Table of Contents

1 R Internal Structures ........................................... 1
   1.1 SEXPs .......................................................... 1
   1.1.1 SEXPTYPEs .................................................. 1
   1.1.2 Rest of header .............................................. 2
   1.1.3 The ‘data’ .................................................. 3
   1.1.4 Allocation classes ......................................... 5
   1.2 Environments and variable lookup .............................. 5
   1.2.1 Search paths ............................................... 6
   1.2.2 Namespaces ................................................ 6
   1.2.3 Hash table ................................................. 6
   1.3 Attributes ................................................... 7
   1.4 Contexts ..................................................... 8
   1.5 Argument evaluation .......................................... 9
   1.5.1 Missingness ............................................... 10
   1.5.2 Dot-dot-dot arguments .................................... 11
   1.6 Autoprinting .................................................. 11
   1.7 The write barrier and the garbage collector .................. 12
   1.8 Serialization Formats ....................................... 12
   1.9 Encodings for CHARXSPs ..................................... 13
   1.10 The CHARXSP cache .......................................... 14
   1.11 Warnings and errors ......................................... 15
   1.12 S4 objects .................................................. 15
      1.12.1 Representation of S4 objects ....................... 15
      1.12.2 S4 classes .............................................. 16
      1.12.3 S4 methods ............................................ 16
      1.12.4 Mechanics of S4 dispatch ............................. 16
   1.13 Memory allocators ......................................... 17
      1.13.1 Internals of R_alloc ................................. 19
   1.14 Internal use of global and base environments .............. 19
      1.14.1 Base environment ..................................... 19
      1.14.2 Global environment .................................. 19
   1.15 Modules .................................................... 19
   1.16 Visibility .................................................. 20
      1.16.1 Hiding C entry points .............................. 20
      1.16.2 Variables in Windows DLLs ....................... 20
   1.17 Lazy loading ................................................ 21

2 Internal vs Primitive ........................................... 22
   2.1 Special primitives .......................................... 24
   2.2 Special internals ........................................... 24
   2.3 Prototypes for primitives ................................... 25
   2.4 Adding a primitive .......................................... 25

3 Internationalization in the R sources ........................... 27
   3.1 R code ....................................................... 27
   3.2 Main C code ................................................ 27
   3.3 Windows-GUI-specific code ................................ 27
1 R Internal Structures

This chapter is the beginnings of documentation about R internal structures. It is written for the core team and others studying the code in the src/main directory.

It is a work-in-progress and should be checked against the current version of the source code. Versions for R 2.x.y contain historical comments about when features were introduced: this version is for the 3.x.y series.

1.1 SEXPs

What R users think of as variables or objects are symbols which are bound to a value. The value can be thought of as either a SEXP (a pointer), or the structure it points to, a SEXPREC (and there are alternative forms used for vectors, namely VECSXP pointing to VECTOR_SEXPREC structures). So the basic building blocks of R objects are often called nodes, meaning SEXPRECs or VECTOR_SEXPRECs.

Note that the internal structure of the SEXPREC is not made available to R Extensions: rather SEXP is an opaque pointer, and the internals can only be accessed by the functions provided.

Both types of node structure have as their first three fields a 32-bit sxpinfo header and then three pointers (to the attributes and the previous and next node in a doubly-linked list), and then some further fields. On a 32-bit platform a node\(^1\) occupies 28 bytes: on a 64-bit platform typically 56 bytes (depending on alignment constraints).

The first five bits of the sxpinfo header specify one of up to 32 SEXPTYPEs.

1.1.1 SEXPTYPEs

Currently SEXPTYPEs 0:10 and 13:25 are in use. Values 11 and 12 were used for internal factors and ordered factors and have since been withdrawn. Note that the SEXPTYPE numbers are stored in saved objects and that the ordering of the types is used, so the gap cannot easily be reused.

<table>
<thead>
<tr>
<th>no</th>
<th>SEXPTYPE</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NILSXP</td>
<td>NULL</td>
</tr>
<tr>
<td>1</td>
<td>SYMSXP</td>
<td>symbols</td>
</tr>
<tr>
<td>2</td>
<td>LISTSXP</td>
<td>pairlists</td>
</tr>
<tr>
<td>3</td>
<td>CLOSXP</td>
<td>closures</td>
</tr>
<tr>
<td>4</td>
<td>ENVSXP</td>
<td>environments</td>
</tr>
<tr>
<td>5</td>
<td>PROMSXP</td>
<td>promises</td>
</tr>
<tr>
<td>6</td>
<td>LANGSXP</td>
<td>language objects</td>
</tr>
<tr>
<td>7</td>
<td>SPECIALSXP</td>
<td>special functions</td>
</tr>
<tr>
<td>8</td>
<td>BUILTINSXP</td>
<td>builtin functions</td>
</tr>
<tr>
<td>9</td>
<td>CHARSXP</td>
<td>internal character strings</td>
</tr>
<tr>
<td>10</td>
<td>LGLSXP</td>
<td>logical vectors</td>
</tr>
<tr>
<td>13</td>
<td>INTSXP</td>
<td>integer vectors</td>
</tr>
<tr>
<td>14</td>
<td>REALSXP</td>
<td>numeric vectors</td>
</tr>
<tr>
<td>15</td>
<td>CPLXSXP</td>
<td>complex vectors</td>
</tr>
<tr>
<td>16</td>
<td>STRSXP</td>
<td>character vectors</td>
</tr>
<tr>
<td>17</td>
<td>DOTSXP</td>
<td>dot-dot-dot object</td>
</tr>
<tr>
<td>18</td>
<td>ANYSXP</td>
<td>make “any” args work</td>
</tr>
<tr>
<td>19</td>
<td>VECSXP</td>
<td>list (generic vector)</td>
</tr>
<tr>
<td>20</td>
<td>EXPRSXP</td>
<td>expression vector</td>
</tr>
<tr>
<td>21</td>
<td>BCODESXP</td>
<td>byte code</td>
</tr>
</tbody>
</table>

\(^1\) strictly, a SEXP node; VECTOR_SEXPREC nodes are slightly smaller but followed by data in the node.
Chapter 1: R Internal Structures

<table>
<thead>
<tr>
<th>No.</th>
<th>SEXP Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>EXTPTRSXP</td>
<td>external pointer</td>
</tr>
<tr>
<td>23</td>
<td>WEAKREFSXP</td>
<td>weak reference</td>
</tr>
<tr>
<td>24</td>
<td>RAWSXP</td>
<td>raw vector</td>
</tr>
<tr>
<td>25</td>
<td>S4SXP</td>
<td>S4 classes not of simple type</td>
</tr>
</tbody>
</table>

Many of these will be familiar from R level: the atomic vector types are LGLSXP, INTSXP, REALSXP, CPLXSP, STRSXP and RAWSXP. Lists are VECSXP and names (also known as symbols) are SYMSXP. Pairlists (LISTSXP, the name going back to the origins of R as a Scheme-like language) are rarely seen at R level, but are for example used for argument lists. Character vectors are effectively lists all of whose elements are CHARSXP, a type that is rarely visible at R level.

Language objects (LANGSXP) are calls (including formulae and so on). Internally they are pairlists with first element a reference\(^2\) to the function to be called with remaining elements the actual arguments for the call (and with the tags if present giving the specified argument names). Although this is not enforced, many places in the code assume that the pairlist is of length one or more, often without checking.

Expressions are of type EXPRSXP: they are a vector of (usually language) objects most often seen as the result of parse().

The functions are of types CLOSXP, SPECIALSXP and BUILTINSXP: where SEXPTYPEs are stored in an integer these are sometimes lumped into a pseudo-type FUNSXP with code 99. Functions defined via function are of type CLOSXP and have formals, body and environment.

The SEXPTYPE S4SXP is for S4 objects which do not consist solely of a simple type such as an atomic vector or function.

1.1.2 Rest of header

The sxpinfo header is defined as a 32-bit C structure by

```c
struct sxpinfo_struct {
    SEXPTYPE type : 5; /* discussed above */
    unsigned int obj : 1; /* is this an object with a class attribute? */
    unsigned int named : 2; /* used to control copying */
    unsigned int gp : 16; /* general purpose, see below */
    unsigned int mark : 1; /* mark object as 'in use' in GC */
    unsigned int debug : 1;
    unsigned int trace : 1;
    unsigned int spare : 1; /* debug once */
    unsigned int gcgen : 1; /* generation for GC */
    unsigned int gccls : 3; /* class of node for GC */
}; /* Tot: 32 */
```

The debug bit is used for closures and environments. For closures it is set by debug() and unset by undebug(), and indicates that evaluations of the function should be run under the browser. For environments it indicates whether the browsing is in single-step mode.

The trace bit is used for functions for trace() and for other objects when tracing duplications (see tracemem).

The spare bit is used for closures to mark them for one time debugging.

The named field is set and accessed by the SET_NAMED and NAMED macros, and take values 0, 1 and 2. R has a ‘call by value’ illusion, so an assignment like

```r
b <- a
```

appears to make a copy of a and refer to it as b. However, if neither a nor b are subsequently altered there is no need to copy. What really happens is that a new symbol b is bound to the

---

\(^2\) a pointer to a function or a symbol to look up the function by name, or a language object to be evaluated to give a function.
same value as \( a \) and the named field on the value object is set (in this case to 2). When an object is about to be altered, the named field is consulted. A value of 2 means that the object must be duplicated before being changed. (Note that this does not say that it is necessary to duplicate, only that it should be duplicated whether necessary or not.) A value of 0 means that it is known that no other SEXP shares data with this object, and so it may safely be altered. A value of 1 is used for situations like

\[
dim(a) <- c(7, 2)
\]

where in principle two copies of \( a \) exist for the duration of the computation as (in principle)

\[
 a <- \texttt{\`dim<=\'}(a, c(7, 2))
\]

but for no longer, and so some primitive functions can be optimized to avoid a copy in this case.

The \( gp \) bits are by definition ‘general purpose’. We label these from 0 to 15. Bits 0–5 and bits 14–15 have been used as described below (mainly from detective work on the sources).

The bits can be accessed and set by the LEVELS and SETLEVELS macros, which names appear to date back to the internal factor and ordered types and are now used in only a few places in the code. The \( gp \) field is serialized/unserialized for the SEXPTYPEs other than NILSXP, SYMSXP and ENVSXP.

Bits 14 and 15 of \( gp \) are used for ‘fancy bindings’. Bit 14 is used to lock a binding or an environment, and bit 15 is used to indicate an active binding. (For the definition of an ‘active binding’ see the header comments in file src/main/envir.c.) Bit 15 is used for an environment to indicate if it participates in the global cache.

The macros ARGUSED and SET_ARGUSED are used when matching actual and formal function arguments, and take the values 0, 1 and 2.

The macros MISSING and SET_MISSING are used for pairlists of arguments. Four bits are reserved, but only two are used (and exactly what for is not explained). It seems that bit 0 is used by matchArgs to mark missingness on the returned argument list, and bit 1 is used to mark the use of a default value for an argument copied to the evaluation frame of a closure.

Bit 0 is used by macros DDVAL and SET_DDVAL. This indicates that a SYMSXP is one of the symbols \( \ldots n \) which are implicitly created when \( \ldots \) is processed, and so indicates that it may need to be looked up in a DOTSXP.

Bit 0 is used for PRSEEN, a flag to indicate if a promise has already been seen during the evaluation of the promise (and so to avoid recursive loops).

Bit 0 is used for HASHASH, on the PRINTNAME of the TAG of the frame of an environment. (This bit is not serialized for CHARSXP objects.)

Bits 0 and 1 are used for weak references (to indicate ‘ready to finalize’, ‘finalize on exit’).

Bit 0 is used by the condition handling system (on a VECSXP) to indicate a calling handler.

Bit 4 is turned on to mark S4 objects.

Bits 1, 2, 3, 5 and 6 are used for a CHARSXP to denote its encoding. Bit 1 indicates that the CHARSXP should be treated as a set of bytes, not necessarily representing a character in any known encoding. Bits 2, 3 and 6 are used to indicate that it is known to be in Latin-1, UTF-8 or ASCII respectively.

