

C-Extensions – README.EXT

Übersetzte und ergänzte Fassung

Aus dem Englischen von Quintus Quintilianus Curiosus <sutniuq@@gmx@net>

Original von Yukihiro »Matz« Matsumoto

Dieses Dokument erklärt, wie man C-Extensions für die Programmiersprache Ruby erstellt. Wo immer ich Anmerkungen gemacht habe, stehen diese in eckigen Klammern [und], mit einem abschließenden »Anm. d. Übers.«.

Die behandelte Ruby-Version ist 1.9.1-p378.

Inhalt

S.1.....	Inhalt
S.3.....	1. Basiswissen
S.3.....	1.1 Datentypen
S.4.....	1.2 Den Datentyp eines VALUEs überprüfen
S.5.....	1.3 Einen VALUE in C-Daten konvertieren
S.6.....	1.4 Konvertieren von C-Daten in VALUEs
S.6.....	Verändern von Ruby-Objekten
S.6.....	String-Funktionen
S.8.....	Array-Funktionen
S.9.....	2. Ruby mit C erweitern
S.9.....	2.1 Neue Features zu Ruby hinzufügen
S.9.....	2.1.1 Klassen/Modul-Definition
S.9.....	2.1.2 Methoden/Singleton-Methoden-Definition
S.11.....	2.1.3 Konstanten-Definition
S.11.....	2.2 Ruby-Features von C aus nutzen
S.11.....	2.2.1 Evaluieren von Ruby-Programmen in Form von Strings
S.12.....	2.2.2 ID oder Symbol
S.12.....	2.2.3 Ruby-Methoden von C aus aufrufen
S.13.....	2.2.4 Abrufen von Variablen und Konstanten
S.14.....	3. Teilen von Informationen zwischen Ruby und C
S.14.....	3.1 Ruby-Konstanten, die von C aus abgerufen werden können
S.14.....	3.2 Globale Variablen zwischen Ruby und C
S.15.....	3.3 C-Daten in ein Ruby-Objekt einbinden
S.17.....	4. Beispiel – Erstellen einer DBM-Extension
S.23.....	Anhang A. Übersicht der Ruby-Quelldateien
S.27.....	Anhang B. Referenz zum Ruby-Extension-API
S.27.....	Typen
S.27.....	Wrapping von C-Pointern

S.27.....	Überprüfen von Datentypen
S.28.....	Konvertierung von Datentypen
S.28.....	Klassen/Modul-Definition
S.29.....	Definieren von globalen Variablen
S.30.....	Konstantendefinition
S.30.....	Methoden-Definition
S.31.....	Aufruf von Ruby-Methoden
S.31.....	Instanzvariablen
S.31.....	Kontrollstrukturen
S.32.....	Fehler und Exceptions
S.33.....	Initialisierung und Start des Interpreters
S.34.....	Hooks für Events im Interpreter
S.35.....	Anhang C. In extconf.rb verfügbare Methoden

1. Basiswissen

In C haben Variablen Typen und Daten nicht. Im Gegensatz dazu haben Ruby-Variablen keinen festen Typ, aber die Daten selber haben Typen, weshalb die Daten zwischen den beiden Programmiersprachen konvertiert werden müssen.

Ruby-Daten werden in C durch den Typ »VALUE« beschrieben. Jedes VALUE-Objekt besitzt einen Datentyp [der der Ruby-Klasse entspricht, Anm. d. Übers.].

Um die C-Daten eines VALUEs zu erhalten, musst du:

1. Den Datentyp des VALUEs herausfinden
2. Den VALUE in C-Daten konvertieren

Eine Konvertierung zum falschen C-Typ kann schwerwiegende Folgen haben.

1.1 Datentypen

Der Ruby-Interpreter kennt die folgenden Datentypen:

Datentyp (C-Makro)	Beschreibung
T_NIL	nil
T_OBJECT	Gewöhnliches Object
T_CLASS	Klasse
T_MODULE	Modul
T_FLOAT	Gleitkommazahl, Float
T_STRING	String
T_REGEXP	Regulärer Ausdruck
T_ARRAY	Array
T_HASH	Assoziatives Array, Hash
T_STRUCT	(Ruby)-Struktur
T_BIGNUM	Multi-Precision-Integer, große Zahlen
T_FIXNUM	Fixnum (31- oder 63-Bit-Integer), kleine Zahlen
T_COMPLEX	Komplexe Zahl
T_RATIONAL	Rational-Objekt, Bruchzahlen
T_FILE	IO-Stream
T_TRUE	true

T_FALSE	false
T_DATA	Data
T_SYMBOL	Symbol

Außerdem werden einige weitere Typen intern benutzt:

- T_ICLASS
- T_MATCH
- T_UNDEF
- T_NODE
- T_ZOMBIE

Die meisten dieser Typen werden durch C-Structs wiedergegeben.

1.2 Den Datentyp eines VALUEs überprüfen

Das Makro `TYPE()`, das in der Datei `»ruby.h«` definiert ist, ermöglicht die Überprüfung des Datentyps eines VALUEs. `TYPE()` gibt eines der obigen `T_XXXX`-Makros (es handelt sich hierbei ausschließlich um Zahlen) zurück. Um verschiedene Datentypen zu behandeln, wird dein Code vermutlich so aussehen:

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

Daneben gibt es noch die `»Check_Type«`-Funktion

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

, die eine Exception [der Klasse `»TypeError«`, Anm. d. Übers.] wirft, wenn der VALUE nicht den angegebenen Typ besitzt.

Für Fixnum und nil gibt es schnellere Makros:

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

1.3 Einen VALUE in C-Daten konvertieren

Die Daten für die Datentypen `T_NIL`, `T_FALSE` und `T_TRUE` sind `nil`, `false` sowie `true`. Hierbei handelt es sich um Singletons.

Die Daten des `T_FIXNUM`-Datentyps sind 31-Bit-Integers (63 Bit Länge bei einigen Computern), die man mithilfe der `FIX2INT()`- und `FIX2LONG()`-Makros in einen C-Integer konvertieren kann. Obwohl du vor deren Gebrauch prüfen musst, ob die Daten tatsächlich ein Fixnum sind, sind sie schneller. `FIX2LONG()` wirft niemals Exceptions, aber `FIX2INT()` wirft einen `RangeError` wenn das Ergebnis kleiner oder größer als der Datenbereich von `int` ist.

Außerdem gibt es noch `NUM2INT()` sowie `NUM2LONG()`, die jede Ruby-Zahl in einen C-Integer konvertieren. Diese Makros enthalten zudem eine Typüberprüfung, weshalb sie eine Exception werfen, wenn die Konvertierung fehlschlägt. Um `double floats` zu erhalten, kann man `NUM2DBL()` verwenden.

Ab einschließlich Version 1.7 sollten die neuen `StringValue()` und `StringValuePtr()`-Makros verwendet werden, um einen `char*` aus einem VALUE zu erhalten.

