument indicates the number of requested bytes, to be put into the *buffer* array starting at *offset*. The return value is the number of bytes actually read, or `EOF` if an end-of-file is encountered without reading any bytes.

If *only_avail* is `0`, then the function blocks until *size* bytes are read or an end-of-file is reached. If *only_avail* is `1`, the function blocks only until at least one byte is read. If *only_avail* is `2`, the function never blocks. If *only_avail* is `-1`, the function blocks only until at least one byte is read but also allows breaks (with the guarantee that bytes are read or a break is raised, but not both).

If *peek* is non-zero, then the port is peeked instead of read. The *peek_skip* argument indicates a portion of the input stream to skip as a non-negative, exact integer (fixnum or bignum). In this case, an *only_avail* value of `1` means to continue the skip until at least one byte can be returned, even if it means multiple blocking reads to skip bytes.

If *peek* is zero, then *peek_skip* should be either `NULL` (which means zero) or the fixnum zero.

```
intptr_t scheme_get_char_string(const char* who,
                                Scheme_Object* port,
                                char* buffer,
                                int offset,
                                intptr_t size,
                                int peek,
                                Scheme_Object* peek_skip)
```

Like `scheme_get_byte_string`, but for characters (by decoding bytes as UTF-8), and without the non-blocking option.

```
intptr_t scheme_get_bytes(Scheme_Object* port,
                          intptr_t size,
                          char* buffer,
                          int offset)
```

For backward compatibility: calls `scheme_get_byte_string` in essentially the obvious way with *only_avail* as `0`; if *size* is negative, then it reads -*size* bytes with *only_avail* as `1`.

```
void scheme_ungetc(int ch,
                   Scheme_Object* port)
```

Puts the byte *ch* back as the next character to be read from the given input port. The character need not have been read from *port*, and `scheme_ungetc` can be called to insert up to five characters at the start of *port*.

Use `scheme_get_byte` followed by `scheme_ungetc` only when your program will certainly call `scheme_get_byte` again to consume the byte. Otherwise, use `scheme_peek_byte`, because some a port may implement peeking and getting differently.

```
int scheme_byte_ready(Scheme_Object* port)
```

Returns 1 if a call to `scheme_get_byte` is guaranteed not to block for the given input port.

```
int scheme_char_ready(Scheme_Object* port)
```

Returns 1 if a call to `scheme_getc` is guaranteed not to block for the given input port.

```
void scheme_need_wakeup(Scheme_Object* port,
                        void* fds)
```

Requests that appropriate bits are set in *fds* to specify which file descriptors(s) the given input port reads from. (*fds* is sortof a pointer to an `fd_set` struct; see §17.4.1 “Callbacks for Blocked Threads”.)

```
intptr_t scheme_tell(Scheme_Object* port)
```

Returns the current read position of the given input port, or the current file position of the given output port.

```
intptr_t scheme_tell_line(Scheme_Object* port)
```

Returns the current read line of the given input port. If lines are not counted, -1 is returned.

```
void scheme_count_lines(Scheme_Object* port)
```

Turns on line-counting for the given input port. To get accurate line counts, call this function immediately after creating a port.

```
intptr_t scheme_set_file_position(Scheme_Object* port,
                                  intptr_t pos)
```

Sets the file position of the given input or output port (from the start of the file). If the port does not support position setting, an exception is raised.

```
void scheme_close_input_port(Scheme_Object* port)
```

Closes the given input port.

```
void scheme_close_output_port(Scheme_Object* port)
```

Closes the given output port.

```
int scheme_get_port_file_descriptor(Scheme_Object* port,
                                    intptr_t* fd)
```

Fills **fd* with a file-descriptor value for *port* if one is available (i.e., the port is a file-stream port and it is not closed). The result is non-zero if the file-descriptor value is available, zero otherwise. On Windows, a “file descriptor” is a file `HANDLE`.

```
intptr_t scheme_get_port_fd(Scheme_Object* port)
```

Like `scheme_get_port_file_descriptor`, but a file descriptor or `HANDLE` is returned directly, and the result is `-1` if no file descriptor or `HANDLE` is available.

```
intptr_t scheme_get_port_socket(Scheme_Object* port,
                                intptr_t* s)
```

Fills **s* with a socket value for *port* if one is available (i.e., the port is a TCP port and it is not closed). The result is non-zero if the socket value is available, zero otherwise. On Windows, a socket value has type `SOCKET`.

```
Scheme_Object* scheme_make_port_type(char* name)
```

Creates a new port subtype.

```
 Scheme_Input_Port*
scheme_make_input_port(Scheme_Object* subtype,
                       void* data,
                       Scheme_Object* name,
                       Scheme_Get_String_Fun get_bytes_fun,
                       Scheme_Peek_String_Fun peek_bytes_fun,
                       Scheme_Progress_Evt_Fun progress_evt_fun,
                       Scheme_Peeked_Read_Fun peeked_read_fun,
                       Scheme_In_Ready_Fun char_ready_fun,
                       Scheme_Close_Input_Fun close_fun,
                       Scheme_Need_Wakeup_Input_Fun need_wakeup_fun,
                       int must_close)
```

Creates a new input port with arbitrary control functions. The *subtype* is an arbitrary value to distinguish the port’s class. The pointer *data* will be installed as the port’s user data, which can be extracted/set with the `SCHEME_INPORT_VAL` macro. The *name* object is used as the port's name (for `object-name` and as the default source name for `read-syntax`).

If *must_close* is non-zero, the new port will be registered with the current custodian, and *close_fun* is guaranteed to be called before the port is garbage-collected.

Although the return type of `scheme_make_input_port` is `Scheme_Input_Port*`, it can be cast into a `Scheme_Object*`.

The functions are as follows.

```
intptr_t get_bytes_fun(Scheme_Input_Port* port,
                       char* buffer,
                       intptr_t offset,
                       intptr_t size,
                       int nonblock,
                       Scheme_Object* unless)
```

Reads bytes into *buffer*, starting from *offset*, up to *size* bytes (i.e., *buffer* is at least *offset* plus *size* long). If *nonblock* is `0`, then the function can block indefinitely, but it should return when at least one byte of data is available. If *nonblock* is `1`, the function should never block. If *nonblock* is `2`, a port in unbuffered mode should return only bytes previously forced to be buffered; other ports should treat a *nonblock* of `2` like `1`. If *nonblock* is `-1`, the function can block, but should enable breaks while blocking. The function should return `0` if no bytes are ready in non-blocking mode. It should return `EOF` if an end-of-file is reached (and no bytes were read into *buffer*). Otherwise, the function should return the number of read bytes. The function can raise an exception to report an error.

The *unless* argument will be non-`NULL` only when *nonblocking* is non-zero (except as noted below), and only if the port supports progress events. If *unless* is non-`NULL` and `SCHEME_CDR(`*unless*`)` is non-`NULL`, the latter is a progress event specific to the port. The *get_bytes_fun* function should return `SCHEME_UNLESS_READY` instead of reading bytes if the event in *unless* becomes ready before bytes can be read. In particular, *get_bytes_fun* should check the event in *unless* before taking any action, and it should check the event in *unless* after any operation that may allow Racket thread swaps. If the read must block, then it should unblock if the event in *unless* becomes ready.

If `scheme_progress_evt_via_get` is used for *progress_evt_fun*, then *unless* can be non-`NULL` even when *nonblocking* is `0`. In all modes, *get_bytes_fun* must call `scheme_unless_ready` to check *unless_evt*. Furthermore, after any potentially thread-swapping operation, *get_bytes_fun* must call `scheme_wait_input_allowed`, because another thread may be attempting to commit, and *unless_evt* must be checked after `scheme_wait_input_allowed` returns. To block, the port should use `scheme_block_until_unless` instead of `scheme_block_until`. Finally, in blocking mode, *get_bytes_fun* must return after immediately reading data, without allowing a Racket thread swap.

```
intptr_t peek_bytes_fun(Scheme_Input_Port* port,
                        char* buffer,
                        intptr_t offset,
                        intptr_t size,
                        Scheme_Object* skip,
                        int nonblock,
                        Scheme_Object* unless_evt)
```

Can be `NULL` to use a default implementation of peeking that uses *get_bytes_fun*. Otherwise, the protocol is the same as for *get_bytes_fun*, except that an extra *skip* argument indicates the number of input elements to skip (but *skip* does not apply to *buffer*). The *skip* value will be a non-negative exact integer, either a fixnum or a bignum.

```
Scheme_Object* progress_evt_fun(Scheme_Input_Port* port)
```

Called to obtain a progress event for the port, such as for `port-progress-evt`. This function can be `NULL` if the port does not support progress events. Use `scheme_progress_evt_via_get` to obtain a default implementation, in which case *peeked_read_fun* should be `scheme_peeked_read_via_get`, and *get_bytes_fun* and *peek_bytes_fun* should handle *unless* as described above.

```
int peeked_read_fun(Scheme_Input_Port* port,
                    intptr_t amount,
                    Scheme_Object* unless_evt,
                    Scheme_Object* target_ch)
```

Called to commit previously peeked bytes, just like the sixth argument to `make-input-port`. Use `scheme_peeked_read_via_get` for the default implementation of commits when *progress_evt_fun* is `scheme_progress_evt_via_get`.