Bit 5 for a CHARSXP indicates that it is hashed by its address, that is NA_STRING or is in the CHARSXP cache (this is not serialized). Only exceptionally is a CHARSXP not hashed, and this should never happen in end-user code.

1.1.3 The ‘data’

A SEXPREC is a C structure containing the 32-bit header as described above, three pointers (to the attributes, previous and next node) and the node data, a union
union {
    struct primsxp_struct primsxp;
    struct symsxp_struct symsxp;
    struct listsxp_struct listsxp;
    struct envsxp_struct envsxp;
    struct closxp_struct closxp;
    struct promsxp_struct promsxp;
} u;

All of these alternatives apart from the first (an int) are three pointers, so the union occupies three words.

The vector types are RAWSXP, CHARSX, LGALSXP, INTSX, REALSX, CPLXSXP, STRSX, VECSXP, EXPRSXP and WEAKREFSX. Remember that such types are a VECTOR_SEXPREC, which again consists of the header and the same three pointers, but followed by two integers giving the length and ‘true length’ of the vector, and then followed by the data (aligned as required: on most 32-bit systems with a 24-byte VECTOR_SEXPREC node the data can follow immediately after the node). The data are a block of memory of the appropriate length to store ‘true length’ elements (rounded up to a multiple of 8 bytes, with the 8-byte blocks being the ‘Vcells’ referred in the documentation for gc()).

The ‘data’ for the various types are given in the table below. A lot of this is interpretation, i.e. the types are not checked.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NILSXP</td>
<td>There is only one object of type NILSXP, R_NilValue, with no data.</td>
</tr>
<tr>
<td>SYMSXP</td>
<td>Pointers to three nodes, the name, value and internal, accessed by PRINTNAME (a CHARSX), SYMVVALUE and INTERNAL. (If the symbol’s value is a .Internal function, the last is a pointer to the appropriate SEXPREC.) Many symbols have SYMVVALUE R_UnboundValue.</td>
</tr>
<tr>
<td>LISTSXP</td>
<td>Pointers to the CAR, CDR (usually a LISTSXP or NULL) and TAG (a SYMSXP or NULL).</td>
</tr>
<tr>
<td>CLOSXP</td>
<td>Pointers to the formals (a pairlist), the body and the environment.</td>
</tr>
<tr>
<td>ENVSXP</td>
<td>Pointers to the frame, enclosing environment and hash table (NULL or a VECSXP). A frame is a tagged pairlist with tag the symbol and CAR the bound value.</td>
</tr>
<tr>
<td>PROMSXP</td>
<td>Pointers to the value, expression and environment (in which to evaluate the expression). Once an promise has been evaluated, the environment is set to NULL.</td>
</tr>
<tr>
<td>LANGSX</td>
<td>A special type of LISTSXP used for function calls. (The CAR references the function (perhaps via a symbol or language object), and the CDR the argument list with tags for named arguments.) R-level documentation references to ‘expressions’ / ‘language objects’ are mainly LANGSXs, but can be symbols (SYMXPs) or expression vectors (EXPRSXPs).</td>
</tr>
<tr>
<td>SPECIALSX</td>
<td>An integer giving the offset into the table of primitives/.Internals.</td>
</tr>
<tr>
<td>BUILTINSXP</td>
<td>length, truelength followed by a block of bytes (allowing for the null terminator).</td>
</tr>
<tr>
<td>CHARSX</td>
<td>length, truelength followed by a block of C ints (which are 32 bits on all R platforms).</td>
</tr>
<tr>
<td>LGALSXP</td>
<td>length, truelength followed by a block of C ints (which are 32 bits on all R platforms).</td>
</tr>
<tr>
<td>INTSX</td>
<td>length, truelength followed by a block of C ints (which are 32 bits on all R platforms).</td>
</tr>
</tbody>
</table>

---

3 This is almost unused. The only current use is for hash tables of environments (VECSXP), where length is the size of the table and truelength is the number of primary slots in use, and for the reference hash tables in serialization (VECSXP), where truelength is the number of slots in use.
REALSXP  length, truelength followed by a block of C doubles.

CPLXSXP  length, truelength followed by a block of C99 double complexes.

STRSXP   length, truelength followed by a block of pointers (SEXP s pointing to CHARSXPs).

DOTSXP   A special type of LISTSXP for the value bound to a ... symbol: a pairlist of promises.

ANYSXP   This is used as a place holder for any type: there are no actual objects of this type.

VECSXP   length, truelength followed by a block of pointers. These are internally identical (and identical to STRSXP) but differ in the interpretations placed on the elements.

EXPRSXP  length, truelength followed by a block of C99 double complex s.

BCODESXP For the 'byte-code' objects generated by the compiler.

EXTPTRSXP Has three pointers, to the pointer, the protection value (an R object which if alive protects this object) and a tag (a SYMSXP?).

WEAKREFSXP A WEAKREFSXP is a special VECSXP of length 4, with elements 'key', 'value', 'finalizer' and 'next'. The 'key' is NULL, an environment or an external pointer, and the 'finalizer' is a function or NULL.

RAWSXP   length, truelength followed by a block of bytes.

S4SXP     two unused pointers and a tag.

1.1.4 Allocation classes
As we have seen, the field gccls in the header is three bits to label up to 8 classes of nodes. Non-vector nodes are of class 0, and 'small' vector nodes are of classes 1 to 5, with a class for custom allocator vector nodes 6 and 'large' vector nodes being of class 7. The 'small' vector nodes are able to store vector data of up to 8, 16, 32, 64 and 128 bytes: larger vectors are malloc-ed individually whereas the 'small' nodes are allocated from pages of about 2000 bytes. Vector nodes allocated using custom allocators (via allocVector3) are not counted in the gc memory usage statistics since their memory semantics is not under R's control and may be non-standard (e.g., memory could be partially shared across nodes).

1.2 Environments and variable lookup
What users think of as 'variables' are symbols which are bound to objects in 'environments'. The word ‘environment’ is used ambiguously in R to mean either the frame of an ENVSXP (a pairlist of symbol-value pairs) or an ENVSXP, a frame plus an enclosure.

There are additional places that ‘variables’ can be looked up, called ‘user databases’ in comments in the code. These seem undocumented in the R sources, but apparently refer to the RObjectTable package at http://www.omegahat.org/RObjectTables/.

The base environment is special. There is an ENVSXP environment with enclosure the empty environment R_EmptyEnv, but the frame of that environment is not used. Rather its bindings are part of the global symbol table, being those symbols in the global symbol table whose values are not R_UnboundValue. When R is started the internal functions are installed (by C code) in the symbol table, with primitive functions having values and .Internal functions having what would be their values in the field accessed by the INTERNAL macro. Then .Platform and .Machine are computed and the base package is loaded into the base environment followed by the system profile.

The frames of environments (and the symbol table) are normally hashed for faster access (including insertion and deletion).
By default R maintains a (hashed) global cache of ‘variables’ (that is symbols and their bindings) which have been found, and this refers only to environments which have been marked to participate, which consists of the global environment (aka the user workspace), the base environment plus environments \(^4\) which have been attached. When an environment is either attached or detached, the names of its symbols are flushed from the cache. The cache is used whenever searching for variables from the global environment (possibly as part of a recursive search).

1.2.1 Search paths

S has the notion of a ‘search path’: the lookup for a ‘variable’ leads (possibly through a series of frames) to the ‘session frame’ the ‘working directory’ and then along the search path. The search path is a series of databases (as returned by `search()` which contain the system functions (but not necessarily at the end of the path, as by default the equivalent of packages are added at the end).

R has a variant on the S model. There is a search path (also returned by `search()` which consists of the global environment (aka user workspace) followed by environments which have been attached and finally the base environment. Note that unlike S it is not possible to attach environments before the workspace nor after the base environment.

However, the notion of variable lookup is more general in R, hence the plural in the title of this subsection. Since environments have enclosures, from any environment there is a search path found by looking in the frame, then the frame of its enclosure and so on. Since loops are not allowed, this process will eventually terminate: it can terminate at either the base environment or the empty environment. (It can be conceptually simpler to think of the search always terminating at the empty environment, but with an optimization to stop at the base environment.) So the ‘search path’ describes the chain of environments which is traversed once the search reaches the global environment.

1.2.2 Namespaces

Namespaces are environments associated with packages (and once again the base package is special and will be considered separately). A package `pkg` with a namespace defines two environments `namespace:pkg` and `package:pkg`: it is `package:pkg` that can be attached and form part of the search path.

The objects defined by the R code in the package are symbols with bindings in the `namespace:pkg` environment. The `package:pkg` environment is populated by selected symbols from the `namespace:pkg` environment (the exports). The enclosure of this environment is an environment populated with the explicit imports from other namespaces, and the enclosure of that environment is the base namespace. (So the illusion of the imports being in the namespace environment is created via the environment tree.) The enclosure of the base namespace is the global environment, so the search from a package namespace goes via the (explicit and implicit) imports to the standard ‘search path’.

The base namespace environment `R_BaseNamespace` is another `ENVSXP` that is special-cased. It is effectively the same thing as the base environment `R_BaseEnv` except that its enclosure is the global environment rather than the empty environment: the internal code diverts lookups in its frame to the global symbol table.

1.2.3 Hash table

Environments in R usually have a hash table, and nowadays that is the default in `new.env()`. It is stored as a `VECSXP` where `length` is used for the allocated size of the table and `truelength`
is the number of primary slots in use—the pointer to the \texttt{VECSXP} is part of the header of a \texttt{SEXP} of type \texttt{ENVSXP}, and this points to \texttt{R NilValue} if the environment is not hashed.

For the pros and cons of hashing, see a basic text on Computer Science.

The code to implement hashed environments is in \texttt{src/main/envir.c}. Unless set otherwise (e.g. by the \texttt{size} argument of \texttt{new.env()}) the initial table size is 29. The table will be resized by a factor of 1.2 once the load factor (the proportion of primary slots in use) reaches 85%.

The hash chains are stored as pairlist elements of the \texttt{VECSXP}: items are inserted at the front of the pairlist. Hashing is principally designed for fast searching of environments, which are from time to time added to but rarely deleted from, so items are not actually deleted but have their value set to \texttt{R UnboundValue}.

### 1.3 Attributes

As we have seen, every \texttt{SEXPREC} has a pointer to the attributes of the node (default \texttt{R NilValue}). The attributes can be accessed/set by the macros/functions \texttt{ATTRIB} and \texttt{SET_ATTRIB}, but such direct access is normally only used to check if the attributes are NULL or to reset them. Otherwise access goes through the functions \texttt{getAttribute} and \texttt{setAttrib} which impose restrictions on the attributes. One thing to watch is that if you copy attributes from one object to another you may (un)set the "class" attribute and so need to copy the object and S4 bits as well. There is a macro/function \texttt{DUPLICATE_ATTRIB} to automate this.

Note that the ‘attributes’ of a \texttt{CHARSXPP} are used as part of the management of the \texttt{CHARSXPP} cache: of course \texttt{CHARSXPP}'s are not user-visible but C-level code might look at their attributes.

The code assumes that the attributes of a node are either \texttt{R NilValue} or a pairlist of non-zero length (and this is checked by \texttt{SET_ATTRIB}). The attributes are named (via tags on the pairlist). The replacement function \texttt{attributes<-} ensures that "dim" precedes "dimnames" in the pairlist. Attribute "dim" is one of several that is treated specially: the values are checked, and any "names" and "dimnames" attributes are removed. Similarly, you cannot set "dimnames" without having set "dim", and the value assigned must be a list of the correct length and with elements of the correct lengths (and all zero-length elements are replaced by NULL).