`StringValue(var)` ersetzt den Wert von `var` mit dem Ergebnis von `»var.to_str()«`.

`StringValuePtr(var)` macht das Gleiche, gibt jedoch die `char*`-Repräsentation von `var` zurück. Beide Makros werden den Ersetzungsschritt auslassen, falls `var` bereits ein String ist. Ebenfalls wichtig ist, dass diese Makros nur Lvalues als Argumente annehmen.

Du kannst auch ein Makro namens `StringValueCStr()` benutzen. Prinzipiell ist das das Gleiche wie `StringValuePtr()`, aber es fügt immer noch ein extra NUL-Zeichen an das Ende an. Wenn das Ergebnis ein NUL-Zeichen enthält, wirft dieses Makro einen `ArgumentError`. `StringValuePtr()` garantiert das abschließende NUL nicht und das Ergebnis kann ein NUL-Zeichen am Ende haben.

In Version 1.6 oder früher war es üblich, dasselbe mithilfe von `STR2CSTR()` zu erreichen, aber seit 1.7 ist `STR2CSTR` *deprecated*, weil in der impliziten `to_str()`-Konvertierung das Sicherheitsrisiko eines Dangling-Pointer-Problems bestand.

Andere Datentypen haben passende C-Strukturen, beispielsweise der `RArray`-Struct für `T_ARRAY`, etc. Ein VALUE, der eine passende C-Struktur besitzt, kann zu einem Pointer auf den Struct gecastet werden. Makros, die einen solchen Cast durchführen, haben die Form `RXXXX`; beispielsweise `RARRAY(obj)`. Siehe dazu die Datei `»ruby.h«`.

Es gibt ein paar Zugriffsmakros für Struct-Members, zum Beispiel `»RSTRING_LEN(s)«`, um die Länge eines Ruby-String-Objekts zu erhalten. Auf den allokierten Speicher kann mithilfe von `»RSTRING_PTR(str)«` zugegriffen werden. Für Arrays gibt es dementsprechend `»RARRAY_LEN(ary)«` und `»RARRAY_PTR(ary)«`.

Hinweis: Ändere niemals direkt einen Wert einer Struktur, wenn du nicht für das Ergebnis geradestehen willst. Solche Sachen sind oftmals Anlass für einige interessante Bugs.

1.4 Konvertieren von C-Daten in VALUEs

- Fixnums: 1 Bit nach links verschieben, dann das LSB setzen.
- Andere Objekt-Pointer: Nach VALUE casten.

Um festzustellen, ob ein VALUE ein Pointer ist oder nicht [alle Ruby-Objekte außer Immediate Values sind Pointer, Anm. d. Übers.], kannst du durch Überprüfung des LSBs feststellen.

Sei dir klar darüber, dass in Ruby nicht jeder Objekt-Pointer ein VALUE sein muss – es könnten Pointer auf Ruby-Structs sein. Alle möglichen Ruby-Structs sind in »ruby.h« definiert.

Um C-Zahlen in Ruby-Zahlen zu verwandeln, kannst du diese Makros benutzen:

- INT2FIX(): Für Integers mit 31 Bit Länge.
- INT2NUM(): Für Integers jeder Größe.

INT2NUM() konvertiert einen Integer in ein Bignum, wenn er außerhalb des Fixnum-Bereichs liegt, ist aber ein bisschen langsamer.

Verändern von Ruby-Objekten

Wie ich bereits beschrieben habe, empfehle ich nicht, die interne Struktur eines Objekts zu verändern. Um Objekte zu bearbeiten, nutze die Funktionen, die dir der Ruby-Interpreter anbietet. Viele (aber nicht alle) dieser nützlichen Funktionen sind hier aufgeführt:

String-Funktionen

[Ich habe hier einige Funktionen hinzugefügt, die mir wichtig erschienen, Anm. d. Übers.]

Funktion und Aufruf	Beschreibung
<code>rb_str_new(const char *ptr, long len)</code>	Erstellt einen Ruby-String mit bestimmter Länge.
<code>rb_str_new2(const char *ptr)</code> <code>rb_str_new_cstr(const char *ptr)</code>	Erstellt einen Ruby-String aus einen C-String. Das ist das Gleiche wie <code>rb_str_new(ptr, strlen(ptr))</code> .
<code>rb_enc_str_new(const char *ptr, long len, rb_encoding *enc)</code>	Erstellt einen neuen String mit dem angegebenen Encoding [<code>*ptr</code> sollte den String auch entsprechend kodiert enthalten, Ruby setzt nur das Encoding-

	Tag, Anm, d, Übers.].
rb_usascii_str_new(const char *ptr, long len) rb_usascii_str_new_cstr(const char *ptr)	Erstellt einen neuen String mit US-ASCII-Encoding.
rb_utf8_encoding()	Gibt das rb_encoding* für UTF-8 zurück. [Anm. d. Übers.]
rb_enc_find(const char *name)	Gibt das rb_encoding* für name zurück. [Anm. d. Übers.]
rb_tainted_str_new(const char *ptr, long len)	Erstellt einen Ruby-String, der <i>tainted</i> ist. Strings, die aus externen Quellen kommen, sollten als <i>tainted</i> markiert werden [das gehört zu Rubys Sicherheitskonzept, Anm. d. Übers.].
rb_tainted_str_new2(const char *ptr) rb_tainted_str_new_cstr(const char *ptr)	Erstellt einen neuen <i>tainted</i> -String. Funktioniert wie rb_str_new_cstr.
rb_sprintf(const char *format, ...) rb_vsprintf(const char *format, va_list ap)	Erstellt einen neuen Ruby-String mithilfe des printf(3)-Formats.
rb_str_cat(VALUE str, const char *ptr, long len)	Hängt len Bytes von ptr an den Ruby-String an.
rb_str_cat2(VALUE str, const char* ptr)	Hängt einen C-String an einen Ruby-String an. Das ist das Gleiche wie 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)	Hängt einen C-String mithilfe eines printf-artigen Formats an einen Ruby-String an. Diese Funktionen sind prinzipiell das Gleiche wie rb_str_cat2(str, rb_sprintf(format, ...)) und rb_str_cat2(str, rb_vsprintf(format, ap)).
rb_str_export_to_enc(VALUE str, rb_encoding *enc)	Äquivalent zu str.encode(enc) in Ruby. [Anm. d. Übers.]

Array-Funktionen

Funktionsname und -anwendung	Beschreibung
<code>rb_ary_new()</code>	Erstellt ein Array ohne Inhalt.
<code>rb_ary_new2(long len)</code>	Erstellt ein Array ohne Inhalt, allokiert aber internen Speicher für <code>len</code> Elemente.
<code>rb_ary_new3(long n, ...)</code>	Erstellt ein <code>n</code> -elementiges Array aus den Argumenten [bei denen es sich um VALUE-Objekte handeln muss, Anm. d. Übers.].
<code>rb_ary_new4(long n, VALUE *elts)</code>	Erstellt aus einem C-Array einen Ruby-Array.
<code>rb_ary_to_ary(VALUE obj)</code>	Konvertiert das Objekt in ein Array. Entspricht <code>Object#to_ary</code> .