The *peeked_read_fun* function must call `scheme_port_count_lines` on a successful commit to adjust the port's position. If line counting is enabled for the port and if line counting uses the default implementation, *peeked_read_fun* should supply a non-`NULL` byte-string argument to `scheme_port_count_lines`, so that character and line counts can be tracked correctly.

```
int char_ready_fun(Scheme_Input_Port* port)
```

Returns `1` when a non-blocking *get_bytes_fun* will return bytes or an `EOF`.

```
void close_fun(Scheme_Input_Port* port)
```

Called to close the port. The port is not considered closed until the function returns.

```
void need_wakeup_fun(Scheme_Input_Port* port,
                     void* fds)
```

Called when the port is blocked on a read; *need_wakeup_fun* should set appropriate bits in *fds* to specify which file descriptor(s) it is blocked on. The *fds* argument is conceptually an array of three `fd_set` structs (one for read, one for write, one for exceptions), but manipulate this array using `scheme_get_fdset` to get a particular element of the array, and use `MZ_FD_XXX` instead of `FD_XXX` to manipulate a single "`fd_set`". On Windows, the first "`fd_set`" can also contain OS-level semaphores or other handles via `scheme_add_fd_handle`.

```
 Scheme_Output_Port*
scheme_make_output_port(Scheme_Object* subtype,
                        void* data,
                        Scheme_Object* name,
                        Scheme_Write_String_Evt_Fun write_bytes_evt_fun,
                        Scheme_Write_String_Fun write_bytes_fun,
                        Scheme_Out_Ready_Fun char_ready_fun,
                        Scheme_Close_Output_Fun close_fun,
                        Scheme_Need_Wakeup_Output_Fun need_wakeup_fun,
                        Scheme_Write_Special_Evt_Fun write_special_evt_fun,
                        Scheme_Write_Special_Fun write_special_fun,
                        int must_close)
```

Creates a new output port with arbitrary control functions. The *subtype* is an arbitrary value to distinguish the port's class. The pointer *data* will be installed as the port's user data, which can be extracted/set with the `SCHEME_OUTPORT_VAL` macro. The *name* object is used as the port's name.

If *must_close* is non-zero, the new port will be registered with the current custodian, and *close_fun* is guaranteed to be called before the port is garbage-collected.

Although the return type of `scheme_make_output_port` is `Scheme_Output_Port*`, it can be cast into a `Scheme_Object*`.

The functions are as follows.

```
intptr_t write_bytes_evt_fun(Scheme_Output_Port* port,
                             const char* buffer,
                             intptr_t offset,
                             intptr_t size)
```

Returns an event that writes up to *size* bytes atomically when event is chosen in a synchronization. Supply `NULL` if bytes cannot be written atomically, or supply `scheme_write_evt_via_write` to use the default implementation in terms of `write_bytes_fun` (with *rarely_block* as 2).

```
intptr_t write_bytes_fun(Scheme_Output_Port* port,
                         const char* buffer,
                         intptr_t offset,
                         intptr_t size,
                         int rarely_block,
                         int enable_break)
```

Write bytes from *buffer*, starting from *offset*, up to *size* bytes (i.e., *buffer* is at least *offset* plus *size* long). If *rarely_block* is `0`, then the function can block indefinitely, and it can buffer output. If *rarely_block* is `2`, the function should never block, and it should not buffer output. If *rarely_block* is `1`, the function should not buffer data, and it should block only until writing at least one byte, either from *buffer* or an internal buffer. The function should return the number of bytes from *buffer* that were written; when *rarely_block* is non-zero and bytes remain in an internal buffer, it should return `-1`. The *size* argument can be `0` when *rarely_block* is `0` for a blocking flush, and it can be `0` if *rarely_block* is `2` for a non-blocking flush. If *enable_break* is true, then it should enable breaks while blocking. The function can raise an exception to report an error.

```
int char_ready_fun(Scheme_Output_Port* port)
```

Returns `1` when a non-blocking *write_bytes_fun* will write at least one byte or flush at least one byte from the port's internal buffer.

```
void close_fun(Scheme_Output_Port* port)
```

Called to close the port. The port is not considered closed until the function returns. This function is allowed to block (usually to flush a buffer) unless `scheme_close_should_force_port_closed` returns a non-zero result, in which case the function must return without blocking.

```
void need_wakeup_fun(Scheme_Output_Port* port,
                     void* fds)
```

Called when the port is blocked on a write; *need_wakeup_fun* should set appropriate bits in *fds* to specify which file descriptor(s) it is blocked on. The *fds* argument is conceptually an array of three `fd_set` structs (one for read, one for write, one for exceptions), but manipulate this array using `scheme_get_fdset` to get a particular element of the array, and use `MZ_FD_XXX` instead of `FD_XXX` to manipulate a single "`fd_set`". On Windows, the first "`fd_set`" can also contain OS-level semaphores or other handles via `scheme_add_fd_handle`.

```
int write_special_evt_fun(Scheme_Output_Port* port,
                          Scheme_Object* v)
```

Returns an event that writes *v* atomically when event is chosen in a synchronization. Supply `NULL` if specials cannot be written atomically (or at all), or supply `scheme_write_special_evt_via_write_special` to use the default implementation in terms of `write_special_fun` (with *non_block* as `1`).

```
int write_special_fun(Scheme_Output_Port* port,
                      Scheme_Object* v,
                      int non_block)
```

Called to write the special value *v* for `write-special` (when *non_block* is `0`) or `write-special-avail*` (when *non_block* is `1`). If `NULL` is supplied instead of a function pointer, then `write-special` and `write-special-avail*` produce an error for this port.

```
 void
scheme_set_port_location_fun(Scheme_Port* port,
                             Scheme_Location_Fun location_fun)
```

Sets the implementation of `port-next-location` for *port*, which is used when line counting is enabled for *port*.

```
Scheme_Object* location_fun(Scheme_Port* port)
```

Returns three values: a positive exact integer or `#f` for a line number, a non-negative exact integer or `#f` for a column (which must be `#f` if and only if the line number is `#f`), and a positive exact integer or `#f` for a character position.

```
 void
scheme_set_port_count_lines_fun(Scheme_Port* port,
                                Scheme_Count_Lines_Fun count_lines_fun)
```

Installs a notification callback that is invoked if line counting is subsequently enabled for *port*.

```
void count_lines_fun(Scheme_Port* port)
```

```
void scheme_port_count_lines(Scheme_Port* port,
                             const char* buffer,
                             intptr_t offset,
                             intptr_t got)
```

Updates the position of *port* as reported by `file-position` as well as the locations reported by `port-next-location` when the default implement of character and line counting is used. This function is intended for use by a peek-commit implementation in an input port.

