extension.rdoc - -- *RDoc* -- created at: Mon Aug 7 16:45:54 JST 1995

This document explains how to make extension libraries for Ruby.

Basic Knowledge

In C, variables have types and data do not have types. In contrast, Ruby variables do not have a static type, and data themselves have types, so data will need to be converted between the languages.

Data in Ruby are represented by the C type `VALUE'. Each VALUE data has its data type.

To retrieve C data from a VALUE, you need to:

1. Identify the VALUE's data type
2. Convert the VALUE into C data

Converting to the wrong data type may cause serious problems.

Data Types

The Ruby interpreter has the following data types:

```
T_NIL       :: nil
T_OBJECT    :: ordinary object
T_CLASS     :: class
T_MODULE    :: module
T_FLOAT     :: floating point number
T_STRING    :: string
T_REGEXP    :: regular expression
T_ARRAY     :: array
T_HASH      :: associative array
T_STRUCT    :: (Ruby) structure
T_BIGNUM    :: multi precision integer
T_FIXNUM    :: Fixnum(31bit or 63bit integer)
T_COMPLEX   :: complex number
T_RATIONAL  :: rational number
T_FILE      :: IO
T_TRUE      :: true
T_FALSE     :: false
T_DATA      :: data
T_SYMBOL    :: symbol
```

In addition, there are several other types used internally:

```
T_ICLASS    :: included module
T_MATCH     :: MatchData object
T_UNDEF     :: undefined
T_NODE      :: syntax tree node
T_ZOMBIE    :: object awaiting finalization
```

Most of the types are represented by C structures.

Check Data Type of the VALUE

The macro TYPE() defined in ruby.h shows the data type of the VALUE. TYPE() returns the constant number T_XXXX described above. To handle data types, your code will look something like this:

```
switch (TYPE(obj)) {
  case T_FIXNUM:
    /* process Fixnum */
    break;
  case T_STRING:
    /* process String */
    break;
  case T_ARRAY:
    /* process Array */
    break;
  default:
    /* raise exception */
    rb_raise(rb_eTypeError, "not valid value");
    break;
}
```

There is the data type check function

```
void Check_Type(VALUE value, int type)
```

which raises an exception if the VALUE does not have the type specified.

There are also faster check macros for fixnums and nil.

```
FIXNUM_P(obj)
NIL_P(obj)
```

Convert VALUE into C Data

The data for type T_NIL, T_FALSE, T_TRUE are nil, false, true respectively. They are singletons for the data type. The equivalent C constants are: Qnil, Qfalse, Qtrue. Note that Qfalse is false in C also (i.e. 0), but not Qnil.

The T_FIXNUM data is a 31bit or 63bit length fixed integer. This size depends on the size of long: if long is 32bit then T_FIXNUM is 31bit, if long is 64bit then T_FIXNUM is 63bit. T_FIXNUM can be converted to a C integer by using the FIX2INT() macro or FIX2LONG(). Though you have to check that the data is really FIXNUM before using them, they are faster. FIX2LONG() never raises exceptions, but FIX2INT() raises RangeError if the result is bigger or smaller than the size of int. There are also NUM2INT() and NUM2LONG() which converts any Ruby numbers into C integers. These macros include a type check, so an exception will be raised if the conversion failed. NUM2DBL() can be used to retrieve the double float value in the same way.

You can use the macros StringValue() and StringValuePtr() to get a char* from a VALUE. StringValue(var) replaces var's value with the result of "var.to_str()". StringValuePtr(var) does the same replacement and returns the char* representation of var. These macros will skip the replacement if var is a String. Notice that the macros take only the lvalue as their argument, to change the value of var in place.

You can also use the macro named StringValueCStr(). This is just like StringValuePtr(), but always adds a NUL character at the end of the result. If the result contains a NUL character, this macro causes the ArgumentError exception. StringValuePtr() doesn't guarantee the existence of a NUL at the end of the result, and the result may contain NUL.

Other data types have corresponding C structures, e.g. struct RArray for T_ARRAY etc. The VALUE of the type which has the corresponding structure can be cast to retrieve the pointer to the struct. The casting macro will be of the form RXXXX for each data type; for instance, RARRAY(obj). See "ruby.h". However, we do not recommend to access RXXXX data directly because these data structures are complex. Use corresponding rb_xxx() functions to access the internal struct. For example, to access an entry of array, use rb_ary_entry(ary, offset) and rb_ary_store(ary, offset, obj).

There are some accessing macros for structure members, for example RSTRING_LEN(str)' to get the size of the Ruby String object. The allocated region can be accessed byRSTRING_PTR(str)'.

Notice: Do not change the value of the structure directly, unless you are responsible for the result. This ends up being the cause of interesting bugs.

Convert C Data into VALUE

To convert C data to Ruby values:

```
FIXNUM ::  left shift 1 bit, and turn on its least significant bit (LSB).
```

```
Other pointer values ::  cast to VALUE.
```

You can determine whether a VALUE is a pointer or not by checking its LSB.

Notice: Ruby does not allow arbitrary pointer values to be a VALUE. They should be pointers to the structures which Ruby knows about. The known structures are defined in <ruby.h>.

To convert C numbers to Ruby values, use these macros:

```
INT2FIX() :: for integers within 31bits.
INT2NUM() :: for arbitrary sized integers.
```

INT2NUM() converts an integer into a Bignum if it is out of the FIXNUM range, but is a bit slower.

Manipulating Ruby Data

As I already mentioned, it is not recommended to modify an object's internal structure. To manipulate objects, use the functions supplied by the Ruby interpreter. Some (not all) of the useful functions are listed below:

String Functions

```
rb_str_new(const char *ptr, long len)
```

Creates a new Ruby string.

```
rb_str_new2(const char *ptr)
rb_str_new_cstr(const char *ptr)
```

Creates a new Ruby string from a C string. This is equivalent to

```
rb_str_new(ptr, strlen(ptr))
```

```
rb_str_new_literal(const char *ptr)
```

Creates a new Ruby string from a C string literal.

```
rb_tainted_str_new(const char *ptr, long len)
```

Creates a new tainted Ruby string. Strings from external data sources should be tainted.

```
rb_tainted_str_new2(const char *ptr)
rb_tainted_str_new_cstr(const char *ptr)
```

Creates a new tainted Ruby string from a C string.

```
rb_sprintf(const char *format, ...)
rb_vsprintf(const char *format, va_list ap)
```

Creates a new Ruby string with printf(3) format.