The other attributes which are given special treatment are "names", "class", "tsp", "comment" and "row.names". For pairlist-like objects the names are not stored as an attribute but (as symbols) as the tags: however the R interface makes them look like conventional attributes, and for one-dimensional arrays they are stored as the first element of the "dimnames" attribute. The C code ensures that the "tsp" attribute is an \texttt{REALSXPP}, the frequency is positive and the implied length agrees with the number of rows of the object being assigned to. Classes and comments are restricted to character vectors, and assigning a zero-length comment or class removes the attribute. Setting or removing a "class" attribute sets the object bit appropriately. Integer row names are converted to and from the internal compact representation.

Care needs to be taken when adding attributes to objects of the types with non-standard copying semantics. There is only one object of type \texttt{NILSXPP}, \texttt{R NilValue}, and that should never have attributes (and this is enforced in \texttt{installAttrib}). For environments, external pointers and weak references, the attributes should be relevant to all uses of the object: it is for example reasonable to have a name for an environment, and also a "path" attribute for those environments populated from R code in a package.

When should attributes be preserved under operations on an object? Becker, Chambers & Wilks (1988, pp. 144–6) give some guidance. Scalar functions (those which operate element-by-element on a vector and whose output is similar to the input) should preserve attributes (except perhaps class, and if they do preserve class they need to preserve the \texttt{OBJECT} and S4 bits). Binary operations normally call \texttt{copyMostAttributes} to copy most attributes from the longer argument (and if they are of the same length from both, preferring the values on the
Chapter 1: R Internal Structures

first). Here ‘most’ means all except the names, dim and dimnames which are set appropriately
by the code for the operator.

Subsetting (other than by an empty index) generally drops all attributes except names, dim
and dimnames which are reset as appropriate. On the other hand, subassignment generally
preserves such attributes even if the length is changed. Coercion drops all attributes. For
example:

```r
> x <- structure(1:8, names=letters[1:8], comm="a comment")
> x
a b c d e f g h
1 2 3 4 5 6 7 8
attr(,"comm")
[1] "a comment"
> x[1:3]
a b c
1 2 3
> x[3] <- 3
> x
a b c d e f g h
1 2 3 4 5 6 7 8
attr(,"comm")
[1] "a comment"
> x[9] <- 9
> x
a b c d e f g h
1 2 3 4 5 6 7 8 9
attr(,"comm")
[1] "a comment"
```

1.4 Contexts

Contexts are the internal mechanism used to keep track of where a computation has got to
(and from where), so that control-flow constructs can work and reasonable information can be
produced on error conditions (such as via traceback), and otherwise (the sys.xxx functions).

Execution contexts are a stack of C structs:

```c
typedef struct RCNTXT {
    struct RCNTXT *nextcontext; /* The next context up the chain */
    int callflag; /* The context ‘type’ */
    JMP_BUF jmpbuf; /* C stack and register information */
    int cstacktop; /* Top of the pointer protection stack */
    int evaldepth; /* Evaluation depth at inception */
    SEXP promargs; /* Promises supplied to closure */
    SEXP callfun; /* The closure called */
    SEXP sysparent; /* Environment the closure was called from */
    SEXP call; /* The call that effected this context */
    SEXP cloenv; /* The environment */
    SEXP conexit; /* Interpreted on.exit code */
    void (*cend)(void *); /* C on.exit thunk */
    void *cenddata; /* Data for C on.exit thunk */
    char *vmax; /* Top of the R_alloc stack */
    int intsusp; /* Interrupts are suspended */
    SEXP handlerstack; /* Condition handler stack */
} RCNTXT;
```
Chapter 1: R Internal Structures

```c
SEXP restartstack;  /* Stack of available restarts */
struct RPRSTACK *prstack;  /* Stack of pending promises */
} RCNTXT, *context;
```

plus additional fields for the byte-code compiler. The ‘types’ are from

```c
enum {
    CTXT_TOPLEVEL = 0, /* toplevel context */
    CTXT_NEXT = 1, /* target for next */
    CTXT_BREAK = 2, /* target for break */
    CTXT_LOOP = 3, /* break or next target */
    CTXT_FUNCTION = 4, /* function closure */
    CTXT_CCODE = 8, /* other functions that need error cleanup */
    CTXT_RETURN = 12, /* return() from a closure */
    CTXT_BROWSER = 16, /* return target on exit from browser */
    CTXT_GENERIC = 20, /* rather, running an S3 method */
    CTXT_RESTART = 32, /* a call to restart was made from a closure */
    CTXT_BUILTIN = 64 /* builtin internal function */
};
```

where the CTXT_FUNCTION bit is on wherever function closures are involved.

Contexts are created by a call to `begincontext` and ended by a call to `endcontext`: code can search up the stack for a particular type of context via `findcontext` (and jump there) or jump to a specific context via `R_JumpToContext`. `R_ToplevelContext` is the ‘idle’ state (normally the command prompt), and `R_GlobalContext` is the top of the stack.

Note that whilst calls to closures and builtins set a context, those to special internal functions never do.

Dispatching from a S3 generic (via `UseMethod` or its internal equivalent) or calling `NextMethod` sets the context type to `CTXT_GENERIC`. This is used to set the `sysparent` of the method call to that of the `generic`, so the method appears to have been called in place of the `generic` rather than from the `generic`.

The R `sys.frame` and `sys.call` functions work by counting calls to closures (type `CTXT_FUNCTION`) from either end of the context stack.

Note that the `sysparent` element of the structure is not the same thing as `sys.parent()`. Element `sysparent` is primarily used in managing changes of the function being evaluated, i.e. by `Recall` and method dispatch.

`CTXT_CCODE` contexts are currently used in `cat()`, `load()`, `scan()` and `write.table()` (to close the connection on error), by `PROTECT`, serialization (to recover from errors, e.g. free buffers) and within the error handling code (to raise the C stack limit and reset some variables).

1.5 Argument evaluation

As we have seen, functions in R come in three types, closures (`SEXPTYPE CLOSXP`), specials (`SPECIALSXP`) and builtins (`BUILTINSXP`). In this section we consider when (and if) the actual arguments of function calls are evaluated. The rules are different for the internal (special/builtin) and R-level functions (closures).

For a call to a closure, the actual and formal arguments are matched and a matched call (another `LANGSXP`) is constructed. This process first replaces the actual argument list by a list of promises to the values supplied. It then constructs a new environment which contains the names of the formal parameters matched to actual or default values: all the matched values are promises, the defaults as promises to be evaluated in the environment just created. That environment is then used for the evaluation of the body of the function, and promises will be forced (and hence actual or default arguments evaluated) when they are encountered. (Evaluating a
promise sets \texttt{NAMED = 2} on its value, so if the argument was a symbol its binding is regarded as having multiple references during the evaluation of the closure call.)

If the closure is an S3 generic (that is, contains a call to \texttt{UseMethod}) the evaluation process is the same until the \texttt{UseMethod} call is encountered. At that point the argument on which to do dispatch (normally the first) will be evaluated if it has not been already. If a method has been found which is a closure, a new evaluation environment is created for it containing the matched arguments of the method plus any new variables defined so far during the evaluation of the body of the generic. (Note that this means changes to the values of the formal arguments in the body of the generic are discarded when calling the method, but \textit{actual} argument promises which have been forced retain the values found when they were forced. On the other hand, missing arguments have values which are promises to use the default supplied by the method and not by the generic.) If the method found is a primitive it is called with the matched argument list of promises (possibly already forced) used for the generic.

The essential difference\footnote{There is currently one other difference: when profiling builtin functions are counted as function calls but specials are not.} between special and builtin functions is that the arguments of specials are not evaluated before the C code is called, and those of builtins are. Note that being a special/builtin is separate from being primitive or \texttt{Internal}: \texttt{quote} is a special primitive, + is a builtin primitive, \texttt{cbind} is a special \texttt{Internal} and \texttt{grep} is a builtin \texttt{Internal}.

Many of the internal functions are internal generics, which for specials means that they do not evaluate their arguments on call, but the C code starts with a call to \texttt{DispatchOrEval}. The latter evaluates the first argument, and looks for a method based on its class. (If S4 dispatch is on, S4 methods are looked for first, even for S3 classes.) If it finds a method, it dispatches to that method with a call based on promises to evaluate the remaining arguments. If no method is found, the remaining arguments are evaluated before return to the internal generic.

The other way that internal functions can be generic is to be group generic. Most such functions are builtins (so immediately evaluate all their arguments), and all contain a call to the C function \texttt{DispatchGeneric}. There are some peculiarities over the number of arguments for the "Math" group generic, with some members allowing only one argument, some having two (with a default for the second) and \texttt{trunc} allows one or more but the default method only accepts one.

### 1.5.1 Missingness

Actual arguments to (non-internal) R functions can be fewer than are required to match the formal arguments of the function. Having unmatched formal arguments will not matter if the argument is never used (by lazy evaluation), but when the argument is evaluated, either its default value is evaluated (within the evaluation environment of the function) or an error is thrown with a message along the lines of

\begin{verbatim}
argument "foobar" is missing, with no default
\end{verbatim}

Internally missingness is handled by two mechanisms. The object \texttt{R_MissingArg} is used to indicate that a formal argument has no (default) value. When matching the actual arguments to the formal arguments, a new argument list is constructed from the formals all of whose values are \texttt{R_MissingArg} with the first \texttt{MISSING} bit set. Then whenever a formal argument is matched to an actual argument, the corresponding member of the new argument list has its value set to that of the matched actual argument, and if that is not \texttt{R_MissingArg} the missing bit is unset.

This new argument list is used to form the evaluation frame for the function, and if named arguments are subsequently given a new value (before they are evaluated) the missing bit is cleared.

Missingness of arguments can be interrogated via the \texttt{missing()} function. An argument is clearly missing if its missing bit is set or if the value is \texttt{R_MissingArg}. However, missingness
can be passed on from function to function, for using a formal argument as an actual argument in a function call does not count as evaluation. So missing() has to examine the value (a promise) of a non-yet-evaluated formal argument to see if it might be missing, which might involve investigating a promise and so on.

Special primitives also need to handle missing arguments, and in some case (e.g. log) that is why they are special and not builtin. This is usually done by testing if an argument’s value is R_MissingArg.

1.5.2 Dot-dot-dot arguments

Dot-dot-dot arguments are convenient when writing functions, but complicate the internal code for argument evaluation.

The formals of a function with a ... argument represent that as a single argument like any other argument, with tag the symbol R_DotsSymbol. When the actual arguments are matched to the formals, the value of the ... argument is of SEXPTYPE DOTSXP, a pairlist of promises (as used for matched arguments) but distinguished by the SEXPTYPE.

Recall that the evaluation frame for a function initially contains the name=value pairs from the matched call, and hence this will be true for ... as well. The value of ... is a (special) pairlist whose elements are referred to by the special symbols ..1, ..2, ... which have the DDVAL bit set: when one of these is encountered it is looked up (via ddfindVar) in the value of the ... symbol in the evaluation frame.

Values of arguments matched to a ... argument can be missing.

Special primitives may need to handle ... arguments: see for example the internal code of switch in file src/main/builtin.c.

1.6 Autoprinting

Whether the returned value of a top-level R expression is printed is controlled by the global boolean variable R_Visible. This is set (to true or false) on entry to all primitive and internal functions based on the eval column of the table in file src/main/names.c: the appropriate setting can be extracted by the macro PRIMPRINT.

The R primitive function invisible makes use of this mechanism: it just sets R_Visible = FALSE before entry and returns its argument.

For most functions the intention will be that the setting of R_Visible when they are entered is the setting used when they return, but there need to be exceptions. The R functions identify, options, system and writeBin determine whether the result should be visible from the arguments or user action. Other functions themselves dispatch functions which may change the visibility flag: examples are .Internal, do.call, eval, withVisible, if, NextMethod, Recall, recordGraphics, standardGeneric, switch and UseMethod.

‘Special’ primitive and internal functions evaluate their arguments internally after R_Visible has been set, and evaluation of the arguments (e.g. an assignment as in PR#9263) can change the value of the flag.