The *got* argument indicates the number of bytes read from or written to *port*. The *buffer* argument is used only when line counting is enabled, and it represents specific bytes read or written for the purposes of character and line coutning. The *buffer* argument can be `NULL`, in which case *got* non-newline characters are assumed. The *offset* argument indicates a starting offset into *buffer*, so `"buffer"` must be at least *offset* plus *got* bytes long.

```
Scheme_Object* scheme_make_file_input_port(FILE* fp)
```

Creates a Scheme input file port from an ANSI C file pointer. The file must never block on reads.

```
Scheme_Object* scheme_open_input_file(const char* filename,
                                      const char* who)
```

Opens *filename* for reading. If an exception is raised, the exception message uses *who* as the name of procedure that raised the exception.

```
 Scheme_Object*
scheme_make_named_file_input_port(FILE* fp,
                                  Scheme_Object* name)
```

Creates a Racket input file port from an ANSI C file pointer. The file must never block on reads. The *name* argument is used as the port's name.

```
Scheme_Object* scheme_open_output_file(const char* filename,
                                       const char* who)
```

Opens *filename* for writing in `'truncate/replace` mode. If an exception is raised, the exception message uses *who* as the name of procedure that raised the exception.

```
Scheme_Object* scheme_make_file_output_port(FILE* fp)
```

Creates a Racket output file port from an ANSI C file pointer. The file must never block on writes.

```
Scheme_Object* scheme_make_fd_input_port(int fd,
                                         Scheme_Object* name,
                                         int regfile,
                                         int win_textmode)
```

Creates a Racket input port for a file descriptor *fd*. On Windows, *fd* can be a `HANDLE` for a stream, and it should never be a file descriptor from the C library or a WinSock socket.

The *name* object is used for the port's name. Specify a non-zero value for *regfile* only if the file descriptor corresponds to a regular file (which implies that reading never blocks, for example).

On Windows, *win_textmode* can be non-zero to make trigger auto-conversion (at the byte level) of CRLF combinations to LF.

Closing the resulting port closes the file descriptor.

Instead of calling both `scheme_make_fd_input_port` and `scheme_make_fd_output_port` on the same file descriptor, call `scheme_make_fd_output_port` with a non-zero last argument. Otherwise, closing one of the ports causes the file descriptor used by the other to be closed as well.

```
Scheme_Object* scheme_make_fd_output_port(int fd,
                                          Scheme_Object* name,
                                          int regfile,
                                          int win_textmode,
                                          int read_too)
```

Creates a Racket output port for a file descriptor *fd*. On Windows, *fd* can be a `HANDLE` for a stream, and it should never be a file descriptor from the C library or a WinSock socket.

The *name* object is used for the port's name. Specify a non-zero value for *regfile* only if the file descriptor corresponds to a regular file (which implies that reading never blocks, for example).

On Windows, *win_textmode* can be non-zero to make trigger auto-conversion (at the byte level) of CRLF combinations to LF.

Closing the resulting port closes the file descriptor.

If *read_too* is non-zero, the function produces multiple values (see §15.3 "Multiple Values") instead of a single port. The first result is an input port for *fd*, and the second is an output port for *fd*. These ports are connected in that the file descriptor is closed only when both of the ports are closed.

```
void scheme_socket_to_ports(intptr_t s,
                            const char* name,
                            int close,
                            Scheme_Object** inp,
                            Scheme_Object** outp)
```

Creates Racket input and output ports for a TCP socket *s*. The *name* argument supplies the name for the ports. If *close* is non-zero, then the ports assume responsibility for closing the socket. The resulting ports are written to *inp* and *outp*.

Whether *close* is zero or not, closing the resulting ports unregisters the file descriptor with `scheme_fd_to_semaphore`. So, passing zero for *close* and also using the file descriptor with other ports or with `scheme_fd_to_semaphore` will not work right.

Changed in version 6.9.0.6: Changed ports to always unregister with `scheme_fd_to_semaphore`, since it's not safe to skip that step.

```
Scheme_Object* scheme_fd_to_semaphore(intptr_t fd,
                                      int mode,
                                      int is_socket)
```

Creates or finds a Racket semaphore that becomes ready when *fd* is ready. The semaphore reflects a registration with the operating system's underlying mechanisms for efficient polling.

When a semaphore is created, it remains findable via `scheme_fd_to_semaphore` for a particular read/write mode as long as *fd* has not become ready in the read/write mode since the creation of the semaphore, or unless `MZFD_REMOVE` has been used to remove the registered semaphore. The *is_socket* argument indicates whether *fd* is a socket or a filesystem descriptor; the difference matters for Windows, and it matters for BSD-based platforms where sockets are always supported and other file descriptors are tested for whether they correspond to a directory or regular file.

The *mode* argument is one of the following:

- `MZFD_CREATE_READ` (= `1`) — creates or finds a semaphore to reflect whether *fd* is ready for reading.
- `MZFD_CREATE_WRITE` (= 2) — creates or finds a semaphore to reflect whether *fd* is ready for writing.
- `MZFD_CHECK_READ` (= 3) — finds a semaphore to reflect whether *fd* is ready for reading; the result is `NULL` if no semaphore was previously created for *fd* in read mode or if such a semaphore has been posted or removed.
- `MZFD_CHECK_WRITE` (= 4) — like `MZFD_CREATE_READ`, but for write mode.
- `MZFD_REMOVE` (= 5) — removes all recorded semaphores for *fd* (unregistering a poll with the operating system) and returns `NULL`.
- `MZFD_CREATE_VNODE` (= `6`) — creates or finds a semaphore to reflect whether *fd* changes; on some platforms, `MZFD_CREATE_VNODE` is the same as `MZFD_CREATE_READ`; on other platforms, only one or the other can be used on a given file descriptor.
- `MZFD_CHECK_VNODE` (= `7`) — like `MZFD_CHECK_READ`, but to find a semaphore recorded via `MZFD_CREATE_VNODE`.
- `MZFD_REMOVE_VNODE` (= `8`) — like `MZFD_REMOVE`, but to remove a semaphore recorded via `MZFD_CREATE_VNODE`.

`Scheme_Object* scheme_make_byte_string_input_port(char*` *str*`)`

Creates a Racket input port from a byte string; successive `read-char`s on the port return successive bytes in the string.

`Scheme_Object* scheme_make_byte_string_output_port()`

Creates a Racket output port; all writes to the port are kept in a byte string, which can be obtained with `scheme_get_byte_string_output`.

`char* scheme_get_byte_string_output(Scheme_Object*` *port*`)`

Returns (in a newly allocated byte string) all data that has been written to the given string output port so far. (The returned string is nul-terminated.)

```
char* scheme_get_sized_byte_string_output(Scheme_Object* port,
                                          intptr_t* len)
```

Returns (in a newly allocated byte string) all data that has been written to the given string output port so far and fills in `*len` with the length of the string in bytes (not including the nul terminator).

```
void scheme_pipe(Scheme_Object** read,
                 Scheme_Object** write)
```

Creates a pair of ports, setting `*`*read* and `*`*write*; data written to `*`*write* can be read back out of `*`*read*. The pipe can store arbitrarily many unread characters,

```
void scheme_pipe_with_limit(Scheme_Object** read,
                            Scheme_Object** write,
                            int limit)
```

Like `scheme_pipe` if *limit* is `0`. If *limit* is positive, creates a pipe that stores at most *limit* unread characters, blocking writes when the pipe is full.

```
Scheme_Input_Port* scheme_input_port_record(Scheme_Object* port)
```

Returns the input-port record for *port*, which may be either a raw-port object with type `scheme_input_port_type` or a structure with the `prop:input-port` property.

```
 Scheme_Output_Port*
scheme_output_port_record(Scheme_Object* port)
```

Returns the output-port record for *port*, which may be either a raw-port object with type `scheme_output_port_type` or a structure with the `prop:output-port` property.

```
int scheme_file_exists(char* name)
```

Returns 1 if a file by the given name exists, 0 otherwise. If *name* specifies a directory, FALSE is returned. The *name* should be already expanded.

```
int scheme_directory_exists(char* name)
```

Returns 1 if a directory by the given name exists, 0 otherwise. The *name* should be already expanded.

```
char* scheme_expand_filename(const char* name,
                             int len,
                             const char* where,
                             int* expanded,
                             int checks)
```

Cleanses the pathname *name* (see `cleanse-path`) and resolves relative paths with respect to the current directory parameter. The *len* argument is the length of the input string; if it is -1, the string is assumed to be null-terminated. The *where* argument is used to raise an exception if there is an error in the filename; if this is `NULL`, an error is not reported and `NULL` is returned instead. If *expanded* is not `NULL`, **expanded* is set to 1 if some expansion takes place, or 0 if the input name is simply returned.