Note: In the format string, "%"PRIsVALUE can be used for Object#to_s (or Object#inspect if '+' flag is set) output (and related argument must be a VALUE). Since it conflicts with "%i", for integers in format strings, use "%d".

```
rb_str_cat(VALUE str, const char *ptr, long len)
```

Appends len bytes of data from ptr to the Ruby string.

```
rb_str_cat2(VALUE str, const char* ptr)
rb_str_cat_cstr(VALUE str, const char* ptr)
```

Appends C string ptr to Ruby string str. This function is equivalent to

```
rb_str_cat(str, ptr, strlen(ptr)).
```

```
rb_str_catf(VALUE str, const char* format, ...)
rb_str_vcatf(VALUE str, const char* format, va_list ap)
```

Appends C string format and successive arguments to Ruby string str according to a printf-like format. These functions are equivalent to rb_str_cat2(str, rb_sprintf(format, ...)) and rb_str_cat2(str, rb_vsprintf(format, ap)), respectively.

```
rb_enc_str_new(const char *ptr, long len, rb_encoding *enc)
rb_enc_str_new_cstr(const char *ptr, rb_encoding *enc)
```

Creates a new Ruby string with the specified encoding.

```
rb_enc_str_new_literal(const char *ptr)
```

Creates a new Ruby string from a C string literal with the specified encoding.

```
rb_usascii_str_new(const char *ptr, long len)
rb_usascii_str_new_cstr(const char *ptr)
```

Creates a new Ruby string with encoding US-ASCII.

```
rb_usascii_str_new_literal(const char *ptr)
```

Creates a new Ruby string from a C string literal with encoding US-ASCII.

```
rb_utf8_str_new(const char *ptr, long len)
rb_utf8_str_new_cstr(const char *ptr)
```

Creates a new Ruby string with encoding UTF-8.

```
rb_utf8_str_new_literal(const char *ptr)
```

Creates a new Ruby string from a C string literal with encoding UTF-8.

```
rb_str_resize(VALUE str, long len)
```

Resizes a Ruby string to len bytes. If str is not modifiable, this function raises an exception. The length of str must

be set in advance. If len is less than the old length the content beyond len bytes is discarded, else if len is greater than the old length the content beyond the old length bytes will not be preserved but will be garbage. Note that RSTRING_PTR(str) may change by calling this function.

```
rb_str_set_len(VALUE str, long len)
```

Sets the length of a Ruby string. If str is not modifiable, this function raises an exception. This function preserves the content up to len bytes, regardless RSTRING_LEN(str). len must not exceed the capacity of str.

Array Functions

```
rb_ary_new()
```

Creates an array with no elements.

```
rb_ary_new2(long len)
rb_ary_new_capa(long len)
```

Creates an array with no elements, allocating internal buffer for len elements.

```
rb_ary_new3(long n, ...)
rb_ary_new_from_args(long n, ...)
```

Creates an n-element array from the arguments.

```
rb_ary_new4(long n, VALUE *elts)
rb_ary_new_from_values(long n, VALUE *elts)
```

Creates an n-element array from a C array.

```
rb_ary_to_ary(VALUE obj)
```

Converts the object into an array. Equivalent to Object#to_ary.

There are many functions to operate an array. They may dump core if other types are given.

```
rb_ary_aref(argc, VALUE *argv, VALUE ary)
```

Equivalent to Array#[].

```
rb_ary_entry(VALUE ary, long offset)
```

\ary[offset]

```
rb_ary_store(VALUE ary, long offset, VALUE obj)
```

\ary[offset] = obj

```
rb_ary_subseq(VALUE ary, long beg, long len)
```

ary[beg, len]

```
rb_ary_push(VALUE ary, VALUE val)
rb_ary_pop(VALUE ary)
rb_ary_shift(VALUE ary)
rb_ary_unshift(VALUE ary, VALUE val)
```

ary.push, ary.pop, ary.shift, ary.unshift

```
rb_ary_cat(VALUE ary, const VALUE *ptr, long len)
```

Appends len elements of objects from ptr to the array.

Extending Ruby with C

Adding New Features to Ruby

You can add new features (classes, methods, etc.) to the Ruby interpreter. Ruby provides APIs for defining the following things:

- Classes, Modules
- Methods, Singleton Methods
- Constants

Class and Module Definition

To define a class or module, use the functions below:

```
VALUE rb_define_class(const char *name, VALUE super)
VALUE rb_define_module(const char *name)
```

These functions return the newly created class or module. You may want to save this reference into a variable to use later.

To define nested classes or modules, use the functions below:

```
VALUE rb_define_class_under(VALUE outer, const char *name, VALUE super)
VALUE rb_define_module_under(VALUE outer, const char *name)
```

Method and Singleton Method Definition

To define methods or singleton methods, use these functions:

```
void rb_define_method(VALUE klass, const char *name,
                      VALUE (*func)(), int argc)

  void rb_define_singleton_method(VALUE object, const char *name,
                                  VALUE (*func)(), int argc)
```

The `argc' represents the number of the arguments to the C function, which must be less than 17. But I doubt you'll need that many.

If `argc' is negative, it specifies the calling sequence, not number of the arguments.

If argc is -1, the function will be called as:

```
VALUE func(int argc, VALUE *argv, VALUE obj)
```

where argc is the actual number of arguments, argv is the C array of the arguments, and obj is the receiver.

If argc is -2, the arguments are passed in a Ruby array. The function will be called like:

```
VALUE func(VALUE obj, VALUE args)
```

where obj is the receiver, and args is the Ruby array containing actual arguments.

There are some more functions to define methods. One takes an ID as the name of method to be defined. See also ID or Symbol below.

```
void rb_define_method_id(VALUE klass, ID name,
                         VALUE (*func)(ANYARGS), int argc)
```

There are two functions to define private/protected methods:

```
void rb_define_private_method(VALUE klass, const char *name,
                              VALUE (*func)(), int argc)
  void rb_define_protected_method(VALUE klass, const char *name,
                                  VALUE (*func)(), int argc)
```

At last, rb_define_module_function defines a module function, which are private AND singleton methods of the module. For example, sqrt is a module function defined in the Math module. It can be called in the following way:

```
Math.sqrt(4)
```

or

```
include Math
sqrt(4)
```

To define module functions, use:

```
void rb_define_module_function(VALUE module, const char *name,
                               VALUE (*func)(), int argc)
```

In addition, function-like methods, which are private methods defined in the Kernel module, can be defined using:

```
void rb_define_global_function(const char *name, VALUE (*func)(), int argc)
```

To define an alias for the method,

```
void rb_define_alias(VALUE module, const char* new, const char* old);
```

To define a reader/writer for an attribute,

```
void rb_define_attr(VALUE klass, const char *name, int read, int write)
```

To define and undefine the `allocate' class method,

```
void rb_define_alloc_func(VALUE klass, VALUE (*func)(VALUE klass));
void rb_undef_alloc_func(VALUE klass);
```

func has to take the klass as the argument and return a newly allocated instance. This instance should be as empty as possible, without any expensive (including external) resources.