The R_Visible flag can also get altered during the evaluation of a function, with comments in the code about warning, writeChar and graphics functions calling GText (PR#7397). (Since the C-level function eval sets R_Visible, this could apply to any function calling it. Since it is called when evaluating promises, even object lookup can change R_Visible.) Internal and primitive functions force the documented setting of R_Visible on return, unless the C code is allowed to change it (the exceptions above are indicated by PRIMPRINT having value 2).

The actual autoprinting is done by PrintValueEnv in file print.c. If the object to be printed has the S4 bit set and S4 methods dispatch is on, show is called to print the object. Otherwise, if

---

6 the other current example is left brace, which is implemented as a primitive.
the object bit is set (so the object has a "class" attribute), print is called to dispatch methods: for objects without a class the internal code of print.default is called.

1.7 The write barrier and the garbage collector

R has long had a generational garbage collector, and bit gcgen in the sxpinfo header is used in the implementation of this. This is used in conjunction with the mark bit to identify two previous generations.

There are three levels of collections. Level 0 collects only the youngest generation, level 1 collects the two youngest generations and level 2 collects all generations. After 20 level-0 collections the next collection is at level 1, and after 5 level-1 collections at level 2. Further, if a level-n collection fails to provide 20% free space (for each of nodes and the vector heap), the next collection will be at level \( n+1 \). (The R-level function gc() performs a level-2 collection.)

A generational collector needs to efficiently ‘age’ the objects, especially list-like objects (including STRSXP). This is done by ensuring that the elements of a list are regarded as at least as old as the list when they are assigned. This is handled by the functions SET_VECTOR_ELT and SET_STRING_ELT, which is why they are functions and not macros. Ensuring the integrity of such operations is termed the write barrier and is done by making the SEXP opaque and only providing access via functions (which cannot be used as lvalues in assignments in C).

All code in R extensions is by default behind the write barrier. The only way to obtain direct access to the internals of the SEXPRECs is to define ‘USE_RINTERNALS’ before including header file Rinternals.h, which is normally defined in Defn.h. To enable a check on the way that the access is used, R can be compiled with flag --enable-strict-barrier which ensures that header Defn.h does not define ‘USE_RINTERNALS’ and hence that SEXP is opaque in most of R itself. (There are some necessary exceptions: foremost in file memory.c where the accessor functions are defined and also in file size.c which needs access to the sizes of the internal structures.)

For background papers see [http://www.stat.uiowa.edu/~luke/R/barrier.html](http://www.stat.uiowa.edu/~luke/R/barrier.html) and [http://www.stat.uiowa.edu/~luke/R/gengcnotes.html](http://www.stat.uiowa.edu/~luke/R/gengcnotes.html).

1.8 Serialization Formats

Serialized versions of R objects are used by load/save and also at a slightly lower level by saveRDS/readRDS (and their earlier ‘internal’ dot-name versions) and serialize/unserialize. These differ in what they serialize to (a file, a connection, a raw vector) and whether they are intended to serialize a single object or a collection of objects (typically the workspace). save writes a header at the beginning of the file (a single LF-terminated line) which the lower-level versions do not.

save and saveRDS allow various forms of compression, and gzip compression is the default (except for ASCII saves). Compression is applied to the whole file stream, including the headers, so serialized files can be uncompressed or re-compressed by external programs. Both load and readRDS can read gzip, bzip2 and xz forms of compression when reading from a file, and gzip compression when reading from a connection.

R has used the same serialization format since R 1.4.0 in December 2001. Earlier formats are still supported via load and save but such formats are not described here. The current serialization format is called ‘version 2’, and has been expanded in back-compatible ways since its inception, for example to support additional SEXPTYPES.

save works by writing a single-line header (typically RDX2\n for a binary save: the only other current value is RDA2\n for save(files=TRUE)), then creating a tagged pairlist of the objects to be saved and serializing that single object. load reads the header line, unserializes a single object (a pairlist or a vector list) and assigns the elements of the object in the specified
environment. The header line serves two purposes in R: it identifies the serialization format so `load` can switch to the appropriate reader code, and the linefeed allows the detection of files which have been subjected to a non-binary transfer which re-mapped line endings. It can also be thought of as a ‘magic number’ in the sense used by the `file` program (although R save files are not yet by default known to that program).

Serialization in R needs to take into account that objects may contain references to environments, which then have enclosing environments and so on. (Environments recognized as package or name space environments are saved by name.) There are ‘reference objects’ which are not duplicated on copy and should remain shared on unserialization. These are weak references, external pointers and environments other than those associated with packages, namespaces and the global environment. These are handled via a hash table, and references after the first are written out as a reference marker indexed by the table entry.

Version-2 serialization first writes a header indicating the format (normally ‘X\n’ for an XDR format binary save, but ‘A\n’, ASCII, and ‘B\n’, native word-order binary, can also occur) and then three integers giving the version of the format and two R versions (packed by the `R_Version` macro from `Rversion.h`). (Unserialization interprets the two versions as the version of R which wrote the file followed by the minimal version of R needed to read the format.) Serialization then writes out the object recursively using function `WriteItem` in file `src/main/serialize.c`.

Some objects are written as if they were `SEXPTYPE`s: such pseudo-`SEXPTYPE`s cover `R_NilValue`, `R_EmptyEnv`, `R_BaseEnv`, `R_GlobalEnv`, `R_UnboundValue`, `R_MissingArg` and `R_BaseNamespace`.

For all `SEXPTYPE`s except `NILSXP`, `SYMSXP` and `ENVSXP` serialization starts with an integer with the `SEXPTYPE` in bits 0:7 followed by the object bit, two bits indicating if there are any attributes and if there is a tag (for the pairlist types), an unused bit and then the `gp` field in bits 12:27. Pairlist-like objects write their attributes (if any), tag (if any), CAR and then CDR (using tail recursion): other objects write their attributes after themselves. Atomic vector objects write their length followed by the data: generic vector-list objects write their length followed by a call to `WriteItem` for each element. The code for `CHARSXP`s special-cases `NA_STRING` and writes it as length -1 with no data. Lengths no more than 2^31 - 1 are written in that way and larger lengths (which only occur on 64-bit systems) as -1 followed by the upper and lower 32-bits as integers (regarded as unsigned).

Environments are treated in several ways: as we have seen, some are written as specific pseudo-`SEXPTYPE`s. Package and namespace environments are written with pseudo-`SEXPTYPE`s followed by the name. ‘Normal’ environments are written out as `ENVSXP`s with an integer indicating if the environment is locked followed by the enclosure, frame, ‘tag’ (the hash table) and attributes.

In the ‘XDR’ format integers and doubles are written in bigendian order: however the format is not fully XDR (as defined in RFC 1832) as byte quantities (such as the contents of `CHARSXP` and `RAWSXP` types) are written as-is and not padded to a multiple of four bytes. The ‘ASCII’ format writes 7-bit characters. Integers are formatted with `%d` (except that `NA_integer_` is written as `NA`), doubles formatted with `%_.16g` (plus `NA`, `Inf` and `-Inf`) and bytes with `%02x`. Strings are written using standard escapes (e.g. `\t` and `\013`) for non-printing and non-ASCII bytes.

### 1.9 Encodings for `CHARSXP`s

Character data in R are stored in the sexptype `CHARSXP`.

---

7 only bits 0:4 are currently used for `SEXPTYPE`s but values 241:255 are used for pseudo-`SEXPTYPE`s.
8 Currently the only relevant bits are 0:1, 4, 14:15.
There is support for encodings other than that of the current locale, in particular UTF-8 and the multi-byte encodings used on Windows for CJK languages. A limited means to indicate the encoding of a `CHARSXP` is via two of the ‘general purpose’ bits which are used to declare the encoding to be either Latin-1 or UTF-8. (Note that it is possible for a character vector to contain elements in different encodings.) Both printing and plotting notice the declaration and convert the string to the current locale (possibly using `<xx>` to display in hexadecimal bytes that are not valid in the current locale). Many (but not all) of the character manipulation functions will either preserve the declaration or re-encode the character string.

Strings that refer to the OS such as file names need to be passed through a wide-character interface on some OSes (e.g. Windows).

When are character strings declared to be of known encoding? One way is to do so directly via `Encoding`. The parser declares the encoding if this is known, either via the `encoding` argument to `parse` or from the locale within which parsing is being done at the R command line. (Other ways are recorded on the help page for `Encoding`.)

It is not necessary to declare the encoding of ASCII strings as they will work in any locale. ASCII strings should never have a marked encoding, as any encoding will be ignored when entering such strings into the `CHARSXP` cache.

The rationale behind considering only UTF-8 and Latin-1 was that most systems are capable of producing UTF-8 strings and this is the nearest we have to a universal format. For those that do not (for example those lacking a powerful enough `iconv`), it is likely that they work in Latin-1, the old R assumption. The the parser can return a UTF-8-encoded string if it encounters a `\uxxx` escape for a Unicode point that cannot be represented in the current charset. (This needs MBCS support, and was only enabled\(^9\) on Windows.) This is enabled for all platforms, and a `\uxxx` or `\Uxxxxxxxx` escape ensures that the parsed string will be marked as UTF-8.

Most of the character manipulation functions now preserve UTF-8 encodings: there are some notes as to which at the top of file `src/main/character.c` and in file `src/library/base/man/Encoding.Rd`.

Graphics devices are offered the possibility of handing UTF-8-encoded strings without re-encoding to the native character set, by setting `hasTextUTF8` to `TRUE` and supplying functions `textUTF8` and `strWidthUTF8` that expect UTF-8-encoded inputs. Normally the symbol font is encoded in Adobe Symbol encoding, but that can be re-encoded to UTF-8 by setting `wantSymbolUTF8` to `TRUE`. The Windows’ port of cairographics has a rather peculiar assumption: it wants the symbol font to be encoded in UTF-8 as if it were encoded in Latin-1 rather than Adobe Symbol: this is selected by `wantSymbolUTF8 = NA_LOGICAL`.

Windows has no UTF-8 locales, but rather expects to work with UCS-2\(^{10}\) strings. R (being written in standard C) would not work internally with UCS-2 without extensive changes. The Rgui console\(^{11}\) uses UCS-2 internally, but communicates with the R engine in the native encoding. To allow UTF-8 strings to be printed in UTF-8 in `Rgui.exe`, an escape convention is used (see header file `rgui_UTF8.h`) which is used by `cat`, `print` and autoprinting.

‘Unicode’ (UCS-2LE) files are common in the Windows world, and `readLines` and `scan` will read them into UTF-8 strings on Windows if the encoding is declared explicitly on an unopened connection passed to those functions.

### 1.10 The CHARSXP cache

There is a global cache for `CHARSXP`s created by `mkChar` — the cache ensures that most `CHARSXP`s with the same contents share storage (‘contents’ including any declared encoding). Not all

---

\(^9\) See define `USE_UTF8_IF_POSSIBLE` in file `src/main/gram.c`.

\(^{10}\) or UTF-16 if support for surrogates is enabled in the OS, which it is not normally so at least for Western versions of Windows, despite some claims to the contrary on the Microsoft website.

\(^{11}\) but not the GraphApp toolkit.
CHARSXP are part of the cache – notably ‘NA_STRING’ is not. CHARSPs reloaded from the save formats of R prior to 0.99.0 are not cached (since the code used is frozen and very few examples still exist).

The cache records the encoding of the string as well as the bytes: all requests to create a CHARSP should be via a call to mkCharLenCE. Any encoding given in mkCharLenCE call will be ignored if the string’s bytes are all ASCII characters.

1.11 Warnings and errors

Each of warning and stop have two C-level equivalents, warning, warningcall, error and errorcall. The relationship between the pairs is similar: warning tries to fathom out a suitable call, and then calls warningcall with that call as the first argument if it succeeds, and with call = R_NilValue if it does not. When warningcall is called, it includes the deparsed call in its printout unless call = R_NilValue.

warning and error look at the context stack. If the topmost context is not of type CTXT_BUILTIN, it is used to provide the call, otherwise the next context provides the call. This means that when these functions are called from a primitive or .Internal, the imputed call will not be to primitive/.Internal but to the function calling the primitive/.Internal. This is exactly what one wants for a .Internal, as this will give the call to the closure wrapper. (Further, for a .Internal, the call is the argument to .Internal, and so may not correspond to any R function.) However, it is unlikely to be what is needed for a primitive.