Es gibt viele Funktionen, die mit Arrays arbeiten – **übergibt man ihnen ein Objekt eines anderen Typs, können sie den Ruby-Interpreter zum Absturz bringen [dies gilt für nahezu alle Funktionen, die keinen Typcheck durchführen, Anm. d. Übers.]**.

Funktionsname und -anwendung	Beschreibung
<code>rb_ary_aref(argc, *argv, VALUE ary)</code>	Äquivalent zu <code>Array#[]</code> .
<code>rb_ary_entry(VALUE ary, long offset)</code>	<code>ary[offset]</code>
<code>rb_ary_subseq(VALUE ary, long beg, long len)</code>	<code>ary[beg, len]</code>
<code>rb_ary_push(VALUE ary, VALUE val)</code>	Hängt <code>val</code> an <code>ary</code> an [Anm. d. Übers.]
<code>rb_ary_pop(VALUE ary)</code>	Entfernt das letzte Element aus dem Array und gibt es zurück [Anm. d. Übers.],
<code>rb_ary_shift(VALUE ary)</code>	Entfernt das erste Element aus dem Array und gibt es zurück [Anm. d. Übers.].
<code>rb_ary_unshift(VALUE ary, VALUE val)</code>	Fügt <code>val</code> als das erste Element des Arrays ein; die restlichen Elemente werden um 1 nach hinten geschoben [Anm. d. Übers.].

2. Ruby mit C erweitern

2.1 Neue Features zu Ruby hinzufügen

Du kannst neue Features (Klassen, Module, etc.) zum Ruby-Interpreter hinzufügen. Ruby bietet ein API für die folgenden Dinge:

- Klassen, Module
- Methoden, Singleton-Methoden
- Konstanten

2.1.1 Klassen/Modul-Definition

Benutze die folgenden Funktionen, um eine Klasse oder ein Modul zu definieren:

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

Diese Funktionen geben die neue Klasse oder das neue Modul zurück. Du solltest diese Referenz in einer Variablen speichern.

Mithilfe der folgenden Funktionen kannst du verschachtelte Klassen und Module erstellen:

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

2.1.2 Methoden/Singleton-Methoden-Definition

Methoden und Singleton-Methoden kannst du mit diesen Funktionen definieren:

```
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)
```

»argc« gibt die Anzahl der Argumente für die C-Funktion an. Diese Zahl muss kleiner 17 sein, aber ich bezweifle, dass du so viele brauchst. [*func ist der Pointer zu der C-Funktion, die bei Aufruf ausgeführt werden soll, *name der Name, unter dem Ruby die Methode definiert. Anm. d. Übers.]

Wenn »argc« negativ ist, gibt es die Calling Sequence an, nicht die Anzahl der Argumente.

Wenn argc -1 ist, wird die Funktion als

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

aufgerufen, wobei `argc` die übergebene Anzahl an Argumenten ist, `argv` ist ein C-Array, das die Argumente beinhaltet und `obj` ist der Sender [der eigentlich immer `self` genannt wird, Anm. d. Übers.].

Ist `argc -2`, werden die Argumente als Ruby-Array übergeben. Die Funktion wird als

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

aufgerufen, wobei `obj` der Sender ist und `args` das Ruby-Array, das die Argumente enthält.

Es gibt noch einige weitere Funktionen, um Methoden zu definieren. Eine nimmt eine ID als den Namen der zu definierenden Methode an. Siehe 2.2.2 für IDs.

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

Dann gibt es noch zwei Funktionen, um private- und protected-Methoden zu erstellen:

```
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)
```

Zu guter Letzt, `rb_define_module_function()` definiert Modulmethoden, die private- UND Singleton-Methoden eines Moduls sind.

Beispielsweise ist `sqrt` eine Modulfunktion des `Math`-Moduls. Sie kann wie folgt aufgerufen werden:

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

oder aber:

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

Um Modulmethoden zu definieren, nutze:

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

Oh, funktionsähnliche Methoden, bei denen es sich um private-Methoden im Kernel-Modul handelt [z.B. `#puts`, Anm. d. Übers.], können mithilfe der folgenden Funktion definiert werden:

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

```
(), int argc)
```

Um einen Alias für eine Methode anzulegen, gibt es:

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

Attribut-lese- und -schreibmethoden:

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

Um die »allocate«-Klassenmethode zu definieren oder undefinieren:

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

func muss dabei das klass-Argument annehmen und eine neu erstellte Instanz zurückgeben. Diese sollte so leer wie irgend möglich sein, ohne jede Extra-Ressource (das schließt Externe mit ein).

2.1.3 Konstanten-Definition

Es gibt 2 Funktionen, um Konstanten zu definieren:

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

Die erstere definiert eine Konstante im angegebenen Modul oder der angegebenen Klasse, die letztere definiert eine globale Konstante [wie z.B. RUBY_VERSION eine ist, Anm. d. Übers.].

2.2 Ruby-Features von C aus nutzen

Es gibt etliche Möglichkeiten, Rubys Features von C aus aufzurufen.

2.2.1 Evaluieren von Ruby-Programmen in Form von Strings

Die einfachste Art, Rubys Möglichkeiten aus einem C-Programm heraus zu nutzen, ist einen String als Ruby-Programm zu evaluieren. Diese Funktion erledigt das:

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

Die Evaluierung wird im momentanen Programmkontext durchgeführt, daher sind die lokalen Variablen der innersten Methode (die in Ruby definiert wurde), verfügbar.

Es kann passieren, dass diese Evaluierung eine Exception wirft. Es gibt eine sicherere Funktion:

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

Diese gibt nil zurück, wenn ein Error auftritt. Außerdem ist `*state` Null [die Zahl, Anm. d. Übers.], wenn `str` erfolgreich evaluiert wurde, sonst Nicht-Null.

2.2.2 ID oder Symbol

Du kannst Methoden auch direkt aufrufen, ohne einen String zu parsen. Dafür muss ich zunächst IDs erklären. Eine ID ist der Integer, der dazu dient, Dinge wie Variablen zu identifizieren. Der Ruby-Datentyp für IDs ist Symbol. Er kann von Ruby in der Form

```
:Identifizier
```

oder

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

erhalten werden.

Um den ID-Wert eines Strings von C aus zu erhalten, nutze

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

Um aus einem Ruby-Objekt (Symbol oder String) eine ID zu bekommen, nimm:

```
rb_to_id(VALUE symbol)
```

Du kannst eine C-ID in ein Ruby-Symbol konvertieren:

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

Und ein Ruby-Symbol in eine ID:

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

2.2.3 Ruby-Methoden von C aus aufrufen

Um Methoden direkt aufzurufen kannst du die folgende Funktion verwenden:

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

Diese Funktion ruft eine Methode, die durch `mid` bezeichnet wird, auf. [`argc` gibt die Anzahl der zu übergebenden Argumente an, danach folgenden die Argumente als VALUES. Anm. d. Übers.]