If you are overriding an existing method of any ancestor of your class, you may rely on:

```
VALUE rb_call_super(int argc, const VALUE *argv)
```

To achieve the receiver of the current scope (if no other way is available), you can use:

```
VALUE rb_current_receiver(void)
```

Constant Definition

We have 2 functions to define constants:

```
void rb_define_const(VALUE klass, const char *name, VALUE val)
void rb_define_global_const(const char *name, VALUE val)
```

The former is to define a constant under specified class/module. The latter is to define a global constant.

```
void rb_define_const(VALUE klass, const char *name, VALUE val)
```

Defines a new constant under the class/module.

```
void rb_define_global_const(const char *name, VALUE val)
```

Defines a global constant. This is just the same as

```
rb_define_const(rb_cObject, name, val)
```

Ruby Constants That Can Be Accessed From C

As stated in section 1.3, the following Ruby constants can be referred from C.

```
Qtrue
Qfalse
```

Boolean values. Qfalse is false in C also (i.e. 0).

```
Qnil
```

Ruby nil in C scope.

Use Ruby Features from C

There are several ways to invoke Ruby's features from C code.

Evaluate Ruby Programs in a String

The easiest way to use Ruby's functionality from a C program is to evaluate the string as Ruby program. This function will do the job:

```
VALUE rb_eval_string(const char *str)
```

Evaluation is done under the current context, thus current local variables of the innermost method (which is defined by Ruby) can be accessed.

Note that the evaluation can raise an exception. There is a safer function:

```
VALUE rb_eval_string_protect(const char *str, int *state)
```

It returns nil when an error occurred. Moreover, *state is zero if str was successfully evaluated, or nonzero otherwise.

ID or Symbol

You can invoke methods directly, without parsing the string. First I need to explain about ID. ID is the integer number to represent Ruby's identifiers such as variable names. The Ruby data type corresponding to ID is

Symbol. It can be accessed from Ruby in the form:

```
:Identifier
```

or

```
:"any kind of string"
```

You can get the ID value from a string within C code by using

```
rb_intern(const char *name)
rb_intern_str(VALUE name)
```

You can retrieve ID from Ruby object (Symbol or String) given as an argument by using

```
rb_to_id(VALUE symbol)
rb_check_id(volatile VALUE *name)
rb_check_id_cstr(const char *name, long len, rb_encoding *enc)
```

These functions try to convert the argument to a String if it was not a Symbol nor a String. The second function stores the converted result into *name, and returns 0 if the string is not a known symbol. After this function returned a non-zero value, *name is always a Symbol or a String, otherwise it is a String if the result is 0. The third function takes NUL-terminated C string, not Ruby VALUE.

You can retrieve Symbol from Ruby object (Symbol or String) given as an argument by using

```
rb_to_symbol(VALUE name)
rb_check_symbol(volatile VALUE *namep)
rb_check_symbol_cstr(const char *ptr, long len, rb_encoding *enc)
```

These functions are similar to above functions except that these return a Symbol instead of an ID.

You can convert C ID to Ruby Symbol by using

```
VALUE ID2SYM(ID id)
```

and to convert Ruby Symbol object to ID, use

```
ID SYM2ID(VALUE symbol)
```

Invoke Ruby Method from C

To invoke methods directly, you can use the function below

```
VALUE rb_funcall(VALUE recv, ID mid, int argc, ...)
```

This function invokes a method on the recv, with the method name specified by the symbol mid.

Accessing the Variables and Constants

You can access class variables and instance variables using access functions. Also, global variables can be shared between both environments. There's no way to access Ruby's local variables.

The functions to access/modify instance variables are below:

```
VALUE rb_ivar_get(VALUE obj, ID id)
VALUE rb_ivar_set(VALUE obj, ID id, VALUE val)
```

id must be the symbol, which can be retrieved by rb_intern().

To access the constants of the class/module:

```
VALUE rb_const_get(VALUE obj, ID id)
```

See also Constant Definition above.

Information Sharing Between Ruby and C

Global Variables Shared Between C and Ruby

Information can be shared between the two environments using shared global variables. To define them, you can use functions listed below:

```
void rb_define_variable(const char *name, VALUE *var)
```

This function defines the variable which is shared by both environments. The value of the global variable pointed to by var can be accessed through Ruby's global variable named name.

You can define read-only (from Ruby, of course) variables using the function below.

```
void rb_define_readonly_variable(const char *name, VALUE *var)
```

You can define hooked variables. The accessor functions (getter and setter) are called on access to the hooked variables.

```
  void rb_define_hooked_variable(const char *name, VALUE *var,
       VALUE (*getter)(), void (*setter)())
```

If you need to supply either setter or getter, just supply 0 for the hook you don't need. If both hooks are 0, rb_define_hooked_variable() works just like rb_define_variable().

The prototypes of the getter and setter functions are as follows:

```
VALUE (*getter)(ID id, VALUE *var);
void (*setter)(VALUE val, ID id, VALUE *var);
```

Also you can define a Ruby global variable without a corresponding C variable. The value of the variable will be set/get only by hooks.

```
void rb_define_virtual_variable(const char *name,
     VALUE (*getter)(), void (*setter)())
```

The prototypes of the getter and setter functions are as follows:

```
VALUE (*getter)(ID id);
void (*setter)(VALUE val, ID id);
```

Encapsulate C Data into a Ruby Object

Sometimes you need to expose your struct in the C world as a Ruby object. In a situation like this, making use of the TypedData_XXX macro family, the pointer to the struct and the Ruby object can be mutually converted.

-- The old (non-Typed) Data_XXX macro family has been deprecated. In the future version of Ruby, it is possible old macros will not work. ++

C struct to Ruby object

You can convert sval, a pointer to your struct, into a Ruby object with the next macro.

```
TypedData_Wrap_Struct(klass, data_type, sval)
```

TypedData_Wrap_Struct() returns a created Ruby object as a VALUE.

The klass argument is the class for the object. data_type is a pointer to a const rb_data_type_t which describes how Ruby should manage the struct.

It is recommended that klass derives from a special class called Data (rb_cData) but not from Object or other ordinal classes. If it doesn't, you have to call rb_undef_alloc_func(klass).

rb_data_type_t is defined like this. Let's take a look at each member of the struct.

```
struct rb_data_type_struct {
  const char *wrap_struct_name;
  struct {
    void (*dmark)(void*);
    void (*dfree)(void*);
    size_t (*dsize)(const void *);
    void *reserved[2];
  } function;
```

```
  const rb_data_type_t *parent;
  void *data;
  VALUE flags;
};
```

wrap_struct_name is an identifier of this instance of the struct. It is basically used for collecting and emitting statistics. So the identifier must be unique in the process, but doesn't need to be valid as a C or Ruby identifier.

These dmark / dfree functions are invoked during GC execution. No object allocations are allowed during it, so do not allocate ruby objects inside them.

dmark is a function to mark Ruby objects referred from your struct. It must mark all references from your struct with rb_gc_mark or its family if your struct keeps such references.

-- Note that it is recommended to avoid such a reference. ++

dfree is a function to free the pointer allocation. If this is -1, the pointer will be just freed.

dsize calculates memory consumption in bytes by the struct. Its parameter is a pointer to your struct. You can pass 0 as dsize if it is hard to implement such a function. But it is still recommended to avoid 0.

You have to fill reserved and parent with 0.