The upshot is that that warningcall and errorcall should normally be used for code called from a primitive, and warning and error should be used for code called from a .Internal (and necessarily from .Call, .C and so on, where the call is not passed down). However, there are two complications. One is that code might be called from either a primitive or a .Internal, in which case probably warningcall is more appropriate. The other involves replacement functions, where the call was once of the form

> length(x) <- y ~ x

Error in "length<"("*tmp*", value = y ~ x) : invalid value

which is unpalatable to the end user. For replacement functions there will be a suitable context at the top of the stack, so warning should be used. (The results for .Internal replacement functions such as substr< are not ideal.)

1.12 S4 objects

[This section is currently a preliminary draft and should not be taken as definitive. The description assumes that R_NO_METHODS_TABLES has not been set.]

1.12.1 Representation of S4 objects

S4 objects can be of any SEXPTYPE. They are either an object of a simple type (such as an atomic vector or function) with S4 class information or of type S4SXP. In all cases, the ‘S4 bit’ (bit 4 of the ‘general purpose’ field) is set, and can be tested by the macro/function IS_S4_OBJECT.

S4 objects are created via new()\footnote{This can also create non-S4 objects, as in new("integer").} and thence via the C function R_do_new_object. This duplicates the prototype of the class, adds a class attribute and sets the S4 bit. All S4 class attributes should be character vectors of length one with an attribute giving (as a character string) the name of the package (or .GlobalEnv) containing the class definition. Since S4 objects have a class attribute, the OBJECT bit is set.

It is currently unclear what should happen if the class attribute is removed from an S4 object, or if this should be allowed.
1.12.2 S4 classes

S4 classes are stored as R objects in the environment in which they are created, with names `.C__classname` as such they are not listed by default by `ls`.

The objects are S4 objects of class "classRepresentation" which is defined in the `methods` package.

Since these are just objects, they are subject to the normal scoping rules and can be imported and exported from namespaces like other objects. The directives `importClassesFrom` and `exportClasses` are merely convenient ways to refer to class objects without needing to know their internal 'metaname' (although `exportClasses` does a little sanity checking via `isClass`).

1.12.3 S4 methods

Details of methods are stored in S4 objects of class "MethodsList". They have a non-syntactic name of the form `.M__generic:package` for all methods defined in the current environment for the named generic derived from a specific package (which might be `.GlobalEnv`).

There is also environment `.T__generic:package` which has names the signatures of the methods defined, and values the corresponding method functions. This is often referred to as a ‘methods table’.

When a package without a namespace is attached these objects become visible on the search path. `library` calls `methods:::cacheMetaData` to update the internal tables.

During an R session there is an environment associated with each non-primitive generic containing objects `.AllMTable`, `.Generic`, `.Methods`, `.MTable`, `.SigArg` and `.SigLength`. `.MTable` and `.AllMTable` are merged methods tables containing all the methods defined directly and via inheritance respectively. `.Methods` is a merged methods list.

Exporting methods from a namespace is more complicated than exporting a class. Note first that you do not export a method, but rather the directive `exportMethods` will export all the methods defined in the namespace for a specified generic: the code also adds to the list of generics any that are exported directly. For generics which are listed via `exportMethods` or exported themselves, the corresponding "MethodsList" and environment are exported and so will appear (as hidden objects) in the package environment.

Methods for primitives which are internally S4 generic (see below) are always exported, whether mentioned in the `NAMESPACE` file or not.

Methods can be imported either via the directive `importMethodsFrom` or via importing a namespace by `import`. Also, if a generic is imported via `importFrom`, its methods are also imported. In all cases the generic will be imported if it is in the namespace, so `importMethodsFrom` is most appropriate for methods defined on generics in other packages. Since methods for a generic could be imported from several different packages, the methods tables are merged.

When a package with a namespace is attached `methods:::cacheMetaData` is called to update the internal tables: only the visible methods will be cached.

1.12.4 Mechanics of S4 dispatch

This subsection does not discuss how S4 methods are chosen: see `http://developer.r-project.org/howMethodsWork.pdf`.

For all but primitive functions, setting a method on an existing function that is not itself S4 generic creates a new object in the current environment which is a call to `standardGeneric` with the old definition as the default method. Such S4 generics can also be created via a call to `setGeneric` and are standard closures in the R language, with environment the environment within which they are created. With the advent of namespaces this is somewhat problematic:

13 although this is not recommended as it is less future-proof.
if myfn was previously in a package with a name space there will be two functions called myfn on the search paths, and which will be called depends on which search path is in use. This is starkest for functions in the base namespace, where the original will be found ahead of the newly created function from any other package with a namespace.

Primitive functions are treated quite differently, for efficiency reasons: this results in different semantics. setGeneric is disallowed for primitive functions. The methods namespace contains a list .BasicFunsList named by primitive functions: the entries are either FALSE or a standard S4 generic showing the effective definition. When setMethod (or setReplaceMethod) is called, it either fails (if the list entry is FALSE) or a method is set on the effective generic given in the list.

Actual dispatch of S4 methods for almost all primitives piggy-backs on the S3 dispatch mechanism, so S4 methods can only be dispatched for primitives which are internally S3 generic. When a primitive that is internally S3 generic is called with a first argument which is an S4 object and S4 dispatch is on (that is, the methods namespace is loaded), DispatchOrEval calls R_possible_dispatch (defined in file src/main/objects.c). (Members of the S3 group generics, which includes all the generic operators, are treated slightly differently: the first two arguments are checked and DispatchGroup is called.) R_possible_dispatch first checks an internal table to see if any S4 methods are set for that generic (and S4 dispatch is currently enabled for that generic), and if so proceeds to S4 dispatch using methods stored in another internal table. All primitives are in the base namespace, and this mechanism means that S4 methods can be set for (some) primitives and will always be used, in contrast to setting methods on non-primitives.

The exception is %*%, which is S4 generic but not S3 generic as its C code contains a direct call to R_possible_dispatch.

The primitive as.double is special, as as.numeric and as.real are copies of it. The methods package code partly refers to generics by name and partly by function, and maps as.double and as.real to as.numeric (since that is the name used by packages exporting methods for it).

Some elements of the language are implemented as primitives, for example {. This includes the subset and subassignment ‘functions’ and they are S4 generic, again piggybacking on S3 dispatch.

.BasicFunsList is generated when methods is installed, by computing all primitives, initially disallowing methods on all and then setting generics for members of .GenericArgsEnv, the S4 group generics and a short exceptions list in file BasicFunsList.R: this currently contains the subsetting and subassignment operators and an override for c.

1.13 Memory allocators

R’s memory allocation is almost all done via routines in file src/main/memory.c. It is important to keep track of where memory is allocated, as the Windows port (by default) makes use of a memory allocator that differs from malloc etc as provided by MinGW. Specifically, there are entry points Rm_malloc, Rm_free, Rm_calloc and Rm_free provided by file src/gnuwin32/malloc.c. This was done for two reasons. The primary motivation was performance: the allocator provided by MSVCRT via MinGW was far too slow at handling the many small allocations that the allocation system for SEXPReCs uses. As a side benefit, we can set a limit on the amount of allocated memory: this is useful as whereas Windows does provide virtual memory it is relatively far slower than many other R platforms and so limiting R’s use of swapping is highly advantageous. The high-performance allocator is only called from src/main/memory.c, src/main/regex.c, src/extra/pcre and src/extra/xdr: note that this means that it is not used in packages.
The rest of R should where possible make use of the allocators made available by file
\texttt{src/main/memory.c}, which are also the methods recommended in Section “Memory allocation”
in \textit{Writing R Extensions} for use in R packages, namely the use of \texttt{R\_alloc}, \texttt{Calloc}, \texttt{Realloc}
and \texttt{Free}. Memory allocated by \texttt{R\_alloc} is freed by the garbage collector once the ‘watermark’
has been reset by calling \texttt{vmaxset}. This is done automatically by the wrapper code calling
primitives and \texttt{.Internal} functions (and also by the wrapper code to \texttt{.Call} and \texttt{.External}),
but \texttt{vmaxget} and \texttt{vmaxset} can be used to reset the watermark from within internal code if the
memory is only required for a short time.

All of the methods of memory allocation mentioned so far are relatively expensive. All R
platforms support \texttt{alloca}, and in almost all cases\footnote{but apparently not on Windows.} this is managed by the compiler, allocates
memory on the C stack and is very efficient.

There are two disadvantages in using \texttt{alloca}. First, it is fragile and care is needed to
avoid writing (or even reading) outside the bounds of the allocation block returned. Second, it
increases the danger of overflowing the C stack. It is suggested that it is only used for smallish
allocations (up to tens of thousands of bytes), and that

\begin{verbatim}
R\_CheckStack();
\end{verbatim}

is called immediately after the allocation (as R’s stack checking mechanism will warn far
enough from the stack limit to allow for modest use of alloca). (\texttt{do\_makeunique} in file
\texttt{src/main/unique.c} provides an example of both points.)

There is an alternative check,

\begin{verbatim}
R\_CheckStack2(size\_t extra);
\end{verbatim}

to be called immediately before trying an allocation of \texttt{extra} bytes.

An alternative strategy has been used for various functions which require intermediate blocks
of storage of varying but usually small size, and this has been consolidated into the routines in
the header file \texttt{src/main/RBufferUtils.h}. This uses a structure which contains a buffer, the
current size and the default size. A call to

\begin{verbatim}
R\_AllocStringBuffer(size\_t blen, R\_StringBuffer \*buf);
\end{verbatim}

sets \texttt{buf->data} to a memory area of at least \texttt{blen+1} bytes. At least the default size is
used, which means that for small allocations the same buffer can be reused. A call to \texttt{R\_FreeStringBufferL}
releases memory if more than the default has been allocated whereas a call
to \texttt{R\_FreeStringBuffer} frees any memory allocated.

The \texttt{R\_StringBuffer} structure needs to be initialized, for example by

\begin{verbatim}
static R\_StringBuffer ex\_buff = {NULL, 0, MAXELTSIZE};
\end{verbatim}

which uses a default size of \texttt{MAXELTSIZE = 8192} bytes. Most current uses have a static \texttt{R\_StringBuffer}
structure, which allows the (default-sized) buffer to be shared between calls to e.g. \texttt{grep} and even between functions: this will need to be changed if R ever allows concurrent
evaluation threads. So the idiom is

\begin{verbatim}
static R\_StringBuffer ex\_buff = {NULL, 0, MAXELTSIZE};
...
char \*buf;
for(i = 0; i < n; i++) {
    compute len
    buf = R\_AllocStringBuffer(len, &ex\_buff);
    use buf
}
/* free allocation if larger than the default, but leave
   default allocated for future use */
R\_FreeStringBufferL(&ex\_buff);
\end{verbatim}
1.13.1 Internals of R_alloc

The memory used by \texttt{R_alloc} is allocated as R vectors, of type \texttt{RAWSXP}. Thus the allocation is in units of 8 bytes, and is rounded up. A request for zero bytes currently returns \texttt{NULL} (but this should not be relied on). For historical reasons, in all other cases 1 byte is added before rounding up so the allocation is always 1–8 bytes more than was asked for: again this should not be relied on.

The vectors allocated are protected via the setting of \texttt{R_VStack}, as the garbage collector marks everything that can be reached from that location. When a vector is \texttt{R_allocated}, its \texttt{ATTRIB} pointer is set to the current \texttt{R_VStack}, and \texttt{R_VStack} is set to the latest allocation. Thus \texttt{R_VStack} is a single-linked chain of the vectors currently allocated via \texttt{R_alloc}. Function \texttt{vmaxset} resets the location \texttt{R_VStack}, and should be to a value that has previously be obtained via \texttt{vmaxget}: allocations after the value was obtained will no longer be protected and hence available for garbage collection.