2.2.4 Abrufen von Variablen und Konstanten

Du kannst Klassen- und Instanzvariablen mit entsprechenden Methoden abrufen. Auch beide Umgebungen dieselben globalen Variablen benutzen. Es gibt jedoch keine Möglichkeit, Rubys lokale Variablen abzurufen.

Die Funktionen, um Instanzvariablen abzurufen/zu modifizieren, sind:

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

id muss das Symbol sein, welches du mithilfe von `rb_intern()` bekommen kannst. [Das Symbol sollte mit einem At-Zeichen @ beginnen, Anm. d. Übers.]

Auf Konstanten von Klassen und Modulen erhältst du hiermit Zugriff:

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

Siehe 2.1.3 für die Definition neuer Konstanten. [Für globale Konstanten nimmt man als `obj` am besten `rb_cObject`, das ist die Klasse `Object` von C aus gesehen, also z.B.

```
rb_const_get(rb_cObj, rb_intern("RUBY_VERSION")). Anm. d. Übers.]
```

3. Teilen von Informationen zwischen Ruby und C

3.1 Ruby-Konstanten, die von C aus abgerufen werden können

Die folgenden Ruby-Konstanten können von C aus abgerufen werden:

- Qtrue
- Qfalse

Boolsche Werte. Qfalse ist in C auch einfach 0.

- Qnil

Ruby-nil in C.

3.2 Globale Variablen zwischen Ruby und C

Mithilfe von globalen Variablen können Werte sowohl von Ruby als auch von C aus benutzt werden. Benutze die folgenden Funktionen, um sie zu definieren:

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

Diese Funktion definiert eine Variable, die von beiden Umgebungen abrufbar ist.

Der Wert der globalen Variablen namens »var« kann von Ruby aus mithilfe einer globalen Variable namens »name« [mit \$ davor, Anm. d. Übers.] abgerufen werden.

Du kannst read-only-Variablen (natürlich nur von Ruby aus nicht schreibbar) mit dieser Funktion definieren:

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

Du kannst Hook-Variablen definieren. Wenn eine solche Variable abgerufen wird, werden die getter- und setter-Funktionen aufgerufen.

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

Wenn du nur eine der beiden Funktionen nutzen willst, setze einfach 0 für den Hook, den du nicht brauchst. Sind beide Hooks 0, dann macht `rb_define_hooked_variable()` das Gleiche wie `rb_define_variable()`.

Die Prototypen der getter- und setter-Funktionen sehen wie folgt aus:

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

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

Es besteht ebenfalls die Möglichkeit, eine Ruby-globale-Variable ohne eine entsprechende C-Variable zu definieren. Der Wert einer solchen Variablen wird nur durch set- sowie get-Hooks gesetzt und ausgelesen.

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

Die Prototypen der getter- und setter-Funktionen sehen so aus:

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

3.3 C-Daten in ein Ruby-Objekt einbinden

Um einen C-Pointer zu wrappen und als Objekt auszugeben (einen gewrappten C-Pointer nennt man DATA) gibt es die Funktion `Data_Wrap_Struct()`.

```
Data_Wrap_Struct(klass, mark, free, ptr)
```

`Data_Wrap_Struct()` gibt das erstellte DATA-Objekt zurück. Das Argument `klass` gibt die Klasse des DATA-Objekts an. `mark` ist die Funktion, die sämtliche von der DATA referenzierten Ruby-Objekte zu markieren, `free` ist die Funktion um den allokierten Speicher freizugeben. Ist es -1, dann wird lediglich der Pointer freigegeben. Die `free`- und `mark`-Funktionen werden vom Garbage Collector aufgerufen.

Diese `mark/free`-Funktionen werden während der Ausführung des Garbage Collectors aufgerufen. Keine Speicherplatzallokationen für Objekte sind während dieser Phase erlaubt, also erstelle in ihr keine Ruby-Objekte.

[Ruby nutzt einen sogenannten »Mark-and-Sweep«-Garbage Collector, d.h. zunächst wird eine Mark-Phase durchlaufen, in der alle in Gebrauch befindlichen Ruby-Objekte markiert werden. Das heißt für die `mark`-Funktion, dass sie sie in diesem Muster mitspielen muss und die Objekte, die das DATA-Objekt referenziert, markieren muss. Wenn sie dies nämlich nicht tut und das einzige Objekt ist, das eine Referenz auf ein solches Objekt besitzt, wird der Garbage Collector es irrtümlich einsammeln. Dies ist die zweite Phase: Alle Objekte, die keine Markierung besitzen, werden vom Garbage Collector »entsorgt«, d.h. ihr Speicher wird freigegeben. Da Ruby nicht weiß, wie es mit den von dir allokierten Datenstrukturen deines DATA-Objekts umgehen soll, musst du die `free`-Funktion bereitstellen, die der GC aufruft, wenn keine Referenz mehr auf dein Objekt existiert. Anm. d. Übers.]

Du kannst Allokation und Wrapping in einem Schritt durchführen:

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

Dieses Makro gibt das allokierte Data-Objekt zurück, wobei es den Pointer in die Struktur wrappt, die ebenfalls allokiert wird. Die Argumente für `klass`, `mark` und `free` funktionieren wie ihre Gegenstücke in `Data_Wrap_Struct()`. `sval` wird einen Pointer, welcher vom zuvor angegebenen Typ sein sollte, auf die allokierte Struktur erhalten.

Um aus einem Data-Objekt einen C-Pointer zu bekommen, nutze das `Data_Get_Struct()`-Makro.

```
Data_Get_Struct(obj, type, sval)
```

Die Variable `sval` wird einen Pointer auf die Struktur erhalten.

4. Beispiel – Erstellen einer DBM-Extension

[DBM ist ein (uraltet) Datenbanksystem für Unix, Anm. d. Übers.]

OK, hier ein Beispiel für das Erstellen einer Extension-Programmibibliothek. Es handelt sich hierbei um eine Extension, um auf DBMs zuzugreifen. Der komplette Quellcode ist im `ext/`-Verzeichnis von Rubys Quellen enthalten.

1. Erstellen des Verzeichnisses

```
% mkdir ext/dbm
```

Erstellt ein Verzeichnis im `ext`-Ordner.

2. Designen der Programmibibliothek

Du musst die Library erst einmal designen, bevor du sie erstellen kannst.

3. Schreiben des C-Codes.

Du wirst für deine Extension C-Code schreiben müssen. Wenn deine Library nur eine einzelne Quellcode-Datei enthält, sollte der Name »LIBRARY.c« gewählt werden [wobei LIBRARY durch den Namen deiner Programmibibliothek zu ersetzen ist, Anm. d. Übers.]. Besteht deine Library jedoch aus mehreren Quellcode-Dateien, solltest du den Namen »LIBRARY.c« besser nicht wählen, da es auf einigen Plattformen zu Konflikten mit der Datei »LIBRARY.o« kommen kann [, weil sonst mehrere LIBRARY.os während des Kompilierprozesses entstehen würden, Anm. d. Übers.].