You can fill "data" with an arbitrary value for your use. Ruby does nothing with the member.

flags is a bitwise-OR of the following flag values. Since they require deep understanding of garbage collector in Ruby, you can just set 0 to flags if you are not sure.

```
RUBY_TYPED_FREE_IMMEDIATELY
```

This flag makes the garbage collector immediately invoke dfree() during GC when it need to free your struct. You can specify this flag if the dfree never unlocks Ruby's internal lock (GVL).

If this flag is not set, Ruby defers invokation of dfree() and invokes dfree() at the same time as finalizers.

```
RUBY_TYPED_WB_PROTECTED
```

It shows that implementation of the object supports write barriers. If this flag is set, Ruby is better able to do garbage collection of the object.

When it is set, however, you are responsible for putting write barriers in all implementations of methods of that object as appropriate. Otherwise Ruby might crash while running.

More about write barriers can be found in "Generational GC" in Appendix D.

You can allocate and wrap the structure in one step.

```
TypedData_Make_Struct(klass, type, data_type, sval)
```

This macro returns an allocated Data object, wrapping the pointer to the structure, which is also allocated. This macro works like:

```
(sval = ZALLOC(type), TypedData_Wrap_Struct(klass, data_type, sval))
```

Arguments klass and data_type work like their counterparts in TypedData_Wrap_Struct(). A pointer to the allocated structure will be assigned to sval, which should be a pointer of the type specified.

Ruby object to C struct

To retrieve the C pointer from the Data object, use the macro Data_Get_Struct().

```
TypedData_Get_Struct(obj, type, &data_type, sval)
```

A pointer to the structure will be assigned to the variable sval.

See the example below for details.

Example - Creating the dbm Extension

OK, here's the example of making an extension library. This is the extension to access DBMs. The full source is included in the ext/ directory in the Ruby's source tree.

Make the Directory

```
$> mkdir ext/dbm
```

Make a directory for the extension library under ext directory.

Design the Library

You need to design the library features, before making it.

Write the C Code

You need to write C code for your extension library. If your library has only one source file, choosing LIBRARY.c'' as a file name is preferred. On the other hand, in case your library has multiple source files, avoid choosingLIBRARY.c'' for a file name. It may conflict with an intermediate file LIBRARY.o'' on some platforms. Note that some functions in mkmf library described below generate a fileconftest.c'' for checking with compilation. You shouldn't choose ``conftest.c'' as a name of a source file.

Ruby will execute the initializing function named Init_LIBRARY'' in the library. For example,Init_dbm()'' will be executed when loading the library.

Here's the example of an initializing function.

```
void
Init_dbm(void)
```

```
{
  /* define DBM class */
  VALUE cDBM = rb_define_class("DBM", rb_cObject);
  /* DBM includes Enumerable module */
  rb_include_module(cDBM, rb_mEnumerable);

  /* DBM has class method open(): arguments are received as C array */
  rb_define_singleton_method(cDBM, "open", fdbm_s_open, -1);

  /* DBM instance method close(): no args */
  rb_define_method(cDBM, "close", fdbm_close, 0);
  /* DBM instance method []: 1 argument */
  rb_define_method(cDBM, "[]", fdbm_fetch, 1);

  /* ... */

  /* ID for a instance variable to store DBM data */
  id_dbm = rb_intern("dbm");
}
```

The dbm extension wraps the dbm struct in the C environment using Data_Make_Struct.

```
struct dbmdata {
  int  di_size;
  DBM *di_dbm;
};

static const rb_data_type_t dbm_type = {
  "dbm",
  {0, free_dbm, memsize_dbm,}, 0, 0, RUBY_TYPED_FREE_IMMEDIATELY
};

obj = TypedData_Make_Struct(klass, struct dbmdata, &dbm_type, dbmp);
```

This code wraps the dbmdata structure into a Ruby object. We avoid wrapping DBM* directly, because we want to cache size information.

To retrieve the dbmdata structure from a Ruby object, we define the following macro:

```
#define GetDBM(obj, dbmp) do {\
TypedData_Get_Struct((obj), struct dbmdata, &dbm_type, (dbmp));\
  if ((dbmp) == 0) closed_dbm();\
  if ((dbmp)->di_dbm == 0) closed_dbm();\
} while (0)
```

This sort of complicated macro does the retrieving and close checking for the DBM.

There are three kinds of way to receive method arguments. First, methods with a fixed number of arguments receive arguments like this:

```
static VALUE
fdbm_delete(VALUE obj, VALUE keystr)
{
      /* ... */
}
```

The first argument of the C function is the self, the rest are the arguments to the method.

Second, methods with an arbitrary number of arguments receive arguments like this:

```
static VALUE
fdbm_s_open(int argc, VALUE *argv, VALUE klass)
{
  /* ... */
  if (rb_scan_args(argc, argv, "11", &file, &vmode) == 1) {
    mode = 0666;        /* default value */
  }
  /* ... */
}
```

The first argument is the number of method arguments, the second argument is the C array of the method arguments, and the third argument is the receiver of the method.

You can use the function rb_scan_args() to check and retrieve the arguments. The third argument is a string that specifies how to capture method arguments and assign them to the following VALUE references.

You can just check the argument number with rb_check_arity(), this is handy in the case you want to treat the arguments as a list.

The following is an example of a method that takes arguments by Ruby's array:

```
static VALUE
thread_initialize(VALUE thread, VALUE args)
{
    /* ... */
}
```

The first argument is the receiver, the second one is the Ruby array which contains the arguments to the method.

<Notice<: GC should know about global variables which refer to Ruby's objects, but are not exported to the Ruby world. You need to protect them by

```
void rb_global_variable(VALUE *var)
```

Prepare extconf.rb

If the file named extconf.rb exists, it will be executed to generate Makefile.

extconf.rb is the file for checking compilation conditions etc. You need to put

```
require 'mkmf'
```

at the top of the file. You can use the functions below to check various conditions.

```
  have_macro(macro[, headers[, opt]]): check whether macro is defined
  have_library(lib[, func[, headers[, opt]]]): check whether library containing
function exists
  find_library(lib[, func, *paths]): find library from paths
  have_func(func[, headers[, opt]): check whether function exists
  have_var(var[, headers[, opt]]): check whether variable exists
  have_header(header[, preheaders[, opt]]): check whether header file exists
  find_header(header, *paths): find header from paths
  have_framework(fw): check whether framework exists (for MacOS X)
  have_struct_member(type, member[, headers[, opt]]): check whether struct has
member
  have_type(type[, headers[, opt]]): check whether type exists
  find_type(type, opt, *headers): check whether type exists in headers
  have_const(const[, headers[, opt]]): check whether constant is defined
  check_sizeof(type[, headers[, opts]]): check size of type
  check_signedness(type[, headers[, opts]]): check signedness of type
  convertible_int(type[, headers[, opts]]): find convertible integer type
  find_executable(bin[, path]): find executable file path
  create_header(header): generate configured header
  create_makefile(target[, target_prefix]): generate Makefile
```

See MakeMakefile for full documentation of these functions.