1.14 Internal use of global and base environments

This section notes known use by the system of these environments: the intention is to minimize or eliminate such uses.

1.14.1 Base environment

The graphics devices system maintains two variables \texttt{.Device} and \texttt{.Devices} in the base environment: both are always set. The variable \texttt{.Devices} gives a list of character vectors of the names of open devices, and \texttt{.Device} is the element corresponding to the currently active device. The null device will always be open.

There appears to be a variable \texttt{.Options}, a pairlist giving the current options settings. But in fact this is just a symbol with a value assigned, and so shows up as a base variable.

Similarly, the evaluator creates a symbol \texttt{.Last.value} which appears as a variable in the base environment.

Errors can give rise to objects \texttt{.Traceback} and \texttt{last.warning} in the base environment.

1.14.2 Global environment

The seed for the random number generator is stored in object \texttt{.Random.seed} in the global environment.

Some error handlers may give rise to objects in the global environment: for example \texttt{dump.frames} by default produces \texttt{last.dump}.

The \texttt{windows()} device makes use of a variable \texttt{.SavedPlots} to store display lists of saved plots for later display. This is regarded as a variable created by the user.

1.15 Modules

R makes use of a number of shared objects/DLLs stored in the \texttt{modules} directory. These are parts of the code which have been chosen to be loaded ‘on demand’ rather than linked as dynamic libraries or incorporated into the main executable/dynamic library.

For the remaining modules the motivation has been the amount of (often optional) code they will bring in \texttt{via} libraries to which they are linked.

\textit{internet} The internal HTTP and FTP clients and socket support, which link to system-specific support libraries. This may load \texttt{libcurl} and on Windows will load \texttt{wininet.dll} and \texttt{ws2_32.dll}.

\textit{lapack} The code which makes use of the LAPACK library, and is linked to \texttt{libRlapack} or an external LAPACK library.
X11

(Unix-alikes only.) The X11(), jpeg(), png() and tiff() devices. These are optional, and links to some or all of the X11, pango, cairo, jpeg, libpng and libtiff libraries.

1.16 Visibility

1.16.1 Hiding C entry points

We make use of the visibility mechanisms discussed in Section “Controlling visibility” in Writing R Extensions, C entry points not needed outside the main R executable/dynamic library (and in particular in no package nor module) should be prefixed by attribute_hidden. Minimizing the visibility of symbols in the R dynamic library will speed up linking to it (which packages will do) and reduce the possibility of linking to the wrong entry points of the same name. In addition, on some platforms reducing the number of entry points allows more efficient versions of PIC to be used: somewhat over half the entry points are hidden. A convenient way to hide variables (as distinct from functions) is to declare them extern0 in header file Defn.h.

The visibility mechanism used is only available with some compilers and platforms, and in particular not on Windows, where an alternative mechanism is used. Entry points will not be made available in R.dll if they are listed in the file src/gnuwin32/Rdll.hide. Entries in that file start with a space and must be strictly in alphabetic order in the C locale (use sort on the file to ensure this if you change it). It is possible to hide Fortran as well as C entry points via this file: the former are lower-cased and have an underline as suffix, and the suffixed name should be included in the file. Some entry points exist only on Windows or need to be visible only on Windows, and some notes on these are provided in file src/gnuwin32/Maintainers.notes.

Because of the advantages of reducing the number of visible entry points, they should be declared attribute_hidden where possible. Note that this only has an effect on a shared-R-library build, and so care is needed not to hide entry points that are legitimately used by packages. So it is best if the decision on visibility is made when a new entry point is created, including the decision if it should be included in header file Rinternals.h. A list of the visible entry points on shared-R-library build on a reasonably standard Unix-alike can be made by something like

```
mv -g libR.so | grep ' [BCDT] ' | cut -b20-```

1.16.2 Variables in Windows DLLs

Windows is unique in that it conventionally treats importing variables differently from functions: variables that are imported from a DLL need to be specified by a prefix (often ‘_imp_’) when being linked to (‘imported’) but not when being linked from (‘exported’). The details depend on the compiler system, and have changed for MinGW during the lifetime of that port. They are in the main hidden behind some macros defined in header file R_ext/libextern.h.

A (non-function) variable in the main R sources that needs to be referred to outside R.dll (in a package, module or another DLL such as Rgraphapp.dll) should be declared with prefix LibExtern. The main use is in Rinternals.h, but it needs to be considered for any public header and also Defn.h.

It would nowadays be possible to make use of the ‘auto-import’ feature of the MinGW port of ld to fix up imports from DLLs (and if R is built for the Cygwin platform this is what happens). However, this was not possible when the MinGW build of R was first constructed in ca 1998, allows less control of visibility and would not work for other Windows compiler suites.

It is only possible to check if this has been handled correctly by compiling the R sources on Windows.
1.17 Lazy loading

Lazy loading is always used for code in packages but is optional (selected by the package maintainer) for datasets in packages. When a package/namespace which uses it is loaded, the package/namespace environment is populated with promises for all the named objects: when these promises are evaluated they load the actual code from a database.

There are separate databases for code and data, stored in the R and data subdirectories. The database consists of two files, name.rdb and name.rdx. The .rdb file is a concatenation of serialized objects, and the .rdx file contains an index. The objects are stored in (usually) a gzip-compressed format with a 4-byte header giving the uncompressed serialized length (in XDR, that is big-endian, byte order) and read by a call to the primitive lazyLoadDBfetch. (Note that this makes lazy-loading unsuitable for really large objects: the unserialized length of an R object can exceed 4GB.)

The index or ‘map’ file name.rdx is a compressed serialized R object to be read by readRDS. It is a list with three elements variables, references and compressed. The first two are named lists of integer vectors of length 2 giving the offset and length of the serialized object in the name.rdb file. Element variables has an entry for each named object: references serializes a temporary environment used when named environments are added to the database. compressed is a logical indicating if the serialized objects were compressed: compression is always used nowadays. We later added the values compressed = 2 and 3 for bzip2 and xz compression (with the possibility of future expansion to other methods): these formats add a fifth byte to the header for the type of compression, and store serialized objects uncompressed if compression expands them.

The loader for a lazy-load database of code or data is function lazyLoad in the base package, but note that there is a separate copy to load base itself in file R_HOME/base/R/base.

Lazy-load databases are created by the code in src/library/tools/R/makeLazyLoad.R: the main tool is the unexported function makeLazyLoadDB and the insertion of database entries is done by calls to .Call("R_lazyLoadDBinsertValue", ...).

Lazy-load databases of less than 10MB are cached in memory at first use: this was found necessary when using file systems with high latency (removable devices and network-mounted file systems on Windows).

Lazy-load databases are loaded into the exports for a package, but not into the namespace environment itself. Thus they are visible when the package is attached, and also via the :: operator. This was a deliberate design decision, as packages mostly make datasets available for use by the end user (or other packages), and they should not be found preferentially from functions in the package, surprising users who expected the normal search path to be used. (There is an alternative mechanism, sysdata.rda, for ‘system datasets’ that are intended primarily to be used within the package.)

The same database mechanism is used to store parsed Rd files. One or all of the parsed objects is fetched by a call to tools:::fetchRdDB.
2 .Internal vs .Primitive

C code compiled into R at build time can be called directly in what are termed primitives or via the .Internal interface, which is very similar to the .External interface except in syntax. More precisely, R maintains a table of R function names and corresponding C functions to call, which by convention all start with ‘do_’ and return a SEXP. This table (R_FunTab in file src/main/names.c) also specifies how many arguments to a function are required or allowed, whether or not the arguments are to be evaluated before calling, and whether the function is ‘internal’ in the sense that it must be accessed via the .Internal interface, or directly accessible in which case it is printed in R as .Primitive.

Functions using .Internal() wrapped in a closure are in general preferred as this ensures standard handling of named and default arguments. For example, grep is defined as

```r
grep <-
function (pattern, x, ignore.case = FALSE, perl = FALSE, value = FALSE,
fixed = FALSE, useBytes = FALSE, invert = FALSE)
{
  if (!is.character(x)) x <- structure(as.character(x), names = names(x))
  .Internal(grep(as.character(pattern), x, ignore.case, value,
    perl, fixed, useBytes, invert))
}
```

and the use of as.character allows methods to be dispatched (for example, for factors).

However, for reasons of convenience and also efficiency (as there is some overhead in using the .Internal interface wrapped in a function closure), the primitive functions are exceptions that can be accessed directly. And of course, primitive functions are needed for basic operations—for example .Internal is itself a primitive. Note that primitive functions make no use of R code, and hence are very different from the usual interpreted functions. In particular, formals and body return NULL for such objects, and argument matching can be handled differently. For some primitives (including call, switch, .C and .subset) positional matching is important to avoid partial matching of the first argument.

The list of primitive functions is subject to change; currently, it includes the following.

1. “Special functions” which really are language elements, but implemented as primitive functions:

```r
{ ( if for while repeat break next
  return function quote switch
}
```

2. Language elements and basic operators (i.e., functions usually not called as foo(a, b, ...)) for subsetting, assignment, arithmetic, comparison and logic:

```r
[ [, [ [ $ @
<- <<- = [<- [<- $<- @<-
+ - * / ^ %% %*% %/%
< <= == != >= >
| || & && !
```

When the arithmetic, comparison and logical operators are called as functions, any argument names are discarded so positional matching is used.

3. “Low level” 0– and 1–argument functions which belong to one of the following groups of functions:

   a. Basic mathematical functions with a single argument, i.e.,
Chapter 2: .Internal vs .Primitive

abs    sign    sqrt
floor  ceiling

exp    expm1
log2   log10   log1p
cos    sin     tan
acos   asin    atan
cosh   sinh    tanh
acosh  asinh   atanh
cosp   sinpi   tanpi

gamma  lgamma  digamma trigamma

cumsum cumprod cummax cummin

Im    Re    Arg   Conj   Mod

log is a primitive function of one or two arguments with named argument matching.
trunc is a difficult case: it is a primitive that can have one or more arguments: the
default method handled in the primitive has only one.
b. Functions rarely used outside of “programming” (i.e., mostly used inside other func-
tions), such as

nargs    missing    on.exit    interactive
as.call  as.character as.complex as.double
as.environment as.integer as.logical as.raw
is.array  is.atomic    is.call    is.character
is.complex is.double    is.environment is.expression
is.finite  is.function is.infinite is.integer
is.language is.list    is.logical is.matrix
is.na     is.name    is.nan    is.null
is.numeric is.object    is.pairlist is.raw
is.real   is.recursive is.single    is.symbol
baseenv   emptyenv  globalenv pos.to.env
unclass   invisible seq_along seq_len

c. The programming and session management utilities

browser proc.time gc.time tracemem retracemem untracemem

4. The following basic replacement and extractor functions

length   length<-
class    class<-
oldClass oldClass<-
attr     attr<-
attributes attributes<-
names    names<-
dim      dim<-
dimnames dimnames<-
environment<-levels<-storage.mode<-
hence replacement functions should where possible be primitive to avoid copying (at least in their default methods).

5. The following functions are primitive for efficiency reasons:

```r
: ~ c list
call expression substitute
UseMethod standardGeneric
.C .Fortran .Call .External
round signif rep seq.int
```

as well as the following internal-use-only functions

```r
.Primitive .Internal
.Call.graphics .External.graphics
.subset .subset2
.primTrace .primUntrace
lazyLoadDBfetch
```

The multi-argument primitives

```r
call switch
.C .Fortran .Call .External
```

intentionally use positional matching, and need to do so to avoid partial matching to their first argument. They do check that the first argument is unnamed or for the first two, partially matches the formal argument name. On the other hand,