If *guards* is not `0`, then `scheme_security_check_file` (see §24 "Security Guards (BC)") is called with *name*, *where*, and *checks* (which implies that *where* should never be `NULL` unless *guards* is `0`). Normally, *guards* should be `SCHEME_GUARD_FILE_EXISTS` at a minimum. Note that a failed access check will result in an exception.

```
char* scheme_expand_string_filename(Scheme_Object* name,
                                    const char* where,
                                    int* expanded,
                                    int checks)
```

Like `scheme_expand_string`, but given a *name* that can be a character string or a path value.

```
Scheme_Object* scheme_char_string_to_path(Scheme_Object* s)
```

Converts a Racket character string into a Racket path value.

```
Scheme_Object* scheme_path_to_char_string(Scheme_Object* s)
```

Converts a Racket path value into a Racket character string.

```
Scheme_Object* scheme_make_path(char* bytes)
```

Makes a path value given a byte string. The *bytes* string is copied.

```
Scheme_Object* scheme_make_path_without_copying(char* bytes)
```

Like `scheme_make_path`, but the string is not copied.

```
Scheme_Object* scheme_make_sized_path(char* bytes,
                                      intptr_t len,
                                      int copy)
```

Makes a path whose byte form has size *len*. A copy of *bytes* is made if *copy* is not 0. The string *bytes* should contain *len* bytes, and if *copy* is zero, *bytes* must have a nul terminator in addition. If *len* is negative, then the nul-terminated length of *bytes* is used for the length.

```
Scheme_Object* scheme_make_sized_offset_path(char* bytes,
                                             intptr_t d,
                                             intptr_t len,
                                             int copy)
```

Like `scheme_make_sized_path`, except the *len* bytes start from position *d* in *bytes*. If *d* is non-zero, then *copy* must be non-zero.

```
char* scheme_build_mac_filename(FSSpec* spec,
                                int isdir)
```

Mac OS only: Converts an `FSSpec` record (defined by Mac OS) into a pathname string. If *spec* contains only directory information (via the `vRefNum` and `parID` fields), *isdir* should be `1`, otherwise it should be `0`.

```
int scheme_mac_path_to_spec(const char* filename,
                            FSSpec* spec,
                            intptr_t* type)
```

Mac OS only: Converts a pathname into an `FSSpec` record (defined by Mac OS), returning `1` if successful and `0` otherwise. If *type* is not `NULL` and *filename* is a file that exists, *type* is filled with the file's four-character Mac OS type. If *type* is not `NULL` and *filename* is not a file that exists, *type* is filled with `0`.

```
char* scheme_os_getcwd(char* buf,
                       int buflen,
                       int* actlen,
                       int noexn)
```

Gets the current working directory according to the operating system. This is separate from Racket's current directory parameter.

The directory path is written into *buf*, of length *buflen*, if it fits. Otherwise, a new (collectable) string is allocated for the directory path. If *actlen* is not `NULL`, **actlen* is set to the length of the current directory path. If *noexn* is no 0, then an exception is raised if the operation fails.

```
int scheme_os_setcwd(char* buf,
                     int noexn)
```

Sets the current working directory according to the operating system. This is separate from Racket's current directory parameter.

If *noexn* is not 0, then an exception is raised if the operation fails.

```
char* scheme_format(mzchar* format,
                    int flen,
                    int argc,
                    Scheme_Object** argv,
                    intptr_t* rlen)
```

Creates a string like Racket's `format` procedure, using the format string *format* (of length *flen*) and the extra arguments specified in *argc* and *argv*. If *rlen* is not `NULL`, `*`*rlen* is filled with the length of the resulting string.

```
void scheme_printf(char* format,
                   int flen,
                   int argc,
                   Scheme_Object** argv)
```

Writes to the current output port like Racket's `printf` procedure, using the format string *format* (of length *flen*) and the extra arguments specified in *argc* and *argv*.

```
char* scheme_format_utf8(char* format,
                         int flen,
                         int argc,
                         Scheme_Object** argv,
                         intptr_t* rlen)
```

Like `scheme_format`, but takes a UTF-8-encoding byte string.

```
void scheme_printf_utf8(char* format,
                        int flen,
                        int argc,
                        Scheme_Object** argv)
```

Like `scheme_printf`, but takes a UTF-8-encoding byte string.

```
int scheme_close_should_force_port_closed()
```

This function must be called by the close function for a port created with `scheme_make_output_port`.

23 Structures (BC)

A new Racket structure type is created with `scheme_make_struct_type`. This creates the structure type, but does not generate the constructor, etc. procedures. The `scheme_make_struct_values` function takes a structure type and creates these procedures. The `scheme_make_struct_names` function generates the standard structure procedures names given the structure type's name. Instances of a structure type are created with `scheme_make_struct_instance` and the function `scheme_is_struct_instance` tests a structure's type. The `scheme_struct_ref` and `scheme_struct_set` functions access or modify a field of a structure.

The structure procedure values and names generated by `scheme_make_struct_values` and `scheme_make_struct_names` can be restricted by passing any combination of these flags:

- `SCHEME_STRUCT_NO_TYPE` — the structure type value/name is not returned.
- `SCHEME_STRUCT_NO_CONSTR` — the constructor procedure value/name is not returned.
- `SCHEME_STRUCT_NO_PRED`— the predicate procedure value/name is not returned.
- `SCHEME_STRUCT_NO_GET` — the selector procedure values/names are not returned.
- `SCHEME_STRUCT_NO_SET` — the mutator procedure values/names are not returned.
- `SCHEME_STRUCT_GEN_GET` — the field-independent selector procedure value/name is returned.
- `SCHEME_STRUCT_GEN_SET` — the field-independent mutator procedure value/name is returned.
- `SCHEME_STRUCT_NO_MAKE_PREFIX` — the constructor name omits a `make-` prefix, like `struct` instead of `define-struct`.

When all values or names are returned, they are returned as an array with the following order: structure type, constructor, predicate, first selector, first mutator, second selector, etc., field-independent select, field-independent mutator. When particular values/names are omitted, the array is compressed accordingly.

```
Scheme_Object* scheme_make_struct_type(Scheme_Object* base_name,
                                       Scheme_Object* super_type,
                                       Scheme_Object* inspector,
                                       int num_init_fields,
                                       int num_auto_fields,
                                       Scheme_Object* auto_val,
                                       Scheme_Object* properties,
                                       Scheme_Object* guard)
```

Creates and returns a new structure type. The *base_name* argument is used as the name of the new structure type; it must be a symbol. The *super_type* argument should be `NULL` or an existing structure type to use as the super-type. The *inspector* argument should be `NULL` or an inspector to manage the type. The *num_init_fields* argument specifies the number of fields for instances of this structure type that have corresponding constructor arguments. (If a super-type is used, this is the number of additional fields, rather than the total number.) The *num_auto_fields* argument specifies the number of additional fields that have no corresponding constructor arguments, and they are initialized to *auto_val*. The *properties* argument is a list of property-value pairs. The *guard* argument is either NULL or a procedure to use as a constructor guard.

```
 Scheme_Object**
scheme_make_struct_names(Scheme_Object* base_name,
                         Scheme_Object* field_names,
                         int flags,
                         int* count_out)
```

Creates and returns an array of standard structure value name symbols. The *base_name* argument is used as the name of the structure type; it should be the same symbol passed to the associated call to `scheme_make_struct_type`. The *field_names* argument is a (Racket) list of field name symbols. The *flags* argument specifies which names should be generated, and if *count_out* is not `NULL`, *count_out* is filled with the number of names returned in the array.

```
 Scheme_Object**
scheme_make_struct_values(Scheme_Object* struct_type,
                          Scheme_Object** names,
                          int count,
                          int flags)
```

Creates and returns an array of the standard structure value and procedure values for *struct_type*. The *struct_type* argument must be a structure type value created by `scheme_make_struct_type`. The *names* procedure must be an array of name symbols, generally the array returned by `scheme_make_struct_names`. The *count* argument specifies the length of the *names* array (and therefore the number of expected return values) and the *flags* argument specifies which values should be generated.

```
 Scheme_Object*
scheme_make_struct_instance(Scheme_Object* struct_type,
                            int argc,
                            Scheme_Object** argv)
```

Creates an instance of the structure type *struct_type*. The *argc* and *argv* arguments provide the field values for the new instance.

```
int scheme_is_struct_instance(Scheme_Object* struct_type,
                              Scheme_Object* v)
```

Returns 1 if *v* is an instance of *struct_type* or 0 otherwise.

```
Scheme_Object* scheme_struct_ref(Scheme_Object* s,
                                 int n)
```

Returns the *n*th field (counting from 0) in the structure *s*.

```
void scheme_struct_set(Scheme_Object* s,
                       int n,
                       Scheme_Object* v)
```

Sets the *n*th field (counting from 0) in the structure *s* to *v*.