The value of the variables below will affect the Makefile.

```
  $CFLAGS: included in CFLAGS make variable (such as -O)
  $CPPFLAGS: included in CPPFLAGS make variable (such as -I, -D)
  $LDFLAGS: included in LDFLAGS make variable (such as -L)
  $objs: list of object file names
```

Normally, the object files list is automatically generated by searching source files, but you must define them explicitly if any sources will be generated while building.

If a compilation condition is not fulfilled, you should not call ``create_makefile''. The Makefile will not be

generated, compilation will not be done.

Prepare Depend (Optional)

If the file named depend exists, Makefile will include that file to check dependencies. You can make this file by invoking

```
$> gcc -MM *.c > depend
```

It's harmless. Prepare it.

Generate Makefile

Try generating the Makefile by:

```
$> ruby extconf.rb
```

If the library should be installed under vendor_ruby directory instead of site_ruby directory, use --vendor option as follows.

```
$> ruby extconf.rb --vendor
```

You don't need this step if you put the extension library under the ext directory of the ruby source tree. In that case, compilation of the interpreter will do this step for you.

Run make

Type

```
$> make
```

to compile your extension. You don't need this step either if you have put the extension library under the ext directory of the ruby source tree.

Debug

You may need to rb_debug the extension. Extensions can be linked statically by adding the directory name in the ext/Setup file so that you can inspect the extension with the debugger.

Done! Now You Have the Extension Library

You can do anything you want with your library. The author of Ruby will not claim any restrictions on your code depending on the Ruby API. Feel free to use, modify, distribute or sell your program.

Appendix A. Ruby Source Files Overview

Ruby Language Core

```
class.c    :: classes and modules
error.c    :: exception classes and exception mechanism
gc.c       :: memory management
load.c     :: library loading
object.c   :: objects
variable.c :: variables and constants
```

Ruby Syntax Parser

```
parse.y       :: grammar definition
parse.c       :: automatically generated from parse.y
defs/keywords :: reserved keywords
lex.c         :: automatically generated from keywords
```

Ruby Evaluator (a.k.a. YARV)

```
  compile.c
  eval.c
  eval_error.c
  eval_jump.c
  eval_safe.c
  insns.def           : definition of VM instructions
  iseq.c              : implementation of VM::ISeq
  thread.c            : thread management and context switching
  thread_win32.c      : thread implementation
  thread_pthread.c    : ditto
  vm.c
  vm_dump.c
  vm_eval.c
  vm_exec.c
  vm_insnhelper.c
  vm_method.c

  defs/opt_insns_unif.def  : instruction unification
  defs/opt_operand.def     : definitions for optimization

    -> insn*.inc           : automatically generated
    -> opt*.inc            : automatically generated
    -> vm.inc              : automatically generated
```

Regular Expression Engine (Oniguruma)

```
  regex.c
  regcomp.c
```

```
  regenc.c
  regerror.c
  regexec.c
  regparse.c
  regsyntax.c
```

Utility Functions

```
debug.c    :: debug symbols for C debugger
dln.c      :: dynamic loading
st.c       :: general purpose hash table
strftime.c :: formatting times
util.c     :: misc utilities
```

Ruby Interpreter Implementation

```
  dmyext.c
  dmydln.c
  dmyencoding.c
  id.c
  inits.c
  main.c
  ruby.c
  version.c

  gem_prelude.rb
  prelude.rb
```

Class Library

```
array.c      :: Array
bignum.c     :: Bignum
compar.c     :: Comparable
complex.c    :: Complex
cont.c       :: Fiber, Continuation
dir.c        :: Dir
enum.c       :: Enumerable
enumerator.c :: Enumerator
file.c       :: File
hash.c       :: Hash
io.c         :: IO
marshal.c    :: Marshal
math.c       :: Math
numeric.c    :: Numeric, Integer, Fixnum, Float
```

```
pack.c       :: Array#pack, String#unpack
proc.c       :: Binding, Proc
process.c    :: Process
random.c     :: random number
range.c      :: Range
rational.c   :: Rational
re.c         :: Regexp, MatchData
signal.c     :: Signal
sprintf.c    :: String#sprintf
string.c     :: String
struct.c     :: Struct
time.c       :: Time

defs/known_errors.def :: Errno::* exception classes
-> known_errors.inc   :: automatically generated
```

Multilingualization

```
encoding.c  :: Encoding
transcode.c :: Encoding::Converter
enc/*.c     :: encoding classes
enc/trans/* :: codepoint mapping tables
```

goruby Interpreter Implementation

```
goruby.c
  golf_prelude.rb     : goruby specific libraries.
    -> golf_prelude.c : automatically generated
```

Appendix B. Ruby Extension API Reference

Types

```
VALUE
```

The type for the Ruby object. Actual structures are defined in ruby.h, such as struct RString, etc. To refer the values in structures, use casting macros like RSTRING(obj).

Variables and Constants

```
Qnil
```

nil object

```
Qtrue
```

true object (default true value)

```
Qfalse
```

false object

C Pointer Wrapping

```
Data_Wrap_Struct(VALUE klass, void (*mark)(), void (*free)(), void *sval)
```

Wrap a C pointer into a Ruby object. If object has references to other Ruby objects, they should be marked by using the mark function during the GC process. Otherwise, mark should be 0. When this object is no longer referred by anywhere, the pointer will be discarded by free function.

```
Data_Make_Struct(klass, type, mark, free, sval)
```

This macro allocates memory using malloc(), assigns it to the variable sval, and returns the DATA encapsulating the pointer to memory region.

```
Data_Get_Struct(data, type, sval)
```

This macro retrieves the pointer value from DATA, and assigns it to the variable sval.

Checking Data Types

```
RB_TYPE_P(value, type)
```

Is +value+ an internal type (T_NIL, T_FIXNUM, etc.)?

```
TYPE(value)
```

Internal type (T_NIL, T_FIXNUM, etc.)

```
FIXNUM_P(value)
```

Is +value+ a Fixnum?

```
NIL_P(value)
```

Is +value+ nil?

```
RB_INTEGER_TYPE_P(value)
```

Is +value+ an Integer?

```
RB_FLOAT_TYPE_P(value)
```

Is +value+ a Float?

```
void Check_Type(VALUE value, int type)
```

Ensures +value+ is of the given internal +type+ or raises a TypeError

```
SaveStringValue(value)
```

Checks that +value+ is a String and is not tainted

Data Type Conversion

```
FIX2INT(value), INT2FIX(i)
```

Fixnum <-> integer

```
FIX2LONG(value), LONG2FIX(l)
```

Fixnum <-> long

```
NUM2INT(value), INT2NUM(i)
```

Numeric <-> integer

```
NUM2UINT(value), UINT2NUM(ui)
```

Numeric <-> unsigned integer

```
NUM2LONG(value), LONG2NUM(l)
```

Numeric <-> long

```
NUM2ULONG(value), ULONG2NUM(ul)
```

Numeric <-> unsigned long

```
NUM2LL(value), LL2NUM(ll)
```

Numeric <-> long long

```
NUM2ULL(value), ULL2NUM(ull)
```

Numeric <-> unsigned long long

```
NUM2OFFT(value), OFFT2NUM(off)
```

Numeric <-> off_t

```
NUM2SIZET(value), SIZET2NUM(size)
```

Numeric <-> size_t

```
NUM2SSIZET(value), SSIZET2NUM(ssize)
```

Numeric <-> ssize_t

```
rb_integer_pack(value, words, numwords, wordsize, nails, flags),
rb_integer_unpack(words, numwords, wordsize, nails, flags)
```