```r
attr attr<- browser rememtrace substitute UseMethod
log round signif rep seq.int
```

manage their own argument matching and do work in the standard way.

All the one-argument primitives check that if they are called with a named argument that this (partially) matches the name given in the documentation: this is also done for replacement functions with one argument plus `value`.

The net effect is that argument matching for primitives intended for end-user use as functions is done in the same way as for interpreted functions except for the six exceptions where positional matching is required.

### 2.1 Special primitives

A small number of primitives are `specials` rather than `builtins`, that is they are entered with unevaluated arguments. This is clearly necessary for the language constructs and the assignment operators, as well as for `&&` and `||` which conditionally evaluate their second argument, and `~`, `.Internal, call, expression, missing, on.exit, quote and substitute` which do not evaluate some of their arguments.

`rep` and `seq.int` are special as they evaluate some of their arguments conditional on which are non-missing.

`log, round` and `signif` are special to allow default values to be given to missing arguments.

The subsetting, subassignment and `@` operators are all special. (For both extraction and replacement forms, `$` and `@` take a symbol argument, and `[` and `[[` allow missing arguments.)

`UseMethod` is special to avoid the additional contexts added to calls to builtins.

### 2.2 Special internals

There are also special `.Internal` functions: `NextMethod, Recall, withVisible, cbind, rbind` (to allow for the `deparse.level` argument), `eapply, lapply` and `vapply`. 
2.3 Prototypes for primitives

Prototypes are available for the primitive functions and operators, and these are used for printing, \texttt{args} and package checking (e.g. by \texttt{tools::checkS3methods} and by package \texttt{codetools}). There are two environments in the \texttt{base} package (and namespace), \texttt{.GenericArgsEnv} for those primitives which are internal S3 generics, and \texttt{.ArgsEnv} for the rest. Those environments contain closures with the same names as the primitives, formal arguments derived (manually) from the help pages, a body which is a suitable call to \texttt{UseMethod} or \texttt{NULL} and environment the base namespace.

The C code for \texttt{print.default} and \texttt{args} uses the closures in these environments in preference to the definitions in base (as primitives).

The QC function \texttt{undoc} checks that all the functions prototyped in these environments are currently primitive, and that the primitives not included are better thought of as language elements (at the time of writing

\begin{verbatim}
 $ $<- kk ( : @ @<- [ [ [<- [<- { |- <- <- =
break for function if next repeat return while
\end{verbatim}

). One could argue about "," but it is known to the parser and has semantics quite unlike a normal function. And : is documented with different argument names in its two meanings.)

The QC functions \texttt{codoc} and \texttt{checkS3methods} also make use of these environments (effectively placing them in front of base in the search path), and hence the formals of the functions they contain are checked against the help pages by \texttt{codoc}. However, there are two problems with the generic primitives. The first is that many of the operators are part of the S3 group generic \texttt{Ops} and that defines their arguments to be \texttt{e1} and \texttt{e2}: although it would be very unusual, an operator could be called as e.g. "+(\texttt{e1=a}, \texttt{e2=b})" and if method dispatch occurred to a closure, there would be an argument name mismatch. So the definitions in environment \texttt{.GenericArgsEnv} have to use argument names \texttt{e1} and \texttt{e2} even though the traditional documentation is in terms of \texttt{x} and \texttt{y}: \texttt{codoc} makes the appropriate adjustment via \texttt{tools:::.make_S3 Primitive generic env}. The second discrepancy is with the Math group generics, where the group generic is defined with argument list \texttt{(x, ...)}, but most of the members only allow one argument when used as the default method (and \texttt{round} and \texttt{signif} allow two as default methods): again fix-ups are used.

Those primitives which are in \texttt{.GenericArgsEnv} are checked (via \texttt{tests/primitives.R}) to be generic \texttt{via} defining methods for them, and a check is made that the remaining primitives are probably not generic, by setting a method and checking it is not dispatched to (but this can fail for other reasons). However, there is no certain way to know that if other \texttt{.Internal} or primitive functions are not internally generic except by reading the source code.

2.4 Adding a primitive

[For R-core use: reverse this procedure to remove a primitive. Most commonly this is done by changing a \texttt{.Internal} to a primitive or \texttt{vice versa}.]

Primitives are listed in the table \texttt{R FunTab} in \texttt{src/main/names.c}: primitives have ‘\texttt{Y = 0}’ in the ‘eval’ field.

There needs to be an ‘\texttt{\textbackslash(alias)}’ entry in a help file in the \texttt{base} package, and the primitive needs to be added to one of the lists at the start of this section.

Some primitives are regarded as language elements (the current ones are listed above). These need to be added to two lists of exceptions, \texttt{langElts} in \texttt{undoc()} (in file \texttt{src/library/tools/R/QC.R}) and \texttt{lang_elements} in \texttt{tests/primitives.R}.

All other primitives are regarded as functions and should be listed in one of the environments defined in \texttt{src/library/base/R/zzz.R}, either \texttt{.ArgsEnv} or \texttt{.GenericArgsEnv}: internal generics also need to be listed in the character vector \texttt{.S3PrimitiveGenerics}. Note too the discussion
about argument matching above: if you add a primitive function with more than one argument by converting a `.Internal` you need to add argument matching to the C code, and for those with a single argument, add argument-name checking.

Do ensure that `make check-devel` has been run: that tests most of these requirements.
3 Internationalization in the R sources

The process of marking messages (errors, warnings etc) for translation in an R package is described in Section “Internationalization” in Writing R Extensions, and the standard packages included with R have (with an exception in grDevices for the menus of the windows() device) been internationalized in the same way as other packages.

3.1 R code

Internationalization for R code is done in exactly the same way as for extension packages. As all standard packages which have R code also have a namespace, it is never necessary to specify domain, but for efficiency calls to message, warning and stop should include domain = NA when the message is constructed via gettextf, gettext or ngettext.

For each package, the extracted messages and translation sources are stored under package directory po in the source package, and compiled translations under inst/po for installation to package directory po in the installed package. This also applies to C code in packages.

3.2 Main C code

The main C code (e.g. that in files src/*/*.c and in the modules) is where R is closest to the sort of application for which ‘gettext’ was written. Messages in the main C code are in domain R and stored in the top-level directory po with compiled translations under share/locale.

The list of files covered by the R domain is specified in file po/POTFILES.in.

The normal way to mark messages for translation is via _("msg") just as for packages. However, sometimes one needs to mark passages for translation without wanting them translated at the time, for example when declaring string constants. This is the purpose of the N_ macro, for example

{ ERROR_ARGTYPE, N_("invalid argument type")},

from file src/main/errors.c.

The P_ macro

#ifndef ENABLE_NLS
#define P_(StringS, StringP, N) ngettext (StringS, StringP, N)
#else
#define P_(StringS, StringP, N) (N > 1 ? StringP: StringS)
#endif

may be used as a wrapper for ngettext: however in some cases the preferred approach has been to conditionalize (on ENABLE_NLS) code using ngettext.

The macro _("msg") can safely be used in directory src/app1; the header for standalone ‘nmath’ skips possible translation. (This does not apply to N_ or P_).

3.3 Windows-GUI-specific code

Messages for the Windows GUI are in a separate domain ‘RGui’. This was done for two reasons:

- The translators for the Windows version of R might be separate from those for the rest of R (familiarity with the GUI helps), and
- Messages for Windows are most naturally handled in the native charset for the language, and in the case of CJK languages the charset is Windows-specific. (It transpires that as the iconv we ported works well under Windows, this is less important than anticipated.)

Messages for the ‘RGui’ domain are marked by G_("msg"), a macro that is defined in header file src/gnuwin32/win-nls.h. The list of files that are considered is hardcoded in
the RGui.pot-update target of file po/Makefile.in.in: note that this includes devWindows.c as the menus on the windows device are considered to be part of the GUI. (There is also GN_("msg"), the analogue of N_("msg").)

The template and message catalogs for the 'RGui' domain are in the top-level po directory.

3.4 OS X GUI

This is handled separately: see http://developer.r-project.org/Translations.html.

3.5 Updating

See file po/README for how to update the message templates and catalogs.
Chapter 4: Structure of an Installed Package

4 Structure of an Installed Package

The structure of a source packages is described in Section “Creating R packages” in Writing R Extensions; this chapter is concerned with the structure of installed packages.

An installed package has a top-level file DESCRIPTION, a copy of the file of that name in the package sources with a 'Built' field appended, and file INDEX, usually describing the objects on which help is available, a file NAMESPACE if the package has a name space, optional files such as CITATION, LICENCE and NEWS, and any other files copied in from inst. It will have directories Meta, help and html (even if the package has no help pages), almost always has a directory R and often has a directory libs to contain compiled code. Other directories with known meaning to R are data, demo, doc and po.

Function library looks for a namespace and if one is found passes control to loadNamespace. Then library or loadNamespace looks for file R/pkgname, warns if it is not found and otherwise sources the code (using sys.source) into the package’s environment, then lazy-loads a database R/sysdata if present. So how R code gets loaded depends on the contents of R/pkgname: a standard template to load lazy-load databases are provided in share/R/nspackloader.R.

Compiled code is usually loaded when the package’s namespace is loaded by a useDynlib directive in a NAMESPACE file or by the package’s .onLoad function. Conventionally compiled code is loaded by a call to library.dynam and this looks in directory libs (and in an appropriate sub-directory if sub-architectures are in use) for a shared object (Unix-alike) or DLL (Windows).

Subdirectory data serves two purposes. In a package using lazy-loading of data, it contains a lazy-load database Rdata, plus a file Rdata.rds which contain a named character vector used by data() in the (unusual) event that it is used for such a package. Otherwise it is a copy of the data directory in the sources, with saved images re-compressed if R CMD INSTALL --resave-data was used.

Subdirectory demo supports the demo function, and is copied from the sources.

Subdirectory po contains (in subdirectories) compiled message catalogs.

4.1 Metadata

Directory Meta contains several files in .rds format, that is serialized R objects written by saveRDS. All packages have files Rd.rds, hsearch.rds, links.rds and package.rds. Packages with namespaces have a file nsInfo.rds, and those with data, demos or vignettes have data.rds, demo.rds or vignette.rds files.

The structure of these files (and their existence and names) is private to R, so the description here is for those trying to follow the R sources: there should be no reference to these files in non-base packages.

File package.rds is a dump of information extracted from the DESCRIPTION file. It is a list of several components. The first, 'DESCRIPTION', is a character vector, the DESCRIPTION file as read by read.dcf. Further elements ‘Depends’, ‘Suggests’, ‘Imports’, ‘Rdepends’ and ‘Rdepends2’ record the ‘Depends’, ’Suggests’ and ’Imports’ fields. These are all lists, and can be empty. The first three have an entry for each package named, each entry being a list of length 1 or 3, which element ‘name’ (the package name) and optional elements ‘op’ (a character string) and ’version’ (an object of class ‘package_version’). Element ‘Rdepends’ is used for the first version dependency on R, and ‘Rdepends2’ is a list of zero or more R version dependencies—each is a three-element list of the form described for packages. Element ‘Rdepends’ is no longer used, but it is still potentially needed so R < 2.7.0 can detect that the package was not installed for it.

File nsInfo.rds records a list, a parsed version of the NAMESPACE file.

File Rd.rds records a data frame with one row for each help file. The columns are ‘File’ (the file name with extension), ‘Name’ (the ‘\name’ section), ‘Type’ (from the optional ‘\docType’
section), ‘Title’, ‘Encoding’, ‘Aliases’, ‘Concepts’ and ‘Keywords’. All columns are character vectors apart from ‘Aliases’, which is a list of character vectors.

File hsearch.rds records the information to be used by ‘help.search’. This is a list of four unnamed elements which are character matrices for help files, aliases, keywords and concepts. All the matrices have columns ‘ID’ and ‘Package’ which are used to tie the aliases, keywords and concepts to a particular help file. The first element has further columns ‘LibPath’ (stored as "" and filled in what the file is loaded), ‘name’, ‘title’, ‘topic’ (the first alias, used when presenting the results as ‘pkgname::topic’) and ‘Encoding’.

File links.rds records a named character vector, the names being aliases and the values character strings of the form ""/pkgname/html/filename.html"

File data.rds records a two-column character matrix with columns of dataset names and titles from the corresponding help file. File demo.rds has the same structure for package demos.

File vignette.rds records a dataframe with one row for each ‘vignette’ (.RSnw file in inst/doc) and with columns ‘File’ (the full file path in the sources), ‘Title’, ‘PDF’ (the pathless file name of the installed PDF version, if present), ‘Depends’, ‘Keywords’ and ‘R’ (the pathless file name of the installed R code, if present).

4.2 Help

All installed packages, whether they had any .Rd files or not, have help and html directories. The latter normally only contains the single file 00Index.html, the package index which has hyperlinks to the help topics (if any).

Directory help contains files AnIndex, paths.rds and pkgname.rd[bx]. The latter two files are a lazy-load database of parsed .Rd files, accessed by tools:::fetchRdDB. File paths.rds is a saved character vector of the original path names of the .Rd files, used when updating the database.

File AnIndex is a two-column tab-delimited file: the first column contains the aliases defined in the help files and the second the basename (without the .Rd or .rd extension) of the file containing that alias. It is read by utils:::index.search to search for files matching a topic (alias), and read by scan in utils:::matchAvailableTopics, part of the completion system.

File aliases.rds is the same information as AnIndex as a named character vector (names the topics, values the file basename), for faster access.
5 Files

R provides many functions to work with files and directories: many of these have been added relatively recently to facilitate scripting in R and in particular the replacement of Perl scripts by R scripts in the management of R itself.

These functions are implemented by standard C/POSIX library calls, except on Windows. That means that filenames must be encoded in the current locale as the OS provides no other means to access the file system: increasingly filenames are stored in UTF-8 and the OS will translate filenames to UTF-8 in other locales. So using a UTF-8 locale gives transparent access to the whole file system.

Windows is another story. There the internal view of filenames is in UTF-16LE (so-called ‘Unicode’), and standard C library calls can only access files whose names can be expressed in the current codepage. To circumvent that restriction, there is a parallel set of Windows-specific calls which take wide-character arguments for filepaths. Much of the file-handling in R has been moved over to using these functions, so filenames can be manipulated in R as UTF-8 encoded character strings, converted to wide characters (which on Windows are UTF-16LE) and passed to the OS. The utilities `RC_fopen` and `filenameToWchar` help this process. Currently `file.copy` to a directory, `list.files`, `list.dirs` and `path.expand` work only with filepaths encoded in the current codepage.

All these functions do tilde expansion, in the same way as `path.expand`, with the deliberate exception of `Sys.glob`.

File names may be case sensitive or not: the latter is the norm on Windows and OS X, the former on other Unix-alikes. Note that this is a property of both the OS and the file system: it is often possible to map names to upper or lower case when mounting the file system. This can affect the matching of patterns in `list.files` and `Sys.glob`.

File names commonly contain spaces on Windows and OS X but not elsewhere. As file names are handled as character strings by R, spaces are not usually a concern unless file names are passed to other process, e.g. by a `system` call.

Windows has another couple of peculiarities. Whereas a POSIX file system has a single root directory (and other physical file systems are mounted onto logical directories under that root), Windows has separate roots for each physical or logical file system (‘volume’), organized under `drives` (with file paths starting D: for an ASCII letter, case-insensitively) and `network shares` (with paths like \netname\topdir\myfiles\a file. There is a current drive, and path names without a drive part are relative to the current drive. Further, each drive has a current directory, and relative paths are relative to that current directory, on a particular drive if one is specified. So D:dir\file and D: are valid path specifications (the last being the current directory on drive D:).
6 Graphics

R’s graphics internals were re-designed to enable multiple graphics systems to be installed on top on the graphics ‘engine’ – currently there are two such systems, one supporting ‘base’ graphics (based on that in S and whose R code\(^1\) is in package graphics) and one implemented in package grid.


At the lowest level is a graphics device, which manages a plotting surface (a screen window or a representation to be written to a file). This implements a set of graphics primitives, to ‘draw’

- a circle, optionally filled
- a rectangle, optionally filled
- a line
- a set of connected lines
- a polygon, optionally filled
- a paths, optionally filled using a winding rule
- text
- a raster image (optional)
- and to set a clipping rectangle

as well as requests for information such as

- the width of a string if plotted
- the metrics (width, ascent, descent) of a single character
- the current size of the plotting surface

and requests/opportunities to take action such as

- start a new ‘page’, possibly after responding to a request to ask the user for confirmation.
- return the position of the device pointer (if any).
- when a device become the current device or stops being the current device (this is usually used to change the window title on a screen device).
- when drawing starts or finishes (e.g. used to flush graphics to the screen when drawing stops).
- wait for an event, for example a mouse click or keypress.
- an ‘onexit’ action, to clean up if plotting is interrupted (by an error or by the user).
- capture the current contents of the device as a raster image.
- close the device.

The device also sets a number of variables, mainly Boolean flags indicating its capabilities. Devices work entirely in ‘device units’ which are up to its developer: they can be in pixels, big points (1/72 inch), twips, . . ., and can differ\(^2\) in the ‘x’ and ‘y’ directions.

The next layer up is the graphics ‘engine’ that is the main interface to the device (although the graphics subsystems do talk directly to devices). This is responsible for clipping lines, rectangles and polygons, converting the \texttt{pch} values \(0\ldots26\) to sets of lines/circles, centring (and

---

\(^1\) The C code is in files \texttt{base.c}, \texttt{graphics.c}, \texttt{par.c}, \texttt{plot.c} and \texttt{plot3d.c} in directory \texttt{src/main}.

\(^2\) although that needs to be handled carefully, as for example the \texttt{circle} callback is given a radius (and that should be interpreted as in the x units).
otherwise adjusting) text, rendering mathematical expressions (‘plotmath’) and mapping colour
descriptions such as names to the internal representation.

Another function of the engine is to manage display lists and snapshots. Some but not all
instances of graphics devices maintain display lists, a ‘list’ of operations that have been performed
on the device to produce the current plot (since the device was opened or the plot was last cleared,
e.g. by plot.new). Screen devices generally maintain a display list to handle repaint and resize
events whereas file-based formats do not—display lists are also used to implement dev.copy() and
friends. The display list is a pairlist of .Internal (base graphics) or .Call.graphics (grid
graphics) calls, which means that the C code implementing a graphics operation will be re-called
when the display list is replayed: apart from the part which records the operation if successful.

Snapshots of the current graphics state are taken by GEcreateSnapshot and replayed later
in the session by GEplaySnapshot. These are used by recordPlot(), replayPlot() and the
GUI menus of the windows() device. The ‘state’ includes the display list.

The top layer comprises the graphics subsystems. Although there is provision for 24 subsys-
tems since about 2001, currently still only two exist, ‘base’ and ‘grid’. The base subsystem is
registered with the engine when R is initialized, and unregistered (via KillAllDevices) when
an R session is shut down. The grid subsystem is registered in its .onLoad function and unreg-
istered in the .onUnload function. The graphics subsystem may also have ‘state’ information
saved in a snapshot (currently base does and grid does not).

Package grDevices was originally created to contain the basic graphics devices (although
X11 is in a separate load-on-demand module because of the volume of external libraries it brings
in). Since then it has been used for other functionality that was thought desirable for use with
grid, and hence has been transferred from package graphics to grDevices. This is principally
cconcerned with the handling of colours and recording and replaying plots.

6.1 Graphics Devices

R ships with several graphics devices, and there is support for third-party packages to provide
additional devices—several packages now do. This section describes the device internals from
the viewpoint of a would-be writer of a graphics device.

6.1.1 Device structures

There are two types used internally which are pointers to structures related to graphics devices.

The DevDesc type is a structure defined in the header file R_ext/GraphicsDevice.h (which is
included by R_ext/GraphicsEngine.h). This describes the physical characteristics of a device,
the capabilities of the device driver and contains a set of callback functions that will be used
by the graphics engine to obtain information about the device and initiate actions (e.g. a new
page, plotting a line or some text). Type pDevDesc is a pointer to this type.

The following callbacks can be omitted (or set to the null pointer, their default value) when
appropriate default behaviour will be taken by the graphics engine: activate, cap, deactivate,
locator, holdflush (API version 9), mode, newFrameConfirm, path, raster and size.

The relationship of device units to physical dimensions is set by the element ipr of the

The GEDevDesc type is a structure defined in R_ext/GraphicsEngine.h (with comments in
the file) as

typedef struct _GEDevDesc GEDevDesc;
struct _GEDevDesc {
   pDevDesc dev;
   Rboolean displayListOn;
   SEXP displayList;

So this is essentially a device structure plus information about the device maintained by the graphics engine and normally\(^3\) visible to the engine and not to the device. Type `pgGEDevDesc` is a pointer to this type.

The graphics engine maintains an array of devices, as pointers to `GEDevDesc` structures. The array is of size 64 but the first element is always occupied by the "null device" and the final element is kept as NULL as a sentinel.\(^4\) This array is reflected in the R variable `.Devices`. Once a device is killed its element becomes available for reallocation (and its name will appear as "" in `.Devices`). Exactly one of the devices is ‘active’: this is the the null device if no other device has been opened and not killed.

Each instance of a graphics device needs to set up a `GEDevDesc` structure by code very similar to