24 Security Guards (BC)

Before a primitive procedure accesses the filesystem or creates a network connection, it should first consult the current security guard to determine whether such access is allowed for the current thread.

File access is normally preceded by a call to `scheme_expand_filename`, which accepts flags to indicate the kind of filesystem access needed, so that the security guard is consulted automatically.

An explicit filesystem-access check can be made by calling `scheme_security_check_file`. Similarly, an explicit network-access check is performed by calling `scheme_security_check_network`.

```
void scheme_security_check_file(const char* who,
                                char* filename,
                                int guards)
```

Consults the current security manager to determine whether access is allowed to *filename*. The *guards* argument should be a bitwise combination of the following:

- `SCHEME_GUARD_FILE_READ`
- `SCHEME_GUARD_FILE_WRITE`
- `SCHEME_GUARD_FILE_EXECUTE`
- `SCHEME_GUARD_FILE_DELETE`
- `SCHEME_GUARD_FILE_EXISTS` (do not combine with other values)

The *filename* argument can be `NULL` (in which case `#f` is sent to the security manager's procedure), and *guards* should be `SCHEME_GUARD_FILE_EXISTS` in that case.

If access is denied, an exception is raised.

```
void scheme_security_check_network(const char* who,
                                   char* host,
                                   int portno)
```

Consults the current security manager to determine whether access is allowed for creating a client connection to *host* on port number *portno*. If *host* is `NULL`, the security manager is consulted for creating a server at port number *portno*.

If access is denied, an exception is raised.

25 Custodians (BC)

When an extension allocates resources that must be explicitly freed (in the same way that a port must be explicitly closed), a Racket object associated with the resource should be placed into the management of the current custodian with `scheme_add_managed`.

Before allocating the resource, call `scheme_custodian_check_available` to ensure that the relevant custodian is not already shut down. If it is, `scheme_custodian_check_available` will raise an exception. If the custodian is shut down when `scheme_add_managed` is called, the close function provided to `scheme_add_managed` will be called immediately, and no exception will be reported.

```
Scheme_Custodian* scheme_make_custodian(Scheme_Custodian* m)
```

Creates a new custodian as a subordinate of *m*. If *m* is `NULL`, then the main custodian is used as the new custodian's supervisor. Do not use `NULL` for *m* unless you intend to create an especially privileged custodian.

```
 Scheme_Custodian_Reference*
scheme_add_managed(Scheme_Custodian* m,
                   Scheme_Object* o,
                   Scheme_Close_Custodian_Client* f,
                   void* data,
                   int strong)
```

Places the value *o* into the management of the custodian *m*. If *m* is `NULL`, the current custodian is used.

The *f* function is called by the custodian if it is ever asked to "shutdown" its values; *o* and *data* are passed on to *f*, which has the type

```
typedef void (*Scheme_Close_Custodian_Client)(Scheme_Object *o,
                                              void *data);
```

If *strong* is non-zero, then the newly managed value will be remembered until either the custodian shuts it down or `scheme_remove_managed` is called. If *strong* is zero, the value is allowed to be garbage collected (and automatically removed from the custodian).

Independent of whether *strong* is zero, the value *o* is initially weakly held and becomes strongly held when the garbage collector attempts to collect it. A value associated with a custodian can therefore be finalized via will executors.

The return value from `scheme_add_managed` can be used to refer to the value's custodian later in a call to `scheme_remove_managed`. A value can be registered with at most one custodian.

If *m* (or the current custodian if *m* is `NULL`) is shut down, then *f* is called immediately, and the result is `NULL`.

See also `register-custodian-shutdown` from `ffi/unsafe/custodian`.

```
 Scheme_Custodian_Reference*
scheme_add_managed_close_on_exit(Scheme_Custodian* m,
                                 Scheme_Object* o,
                                 Scheme_Close_Custodian_Client* f,
                                 void* data)
```

Like `scheme_add_managed` with a `1` final argument, but also causes *f* to be called when Racket exists without an explicit custodian shutdown.

```
void scheme_custodian_check_available(Scheme_Custodian* m,
                                      const char* name,
                                      const char* resname)
```

Checks whether *m* is already shut down, and raises an error if so. If *m* is `NULL`, the current custodian is used. The *name* argument is used for error reporting. The *resname* argument will likely be used for checking pre-set limits in the future; pre-set limits will have symbolic names, and the *resname* string will be compared to the symbols.

```
void scheme_remove_managed(Scheme_Custodian_Reference* mref,
                           Scheme_Object* o)
```

Removes *o* from the management of its custodian. The *mref* argument must be a value returned by `scheme_add_managed` or `NULL`.

See also `unregister-custodian-shutdown` from `ffi/unsafe/custodian`.

```
void scheme_close_managed(Scheme_Custodian* m)
```

Instructs the custodian *m* to shutdown all of its managed values.

```
void scheme_add_atexit_closer(Scheme_Exit_Closer_Func f)
```

Installs a function to be called on each custodian-registered item and its closer when Racket is about to exit. The registered function has the type

```
typedef
void (*Scheme_Exit_Closer_Func)(Scheme_Object *o,
                                Scheme_Close_Custodian_Client *f,
                                void *d);
```

where *d* is the second argument for *f*.

At-exit functions are run in reverse of the order that they are added. An at-exit function is initially registered (and therefore runs last) that flushes each file-stream output port and calls every function registered with `scheme_add_managed_close_on_exit`.

An at-exit function should not necessarily apply the closer function for every object that it is given. In particular, shutting down a file-stream output port would disable the flushing action of the final at-exit function. Typically, an at-exit function ignores most objects while handling a specific type of object that requires a specific clean-up action before the OS-level process terminates.

`int scheme_atexit(Exit_Func` *func*`)`

Identical to calling the system's `atexit` function. Provided to give programs a common interface, different systems link to `atexit` in different ways. The type of *func* must be:

```
typedef void (*func)(void);
```

26 Subprocesses (BC)

On Unix and Mac OS, subprocess handling involves `fork`, `waitpid`, and `SIGCHLD`, which creates a variety of issues within an embedding application. On Windows, subprocess handling is more straightforward, since no `fork` is required, and since Windows provides an abstraction that is a close fit to Racket's subprocess values.

After Racket creates a subprocess via `subprocess` (or `system`, `process`, etc.), it periodically polls the process status using `waitpid`. If the process is created as its own group, then the call to `waitpid` uses the created subprocess's process ID; for all other subprocesses, polling uses a single call to `waitpid` with the first argument as `0`. Using `0`, in particular, can interfere with other libraries in an embedding context, so Racket refrains from calling `waitpid` if no subprocesses are pending.

Racket may or may not rely on a `SIGCHLD` handler, and it may or may not block `SIGCHLD`. Currently, when Racket is compiled to support places, Racket blocks `SIGCHLD` on start up with the expectation that all created threads have `SIGCHLD` blocked. When Racket is not compiled to support places, then a `SIGCHLD` handler is installed.

Using `fork` in an application that embeds Racket is problematic for several reasons: Racket may install a `SIGALRM` handler and schedule alarms to implement context switches, it may have file descriptors open that should be closed in a child process, and it may have changed the disposition of signals such as `SIGCHLD`. Consequently, embedding Racket in a process that `fork`s is technically not supported; in the future, Racket may provide better support for such applications.