Numeric <-> Arbitrary size integer buffer

```
NUM2DBL(value)
```

Numeric -> double

```
rb_float_new(f)
```

double -> Float

```
RSTRING_LEN(str)
```

String -> length of String data in bytes

```
RSTRING_PTR(str)
```

String -> pointer to String data Note that the result pointer may not be NUL-terminated

```
StringValue(value)
```

Object with #to_str -> String

```
StringValuePtr(value)
```

Object with #to_str -> pointer to String data

```
StringValueCStr(value)
```

Object with #to_str -> pointer to String data without NUL bytes It is guaranteed that the result data is NUL-terminated

```
rb_str_new2(s)
```

char * -> String

Defining Classes and Modules

```
VALUE rb_define_class(const char *name, VALUE super)
```

Defines a new Ruby class as a subclass of super.

```
VALUE rb_define_class_under(VALUE module, const char *name, VALUE super)
```

Creates a new Ruby class as a subclass of super, under the module's namespace.

```
VALUE rb_define_module(const char *name)
```

Defines a new Ruby module.

```
VALUE rb_define_module_under(VALUE module, const char *name)
```

Defines a new Ruby module under the module's namespace.

```
void rb_include_module(VALUE klass, VALUE module)
```

Includes module into class. If class already includes it, just ignored.

```
void rb_extend_object(VALUE object, VALUE module)
```

Extend the object with the module's attributes.

Defining Global Variables

```
void rb_define_variable(const char *name, VALUE *var)
```

Defines a global variable which is shared between C and Ruby. If name contains a character which is not allowed to be part of the symbol, it can't be seen from Ruby programs.

```
void rb_define_readonly_variable(const char *name, VALUE *var)
```

Defines a read-only global variable. Works just like rb_define_variable(), except the defined variable is read-only.

```
void rb_define_virtual_variable(const char *name, VALUE (*getter)(), void (*setter)
```

```
())
```

Defines a virtual variable, whose behavior is defined by a pair of C functions. The getter function is called when the variable is referenced. The setter function is called when the variable is set to a value. The prototype for getter/setter functions are:

```
VALUE getter(ID id)
void setter(VALUE val, ID id)
```

The getter function must return the value for the access.

```
void rb_define_hooked_variable(const char *name, VALUE *var, VALUE (*getter)(), void
(*setter)())
```

Defines hooked variable. It's a virtual variable with a C variable. The getter is called as

```
VALUE getter(ID id, VALUE *var)
```

returning a new value. The setter is called as

```
void setter(VALUE val, ID id, VALUE *var)
```

```
void rb_global_variable(VALUE *var)
```

GC requires C global variables which hold Ruby values to be marked. rb_global_variable tells GC to protect these variables.

Method Definition

```
rb_define_method(VALUE klass, const char *name, VALUE (*func)(), int argc)
```

Defines a method for the class. func is the function pointer. argc is the number of arguments. if argc is -1, the function will receive 3 arguments: argc, argv, and self. if argc is -2, the function will receive 2 arguments, self and args, where args is a Ruby array of the method arguments.

```
rb_define_private_method(VALUE klass, const char *name, VALUE (*func)(), int argc)
```

Defines a private method for the class. Arguments are same as rb_define_method().

```
rb_define_singleton_method(VALUE klass, const char *name, VALUE (*func)(), int argc)
```

Defines a singleton method. Arguments are same as rb_define_method().

```
rb_check_arity(int argc, int min, int max)
```

Check the number of arguments, argc is in the range of min..max. If max is UNLIMITED_ARGUMENTS, upper bound is not checked. If argc is out of bounds, an ArgumentError will be raised.

```
rb_scan_args(int argc, VALUE *argv, const char *fmt, ...)
```

Retrieve argument from argc and argv to given VALUE references according to the format string. The format can be described in ABNF as follows:

```
    scan-arg-spec  := param-arg-spec [option-hash-arg-spec] [block-arg-spec]

    param-arg-spec := pre-arg-spec [post-arg-spec] / post-arg-spec /
                      pre-opt-post-arg-spec
    pre-arg-spec   := num-of-leading-mandatory-args [num-of-optional-args]
    post-arg-spec  := sym-for-variable-length-args
                      [num-of-trailing-mandatory-args]
    pre-opt-post-arg-spec := num-of-leading-mandatory-args num-of-optional-args
                             num-of-trailing-mandatory-args
    option-hash-arg-spec := sym-for-option-hash-arg
    block-arg-spec := sym-for-block-arg

    num-of-leading-mandatory-args  := DIGIT ; The number of leading
                                            ; mandatory arguments
    num-of-optional-args           := DIGIT ; The number of optional
                                            ; arguments
    sym-for-variable-length-args   := "*"   ; Indicates that variable
                                            ; length arguments are
                                            ; captured as a ruby array
    num-of-trailing-mandatory-args := DIGIT ; The number of trailing
                                            ; mandatory arguments
    sym-for-option-hash-arg        := ":"   ; Indicates that an option
                                            ; hash is captured if the last
                                            ; argument is a hash or can be
                                            ; converted to a hash with
                                            ; #to_hash.  When the last
                                            ; argument is nil, it is
                                            ; captured if it is not
                                            ; ambiguous to take it as
                                            ; empty option hash; i.e. '*'
                                            ; is not specified and
                                            ; arguments are given more
                                            ; than sufficient.
    sym-for-block-arg              := "&"   ; Indicates that an iterator
                                            ; block should be captured if
                                            ; given
```

For example, "12" means that the method requires at least one argument, and at most receives three (1+2) arguments. So, the format string must be followed by three variable references, which are to be assigned to captured arguments. For omitted arguments, variables are set to Qnil. NULL can be put in place of a variable reference, which means the corresponding captured argument(s) should be just dropped.

The number of given arguments, excluding an option hash or iterator block, is returned.

```
int rb_get_kwargs(VALUE keyword_hash, const ID *table, int required, int optional,
VALUE *values)
```

Retrieves argument VALUEs bound to keywords, which directed by +table+ into +values+, deleting retrieved entries from +keyword_hash+ along the way. First +required+ number of IDs referred by +table+ are mandatory, and succeeding +optional+ (- +optional+ - 1 if +optional+ is negative) number of IDs are optional. If a mandatory key is not contained in +keyword_hash+, raises "missing keyword" +ArgumentError+. If an optional key is not present in +keyword_hash+, the corresponding element in +values+ is not changed. If +optional+ is negative, rest of +keyword_hash+ are stored in the next to optional +values+ as a new Hash, otherwise raises "unknown keyword" +ArgumentError+.

