Using C code with Scheme 48

Mike Sperber
sperber@informatik.uni-tuebingen.de
Richard Kelsey
kelsey@research.nj.nec.com

February 23, 1999

Abstract

This document describes an interface for calling C functions from Scheme, calling Scheme functions from C, and allocating storage in the Scheme heap. These facilities are designed to link existing C libraries into Scheme 48 in order to use them from Scheme. To this end, Scheme 48 manages stub functions in C that negotiate between the calling conventions of Scheme and C and the memory allocation policies of both worlds. No stub generator is available yet, but writing them is a straightforward task.

1 Available Facilities

The following facilities are available for interfacing between Scheme 48 and C:

- Scheme code can call C functions.
- The external interface provides full introspection for all Scheme objects. External code may inspect, modify, and allocate Scheme objects arbitrarily.
- External code may raise exceptions back to Scheme 48 to signal errors.
- External code may call back into Scheme. Scheme 48 correctly unrolls the process stack on non-local exits.
- External modules may register bindings of names to values with a central registry accessible from Scheme. Conversely, Scheme code can register shared bindings for access by C code.

This document has three parts: the first describes how bindings are moved from Scheme to C and vice versa, the second tells how to call C functions from Scheme, and the third covers the C interface to Scheme objects, including calling Scheme procedures, using the Scheme heap, and so forth.

1.1 Scheme structures

The structure `external-calls` has most of the Scheme functions described here. The others are in `dynamic-externals`, which has the functions for dynamic loading and name lookup from Section 5, and `shared-bindings`, which has the additional shared-binding functions described in Section 2.3.

1.2 C naming conventions

The names of all of Scheme 48's visible C bindings begin with '`s48_`' (for procedures and variables) or '`S48_`' (for macros). Whenever a C name is derived from a Scheme identifier, we replace '`-`' with '`_`' and convert letters to lowercase for procedures and uppercase for macros. A final '`?`' converted to '`_p`' ('`_P`' in C macro names). A final '`!`' is dropped. Thus the C macro for Scheme's `pair?` is `S48_PAIR_P` and the one for `set-car!` is `S48_SET_CAR`. Procedures and macros that do not check the types of their arguments have '`unsafe`' in their names.

All of the C functions and macros described have prototypes or definitions in the file `c/scheme48.h`. The C type for Scheme values is defined there to be `s48_value`.

2 Shared bindings

Shared bindings are the means by which named values are shared between Scheme code and C code. There are two separate tables of shared bindings, one for values defined in Scheme and accessed from C and the other for values going the other way. Shared bindings actually bind names to cells, to allow a name to be looked up before it has been assigned. This is necessary because C initialization code may be run before or after the corresponding Scheme code, depending on whether the Scheme code is in the resumed image or is run in the current session.

2.1 Exporting Scheme values to C

- `(define-exported-binding` *name value*`)` → *shared-binding*

- `s48_value s48_get_imported_binding(char *name)`
- `s48_value S48_SHARED_BINDING_REF(s48_value shared_binding)`

`Define-exported-binding` makes *value* available to C code under as *name* which must be a *string*, creating a new shared binding if necessary. The C function `s48_get_imported_binding` returns the shared binding defined for `name`, again creating it if necessary. The C macro `S48_SHARED_BINDING_REF` dereferences a shared binding, returning its current value.

2.2 Exporting C values to Scheme

- `void s48_define_exported_binding(char *name, s48_value value)`

- `(lookup-imported-binding` *string*`)` → *shared-binding*
- `(shared-binding-ref` *shared-binding*`)` → *value*

These are used to define shared bindings from C and to access them from Scheme. Again, if a name is looked up before it has been defined, a new binding is created for it.

The common case of exporting a C function to Scheme can be done using the macro `S48_EXPORT_FUNCTION(`*name*`)`. This expands into

`s48_define_exported_binding("`*name*`", s48_enter_pointer(`*name*`))`

which boxes the function into a Scheme byte vector and then exports it. Note that `s48_enter_pointer` allocates space in the Scheme heap and might trigger a garbage collection; see Section 9.