Zuallererst wird Ruby eine Initialisierungsfunktion namens »Init_LIBRARY« in deiner Programmibibliothek aufrufen, beispielsweise könnte `Init_dbm()` beim Laden der Library aufgerufen werden.

Hier ist ein Beispiel für eine Initialisierungsfunktion:

```
void
Init_dbm(void)
{
    /* Definieren einer DBM-Klasse */
    cDBM = rb_define_class("DBM", rb_cObject);
    /* DBM mischt das Mixin Enumerable ein */
    rb_include_module(cDBM, rb_mEnumerable);

    /* DBM hat eine Klassenmethode open(): Sie bekommt die
Argumente als C-Array */
    rb_define_singleton_method(cDBM, "open", fdbm_s_open, -1);

    /* DBM-Instanzmethode close(): Keine Argumente */
    rb_define_method(cDBM, "close", fdbm_close, 0);
}
```

```

/* DBM-Instanzmethode []: 1 Argument */
rb_define_method(cDBM, "[]", fdbm_fetch, 1);
:

/* ID für eine Instanzvariable, um DBM-Daten zu speichern */
id_dbm = rb_intern("dbm");
}

```

Die DBM-Extension wrappt das DBM-Struct der C-Umgebung mithilfe von `Data_Make_Struct`.

```

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

obj = Data_Make_Struct(klass, struct dbmdata, 0, free_dbm, dbmp);

```

Dieser Code wrappt die `dbmdata`-Struktur in ein Ruby-Objekt. Wir vermeiden das direkte Wrapping von `DBM*`, weil wir die Größeninformationen speichern wollen.

Um die `dbmdata`-Struktur wieder aus unserem Ruby-Objekt herauszubekommen, definieren wir das folgende Makro:

```

#define GetDBM(obj, dbmp) {\
    Data_Get_Struct(obj, struct dbmdata, dbmp);\
    if (dbmp->di_dbm == 0) closed_dbm();\
}

```

Dieses etwas kompliziertere Makro holt für uns die Struktur aus dem Objekt und prüft gleichzeitig, ob die DBM geschlossen ist.

Es gibt drei verschiedene Wege, Methodenargumente zu erhalten. Als erstes gibt es da die Methoden mit einer fixen Länge von Argumenten, welche sie etwa so erhalten:

```

static VALUE
fdbm_delete(VALUE obj, VALUE keystr)
{
    :
}

```

Das erste Argument der C-Funktion ist `self`, der Rest sind die Argumente für die Methode.

Zweitens: Methoden mit einer unbestimmten Anzahl an Argumenten [, was etwa `*args` in Ruby entspricht und zutrifft, wenn die Funktion mit `-1` als Argumentanzahl definiert wurde, Anm. d. Übers.] erhalten ihre Argumente so:

```

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

```

Das erste Argument ist die Anzahl der Methodenargumente, das zweite ist ein C-Array, welches die Argumente enthält und das dritte ist der Empfänger der Methode [alias `self`, Anm. d. Übers.].

Du kannst die `rb_scan_args()`-Funktion benutzen, um die Argumente zu überprüfen. In diesem Beispiel meint "11", dass die Methode mindestens ein Argument benötigt, höchstens jedoch zwei. [Man könnte es auch so sehen, dass die Methode mindestens 1 Argument erfordert und noch 1 Optionales dazukommt, in summa also zwei nehmen kann. Anm. d. Übers.]

Eine andere Möglichkeit für Methoden mit unbestimmter Argumentanzahl ist es, die Argumente als Ruby-Array zu bekommen [, was passiert, wenn man die Funktion mit -2 als Argumentanzahl definiert hat, Anm. d. Übers.]:

```

static VALUE
fdbm_indexes(VALUE obj, VALUE args)
{
    :
}

```

Das erste Argument ist der Empfänger, das zweite ist ein Ruby-Array, welches die Argumente für die Methode enthält.

Anmerkung

Der GC sollte über globale Variablen, die zwar Ruby-Objekte referenzieren, aber nicht zu Rubys Welt gehören, Bescheid wissen. Du musst solche durch den Gebrauch von

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

schützen.

4. *extconf.rb* vorbereiten

Wenn eine Datei namens »*extconf.rb*« existiert, wird sie benutzt werden, um eine Makefile zu generieren.

extconf.rb ist die Datei, die Kompilierbedingungen und Ähnliches überprüft. Du musst

```
require 'mkmf'
```

an den Anfang der Datei schreiben. Danach kannst du die folgenden Funktionen benutzen, um diverse Bedingungen zu überprüfen.

Methoden	Beschreibung
<code>have_library(lib, func)</code>	Prüft, ob eine Library mit der angegebenen Funktion gefunden werden kann.
<code>have_func(func, header)</code>	Prüft, ob der angegebene Header die angegebene Funktion enthält.
<code>have_header(header)</code>	Prüft, ob die angegebene Header-Datei existiert.
<code>create_makefile(target)</code>	Generiert die Makefile.
<code>dir_config(lib)</code>	Fügt die Kommandozeilenoption <code>-with-<lib>-dir=...</code> hinzu, die anschließend von den Library-suchenden Methoden beachtet wird. [Anm. d. Übers.]
<code>find_library(lib, *paths)</code>	Ähnlich wie <code>have_library</code> , aber ermöglicht es, einige Suchpfade für die Library direkt anzugeben. Das spezielle Prüfen auf eine bestimmte Funktion wird nicht unterstützt. [Anm. d. Übers.]

Der Wert der folgenden Variablen beeinflusst die Makefile:

Variable	Beschreibung
<code>\$CFLAGS</code>	Wird in die CFLAGS-make-Variable aufgenommen (z.B. <code>-O</code>) [Optionen für den C-Compiler, Anm. d. Übers.].
<code>\$CPPFLAGS</code>	Wird in die CPPFLAGS-make-Variable aufgenommen (z.B. <code>-I</code> , <code>-D</code>) [Optionen für den C++-Compiler, Anm. d. Übers.].
<code>\$LDFLAGS</code>	Wird in die LDFLAGS-make-Variable aufgenommen (z.B. <code>-L</code>) [Optionen für den Linker, Anm. d. Übers.]
<code>\$objs</code>	Liste der Object-Dateien.

Normalerweise wird die Liste der Object-Dateien automatisch beim Suchen nach Quellcode-Dateien erstellt, aber du musst sie explizit definieren, wenn einige Quellen während des Buildprozesses generiert werden.

Wenn eine Kompilierbedingung nicht erfüllt wird, solltest du nicht `create_makefile` aufrufen. Die Makefile wird nicht generiert, das Kompilieren nicht durchgeführt.