Be warned, handling keyword arguments in the C API is less efficient than handling them in Ruby. Consider using a Ruby wrapper method around a non-keyword C function. ref: https://bugs.ruby-lang.org/issues/11339

```
VALUE rb_extract_keywords(VALUE *original_hash)
```

Extracts pairs whose key is a symbol into a new hash from a hash object referred by +original_hash+. If the original hash contains non-symbol keys, then they are copied to another hash and the new hash is stored through +original_hash+, else 0 is stored.

Invoking Ruby method

```
VALUE rb_funcall(VALUE recv, ID mid, int narg, ...)
```

Invokes a method. To retrieve mid from a method name, use rb_intern(). Able to call even private/protected methods.

```
VALUE rb_funcall2(VALUE recv, ID mid, int argc, VALUE *argv)
VALUE rb_funcallv(VALUE recv, ID mid, int argc, VALUE *argv)
```

Invokes a method, passing arguments as an array of values. Able to call even private/protected methods.

```
VALUE rb_funcallv_public(VALUE recv, ID mid, int argc, VALUE *argv)
```

Invokes a method, passing arguments as an array of values. Able to call only public methods.

```
VALUE rb_eval_string(const char *str)
```

Compiles and executes the string as a Ruby program.

```
ID rb_intern(const char *name)
```

Returns ID corresponding to the name.

```
char *rb_id2name(ID id)
```

Returns the name corresponding ID.

```
char *rb_class2name(VALUE klass)
```

Returns the name of the class.

```
int rb_respond_to(VALUE obj, ID id)
```

Returns true if the object responds to the message specified by id.

Instance Variables

```
VALUE rb_iv_get(VALUE obj, const char *name)
```

Retrieve the value of the instance variable. If the name is not prefixed by `@', that variable shall be inaccessible from Ruby.

```
VALUE rb_iv_set(VALUE obj, const char *name, VALUE val)
```

Sets the value of the instance variable.

Control Structure

```
VALUE rb_block_call(VALUE recv, ID mid, int argc, VALUE * argv, VALUE (*func)
(ANYARGS), VALUE data2)
```

Calls a method on the recv, with the method name specified by the symbol mid, with argc arguments in argv, supplying func as the block. When func is called as the block, it will receive the value from yield as the first argument, and data2 as the second argument. When yielded with multiple values (in C, rb_yield_values(), rb_yield_values2() and rb_yield_splat()), data2 is packed as an Array, whereas yielded values can be gotten via argc/argv of the third/fourth arguments.

```
\[OBSOLETE] VALUE rb_iterate(VALUE (*func1)(), VALUE arg1, VALUE (*func2)(), VALUE
arg2)
```

Calls the function func1, supplying func2 as the block. func1 will be called with the argument arg1. func2 receives the value from yield as the first argument, arg2 as the second argument.

When rb_iterate is used in 1.9, func1 has to call some Ruby-level method. This function is obsolete since 1.9; use rb_block_call instead.

```
VALUE rb_yield(VALUE val)
```

Evaluates the block with value val.

```
VALUE rb_rescue(VALUE (*func1)(ANYARGS), VALUE arg1, VALUE (*func2)(ANYARGS), VALUE
arg2)
```

Calls the function func1, with arg1 as the argument. If an exception occurs during func1, it calls func2 with arg2 as the first argument and the exception object as the second argument. The return value of rb_rescue() is the return value from func1 if no exception occurs, from func2 otherwise.

```
VALUE rb_ensure(VALUE (*func1)(ANYARGS), VALUE arg1, VALUE (*func2)(ANYARGS), VALUE
arg2)
```

Calls the function func1 with arg1 as the argument, then calls func2 with arg2 if execution terminated. The return value from rb_ensure() is that of func1 when no exception occurred.

```
VALUE rb_protect(VALUE (*func) (VALUE), VALUE arg, int *state)
```

Calls the function func with arg as the argument. If no exception occurred during func, it returns the result of func and *state is zero. Otherwise, it returns Qnil and sets *state to nonzero. If state is NULL, it is not set in both cases. You have to clear the error info with rb_set_errinfo(Qnil) when ignoring the caught exception.

```
void rb_jump_tag(int state)
```

Continues the exception caught by rb_protect() and rb_eval_string_protect(). state must be the returned value from those functions. This function never return to the caller.

```
void rb_iter_break()
```

Exits from the current innermost block. This function never return to the caller.

```
void rb_iter_break_value(VALUE value)
```

Exits from the current innermost block with the value. The block will return the given argument value. This function never return to the caller.

Exceptions and Errors

```
void rb_warn(const char *fmt, ...)
```

Prints a warning message according to a printf-like format.

```
void rb_warning(const char *fmt, ...)
```

Prints a warning message according to a printf-like format, if $VERBOSE is true.

```
void rb_raise(rb_eRuntimeError, const char *fmt, ...)
```

Raises RuntimeError. The fmt is a format string just like printf().

```
void rb_raise(VALUE exception, const char *fmt, ...)
```

Raises a class exception. The fmt is a format string just like printf().

```
void rb_fatal(const char *fmt, ...) ::
```

Raises a fatal error, terminates the interpreter. No exception handling will be done for fatal errors, but ensure blocks will be executed.

```
void rb_bug(const char *fmt, ...)
```

Terminates the interpreter immediately. This function should be called under the situation caused by the bug in the interpreter. No exception handling nor ensure execution will be done.

Note: In the format string, "%"PRIsVALUE can be used for Object#to_s (or Object#inspect if '+' flag is set) output (and related argument must be a VALUE). Since it conflicts with "%i", for integers in format strings, use "%d".

Initialize and Start the Interpreter

The embedding API functions are below (not needed for extension libraries):

```
void ruby_init()
```

Initializes the interpreter.

```
void *ruby_options(int argc, char **argv)
```

Process command line arguments for the interpreter. And compiles the Ruby source to execute. It returns an opaque pointer to the compiled source or an internal special value.

```
int ruby_run_node(void *n)
```

Runs the given compiled source and exits this process. It returns EXIT_SUCCESS if successfully runs the source. Otherwise, it returns other value.

```
void ruby_script(char *name)
```

Specifies the name of the script ($0).

Hooks for the Interpreter Events

```
void rb_add_event_hook(rb_event_hook_func_t func, rb_event_flag_t events, VALUE
data)
```

Adds a hook function for the specified interpreter events. events should be OR'ed value of:

```
RUBY_EVENT_LINE
RUBY_EVENT_CLASS
RUBY_EVENT_END
RUBY_EVENT_CALL
RUBY_EVENT_RETURN
RUBY_EVENT_C_CALL
RUBY_EVENT_C_RETURN
RUBY_EVENT_RAISE
RUBY_EVENT_ALL
```

The definition of rb_event_hook_func_t is below:

```
typedef void (*rb_event_hook_func_t)(rb_event_t event, VALUE data,
                                     VALUE self, ID id, VALUE klass)
```

The third argument `data' to rb_add_event_hook() is passed to the hook function as the second argument, which was the pointer to the current NODE in 1.8. See RB_EVENT_HOOKS_HAVE_CALLBACK_DATA below.

```
int rb_remove_event_hook(rb_event_hook_func_t func)
```

Removes the specified hook function.

