R Connection Internals

Matthew S. Shotwell

Table of Contents

1 Introduction **1**

2 General Strategy **2**

3 Specifics **3**

3.1 Components 3
3.2 Important Symbols 3
3.3 Life Cycle 3
3.3.1 Initialization 4
3.3.2 Finalization 5
3.4 Sinks 5
3.5 Character Encoding 6
3.6 Application Programming Interface 6

4 Reference **8**

4.1 R Functions 8
4.2 C Structures 8

Index **12**

1 Introduction

The information in this document was collected through investigation of the R source code, mostly in the files `src/include/Rconnections.h`, `src/main/connections.c`, the files that reference the functions and symbols therein, and the Subversion commit log over the past ten years. The concepts presented in this document are current to R version 2.12.0 (R-devel, r52500). These notes are collected and released purely to satisfy the interests of the author, and hopefully other R enthusiasts.

The content of this document assumes familiarity with the C and R programming languages, and some familiarity with the infomation contained in the R Internals manual. Readers are encouraged to follow along with the R source code in the appropriate sections, which may be obtained from one of the CRAN mirrors.

For the past ten years, the R connections internals have been fairly stable. Most changes to the internals have related to the addition of new connection types, most recently the clipboard connection types, and changes to the character re-encoding mechanism. Currently, and over the past ten years, the primary contributor (committer) to the connections internals is Professor Brian Ripley of the University of Oxford, though several other members of the R core development team have also contributed.

2 General Strategy

The R connection internals are conceptually similar to the Linux device driver API. Linux device drivers register C functions with the Linux kernel to be called when user programs request to read, write, map memory, or otherwise alter a device that the driver controls. Writing the code for a Linux device driver involves creating a kernel module, a concept similar to R's package mechanism.

In a fashion similar to Linux device drivers, R connections register C functions to be called when the R user requests to create, read, write, or modify a connection. Each type of R connection performs a specialized task, yet has a common interface with the R program. This ensures that generic functions like `readChar` and `seek` work on all types of connections.

Every R connection is associated with a structure that contains all the symbols necessary to identify the connection, perform I/O operations, character re-encoding, finalization, and storage of connection-specific data. R level functions, such as `open`, `close`, `readChar`, and `writeChar` ultimately call C functions that access the symbols in this structure to perform their task.

3 Specifics

3.1 Components

The organizational unit of all connections is the `struct Rconn` C structure, which holds references to the components that make up a connection. Each type of R connection consists of a standardized set of input/output methods (C functions),a private structure for connection-specific use, a function to initialize the `struct Rconn` (conventionally named `new`*`connection`*), and a function to interface with R-level code (conventionally named `do_`*`connection`*). The connection-specific input/output methods perform open, close, read, write, and several other operations on the connection. The `struct Rconn` structure additionally holds some symbols that are used by the character re-encoding and finalization mechanisms. Readers will find a complete listing and boilerplate descriptions of the standard input/output routines and other symbols referenced by the `struct Rconn` structure in the section on C structures (see Section 4.2 [C Structures], page 8).

3.2 Important Symbols

A pointer (`Rconnection`) to a `struct Rconn` structure for each connection is stored in a `static` declared, statically allocated array in `src/main/connections.c` named `Connections`. The size of the array is set at compile time by the `NCONNECTIONS` macro, limiting the maximum number of connections in a single R session (currently 128). However, all `struct Rconn` structures, and several other arrays used by connections are dynamically allocated when a connection is created. The first three connections referenced in the `Connections` array are reserved for the special 'terminal' connections with descriptions '`stdin`', '`stdout`', and '`stderr`'. These connections may not be modified.

Several functions are available to operate on the `Connections` array, including

`int NextConnection(void)`
: returns the next free pointer in the `Connections` array

`int ConnIndex(Rconnection)`
: returns the index of `Rconnection` in the `Connections` array

`Rconnection getConnection(int)`
: returns the `Rconnection` pointer at the `int` position of the `Connections` array

`Rconnection getConnection_no_err(int)`
: same as `getConnection(int)` but will return `NULL` rather than call generate an error.