5. `depend` vorbereiten (optional)

Wenn eine Datei namens »`depend`« existiert, wird sie in die Makefile übernommen, um Abhängigkeiten zu überprüfen. Du kannst diese Datei mit diesem Befehl erstellen:

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

Es ist ungefährlich. Bereite diese Datei vor.

6. Generieren der Makefile

Versuche, die Makefile hiermit zu generieren:

```
ruby extconf.rb
```

Wenn die Library im `vendor_ruby`-Verzeichnis anstatt im `site_ruby`-Verzeichnis landen soll, verwende die `--vendor`-Option wie folgt:

```
ruby extconf.rb --vendor
```

Dieser Schritt ist nicht notwendig, wenn du die Extension in das `ext`-Verzeichnis der Ruby-Quellen geschoben hast, in diesem Fall erledigt die Kompilierung des Interpreters das für dich.

7. `make`

Gebe

```
make
```

ein, um die Extension zu kompilieren [oder `nmake` bei der `mswin32`-Version von Ruby, Anm. d. Übers.]. Auch dieser Schritt ist überflüssig, wenn sich die Extension im `ext`-Verzeichnis der Ruby-Quellen befindet.

8. `debug`

Eventuell willst du die Extension `rb_debuggen`. Extensions können statisch gelinkt werden, indem der Datei `ext/Setup` der Verzeichnisname hinzugefügt wird, wodurch du die Extension mit dem Debugger untersuchen kannst.

9. Fertig, du hast jetzt eine Library mit Extension

Du kannst mit deiner Programmbibliothek machen, was du willst. Rubys Autor hat nicht vor,

deinem auf der der Ruby-API basierende Code irgendwelche Restriktionen aufzuerlegen.
Sei frei dein Programm zu benutzen, es zu verändern, zu verteilen oder zu verkaufen.

Anhang A. Übersicht der Ruby-Quelldateien

<u>Sprachkern von Ruby</u>	
Datei	Beschreibung
class.c	Klassen und Module
error.c	Exception-Klassen und der Exception-Mechanismus.
gc.c	Speicherverwaltung
load.c	Laden von Programmbibliotheken
object.c	Objekte
variable.c	Variablen und Konstanten
<u>Ruby-Syntax-Parser</u>	
Datei	Beschreibung
parse.y → parse.c	Automatisch erstellt
keywords → lex.c	Reservierte Schlüsselwörter
<u>Ruby-Evaluierer (alias YARV)</u>	
Datei	Beschreibung
blockinlining.c	
compile.c	
eval.c	
eval_error.c	
eval_jump.c	
eval_safe.c	
insns.def	Definition der VM-Anweisungen
iseq.c	Implementation von VM::ISeq
thread.c	Thread-Verwaltung und Kontextaustausch
thread_win32.c	Thread-Implementation
thread_pthread.c	Dito
vm.c	
vm_dump.c	
vm_eval.c	
vm_exec.c	
vm_inshelper.c	

vm_method.c	
opt_insns_unif.def	Vereinheitlichung der Anweisungen
opt_operand.def	Definitionen zur Optimierung
→ insn*.inc	Automatisch generiert
→ opt*.inc	Automatisch generiert
→ vm.inc	Automatisch generiert
<u>Implementation der Regulären Ausdrücke (Oniguruma)</u>	
Datei	Beschreibung
regex.c	
regcomp.c	
regenc.c	
regerror.c	
regexec.c	
regparse.c	
regsyntax.c	
<u>Nützliche Funktionen</u>	
Datei	Beschreibung
debug.c	Debugging-Symbole für C-Debugger
dln.c	Dynamisches Laden
st.c	Allgemeine Hash-Tabelle
strftime.c	Formatierung von Time-Objekten
util.c	Sonstige Funktionen
<u>Implementation des Ruby-Interpreters</u>	
Datei	Beschreibung
dmyext.c	
dmydln.c	
dmyencoding.c	
id.c	
inits.c	
main.c	
ruby.c	
version.c	

gem_prelude.rb	
prelude.rb	
<u>Klassenbibliothek</u>	
Datei	Klassen & Methoden
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	<i>Zufallszahlen</i>
range.c	Range
rational.c	Rational
re.c	Regexp, MatchData
signal.c	Signal
sprintf.c	[Kernel#sprintf, Kernel#format, Anm. d. Übers.]
string.c	String
struct.c	Struct
time.c	Time
def/known_errors.def	Errno::* <i>Exception-Klassen</i>

→ known_errors.inc	<i>Automatisch generiert</i>
<u>Multilingualisierung [alias m17n, Anm. d. Übers.]</u>	
Datei	Beschreibung
encoding.c	Encoding
transcode.c	Encoding::Converter
enc/*.c	Encoding-Klassen
enc/trans/*	Codepoint-Mapping-Tabellen
<u>Implementation des Goruby-Interpreters</u>	
Datei	Beschreibung
goruby.c	
golf_prelude.rb	Goruby-eigene Programmbibliotheken
→ golf_prelude.c	Automatisch generiert

Anhang B. Referenz zum Ruby-Extension-API

Typen

VALUE. Das ist der Typ für das Ruby-Objekt. Die eigentlichen Strukturen wie z.B. RString sind in der Datei ruby.h definiert. Um Werte aus diesen Strukturen abzufragen, benutze Casting-Makros wie RSTRING(obj).

Variablen und Konstanten

Bezeichner	Beschreibung
Qnil	Konstant: nil-Objekt,
Qtrue	Konstant: true-Objekt (Standard-true-Wert)
Qfalse	Konstant: false-Objekt

Wrapping von C-Pointern

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

Wrappt einen C-Pointer in ein Ruby-Objekt hinein. Wenn dieses Objekt weitere Ruby-Objekte referenziert, sollten diese während des GC-Prozesses von der mark-Funktion markiert werden. Ansonsten sollte mark 0 sein. Wird das Objekt nirgendwo mehr referenziert, so wird die free-Funktion aufgerufen, um den Pointer aufzulösen.

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

Dieses Makro allokiert Speicher mithilfe von malloc() und weist die DATA, welche den Pointer zum allokierten Speicher enthält, der Variablen sval zu,

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

Dieses Makro liest den Pointer aus einer DATA und weist ihn der Variablen sval zu.

Überprüfen von Datentypen

```
TYPE(value)  
FIXNUM_P(value)  
NIL_P(value)  
void Check_Type(VALUE value, int type)  
void Check_SafeStr(VALUE value)
```

Konvertierung von Datentypen

```
FIX2INT(value)
FIX2LONG(value)
INT2FIX(i)
NUM2INT(value)
NUM2LONG(value)
INT2NUM(i)
NUM2DBL(value)
rb_float_new(f)
StringValue(value)
StringValuePtr(value)
StringValueCStr(value)
rb_str_new2(s)
```

Klassen/Modul-Definition

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

Definiert eine neue Ruby-Klasse als Subklasse von `super`. [Für Klassen, die einfach nur Subklasse von `Object` sein sollen, sollte man `rb_cObject` als Superklasse nehmen, Anm. d. Übers.]