27 Miscellaneous Utilities (BC)

The `MZSCHEME_VERSION` preprocessor macro is defined as a string describing the version of Racket. The `MZSCHEME_VERSION_MAJOR` and `MZSCHEME_VERSION_MINOR` macros are defined as the major and minor version numbers, respectively.

```
int scheme_eq(Scheme_Object* obj1,
              Scheme_Object* obj2)
```

Returns 1 if the Scheme values are `eq?`.

```
int scheme_eqv(Scheme_Object* obj1,
               Scheme_Object* obj2)
```

Returns 1 if the Scheme values are `eqv?`.

```
int scheme_equal(Scheme_Object* obj1,
                 Scheme_Object* obj2)
```

Returns 1 if the Scheme values are `equal?`.

```
int scheme_recur_equal(Scheme_Object* obj1,
                       Scheme_Object* obj2,
                       void* cycle_data)
```

Like `scheme_equal`, but accepts an extra value for cycle tracking. This procedure is meant to be called by a procedure installed with `scheme_set_type_equality`.

```
intptr_t scheme_equal_hash_key(Scheme_Object* obj)
```

Returns the primary `equal?`-hash key for *obj*.

```
intptr_t scheme_equal_hash_key2(Scheme_Object* obj)
```

Returns the secondary `equal?`-hash key for *obj*.

```
intptr_t scheme_recur_equal_hash_key(Scheme_Object* obj,
                                     void* cycle_data)
```

Like `scheme_equal_hash_key`, but accepts an extra value for cycle tracking. This procedure is meant to be called by a hashing procedure installed with `scheme_set_type_equality`.

```
intptr_t scheme_recur_equal_hash_key2(Scheme_Object* obj,
                                      void* cycle_data)
```

Like `scheme_equal_hash_key2`, but accepts an extra value for cycle tracking. This procedure is meant to be called by a secondary hashing procedure installed with `scheme_set_type_equality`.

```
Scheme_Object* scheme_build_list(int c,
                                 Scheme_Object** elems)
```

Creates and returns a list of length *c* with the elements *elems*.

```
int scheme_list_length(Scheme_Object* list)
```

Returns the length of the list. If *list* is not a proper list, then the last `cdr` counts as an item. If there is a cycle in *list* (involving only `cdr`s), this procedure will not terminate.

```
int scheme_proper_list_length(Scheme_Object* list)
```

Returns the length of the list, or -1 if it is not a proper list. If there is a cycle in *list* (involving only `cdr`s), this procedure returns -1.

```
Scheme_Object* scheme_car(Scheme_Object* pair)
```

Returns the `car` of the pair.

```
Scheme_Object* scheme_cdr(Scheme_Object* pair)
```

Returns the `cdr` of the pair.

```
Scheme_Object* scheme_cadr(Scheme_Object* pair)
```

Returns the `cadr` of the pair.

```
Scheme_Object* scheme_caddr(Scheme_Object* pair)
```

Returns the `caddr` of the pair.

```
Scheme_Object* scheme_vector_to_list(Scheme_Object* vec)
```

Creates a list with the same elements as the given vector.

```
Scheme_Object* scheme_list_to_vector(Scheme_Object* list)
```

Creates a vector with the same elements as the given list.

```
Scheme_Object* scheme_append(Scheme_Object* lstx,
                             Scheme_Object* lsty)
```

Non-destructively appends the given lists.

```
Scheme_Object* scheme_unbox(Scheme_Object* obj)
```

Returns the contents of the given box.

```
void scheme_set_box(Scheme_Object* b,
                    Scheme_Object* v)
```

Sets the contents of the given box.

```
Scheme_Object* scheme_dynamic_require(int argc,
                                      Scheme_Object** argv)
```

The same as `dynamic-require`. The *argc* argument must be 2, and *argv* contains the arguments.

```
 Scheme_Object*
scheme_namespace_require(Scheme_Object* prim_req_spec)
```

The same as `namespace-require`.

```
Scheme_Object* scheme_load(char* file)
```

Loads the specified Racket file, returning the value of the last expression loaded, or `NULL` if the load fails.

```
Scheme_Object* scheme_load_extension(char* filename)
```

Loads the specified Racket extension file, returning the value provided by the extension's initialization function.

```
Scheme_Hash_Table* scheme_make_hash_table(int type)
```

Creates a hash table. The *type* argument must be either `SCHEME_hash_ptr` or `SCHEME_hash_string`, which determines how keys are compared (unless the hash and compare functions are modified in the hash table record; see below). A `SCHEME_hash_ptr` table hashes on a key's pointer address, while `SCHEME_hash_string` uses a key as a `char*` and hashes on the null-terminated string content. Since a hash table created with `SCHEME_hash_string` (instead of `SCHEME_hash_ptr`) does not use a key as a Racket value, it cannot be used from Racket code.

Although the hash table interface uses the type `Scheme_Object*` for both keys and values, the table functions never inspect values, and they inspect keys only for `SCHEME_hash_string` hashing. Thus, the actual types of the values (and keys, for `SCHEME_hash_ptr` tables) can be anything.

The public portion of the `Scheme_Hash_Table` type is defined roughly as follows:

```
typedef struct Scheme_Hash_Table {
  Scheme_Object so; /* so.type == scheme_hash_table_type */
  /* ... */
  int size;  /* size of keys and vals arrays */
  int count; /* number of mapped keys */
  Scheme_Object **keys;
  Scheme_Object **vals;
  void (*make_hash_indices)(void *v, intptr_t *h1, intptr_t *h2);
  int (*compare)(void *v1, void *v2);
```

```
    /* ... */
  } Scheme_Hash_Table;
```

The `make_hash_indices` and `compare` function pointers can be set to arbitrary hashing and comparison functions (before any mapping is installed into the table). A hash function should fill *h1* with a primary hash value and *h2* with a secondary hash value; the values are for double-hashing, where the caller takes appropriate modulos. Either *h1* or *h2* can be `NULL` if the corresponding hash code is not needed.

To traverse the hash table content, iterate over *keys* and *vals* in parallel from `0` to `size-1`, and ignore *keys* where the corresponding *vals* entry is `NULL`. The `count` field indicates the number of non-`NULL` values that will be encountered.

```
Scheme_Hash_Table* scheme_make_hash_table_equal()
```

Like `scheme_make_hash_table`, except that keys are treated as Racket values and hashed based on `equal?` instead of `eq?`.

```
void scheme_hash_set(Scheme_Hash_Table* table,
                     Scheme_Object* key,
                     Scheme_Object* val)
```

Sets the current value for *key* in *table* to *val*. If *val* is `NULL`, the *key* is unmapped in *table*.

```
Scheme_Object* scheme_hash_get(Scheme_Hash_Table* table,
                               Scheme_Object* key)
```

Returns the current value for *key* in *table*, or `NULL` if *key* has no value.

```
Scheme_Bucket_Table* scheme_make_bucket_table(int size_hint,
                                              int type)
```

Like `make_hash_table`, but bucket tables are somewhat more flexible, in that hash buckets are accessible and weak keys are supported. (They also consume more space than hash tables.)

The *type* argument must be either `SCHEME_hash_ptr`, `SCHEME_hash_string`, or `SCHEME_hash_weak_ptr`. The first two are the same as for hash tables. The last is like `SCHEME_hash_ptr`, but the keys are weakly held.

The public portion of the `Scheme_Bucket_Table` type is defined roughly as follows:

```
  typedef struct Scheme_Bucket_Table {
    Scheme_Object so; /* so.type == scheme_variable_type */
    /* ... */
    int size;  /* size of buckets array */
    int count; /* number of buckets, >= number of mapped keys */
    Scheme_Bucket **buckets;
```

```
    void (*make_hash_indices)(void *v, intptr_t *h1, intptr_t *h2);
    int (*compare)(void *v1, void *v2);
    /* ... */
  } Scheme_Bucket_Table;
```

The `make_hash_indices` and `compare` functions are used as for hash tables. Note that `SCHEME_hash_weak_ptr` supplied as the initial type makes keys weak even if the hash and comparison functions are changed.