`void InitConnections(void)`
: initializes '`stdin`', '`stdout`', and '`stderr`' connections

3.3 Life Cycle

R functions that create the various connection types may accept different arguments and have slightly different initialization steps. However, the life cycle of an R connection is fairly consistent across connections types. Many R functions operate on connections in one stage or another (usually in the Initialization stage, or between Initialization and Finalization)

see Section 4.1 [R Functions], page 8 for a partial listing. Many of these functions ultimately result in a call to one of the I/O methods set in a `struct Rconn`. In this manner, data is passed to and from the connection and the R program, possibly undergoing some useful transformation, such as text re-encoding. Without loss of generality, the following discussion considers the life cycle of a file connection.

3.3.1 Initialization

When a user calls the R function `file`, the `.Internal` calling convention passes the arguments `description`, `open`, `blocking`, `encoding`, and `raw` to the C function `do_url`, defined in `src/main/connections.c` (see the "file" entry in `src/main/names.c` for verification). Briefly, these arguments represent the URL, read/write mode, whether the read/write operations are blocking, the character encoding of text to be read from or written to the connection, and whether the connection should be a '`raw`' connection. Other R functions that create connections typically result in a call to other C functions with the naming convention of the form `do_`*`connection`*.

The `do_url` function first checks the arguments for validity, for example, that `description` is an object of type STRSXP, and not empty. The character encoding of `description` is checked and converted, if necessary to the current locale encoding. In addition, the `NextConnection` function is used to get the index of the next available `Rconnection` in the `Connections` array, raising an error if none are available. Additional tests are performed to determine the file given by `description` is a regular system file, an internet URL specifying the transfer protocol (*i.e.* `http://` or `ftp://`), or one of the supported compressed file types, such as a gzip compressed file.

Assuming the file is a regular (uncompressed) file, data from the `do_url` arguments are then extracted to their C equivalents (*i.e.* `const char *`) and passed to the `newfile` function, which dynamically allocates and initializes an instance of `struct Rconn`. The return value is an `Rconnection` pointer to the newly allocated connection. The `newfile` function is specific to regular file connections, and tailors the `struct Rconn` as such. Other connection types have functions that perform similar tasks, following the naming convention `new`*`connection`*.

The `newfile` function dynamically allocates an instance of `struct Rconn`, next passing it to the function `init_con` to be generically initialized. Code in the `newfile` function then assigns file connection input/output methods to the function pointer members of `struct Rconn`. Finally, the `private` member is set to point to a dynamically allocated instance of `struct fileconn`, which holds a `FILE` stream pointer and other members used by the `file` connection methods. A pointer to the new `struct Rconn`, and program control is then returned to `do_url`.

On return to `do_url`, a pointer to the newly allocated `struct Rconn` is copied to the `Connections` array, in the position returned by `NextConnection`. Other members of the `struct Rconn` structure are also modified at this point, including the `blocking` indicator and character encoding name `encname`.

It is convention that R connections are not immediately opened unless a valid `open` argument is provided to the R-level function that creates a connection. In the case one is provided to `do_url`, the connection is opended using the `open` function pointer set by the `newfile` functionin the newly created `struct Rconn`.

The final steps of the `do_url` function registers finalization code to deallocate memory associated with the connection via the external pointer mechanism (see Section 3.3.2 [Finalization], page 5), and build a return value. The return value is an `INTSXP` with '`conn_id`' attribute set to the external pointer, the class attribute `c("file", "connection")`, and value equal to the index of the `Rconnection` pointer to the newly created `struct Rconn` in the `Connections` array. This index is used by subsequent R function calls to identify to the new connection. Special care has been taken here to avoid exposing references to the connection structures to R code.