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

Definiert eine neue Ruby-Klasse, welche eine Subklasse von `super` ist, im Namespace von `module`.

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

Definiert ein neues Ruby-Modul.

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

Definiert ein neues Ruby-Modul im Namespace von `module`.

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

Inkludiert ein Modul in eine Klasse. Wenn das Modul bereits in die Klasse eingemischt wurde, wird der Aufruf ignoriert. [Siehe `Module#include`, Anm. d. Übers.]

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

Extendet ein Objekt mit den Fähigkeiten von `module`. [Siehe `Module#extend`, Anm. d. Übers.]

Definieren von globalen Variablen

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

Definiert eine globale Variable, auf die sowohl von Ruby als auch von C aus zugegriffen werden kann. Wenn der Name ein Zeichen enthält, das in Ruby-Symbolen nicht erlaubt ist, können Ruby-Programme auf die Variable nicht zugreifen [in der Praxis lässt man, so man tatsächlich Ruby ausschließen will, meist das führende \$-Zeichen weg, was in Ruby 1.9 den gleichen Effekt hat, Anm. d. Übers.].

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

Definiert eine nur-lesbare globale Variable. Funktioniert ansonsten genau wie `rb_define_variable()`.

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

Definiert eine virtuelle Variable, deren Verhalten nur durch ein Paar von C-Funktionen bestimmt wird. Die `getter`-Funktion wird aufgerufen, wenn die Variable referenziert wird, `setter` wird bei Zuweisungen aufgerufen. Die Prototypen für `getter/setter` sind:

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

Die `getter`-Funktion muss den Wert für den Aufruf zurückgeben.

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

Definiert eine *hooked*-Variable. Das ist eine virtuelle Variable, die gleichzeitig auch eine globale C-Variable ist. Der `getter` wird mit

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

aufgerufen und gibt einen neuen Wert zurück. Der `setter` wird so aufgerufen:

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

Für den GC ist es wichtig, dass globale C-Variablen, die Ruby-Objekte halten, markiert werden.

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

Weist den GC an, diese Variablen zu schützen.

Konstantendefinition

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

Definiert eine neue Konstante in einer Klasse/einem Modul.

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

Definiert eine neue globale Konstante. Das ist einfach nur das Gleiche wie:

```
rb_define_const(rb_mKernel, name, val)
```

Methoden-Definition

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

Definiert eine Methode für die angegebene Klasse. `func` ist die aufzurufende Funktion, `argc` die Anzahl der Argumente. Wenn `argc -1` ist, erhält die Funktion drei Argumente: `argc` [Argumentanzahl], `argv` [Argumente als C-VALUE-Array] und `self` [Der Sender, Anmerkungen d. Übers.]. Ist `argc -2`, erhält die Funktion zwei Argumente, `self` und `args`, wobei `args` ein Ruby-Array der Methoden-Argumente ist.

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

Definiert für die angegebene Klasse eine private Methode. Die Argumente sind die gleichen wie für `rb_define_method()`.

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

Definiert eine Singleton-Methode. Argumente sind die gleichen wie für `rb_define_method()`.

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

Holt die Argumente aus `argc` und `argv`. `fmt` ist der Formatstring für die Argumente, etwa "12" für ein nicht-optional Argument und zwei optionale Argumente. Wenn `fmt` auf ein Sternchen * endet, heißt das, dass der Rest der Argumente als Array in die dazugehörige Variable geschrieben wird.

Aufruf von Ruby-Methoden

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

Ruft eine Methode auf. Um `mid` aus einem Methodennamen zu erhalten, verwende `rb_intern()`.

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

Ruft eine Methode auf und übergibt ihr ein Array von VALUES als Argumente.

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

Kompiliert den String und führt ihn als Ruby-Programm aus.

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

Gibt die für den gegebenen Namen die dazugehörige ID zurück.

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

Gibt den Namen einer Klasse zurück.

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

Gibt `true` zurück, wenn das Objekt die durch die `id` angegebene Methode beherrscht.

Instanzvariablen

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

Gibt den Wert einer Instanzvariablen zurück. Wenn der Name nicht mit einem `»@«` beginnt, sollte die Variable von Ruby aus unerreichbar sein.

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

Setzt den Wert einer Instanzvariablen.

Kontrollstrukturen

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

Ruft die angegebene Methode von `recv` auf, wobei der Methodename durch das Symbol `mid`

angegeben wird und `func` den zu übergebenden Block darstellt. `func` wird als erstes das `yield` übergebene Argument erhalten, `data2` als zweites und dann `argc` und `argv` als drittes und viertes.

```
[VERALTET] VALUE rb_iterate(VALUE (*func1)(), void *arg1, VALUE (*func2)(), void *arg2)
```

Ruft `func1` auf und übergibt ihr `func2` als Block. `func1` wird mit `arg1` als Argument aufgerufen, `func2` bekommt den Wert von `yield` als erstes und `arg2` als zweites Argument.

Möchte man `rb_interate` in 1.9 benutzen, so muss `func1` eine Ruby-Methode aufrufen. Diese Funktion ist seit 1.9 veraltet; benutze stattdessen `rb_block_call`.

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

Führt den Block [, der der umgebenden Methode mitgegeben wurde, Anm. d. Übers.] mit dem Wert von `val` aus.

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

Ruft `func1` mit `arg1` als Argument auf. Wenn während der Ausführung von `func1` eine Exception auftritt, wird `func2` mit `arg2` als Argument aufgerufen. Der Rückgabewert von `rb_rescue()` ist, wenn keine Exception auftritt, der von `func1`, sonst der von `func2`.

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

Ruft die Funktion `func1` mit `arg1` als Argument auf und ruft `func2` mit `arg2` als Argument auf, sobald die Ausführung von `func1` beendet wurde. Der Rückgabewert von `rb_ensure()` ist der von `func1`.

Fehler und Exceptions

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

Gibt eine Warnung in einem `printf`-ähnlichen Format aus.

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

Gibt ebenfalls eine Warnung in einem `printf`-ähnlichen Format aus, aber nur, wenn `$VERBOSE true` ist.

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

Wirft einen `RuntimeError`. `fmt` ist ein Format-String genau wie in `printf()`. [Eigentlich ist das

nur ein Spezialfall der folgenden Funktion, Anm. d. Übers.]

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

Wirft eine Exception der angegebenen Klasse. `fmt` ist ein Format-String genau wie in `printf()`.

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

Wirft einen fatalen Fehler [eine Exception der Klasse `fatal`, Anm. d. Übers.] und beendet den Interpreter. Es wird kein Exception-Handling durchgeführt, aber `ensure`-Blöcke werden noch ausgeführt.