See `scheme_bucket_from_table` for information on buckets.

```
void scheme_add_to_table(Scheme_Bucket_Table* table,
                         const char* key,
                         void* val,
                         int const)
```

Sets the current value for *key* in *table* to *val*. If *const* is non-zero, the value for *key* must never be changed.

```
void scheme_change_in_table(Scheme_Bucket_Table* table,
                            const char* key,
                            void* val)
```

Sets the current value for *key* in *table* to *val*, but only if *key* is already mapped in the table.

```
void* scheme_lookup_in_table(Scheme_Bucket_Table* table,
                             const char* key)
```

Returns the current value for *key* in *table*, or `NULL` if *key* has no value.

```
 Scheme_Bucket*
scheme_bucket_from_table(Scheme_Bucket_Table* table,
                         const char* key)
```

Returns the bucket for *key* in *table*. The `Scheme_Bucket` structure is defined as:

```
  typedef struct Scheme_Bucket {
    Scheme_Object so; /* so.type == scheme_bucket_type */
    /* ... */
    void *key;
    void *val;
  } Scheme_Bucket;
```

Setting *val* to `NULL` unmaps the bucket's key, and *key* can be `NULL` in that case as well. If the table holds keys weakly, then *key* points to a (weak) pointer to the actual key, and the weak pointer's value can be `NULL`.

```
Scheme_Hash_Tree* scheme_make_hash_tree(int type)
```

Similar to `scheme_make_hash_table`, but creates a hash tree. A hash tree is equivalent to an immutable hash table created by `hash`. The *type* argument must be either `SCHEME_hashtr_eq`, `SCHEME_hashtr_equal`, or `SCHEME_hashtr_eqv`, which determines how keys are compared.

```
void scheme_hash_tree_set(Scheme_Hash_Tree* table,
                          Scheme_Object* key,
                          Scheme_Object* val)
```

Like `scheme_hash_set`, but operates on `Scheme_Hash_Tree`.

```
Scheme_Object* scheme_hash_tree_get(Scheme_Hash_Tree* table,
                                    Scheme_Object* key)
```

Like `scheme_hash_get`, but operates on `Scheme_Hash_Tree`.

```
intptr_t scheme_double_to_int(char* where,
                              double d)
```

Returns a fixnum value for the given floating-point number *d*. If *d* is not an integer or if it is too large, then an error message is reported; *name* is used for error-reporting.

```
intptr_t scheme_get_milliseconds()
```

Returns the current “time” in milliseconds, just like `current-milliseconds`.

```
intptr_t scheme_get_process_milliseconds()
```

Returns the current process “time” in milliseconds, just like `(current-process-milliseconds)`.

```
intptr_t scheme_get_process_children_milliseconds()
```

Returns the current process group “time” in milliseconds just like `(current-process-milliseconds 'subprocesses)`.

```
char* scheme_banner()
```

Returns the string that is used as the Racket startup banner.

```
char* scheme_version()
```

Returns a string for the executing version of Racket.

```
Scheme_Hash_Table* scheme_get_place_table()
```

Returns an `eq?`-based hash table that is global to the current place.

A key generated by `scheme_malloc_key` can be useful as a common key across multiple places.

```
Scheme_Object* scheme_malloc_key()
```

Generates an uncollectable Racket value that can be used across places. Free the value with `scheme_free_key`.

```
void scheme_free_key(Scheme_Object* key)
```

Frees a key allocated with `scheme_malloc_key`. When a key is freed, it must not be accessible from any GC-travsered reference in any place.

```
void* scheme_register_process_global(const char* key,
                                     void* val)
```

Gets or sets a value in a process-global table (i.e., shared across multiple places, if any). If *val* is `NULL`, the current mapping for *key* is given. If *val* is not `NULL`, and no value has been installed for that *key*, then the value is installed and `NULL` is returned. If a value has already been installed, then no new value is installed and the old value is returned. The given *val* must not refer to garbage-collected memory.

This function is intended for infrequent use with a small number of keys.

See also `register-process-global` from `ffi/unsafe/global`.

```
void* scheme_jit_find_code_end(void* p)
```

Given the address of machine code generated by Racket's compiler, attempts to infer and return the address just after the end of the generated code (for a single source function, typically). The result is `#f` if the address cannot be inferred, which may be because the given *p* does not refer to generated machine code.

Added in version 6.0.1.9.

```
void scheme_jit_now(Scheme_Object* val)
```

If *val* is a procedure that can be JIT-compiled, JIT compilation is forced immediately if it has not been forced already (usually through calling the function).

Added in version 6.0.1.10.

Part III

Appendices

28 Building Racket from Source

The normal Racket distribution includes `".rkt"` sources for collection-based libraries. After modifying library files, run `raco setup` (see §6 “`raco setup`: Installation Management”) to rebuild installed libraries.

The normal Racket distribution does not include the C sources for Racket’s run-time system. To build Racket from scratch, download a source distribution from `http://download.racket-lang.org`. Detailed build instructions are in the `"README.txt"` file in the top-level `"src"` directory. You can also get the latest sources from the `git` repository at `https://github.com/racket/racket`, but beware that the repository is one step away from a normal source distribution, and it provides build modes that are more suitable for developing Racket itself; see `"build.md"` in the `git` repository for more information.

29 Cross-compiling Racket Sources for iOS

See §6.14 “API for Cross-Platform Configuration” for general information on using Racket in cross-build mode. Everything in this section can be adapted to other cross-compilation targets, but iOS is used to make the examples concrete.

After cross-compiling Racket CS for iOS according to the source distribution’s `"src/README.txt"` file, you can use that build ⟨*ios-racket-dir*⟩ in conjunction with the host build it was compiled by to cross-compile Racket modules for iOS by passing the following set of flags to the host executable:

```
racket  \
  --compile-any  \
  --compiled ⟨ios-racket-dir⟩/src/build/cs/c/compiled:  \
  --cross  \
  --cross-compiler tarm64osx ⟨ios-racket-dir⟩/lib  \
  --config ⟨ios-racket-dir⟩/etc  \
  --collects ⟨ios-racket-dir⟩/collects
```

The above command runs the host Racket REPL with support for writing compiled code for both the host machine and for the `tarm64osx` target. The first path to `--compiled` (before the `:`) can be any absolute path, and `".zo"` files for the host platform will be written there; specifying the path `"⟨ios-racket-dir⟩/src/build/cs/c/compiled"` is meant to reuse the directory that was created during cross-compilation installation. The second path to `--compiled` (after `:`) is empty, which causes target-platform `".zo"` files to be written in the usual `"compiled"` subdirectory.

Instruct the host Racket to run library code by passing the `-l` flag. For example, you can setup the target Racket’s installation with the following command:

```
racket  \
  --compile-any  \
  --compiled ⟨ios-racket-dir⟩/src/build/cs/c/compiled:  \
  --cross  \
  --cross-compiler tarm64osx ⟨ios-racket-dir⟩/lib  \
  --config ⟨ios-racket-dir⟩/etc  \
  --collects ⟨ios-racket-dir⟩/collects  \
  -l-  \
  raco setup
```

Finally, you can package up a Racket module and its dependencies for use with `racket_embedded_load_file` (after installing `"compiler-lib"` and `"cext-lib"` for the target Racket) with:

```
racket  \
  --compile-any  \
```

```
--compiled ⟨ios-racket-dir⟩/src/build/cs/c/compiled:  \
--cross  \
--cross-compiler tarm64osx ⟨ios-racket-dir⟩/lib  \
--config ⟨ios-racket-dir⟩/etc  \
--collects ⟨ios-racket-dir⟩/collects  \
-l-  \
raco ctool --mods application.zo src/application.rkt
```

30 Embedding Files in Executable Sections

Locating external files on startup, such as the boot files needed for Racket CS, can be troublesome. An alternative to having separate files is to embed the files in an ELF or Mach-O executable as data segments or in a Windows executable as a resource. Embedding files in that way requires using OS-specific linking steps and runtime libraries.

30.1 Accessing ELF Sections on Linux

On Linux and other ELF-based systems, you can add sections to an executable using `objcopy`. For example, the following command copies `"pre_run"` to `run` while adding boot files as sections:

```
objcopy --add-section .csboot1=petite.boot \
        --set-section-flags .csboot1=noload,readonly \
        --add-section .csboot2=scheme.boot \
        --set-section-flags .csboot2=noload,readonly \
        --add-section .csboot3=racket.boot \
        --set-section-flags .csboot3=noload,readonly \
        ./pre_run ./run
```

Here's an implementation for `"pre_run"` like the one in §2 "Embedding into a Program (CS)", but where boot files are loaded from sections:

`"main.c"`

```
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <errno.h>
#include <elf.h>
#include <fcntl.h>

#include "chezscheme.h"
#include "racketcs.h"

#include "run.c"

static char *get_self_path()
{
  ssize_t len, blen = 256;
  char *s = malloc(blen);
```

```
    while (1) {
      len = readlink("/proc/self/exe", s, blen-1);
      if (len == (blen-1)) {
        free(s);
        blen *= 2;
        s = malloc(blen);
      } else if (len < 0) {
        fprintf(stderr, "failed to get self (%d)\n", errno);
        exit(1);
      } else
        return s;
    }
  }

  static long find_section(const char *exe, const char *sectname)
  {
    int fd, i;
    Elf64_Ehdr e;
    Elf64_Shdr s;
    char *strs;

    fd = open(exe, O_RDONLY, 0);
    if (fd != -1) {
      if (read(fd, &e, sizeof(e)) == sizeof(e)) {
        lseek(fd, e.e_shoff + (e.e_shstrndx * e.e_shentsize),
SEEK_SET);
        if (read(fd, &s, sizeof(s)) == sizeof(s)) {
          strs = (char *)malloc(s.sh_size);
          lseek(fd, s.sh_offset, SEEK_SET);
          if (read(fd, strs, s.sh_size) == s.sh_size) {
            for (i = 0; i < e.e_shnum; i++) {
              lseek(fd, e.e_shoff + (i * e.e_shentsize), SEEK_SET);
              if (read(fd, &s, sizeof(s)) != sizeof(s))
                break;
              if (!strcmp(strs + s.sh_name, sectname)) {
                close(fd);
                return s.sh_offset;
              }
            }
          }
        }
      }
      close(fd);
    }

    fprintf(stderr, "could not find section %s\n", sectname);
```

```
  return -1;
}

int main(int argc, char *argv[])
{
  racket_boot_arguments_t ba;

  memset(&ba, 0, sizeof(ba));

  ba.boot1_path = get_self_path();
  ba.boot2_path = ba.boot1_path;
  ba.boot3_path = ba.boot1_path;

  ba.boot1_offset = find_section(ba.boot1_path, ".csboot1");
  ba.boot2_offset = find_section(ba.boot2_path, ".csboot2");
  ba.boot3_offset = find_section(ba.boot3_path, ".csboot3");

  ba.exec_file = argv[0];

  racket_boot(&ba);

  declare_modules();

  ptr mod = Scons(Sstring_to_symbol("quote"),
                  Scons(Sstring_to_symbol("run"),
                        Snil));

  racket_dynamic_require(mod, Sfalse);

  return 0;
}
```

30.2 Accessing Mac OS Sections

On Mac OS, sections can be added to a Mach-O executable using the `-sectcreate` compiler flag. If `"main.c"` is compiled and linked with

```
gcc main.c libracketcs.a -Ipath/to/racket/include \
    -liconv -lncurses -framework CoreFoundation \
    -sectcreate __DATA __rktboot1 petite.boot \
    -sectcreate __DATA __rktboot2 scheme.boot \
    -sectcreate __DATA __rktboot2 racket.boot
```

then the executable can access is own path using `_NSGetExecutablePath`, and it can locate sections using `getsectbyname`. Here's an example like the one in §2 "Embedding into a Program (CS)":

"main.c"

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "chezscheme.h"
#include "racketcs.h"

#include "run.c"

#include <mach-o/dyld.h>
#include <mach-o/getsect.h>

static char *get_self_path()
{
  char *s;
  uint32_t size = 0;
  int r;

  r = _NSGetExecutablePath(NULL, &size);
  s = malloc(size+1);
  r = _NSGetExecutablePath(s, &size);
  if (!r)
    return s;

  fprintf(stderr, "could not get executable path\n");
  exit(1);
}

static long find_section(char *segname, char *sectname)
{
  const struct section_64 *s = getsectbyname(segname, sectname);
  if (s)
    return s->offset;

  fprintf(stderr, "could not find segment %s section %s\n",
          segname, sectname);
  exit(1);
}

#endif

int main(int argc, char **argv)
```

```
{
  racket_boot_arguments_t ba;

  memset(&ba, 0, sizeof(ba));

  ba.boot1_path = get_self_path();
  ba.boot2_path = ba.boot1_path;
  ba.boot3_path = ba.boot1_path;

  ba.boot1_offset = find_section("__DATA", "__rktboot1");
  ba.boot2_offset = find_section("__DATA", "__rktboot2");
  ba.boot3_offset = find_section("__DATA", "__rktboot3");

  ba.exec_file = argv[0];
  ba.run_file = argv[0];

  racket_boot(&ba);

  declare_modules(); /* defined by "run.c" */

  ptr mod = Scons(Sstring_to_symbol("quote"),
                  Scons(Sstring_to_symbol("run"),
                        Snil));

  racket_dynamic_require(mod, Sfalse);

  return 0;
}
```

30.3 Accessing Windows Resources

On Windows, data is most readily added to an executable as a resource. The following code demonstrates how to find the path to the current executable and how to find a resource in the executable by identifying number, type (usually `1`) and encoding (usual `1033`):

"main.c"

```
/* forward declaration for internal helper */
static DWORD find_by_id(HANDLE fd, DWORD rsrcs, DWORD pos, int id);

static wchar_t *get_self_executable_path()
{
  wchar_t *path;
  DWORD r, sz = 1024;
```

```
    while (1) {
      path = (wchar_t *)malloc(sz * sizeof(wchar_t));
      r = GetModuleFileNameW(NULL, path, sz);
      if ((r == sz)
          && (GetLastError() == ERROR_INSUFFICIENT_BUFFER)) {
        free(path);
        sz = 2 * sz;
      } else
        break;
    }

    return path;
  }

  static long find_resource_offset(wchar_t *path, int id, int type,
int encoding)
  {
    /* Find the resource of type `id` */
    HANDLE fd;

    fd = CreateFileW(path, GENERIC_READ,
                     FILE_SHARE_READ | FILE_SHARE_WRITE,
                     NULL,
                     OPEN_EXISTING,
                     0,
                     NULL);

    if (fd == INVALID_HANDLE_VALUE)
      return 0;
    else {
      DWORD val, got, sec_pos, virtual_addr, rsrcs, pos;
      WORD num_sections, head_size;
      char name[8];

      SetFilePointer(fd, 60, 0, FILE_BEGIN);
      ReadFile(fd, &val, 4, &got, NULL);
      SetFilePointer(fd, val+4+2, 0, FILE_BEGIN); /* Skip "PE\0\0"
tag and machine */
      ReadFile(fd, &num_sections, 2, &got, NULL);
      SetFilePointer(fd, 12, 0, FILE_CURRENT); /* time stamp + symbol
table */
      ReadFile(fd, &head_size, 2, &got, NULL);

      sec_pos = val+4+20+head_size;
      while (num_sections--) {
        SetFilePointer(fd, sec_pos, 0, FILE_BEGIN);
```

```
        ReadFile(fd, &name, 8, &got, NULL);
        if ((name[0] == '.')
            && (name[1] == 'r')
            && (name[2] == 's')
            && (name[3] == 'r')
            && (name[4] == 'c')
            && (name[5] == 0)) {
          SetFilePointer(fd, 4, 0, FILE_CURRENT); /* skip virtual
size */
          ReadFile(fd, &virtual_addr, 4, &got, NULL);
          SetFilePointer(fd, 4, 0, FILE_CURRENT); /* skip file size
*/
          ReadFile(fd, &rsrcs, 4, &got, NULL);
          SetFilePointer(fd, rsrcs, 0, FILE_BEGIN);

          /* We're at the resource table; step through 3 layers */
          pos = find_by_id(fd, rsrcs, rsrcs, id);
          if (pos) {
            pos = find_by_id(fd, rsrcs, pos, type);
            if (pos) {
              pos = find_by_id(fd, rsrcs, pos, encoding);

              if (pos) {
                /* pos is the reource data entry */
                SetFilePointer(fd, pos, 0, FILE_BEGIN);
                ReadFile(fd, &val, 4, &got, NULL);
                pos = val - virtual_addr + rsrcs;

                CloseHandle(fd);

                return pos;
              }
            }
          }

          break;
        }
        sec_pos += 40;
      }

      /* something went wrong */
      CloseHandle(fd);
      return -1;
    }
  }
```

```
/* internal helper function */
static DWORD find_by_id(HANDLE fd, DWORD rsrcs, DWORD pos, int id)
{
  DWORD got, val;
  WORD name_count, id_count;

  SetFilePointer(fd, pos + 12, 0, FILE_BEGIN);
  ReadFile(fd, &name_count, 2, &got, NULL);
  ReadFile(fd, &id_count, 2, &got, NULL);

  pos += 16 + (name_count * 8);
  while (id_count--) {
    ReadFile(fd, &val, 4, &got, NULL);
    if (val == id) {
      ReadFile(fd, &val, 4, &got, NULL);
      return rsrcs + (val & 0x7FFFFFF);
    } else {
      ReadFile(fd, &val, 4, &got, NULL);
    }
  }

  return 0;
}
```

Index