- `(import-definition` *name*`)` syntax
- `(import-definition` *name c-name*`)` syntax

These macros simplify importing definitions from C to Scheme. They expand into

`(define` *name* `(lookup-imported-binding` *c-name*`))`

where *c-name* is as supplied for the second form. For the first form *c-name* is derived from *name* by replacing '`-`' with '`_`' and converting letters to lowercase. For example, `(import-definition my-foo)` expands into

`(define my-foo (lookup-imported-binding "my_foo"))`

2.3 Complete shared binding interface

There are a number of other Scheme functions related to shared bindings; these are in the structure `shared-bindings`.

- `(shared-binding?` *x*`)` → *boolean*
- `(shared-binding-name` *shared-binding*`)` → *string*
- `(shared-binding-is-import?` *shared-binding*`)` → *boolean*
- `(shared-binding-set!` *shared-binding value*`)`
- `(define-imported-binding` *string value*`)`
- `(lookup-exported-binding` *string*`)`
- `(undefine-imported-binding` *string*`)`
- `(undefine-exported-binding` *string*`)`

`Shared-binding?` is the predicate for shared-bindings. `Shared-binding-name` returns the name of a binding. `Shared-binding-is-import?` is true if the binding was defined from C. `Shared-binding-set!` changes the value of a binding. `Define-imported-binding` and `lookup-exported-binding` are Scheme versions of `s48_define_exported_binding` and `s48_lookup_imported_binding`. The two `undefine-` procedures remove bindings from the two tables. They do nothing if the name is not found in the table.

The following C macros correspond to the Scheme functions above.

- `int       S48_SHARED_BINDING_P(x)`
- `int       S48_SHARED_BINDING_IS_IMPORT_P(s48_value s_b)`
- `s48_value S48_SHARED_BINDING_NAME(s48_value s_b)`
- `void      S48_SHARED_BINDING_SET(s48_value s_b, s48_value value)`

3 Calling C Functions from Scheme

There are three different ways to call C functions from Scheme, depending on how the C function was obtained.

- `(call-imported-binding` *binding* arg_0 ...`)` → *value*
- `(call-external` *external* arg_0 ...`)` → *value*
- `(call-external-value` *value* *name* arg_0 ...`)` → *value*

Each of these applies its first argument, a C function, to the rest of the arguments. For `call-imported-binding` the function argument must be an imported binding. For `call-external` the function argument must be an external bound in the current process (see Section 5). For `call-external-value` *value* must be a byte vector whose contents is a pointer to a C function and *name* should be a string naming the function. The *name* argument is used only for printing error messages.

For all of these, the C function is passed the arg_i values and the value returned is that returned by C procedure. Up to twelve arguments may be passed. There is no method supplied for returning multiple values to Scheme from C (or vice versa) (mainly because C does not have multiple return values).

Keyboard interrupts that occur during a call to a C function are ignored until the function returns to Scheme (this is clearly a problem; we are working on a solution).

- `(import-lambda-definition` *name* `(`*formal* `...))` syntax
- `(import-lambda-definition` *name* `(`*formal* `...)` *c-name*`)` syntax

These macros simplify importing functions from C. They define *name* to be a function with the given formals that applies those formals to the corresponding C binding. *C-name*, if supplied, should be a string. These expand into

```
(define temp (lookup-imported-binding c-name))
(define name
  (lambda (formal ...)
    (external-apply temp formal ...)))
```

If *c-name* is not supplied, it is derived from *name* by converting all letters to lowercase and replacing '`-`' with '`_`'.