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

Erzwingt das **sofortige** Beenden des Interpreters. Diese Funktion sollte nur in einer Situation aufgerufen werden, die durch einen Bug im Interpreter hervorgerufen wurde. Weder wird Exception-Handling durchgeführt, noch werden `ensure`-Blöcke beachtet. [Außerdem gibt diese Funktion neben der netten »[BUG] ...«-Meldung noch einen Hinweis darüber aus, dass man über Bugreports für Ruby immer froh sei, Anm. d. Übers.]

Initialisierung und Start des Interpreters

Hier folgt das API für das Einbetten eines Ruby-Interpreters, für Extensions sind diese Funktionen nicht erforderlich.

```
void ruby_init()
```

Initialisiert den Interpreter.

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

Bearbeitet Kommandozeilenoptionen für den Interpreter.

```
void ruby_run()
```

Startet die Ausführung des Interpreters.

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

Gibt den Namen des Skripts an (`$0`).

Hooks für Events im Interpreter

```
void rb_add_event_hook(rb_event_hook_func_t func, rb_event_t
events)
```

Fügt eine *Hook*-Funktion für die angegebenen Interpreter-Events hinzu. `events` sollte eine aus folgenden Werten geORter Wert sein:

- 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

Dies ist die Definition für `rb_event_hook_func_t`:

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

[Diese Funktion macht prinzipiell genau das, was man innerhalb von Ruby mit `Kernel#set_trace_func` erreichen kann, Anm. d. Übers.]

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

Entfernt die angegebene *Hook*-Funktion.

Anhang C. In *extconf.rb* verfügbare Methoden

Diese Methoden sind in der Datei *extconf.rb* [, die zum Erstellen der Makefiles für C-Extensions benutzt wird, Anm. d. Übers.] verfügbar.

```
have_macro(macro, headers)
```

Prüft, ob das Makro im angegebenen Header definiert wurde. Gibt true zurück, wenn das Makro definiert ist.

```
have_library(lib, func)
```

Prüft, ob die angegebene Programmbibliothek die angegebene Funktion enthält. Gibt bei Erfolg true zurück. [Fügt die Library zur Makefile hinzu, Anm. d. Übers.]

```
find_library(lib, func, path...)
```

Sucht nach der angegebenen Programmbibliothek in allen angegebenen Pfaden [zusätzlich zu den Standard-Pfaden, in denen auch *have_library* sucht, Anm. d. Übers.] und prüft, ob die gefundene Library die angegebene Funktion enthält. Gibt im Erfolgsfall true zurück. [Fügt die Library zur Makefile hinzu, Anm. d. Übers.]

```
have_func(func, header)
```

Prüft, ob die angegebene Funktion in *header* existiert. Gibt true zurück, wenn die Funktion existiert. Um Funktionen in einer weiteren Programmbibliothek zu prüfen, musst du erst mit *have_library()* diese Programmbibliothek überprüfen. [Definiert bei Erfolg ein Makro *HAVE_FUNC*, Anm. d. Übers.]

```
have_var(var, header)
```

Prüft, ob *var* in *header* existiert. Gibt im Erfolgsfall true zurück. Um Variablen in einer weiteren Programmbibliothek zu überprüfen, musst du erst mit *have_library()* diese Programmbibliothek überprüfen.

```
have_header(header)
```

Prüft, ob der angegebene Header existiert. Gibt im Erfolgsfall true zurück [und definiert bei Erfolg ein Makro *HAVE_HEADER*, wobei der Punkt im Dateinamen ebenfalls mit einem Unterstrich *_* ersetzt wird, z.B. *HAVE_FOO_H*, Anm. d. Übers.]

```
find_header(header, path...)
```

Prüft, ob der Header in einem der angegebenen Pfade existiert [zusätzlich zu den Standard-Pfaden, Anm. d. Übers.]. Gibt im Erfolgsfall true zurück [und definiert ein Makro *HAVE_HEADER*, Anm. d.

Übers.]

```
have_struct_member(type, member, header)
```

Prüft, ob der angegebene Typ in `header` einen bestimmten Member besitzt. [Und definiert bei Erfolg ein Makro `HAVE_ST_MEMBER`, Anm. d. Übers.]

```
have_type(type, header, opt)
```

Prüft, ob `type` in `header` definiert ist und gibt bei Erfolg `true` zurück. [Durch `opt` übergebene Parameter werden später an den Compiler weitergereicht. Zudem wird bei Erfolg ein Makro `HAVE_TYPE_TYPE` definiert, Anm. d. Übers.]

```
check_sizeof(type, header)
```

Prüft, ob `type` in `header` definiert ist und gibt bei Erfolg die Größe des Typs in `chars` zurück. Ansonsten wird `nil` zurückgegeben. [Definiert ein Makro `SIZEOF_TYPE`, dessen Wert die Größe des Typs ist, Anm. d. Übers.]

```
create_makefile(target)
```

Erstellt die Makefile für die Kompilierung. Wenn du diese Methode nicht aufrufst, wird nicht kompiliert werden. [`target` sollte der Name deiner Extension sein und dem Namen der `Init_`-Funktion entsprechen, Anm. d. Übers.]

```
find_executable(bin, path)
```

Prüft, ob die angegebene Executable in einem der durch `path` angegebenen Pfade gefunden werden kann und gibt bei Erfolg den Pfad zur Executable zurück. `path` ist eine durch `File::PATH_SEPARATOR` getrennte Liste von Verzeichnissen; wird es ausgelassen oder ist `nil`, wird der Wert der Umgebungsvariablen `PATH` benutzt. Gibt bei Misserfolg `nil` zurück.

```
with_config(withval[, default=nil])
```

Bearbeitet die Kommandozeilenoptionen und gibt den Wert von `--with-<withval>` zurück.

```
enable_config(config, *defaults)  
disable_config(config, *defaults)
```

Durchsucht die Kommandozeilenoptionen nach Booleschen Werten. Gibt `true` zurück, wenn `--enable-config` übergeben wurde oder `false`, wenn `--disable-config` übergeben wurde. Ansonsten wird, falls ein Block vorhanden ist, `defaults` dem beigefügten Block übergeben und das Ergebnis des Blocks zurückgegeben, und wenn nicht, wird `defaults` zurückgegeben.

```
dir_config(target[, default_dir])  
dir_config(target[, default_include, default_lib])
```

Bearbeitet die Kommandozeilenoptionen und fügt die Werte der Optionen `--with-<target>-dir`, `--with-<target>-include` und/oder `--with-<target>-lib` zu `$CFLAGS` und/oder `$LDFLAGS` hinzu.

Die Kurzform `--with-<target>-dir=/path` ist das Gleiche wie das separate Übergeben von `--with-<target>-include=/path/include` und `--with-<target>-lib=/path/lib`.

Gibt ein Array der hinzugefügten Verzeichnisse in der Form `[include_dir, lib_dir]` zurück.

```
pkg_config(pkg)
```

Gibt die Informationen, die der Befehl `pkg-config` bei `pkg` ausgibt, zurück. Der tatsächliche Name des Befehls `pkg-config` kann mit der Kommandozeilenoption `--with-pkg-config` überschrieben werden.