Memory usage

```
void rb_gc_adjust_memory_usage(ssize_t diff)
```

Adjusts the amount of registered external memory. You can tell GC how much memory is used by an external library by this function. Calling this function with positive diff means the memory usage is increased; new memory block is allocated or a block is reallocated as larger size. Calling this function with negative diff means the memory usage is decreased; a memory block is freed or a block is reallocated as smaller size. This function may trigger the GC.

Macros for Compatibility

Some macros to check API compatibilities are available by default.

```
NORETURN_STYLE_NEW
```

Means that NORETURN macro is functional style instead of prefix.

```
HAVE_RB_DEFINE_ALLOC_FUNC
```

Means that function rb_define_alloc_func() is provided, that means the allocation framework is used. This is same as the result of have_func("rb_define_alloc_func", "ruby.h").

```
HAVE_RB_REG_NEW_STR
```

Means that function rb_reg_new_str() is provided, that creates Regexp object from String object. This is same as the result of have_func("rb_reg_new_str", "ruby.h").

```
HAVE_RB_IO_T
```

Means that type rb_io_t is provided.

```
USE_SYMBOL_AS_METHOD_NAME
```

Means that Symbols will be returned as method names, e.g., Module#methods, #singleton_methods and so on.

```
HAVE_RUBY_*_H
```

Defined in ruby.h and means corresponding header is available. For instance, when HAVE_RUBY_ST_H is defined you should use ruby/st.h not mere st.h.

```
RB_EVENT_HOOKS_HAVE_CALLBACK_DATA
```

Means that rb_add_event_hook() takes the third argument `data', to be passed to the given event hook function.

Appendix C. Functions available for use in extconf.rb

See documentation for {mkmf}[rdoc-ref:MakeMakefile].

Appendix D. Generational GC

Ruby 2.1 introduced a generational garbage collector (called RGenGC). RGenGC (mostly) keeps compatibility.

Generally, the use of the technique called write barriers is required in extension libraries for generational GC (http://en.wikipedia.org/wiki/Garbage_collection_%28computer_science%29). RGenGC works fine without write barriers in extension libraries.

If your library adheres to the following tips, performance can be further improved. Especially, the "Don't touch pointers directly" section is important.

Incompatibility

You can't write RBASIC(obj)->klass field directly because it is const value now.

Basically you should not write this field because MRI expects it to be an immutable field, but if you want to do it in your extension you can use the following functions:

```
VALUE rb_obj_hide(VALUE obj)
```

Clear RBasic::klass field. The object will be an internal object. ObjectSpace::each_object can't find this object.

```
VALUE rb_obj_reveal(VALUE obj, VALUE klass)
```

Reset RBasic::klass to be klass. We expect the `klass' is hidden class by rb_obj_hide().

Write barriers

RGenGC doesn't require write barriers to support generational GC. However, caring about write barrier can improve the performance of RGenGC. Please check the following tips.

Don't touch pointers directly

In MRI (include/ruby/ruby.h), some macros to acquire pointers to the internal data structures are supported such as RARRAY_PTR(), RSTRUCT_PTR() and so on.

DO NOT USE THESE MACROS and instead use the corresponding C-APIs such as rb_ary_aref(), rb_ary_store() and so on.

Consider whether to insert write barriers

You don't need to care about write barriers if you only use built-in types.

If you support T_DATA objects, you may consider using write barriers.

Inserting write barriers into T_DATA objects only works with the following type objects: (a) long-lived objects, (b) when a huge number of objects are generated and (c) container-type objects that have references to other objects. If your extension provides such a type of T_DATA objects, consider inserting write barriers.

(a): short-lived objects don't become old generation objects. (b): only a few oldgen objects don't have performance impact. (c): only a few references don't have performance impact.

Inserting write barriers is a very difficult hack, it is easy to introduce critical bugs. And inserting write barriers has several areas of overhead. Basically we don't recommend you insert write barriers. Please carefully consider the risks.

Combine with built-in types

Please consider utilizing built-in types. Most built-in types support write barrier, so you can use them to avoid manually inserting write barriers.

For example, if your T_DATA has references to other objects, then you can move these references to Array. A

T_DATA object only has a reference to an array object. Or you can also use a Struct object to gather a T_DATA object (without any references) and an that Array contains references.

With use of such techniques, you don't need to insert write barriers anymore.

Insert write barriers

[AGAIN] Inserting write barriers is a very difficult hack, and it is easy to introduce critical bugs. And inserting write barriers has several areas of overhead. Basically we don't recommend you insert write barriers. Please carefully consider the risks.

Before inserting write barriers, you need to know about RGenGC algorithm (gc.c will help you). Macros and functions to insert write barriers are available in include/ruby/ruby.h. An example is available in iseq.c.

For a complete guide for RGenGC and write barriers, please refer to https://bugs.ruby-lang.org/projects/ruby-trunk/wiki/RGenGC.

Appendix E. RB_GC_GUARD to protect from premature GC

C Ruby currently uses conservative garbage collection, thus VALUE variables must remain visible on the stack or registers to ensure any associated data remains usable. Optimizing C compilers are not designed with conservative garbage collection in mind, so they may optimize away the original VALUE even if the code depends on data associated with that VALUE.

The following example illustrates the use of RB_GC_GUARD to ensure the contents of sptr remain valid while the second invocation of rb_str_new_cstr is running.

```
VALUE s, w;
const char *sptr;

s = rb_str_new_cstr("hello world!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");
sptr = RSTRING_PTR(s);
w = rb_str_new_cstr(sptr + 6); /* Possible GC invocation */

RB_GC_GUARD(s); /* ensure s (and thus sptr) do not get GC-ed */
```

In the above example, RB_GC_GUARD must be placed *after* the last use of sptr. Placing RB_GC_GUARD before dereferencing sptr would be of no use. RB_GC_GUARD is only effective on the VALUE data type, not converted C data types.

RB_GC_GUARD would not be necessary at all in the above example if non-inlined function calls are made on the s' VALUE after sptr is dereferenced. Thus, in the above example, calling any un-inlined function ons' such as:

```
rb_str_modify(s);
```

Will ensure `s' stays on the stack or register to prevent a GC invocation from prematurely freeing it.

Using the RB_GC_GUARD macro is preferable to using the "volatile" keyword in C. RB_GC_GUARD has the

following advantages:

1. the intent of the macro use is clear
2. RB_GC_GUARD only affects its call site, "volatile" generates some extra code every time the variable is used, hurting optimization.
3. "volatile" implementations may be buggy/inconsistent in some compilers and architectures. RB_GC_GUARD is customizable for broken systems/compilers without negatively affecting other systems.

```
:enddoc: Local variables:
:enddoc: fill-column: 70
:enddoc: end:
```