3.3.2 Finalization

A finalization function `conFinalizer` is registered via the external pointer mechanism when the connection is created. This function is executed when a garbage collection event finds that the connection is no longer reachable from the running R program. For more information on the finalization and external pointer mechanism, see the Writing R Extensions manual. The function `conFinalizer` calls `con_destroy`, which calls `con_close1`. The `con_close1` function calls the `close` and `destroy` methods specified in the `struct Rconn` structure associated with the connection, if necessary, and deallocates memory associated with character re-encoding and the `description` and `class` character strings. Finally, on return to `con_destroy`, memory associated with the `struct Rconn` structure is released.

The function `con_destroy` may also be called if the `open` method of an initialized connection fails, and is always called following a call to the R function `close` on non-standard, non-sink connections.

3.4 Sinks

R has a sink mechanism to divert output and messages (messages, warnings, and errors) to connections. By default, output is diverted to the `stdout` connection and messages are diverted to the `stderr` connection. The R `sink` function is used to modify or add output sink connections, and to modify (but not add to) the message connection. Adding or changing sink connections only affect the output generated by the C functions `do_writelines` (`src/main/connections.c`), `Rvprintf`, and `REvprintf` (`src/main/printutils.c`).

Internally, sink connections are managed through a collection of `static` declared variables and functions defined in the file `src/main/connections.c`. Several statically allocated arrays are delared in order to manage the stack of output sink connections, including `SinkCons`, `SinkConsClose`, and `R_SinkSplit`, each of length `NSINKS` (a macro currently defined to 21). Two additional integer variables, `R_SinkNumber` and `R_OutputCon` store the number of sinks in the output stack, and the number of the output connection at the top of the stack, respectively. Initially, `R_SinkNumber` is set to zero and `R_OutputCon` is set to one. The `SinkCons` array holds the number (index in the `Connections` array) of each sink connection. The first index of the `SinkCons` array is reserved for the `stdout` connection, and this may not be modified. The `SinkConsClose` array holds an integer value for each sink connection, that determines the action performed when the connection is popped from the sink stack. If a value in `SinkConsClose` is one or two, the corresponding connection is closed or destroyed, respectively. If the value is zero, no action is performed. The `R_SinkSplit` array holds the value one for each sink connection where output should also be diverted to the next sink connection on the stack. In this manner, it is possible to divert output to many connections simultaneously.

When a connection is added to the sink stack via the `sink` function, in turn calling the C-level `do_sink` function, the associated connection number is assigned to `R_OutputCon` and appended to the `SinkCons` array, `R_SinkNumber` is incremented, and the corresponding members of the `SinkConsClose` and `R_SinkSplit` arrays are assigned according to the arguments of the `sink` function call.

While a connection is in the sink stack, its external pointer is preserved with `R_PreserveObject`, and released with `R_ReleaseObject` once the connection is popped from the sink stack. Hence, while a connection is part of the sink stack, it will not be destroyed, even if it becomes unreachable at the R level. In addition when a sink connection is removed from the stack, the action specified by the corresponding member of `SinkConsClose` is performed, `R_SinkNumber` is decremented, and `R_OutputCon` is reassigned to the next value in the `SinkCons` array.

The connection that recieves R messages is identified by the integer stored in the `R_ErrorCon` variable, initially set to two, the number of the `stderr` connection. This variable may be set and reset using the `sink` function, such that R messages are diverted to other connections. When messages are diverted to a connection, the connection external pointer is preserved using `R_PreserveObject`. Until recently, the external pointer for a message sink connections were never released. This was problem was fixed by BUG 14331 such that external pointers for message sink connections are released the message sink is reset.