4 Adding external modules to the `Makefile`

Getting access to C bindings from Scheme requires that the C code be compiled an linked in with the Scheme 48 virtual machine and that the relevent shared-bindings be created. The Scheme 48 makefile has rules for compiling and linking external code and for specifying initialization functions that should be called on startup. There are three Makefile variables that control which external modules are included in the executable for the virutal machine (`scheme48vm`). `EXTERNAL_OBJECTS` lists the object files to be included in `scheme48vm`, `EXTERNAL_FLAGS` is a list of `ld` flags to be used when creating `scheme48vm`, and `EXTERNAL_INITIALIZERS` is a list of C procedures to be called on startup. The procedures listed in `EXTERNAL_INITIALIZERS` should take no arguments and have a return type of `void`. After changing the definitions of any of these variables you should do `make scheme48vm` to rebuild the virtual machine.

5 Dynamic Loading

External code can be loaded into a running Scheme 48 process and C object-file bindings can be dereferenced at runtime and their values called (although not all versions of Unix support all of this). The required Scheme functions are in the structure `dynamic-externals`.

- `(dynamic-load` *string*`)`

`Dynamic-load` loads the named file into the current process, raising an exception if the file cannot be found or if dynamic loading is not supported by the operating system. The file must have been compiled and linked appropriately. For Linux, the following commands compile `foo.c` into a file `foo.so` that can be loaded dynamically.

```
% gcc -c -o foo.o foo.c
% ld -shared -o foo.so foo.o
```

- `(get-external` *string*`)` → *external*
- `(external?` *x*`)` → *boolean*
- `(external-name` *external*`)` → *string*
- `(external-value` *external*`)` → *byte-vector*

These functions give access to values bound in the current process, and are used for retrieving values from dynamically-loaded files. `Get-external` returns an *external* object that contains the value of *name*, raising an exception if there is no such value in the current process. `External?` is the predicate for externals, and `external-name` and `external-value` return the name and value of an external. The value is returned as byte vector of length four (on 32-bit architectures). The value is that which was extant when `get-external` was called. The following two functions can be used to update the values of externals.

- `(lookup-external` *external*`)` → *boolean*
- `(lookup-all-externals)` → *boolean*

`Lookup-external` updates the value of *external* by looking its name in the current process, returning `#t` if it is bound and `#f` if it is not. `Lookup-all-externals` calls `lookup-external` on all extant externals, returning `#f` any are unbound.

- `(call-external` *external* arg_0 `...)` → *value*

An external whose value is a C procedure can be called using `call-external`. See Section 3 for more information.

In some versions of Unix retrieving a value from the current process may require a non-trivial amount of computation. We recommend that a dynamically-loaded file contain a single initialization procedure that creates shared bindings for the values exported by the file.

6 Compatibility

Scheme 48's old `external-call` function is still available in the structure `externals`, which now also includes `external-name` and `external-value`. The old `scheme48.h` file has been renamed `old-scheme48.h`.

7 Accessing Scheme data from C

The C header file `scheme48.h` provides access to Scheme 48 data structures (for compatibility, the old `scheme48.h` file is available as `old-scheme48.h`). The type `s48_value` is used for Scheme values. When the type of a value is known, such as the integer returned by `vector-length` or the boolean returned by `pair?`, the corresponding C procedure returns a C value of the appropriate type, and not a `s48_value`. Predicates return `1` for true and `0` for false.

7.1 Constants

The following macros denote Scheme constants:

`S48_FALSE` is `#f`.

`S48_TRUE` is `#t`.

`S48_NULL` is the empty list.

`S48_UNSPECIFIC` is a value used for functions which have no meaningful return value (in Scheme this value returned by the nullary procedure `unspecific` in the structure `util`).

`S48_EOF` is the end-of-file object (in Scheme this value is returned by the nullary procedure `eof-object` in the structure `i/o-internal`).

7.2 Converting values

The following functions convert values between Scheme and C representations. The 'extract' ones convert from Scheme to C and the 'enter's go the other way.