3.5 Character Encoding

The R API for chacter re-encoding is covered in the Writing R Extensions manual. However, several connection types (*i.e.* `file`, `fifo`, `pipe`, `gzfile`, `bzfile`, `xzfile`, and `clipboard`) have a special mechanism for interacting with the character re-encoding subsystem. These connection types set the `fgetc` method in the `struct Rconn` structure to a "dummy" function named `dummy_fgetc`. The purpose of this function is to buffer, re-encode, and return characters read using the `fgetc_internal` method. Hence, the `fgetc_internal` method performs the connection-specific work, returning a single byte (`char`) read from the connection. Several connections that support character re-encoding additionally set the `vsprintf` member of `struct Rconn` to `dummy_vsprintf`. This function simultaneously performs the function typically associated with vsprintf (variable argument conversion and printing according to a format string), as well as character re-encoding. This "dummy" mechanism is generic, as evidenced by its use in a variety of connection types, and provides one of the most powerful connection features.

3.6 Application Programming Interface

The R connections internals are generally not available to package developers. The C symbols associated with the connections internals are protected by declarations and definitions in private headers, `static` declarations, and by setting the visibility attribute `attribute_hidden`, where available. However, several of the functions defined in `src/main/connections.c` are called in other places throughout the source code. In addition, comments in the source code hint at a possible public API in future versions of R. The following code exerpt from `src/main/connections.c` (added at revision r19005, 03/29/2002, by Luke Tierney of The University of Iowa) illustrates both points.

```
/* This function allows C code to call the write method of a
```

```
   connection.  It is mainly intended as a means for C code to do a
   buffered write to sockets, but could be the start of a more
   extensive C-level connection API.  LT */
size_t R_WriteConnection(Rconnection con, void *buf, size_t n)
{
    if(!con->isopen) error(_("connection is not open"));
    if(!con->canwrite) error(_("cannot write to this connection"));

    return con->write(buf, 1, n, con);
}
```

The `R_WriteConnection` function is declared and used only in `src/main/serialize.c`. The R serialization mechanism (`save`/`load`) is also the sole user of the two publicly (`Rinternals.h`) declared functions loosely associated with connections: `R_InitConnOutPStream`, and `R_InitConnInPStream`. However, each of these three functions requires an *initialized* `Rconnection` pointer.

The following snippet from `src/include/Rinternals.h` defines an opaque pointer to the `struct Rconn` structure.

```
/* The connection interface is not yet available to packages.  To
   allow limited use of connection pointers this defines the opaque
   pointer type. */
#ifndef HAVE_RCONNECTION_TYPEDEF
typedef struct Rconn  *Rconnection;
#define HAVE_RCONNECTION_TYPEDEF
#endif
```

However, in the absence of a public definition for `struct Rconn`, or another mechanism to obtain a reference to an initialized structure, the existing API functions are not usable by authors of extension packages.

A more complete connections API was proposed by Jeff Horner of Vanderbilt University in an series of R-devel messages (see "[Rd] Connections patch", 11/2/06). The proposed API was essentially a series of functions to compliment the `R_WriteConnection` function, including methods to read, open, close, and create new connections. However, these additions have not been adopted by the R core development team.

4 Reference

4.1 R Functions

The following (possibly not comprehensive) list contains R functions from the `recommended` R packages that are either directly or indirectly related to connections: `bzfile`, `cat`, `clearPushBack`, `close`, `dget`, `dput`, `dump`, `fifo`, `flush`, `getAllConnections`, `getConnection`, `gzcon`, `gzfile`, `isatty`, `isIncomplete`, `isOpen`, `isSeekable`, `load`, `memCompress`, `memDecompress`, `open`, `parse`, `pipe`, `pushBack`, `pushBackLength`, `rawConnection`, `rawConnectionValue`, `readBin`, `readChar`, `read.csv`, `read.csv2`, `read.delim`, `read.delim2`, `readLines`, `read.table`, `save`, `scan`, `seek`, `sink`, `sink.number`, `socketConnection`, `sockSelect`, `source`, `stderr`, `stdin`, `stdout`, `summary.connection`, `textConnection`, `textConnectionValue`, `truncate`, `unz`, `unzip`, `url`, `write`, `writeBin`, `writeChar`, `writeLines`, `xzfile`.