- `unsigned char s48_extract_char(s48_value)`
- `char *    s48_extract_string(s48_value)`
- `long      s48_extract_integer(s48_value)`

- `double    s48_extract_double(s48_value)`
- `s48_value s48_enter_char(unsigned char)`
- `s48_value s48_enter_string(char *)` (may GC)
- `s48_value s48_enter_integer(long)` (may GC)
- `s48_value s48_enter_double(double)` (may GC)

The value returned by `s48_extract_string` points to the actual storage used by the string; it is valid only until the next garbage collection; see Section 9.

`s48_enter_integer()` needs to allocate storage when its argument is too large to fit in a Scheme 48 fixnum. In cases where the number is known to fit within a fixnum (currently 30 bits including the sign), the following procedures can be used. These have the disadvantage of only having a limited range, but the advantage of never causing a garbage collection.

- `long      s48_extract_fixnum(s48_value)`
- `s48_value s48_enter_fixnum(long)`
- `long      S48_MAX_FIXNUM_VALUE`
- `long      S48_MIN_FIXNUM_VALUE`

An error is signalled if `s48_extract_fixnum`'s argument is not a fixnum or if the argument to `s48_enter_fixnum` is less than `S48_MIN_FIXNUM_VALUE` or greater than `S48_MAX_FIXNUM_VALUE` (-2^{29} and $2^{29}-1$ in the current system).

7.3 C versions of Scheme procedures

The following macros and procedures are C versions of Scheme procedures. The names were derived by replacing '`-`' with '`_`', '`?`' with '`p`', and dropping '`!`.

- `int       S48_EQ_P(s48_value)`
- `int       S48_CHAR_P(s48_value)`
- `int       S48_INTEGER_P(s48_value)`

- `int       S48_PAIR_P(s48_value)`
- `s48_value S48_CAR(s48_value)`
- `s48_value S48_CDR(s48_value)`
- `void      S48_SET_CAR(s48_value, s48_value)`
- `void      S48_SET_CDR(s48_value, s48_value)`
- `s48_value s48_cons(s48_value, s48_value)` (may GC)
- `long      s48_length(s48_value)`

- `int       S48_VECTOR_P(s48_value)`
- `long      S48_VECTOR_LENGTH(s48_value)`
- `s48_value S48_VECTOR_REF(s48_value, long)`
- `void      S48_VECTOR_SET(s48_value, long, s48_value)`
- `s48_value s48_make_vector(long, s48_value)` (may GC)

- `int       S48_STRING_P(s48_value)`
- `long      S48_STRING_LENGTH(s48_value)`
- `char      S48_STRING_REF(s48_value, long)`
- `void      S48_STRING_SET(s48_value, long, char)`

- `s48_value s48_make_string(long, char)` (may GC)

- `int       S48_SYMBOL_P(s48_value)`
- `s48_value s48_SYMBOL_TO_STRING(s48_value)`

- `int       S48_BYTE_VECTOR_P(s48_value)`
- `long      S48_BYTE_VECTOR_LENGTH(s48_value)`
- `char      S48_BYTE_VECTOR_REF(s48_value, long)`
- `void      S48_BYTE_VECTOR_SET(s48_value, long, int)`
- `s48_value s48_make_byte_vector(long, int)` (may GC)

8 Calling Scheme functions from C

External code that has been called from Scheme can call back to Scheme procedures using the following function.

- `scheme_value s48_call_scheme(s48_value proc, long nargs, ...)`

This calls the Scheme procedure `proc` on `nargs` arguments, which are passed as additional arguments to `s48_call_scheme`. There may be at most ten arguments. The value returned by the Scheme procedure is returned by the C procedure. Invoking any Scheme procedure may potentially cause a garbage collection.

There are some complications that occur when mixing calls from C to Scheme with continuations and threads. C only supports downward continuations (via `longjmp()`). Scheme continuations that capture a portion of the C stack have to follow the same restriction. For example, suppose Scheme procedure `s0` captures continuation `a` and then calls C procedure `c0`, which in turn calls Scheme procedure `s1`. Procedure `s1` can safely call the continuation `a`, because that is a downward use. When `a` is called Scheme 48 will remove the portion of the C stack used by the call to `c0`. On the other hand, if `s1` captures a continuation, that continuation cannot be used from `s0`, because by the time control returns to `s0` the C stack used by `c0` will no longer be valid. An attempt to invoke an upward continuation that is closed over a portion of the C stack will raise an exception.

In Scheme 48 threads are implemented using continuations, so the downward restriction applies to them as well. An attempt to return from Scheme to C at a time when the appropriate C frame is not on top of the C stack will cause the current thread to block until the frame is available. For example, suppose thread `t0` calls a C procedure which calls back to Scheme, at which point control switches to thread `t1`, which also calls C and then back to Scheme. At this point both `t0` and `t1` have active calls to C on the C stack, with `t1`'s C frame above `t0`'s. If thread `t0` attempts to return from Scheme to C it will block, as its frame is not accessable. Once `t1` has returned to C and from there to Scheme, `t0` will be able to resume. The return to Scheme is required because context switches can only occur while C code is running. `T0` will also be able to resume if `t1` uses a continuation to throw past its call to C.

9 Interacting with the Scheme Heap

Scheme 48 uses a copying, precise garbage collector. Any procedure that allocates objects within the Scheme 48 heap may trigger a garbage collection. Variables bound to values in the Scheme 48 heap need to be registered with the garbage collector so that the value will be retained and so that the variables will be updated if the garbage collector moves the object. The garbage collector has no facility for updating pointers to the interiors of objects, so such pointers, for example the ones returned by `EXTRACT_STRING`, will likely become invalid when a garbage collection occurs.

9.1 Registering Objects with the GC

A set of macros are used to manage the registration of local variables with the garbage collector.

- `S48_DECLARE_GC_PROTECT(`n`)`
- `void S48_GC_PROTECT_`n`(s48_value`$_1$`, ..., s48_value`$_n$`)`
- `void S48_GC_UNPROTECT()`

`S48_DECLARE_GC_PROTECT(`n`)`, where $1 \leq n \leq 9$, allocates storage for registering n variables. At most one use of `S48_DECLARE_GC_PROTECT` may occur in a block. `S48_GC_PROTECT_`n`(`v_1`, ..., `v_n`)` registers the n variables (l-values) with the garbage collector. It must be within scope of a `S48_DECLARE_GC_PROTECT(`n`)` and be before any code which can cause a GC. `S48_GC_UNPROTECT` removes the block's protected variables from the garbage collectors list. It must be called at the end of the block after any code which may cause a garbage collection. Omitting any of the three may cause serious and hard-to-debug problems. Notably, the garbage collector may relocate an object and invalidate `s48_value` variables which are not protected.

A `gc-protection-mismatch` exception is raised if, when a C procedure returns to Scheme, the calls to `S48_GC_PROTECT()` have not been matched by an equal number of calls to `S48_GC_UNPROTECT()`.

Global variables may also be registered with the garbage collector.

- `void S48_GC_PROTECT_GLOBAL(`*value*`)`

`S48_GC_PROTECT_GLOBAL` permanently registers the variable *value* (an l-value) with the garbage collector. There is no way to unregister the variable.

9.2 Keeping C data structures in the Scheme heap

C data structures can be kept in the Scheme heap by embedding them inside byte vectors. The following macros can be used to create and access embedded C objects.

- `s48_value S48_MAKE_VALUE(type)` (may GC)
- `type       S48_EXTRACT_VALUE(s48_value, type)`
- `type *     S48_EXTRACT_VALUE_POINTER(s48_value, type)`

- `void      S48_SET_VALUE(s48_value, type, value)`

`S48_MAKE_VALUE` makes a byte vector large enough to hold an object whose type is *type*. `S48_EXTRACT_VALUE` returns the contents of a byte vector cast to *type*, and `S48_EXTRACT_VALUE_POINTER` returns a pointer to the contents of the byte vector. The value returned by `S48_EXTRACT_VALUE_POINTER` is valid only until the next garbage collection(see Section 9).