4.2 C Structures

Below are the definitions for `struct Rconn` and `struct fileconn` copied from `src/include/Rconnections.h`. Additional comments by this author are inserted between special delimiters of the form `/** comment **/`.

```
struct Rconn {
    /** class name (null terminated) **/
    char* class;

    /** description (null terminated), can be a filename, url, or other
        identifier, depending on the connection type
    **/
    char* description;
    int enc; /* the encoding of 'description' */

    /** file operation mode (null terminated) **/
    char mode[5];

    /** text       - true if connection operates on text
        isopen     - true if connection is open
        incomplete - used in @code{do_readLines}, @code{do_isincomplete},
                     and text_vfprintf, From `?connections`: true if last
                     read was blocked, or for an output text connection whether
                     there is unflushed output
        canread    - true if connection is readable
        canwrite   - true if connection is writable
        canseek    - true if connection is seekable
        blocking   - true if connection reads are blocking
        isGzcon    - true if connection operates on gzip compressed data
    **/
    Rboolean text, isopen, incomplete, canread, canwrite, canseek, blocking,
          isGzcon;

    /** function pointers for I/O operations **/
    /** open - called when the connection should be opened
        args: struct Rconn * - an initialized connection to be opened
        return: Rboolean - true if connection successfully opened, false otherwise
    **/
    Rboolean (*open)(struct Rconn *);
```

```
/** close - called when the connection should be closed
    args: struct Rconn * - a connection to be closed
**/
void (*close)(struct Rconn *); /* routine closing after auto open */
/** destroy - called after the connection is closed in order to free memory,
    and other cleanup tasks
    args: struct Rconn * - a connection to be closed
**/
void (*destroy)(struct Rconn *); /* when closing connection */
/** vfprintf - variable argument list version of printf for a connection
    args: struct Rconn * - a connection where items should be printed
          const char *   - a format string in the style of the printf family
          va_list        - a variable argument list containing the items
                           referred to in the format string
    return: int - number of characters printed, negative on failure
**/
int (*vfprintf)(struct Rconn *, const char *, va_list);
/** fgetc - get a (re-encoded) character from the connection
    args: struct Rconn * - a connection to be read
    return: int - a (re-encoded) character, or R_EOF
**/
int (*fgetc)(struct Rconn *);
/** fgetc_internal - get a character from the connection
    args: struct Rconn * - a connection to be read
    return: int - a character, or R_EOF
**/
int (*fgetc_internal)(struct Rconn *);
/** seek - seek to a new position in the connection
    args: struct Rconn * - a connection to seek
          double         - offset to seek relative to origin, apparently
                           double is used here to avoid using
                           integer types, i.e. long int, which is
                           the prototype of the corresponding parameter
                           in fseek, as defined in stdio.h
          int            - the origin of seeking, 1 (and any except 2 and
                           3) if relative to the beginning of the
                           connection, 2 if relative to the current
                           connection read/write position, 3 if relative to
                           the end of the connection
          int            - currently only used by file_seek to select
                           the read or write position when the offset is NA
    return: double - the read/write position of the connection before
                     seeking, negative on error double is again used to
                     avoid integer types
**/
double (*seek)(struct Rconn *, double, int, int);
/** truncate - truncate the connection at the current read/write position.
    args: struct Rconn * - a connection to be truncated
**/
void (*truncate)(struct Rconn *);
/** fflush - called when the connection should flush internal read/write buffers
    args: struct Rconn * - a connection to be flushed