`S48_SET_VALUE` stores `value` into the byte vector.

9.3 C code and heap images

Scheme 48 uses dumped heap images to restore a previous system state. The Scheme 48 heap is written into a file in a machine-independent and operating-system-independent format. The procedures described above may be used to create objects in the Scheme heap that contain information specific to the current machine, operating system, or process. A heap image containing such objects may not work correctly on when resumed.

To address this problem, a record type may be given a 'resumer' procedure. On startup, the resumer procedure for a type is applied to each record of that type in the image being restarted. This procedure can update the record in a manner appropriate to the machine, operating system, or process used to resume the image.

- `(define-record-resumer` *record-type procedure*`)`

`Define-record-resumer` defines *procedure*, which should accept one argument, to be the resumer for *record-type*. The order in which resumer procedures are called is not specified.

The *procedure* argument to `define-record-resumer` may be `#f`, in which case records of the given type are not written out in heap images. When writing a heap image any reference to such a record is replaced by the value of the record's first field, and an exception is raised after the image is written.

10 Using Scheme records in C code

External modules can create records and access their slots positionally.

- `s48_value S48_MAKE_RECORD(s48_value)` (may GC)
- `int       S48_RECORD_P(s48_value)`
- `s48_value S48_RECORD_TYPE(s48_value)`
- `s48_value S48_RECORD_REF(s48_value, long)`
- `void      S48_RECORD_SET(s48_value, long, s48_value)`

The argument to `S48_MAKE_RECORD` should be a shared binding whose value is a record type. In C the fields of Scheme records are only accessible via offsets, with the first field having offset zero, the second offset one, and so forth. If the order of the fields is changed in the Scheme definition of the record type the C code must be updated as well.

For example, given the following record-type definition

```
(define-record-type thing :thing
  (make-thing a b)
  thing?
  (a thing-a)
  (b thing-b))
```

the identifier `:thing` is bound to the record type and can be exported to C:

```
(define-exported-binding "thing-record-type" :thing)
```

`Thing` records can then be made in C:

```
static scheme_value thing_record_type_binding = SCHFALSE;

void initialize_things(void)
{
  S48_GC_PROTECT_GLOBAL(thing_record_type_binding);
  thing_record_type_binding =
     s48_get_imported_binding("thing-record-type");
}

scheme_value make_thing(scheme_value a, scheme_value b)
{
  s48_value thing;
  s48_DECLARE_GC_PROTECT(2);

  S48_GC_PROTECT_2(a, b);

  thing = s48_make_record(thing_record_type_binding);
  S48_RECORD_SET(thing, 0, a);
  S48_RECORD_SET(thing, 1, b);

  S48_GC_UNPROTECT();

  return thing;
}
```

Note that the variables `a` and `b` must be protected against the possibility of a garbage collection occuring during the call to `s48_make_record()`.

11 Raising exceptions from external code

The following macros explicitly raise certain errors, immediately returning to Scheme 48. Raising an exception performs all necessary clean-up actions to properly return to Scheme 48, including adjusting the stack of protected variables.

- `s48_raise_scheme_exception(int type, int nargs, ...)`

`s48_raise_scheme_exception` is the base procedure for raising exceptions. `type` is the type of exception, and should be one of the `S48_EXCEPTION_`... constants defined in `scheme48arch.h`. `nargs` is the number of additional values to be included in the exception; these follow the `nargs` argument and should all have type `s48_value`. `s48_raise_scheme_exception` never returns.

The following procedures are available for raising particular types of exceptions. Like `s48_raise_scheme_exception` these never return.

- `s48_raise_argument_type_error(scheme_value)`
- `s48_raise_argument_number_error(int nargs, int min, int max)`
- `s48_raise_index_range_error(long value, long min, long max)`
- `s48_raise_closed_channel_error()`
- `s48_raise_os_error(int errno)`
- `s48_raise_out_of_memory_error()`

An argument type error indicates that the given value is of the wrong type. An argument number error is raised when the number of arguments, `nargs`, should be, but isn't, between `min` and `max`, inclusive. Similarly, and index range error is raised when `value` is not between between `min` and `max`, inclusive.

The following macros raise argument type errors if their argument does not have the required type.

- `void S48_CHECK_SYMBOL(s48_value)`
- `void S48_CHECK_PAIR(s48_value)`
- `void S48_CHECK_STRING(s48_value)`
- `void S48_CHECK_INTEGER(s48_value)`
- `void S48_CHECK_CHANNEL(s48_value)`
- `void S48_CHECK_BYTE_VECTOR(s48_value)`
- `void S48_CHECK_RECORD(s48_value)`
- `void S48_CHECK_SHARED_BINDING(s48_value)`

12 Unsafe functions and macros

All of the C procedures and macros described above check that their arguments have the appropriate types and that indexes are in range. The following procedures and macros are identical to those described above, except that they do not perform type and range checks. They are provided for the purpose of writing more efficient code; their general use is not recommended.

- `char      S48_UNSAFE_EXTRACT_CHAR(s48_value)`
- `char *    S48_UNSAFE_EXTRACT_STRING(s48_value)`
- `long      S48_UNSAFE_EXTRACT_INTEGER(s48_value)`
- `long      S48_UNSAFE_EXTRACT_DOUBLE(s48_value)`

- `long      S48_UNSAFE_EXTRACT_FIXNUM(s48_value)`
- `s48_value S48_UNSAFE_ENTER_FIXNUM(long)`

- `s48_value S48_UNSAFE_CAR(s48_value)`

- `s48_value S48_UNSAFE_CDR(s48_value)`
- `void      S48_UNSAFE_SET_CAR(s48_value, s48_value)`
- `void      S48_UNSAFE_SET_CDR(s48_value, s48_value)`

- `long      S48_UNSAFE_VECTOR_LENGTH(s48_value)`
- `s48_value S48_UNSAFE_VECTOR_REF(s48_value, long)`
- `void      S48_UNSAFE_VECTOR_SET(s48_value, long, s48_value)`

- `long      S48_UNSAFE_STRING_LENGTH(s48_value)`
- `char      S48_UNSAFE_STRING_REF(s48_value, long)`
- `void      S48_UNSAFE_STRING_SET(s48_value, long, char)`

- `s48_value S48_UNSAFE_SYMBOL_TO_STRING(s48_value)`

- `long      S48_UNSAFE_BYTE_VECTOR_LENGTH(s48_value)`
- `char      S48_UNSAFE_BYTE_VECTOR_REF(s48_value, long)`
- `void      S48_UNSAFE_BYTE_VECTOR_SET(s48_value, long, int)`

- `s48_value S48_UNSAFE_SHARED_BINDING_REF(s48_value s_b)`
- `int       S48_UNSAFE_SHARED_BINDING_P(x)`
- `int       S48_UNSAFE_SHARED_BINDING_IS_IMPORT_P(s48_value s_b)`
- `s48_value S48_UNSAFE_SHARED_BINDING_NAME(s48_value s_b)`
- `void      S48_UNSAFE_SHARED_BINDING_SET(s48_value s_b, s48_value value)`

- `s48_value S48_UNSAFE_RECORD_TYPE(s48_value)`
- `s48_value S48_UNSAFE_RECORD_REF(s48_value, long)`
- `void      S48_UNSAFE_RECORD_SET(s48_value, long, s48_value)`

- `type      S48_UNSAFE_EXTRACT_VALUE(s48_value, type)`
- `type *    S48_UNSAFE_EXTRACT_VALUE_POINTER(s48_value, type)`
- `void      S48_UNSAFE_SET_VALUE(s48_value, type, value)`