    return: int - zero on success, non-zero otherwise
**/
int (*fflush)(struct Rconn *);
/** read - read in the style of fread
    args: void *         - buffer where data is read into
          size_t         - size (in bytes) of each item to be read
```

```
          size_t         - number of items to be read
          struct Rconn * - a connection to be read
    return: size_t - number of _items_ read
**/
size_t (*read)(void *, size_t, size_t, struct Rconn *);
/** write - write in the style of fwrite
    args: void *         - buffer containing data to be written
          size_t         - size (in bytes) of each item to be written
          size_t         - number of items to be written
          struct Rconn * - a connection to be written
    return: size_t - number of _items_ written
**/
size_t (*write)(const void *, size_t, size_t, struct Rconn *);

/** cached and pushBack data
    nPushBack   - number of lines of cached/pushBack storage
    posPushBack - read position on current line of storage
    PushBack    - cached/pushBack data lines ('\n' delimited)
    save        - used to store the character following a \n, if not \r
    save2       - used to store a character from Rconn_ungetc
**/
int nPushBack, posPushBack; /* number of lines, position on top line */
char **PushBack;
int save, save2;

/** character re-encoding with iconv
    encname   - character encoding string (null terminated), this string
                must be one of the standard encoding strings used by [lib]iconv
    inconv    - input character encoding context (iconv_t)
    outconv   - output character encoding context (iconv_t)
    iconvbuff - input character encoding buffer
    oconvbuff - output character encoding buffer
    next      - only used by dummy_fgetc, points to the next re-encoded
                character for reading
    init_out  - storage for output iconv initialization sequence
    navail    - iconv buffer offset
    inavail   - iconv buffer offset
    EOF_signalled - true if EOF reached
    UTF8out   - true if connection writes UTF8 encoded characters
**/
char encname[101];
/* will be iconv_t, which is a pointer. NULL if not in use */
void *inconv, *outconv;
/* The idea here is that no MBCS char will ever not fit */
char iconvbuff[25], oconvbuff[50], *next, init_out[25];
short navail, inavail;
Rboolean EOF_signalled;
Rboolean UTF8out;

/** finalization pointers
    id     - unique id, used to "ensure that the finalizer does not
             try to close connection after it is alread closed"
             (quoted from source code), but also to identify the
             connection to be finalized. Using an arbitrary but
             unique id here is clever, it means the connections
             internals are further protected from passing references
             to connection structures.
    ex_ptr - external pointer, referenced by finalizer code
```

```
    **/
    void *id;
    void *ex_ptr;

    /** private user data (i.e. FILE *, offsets etc.) **/
    void *private;
};
typedef struct fileconn {
    /** stream pointer for file connection **/
    FILE *fp;

    /** read/write offsets **/
#if defined(HAVE_OFF_T) && defined(HAVE_FSEEKO)
    off_t rpos, wpos;
#else
#ifdef Win32
    off64_t rpos, wpos;
#else
    long rpos, wpos;
#endif
#endif

    /** last_was_write - true if last file operation was write **/
    Rboolean last_was_write;

    /** raw - true if a raw file connection **/
    Rboolean raw;

#ifdef Win32
    /** anon_file - true if file is temporary or 'anonymous' **/
    Rboolean anon_file;

    /** name - expanded filename of temporary file **/
    char name[PATH_MAX+1];
#endif
} *Rfileconn;
```

Index

Structures

C

Connections 3

R

R_ErrorCon 6
R_OutputCon 5
R_SinkNumber 5
R_SinkSplit 5

S

SinkCons 5
SinkConsClose 5
struct fileconn 11
struct Rconn 3, 8

Functions

C

close method 8
con_close1 5
con_destroy 5
conFinalizer 5
ConnIndex 3

D

destroy method 9
do_sink 5
do_url 4
do_writelines 5

F

fflush method 9
fgetc method 9
fgetc_internal method 9
file 4

G

getConnection 3
getConnection_no_err 3

I

InitConnections 3

N

newfile 4
NextConnection 3

O

open method 8

R

R_InitConnInPStream 7
R_InitConnOutPStream 7
R_PreserveObject 6
R_ReleaseObject 6
R_WriteConnection 6, 7
read method 9
REvprintf 5
Rvprintf 5

S

seek method 9
sink 5

T

truncate method 9

V

vfprintf method 9

W

write method 10