```c
pGEDevDesc gdd;

R_GE_checkVersionOrDie(R_GE_version);
R_CheckDeviceAvailable();
BEGIN_SUSPEND_INTERRUPTS {
    pDevDesc dev;
    /* Allocate and initialize the device driver data */
    if (!((dev = (pDevDesc) calloc(1, sizeof(DevDesc))))
        return 0; /* or error() */
    /* set up device driver or free 'dev' and error() */
    gdd = GEcreateDevDesc(dev);
    GEaddDevice2(gdd, "dev_name");
} END_SUSPEND_INTERRUPTS;
```

The `DevDesc` structure contains a `void *` pointer `deviceSpecific` which is used to store data specific to the device. Setting up the device driver includes initializing all the non-zero elements of the `DevDesc` structure.

Note that the device structure is zeroed when allocated: this provides some protection against future expansion of the structure since the graphics engine can add elements that need to be non-NULL/non-zero to be ‘on’ (and the structure ends with 64 reserved bytes which will be zeroed and allow for future expansion).

Rather more protection is provided by the version number of the engine/device API, `R_GE_version` defined in `R_ext/GraphicsEngine.h` together with access functions

```c
int R_GE_getVersion(void);
void R_GE_checkVersionOrDie(int version);
```

If a graphics device calls `R_GE_checkVersionOrDie(R_GE_version)` it can ensure it will only be used in versions of R which provide the API it was designed for and compiled against.

\(^3\) It is possible for the device to find the `GEDevDesc` which points to its `DevDesc`, and this is done often enough that there is a convenience function `desc2GEDesc` to do so.

\(^4\) Calling `R_CheckDeviceAvailable()` ensures there is a free slot or throws an error.
6.1.2 Device capabilities

The following ‘capabilities’ can be defined for the device’s DevDesc structure.

- **canChangeGamma** – Rboolean: can the display gamma be adjusted? This is now ignored, as gamma support has been removed.
- **canHadj** – integer: can the device do horizontal adjustment of text via the text callback, and if so, how precisely? 0 = no adjustment, 1 = \{0, 0.5, 1\} (left, centre, right justification) or 2 = continuously variable (in [0,1]) between left and right justification.
- **canGenMouseDown** – Rboolean: can the device handle mouse down events? This flag and the next three are not currently used by R, but are maintained for back compatibility.
- **canGenMouseMove** – Rboolean: ditto for mouse move events.
- **canGenMouseUp** – Rboolean: ditto for mouse up events.
- **canGenKeybd** – Rboolean: ditto for keyboard events.
- **hasTextUTF8** – Rboolean: should non-symbol text be sent (in UTF-8) to the textUTF8 and strWidthUTF8 callbacks, and sent as Unicode points (negative values) to the metricInfo callback?
- **wantSymbolUTF8** – Rboolean: should symbol text be handled in UTF-8 in the same way as other text? Requires textUTF8 = TRUE.
- **haveTransparency**: does the device support semi-transparent colours?
- **haveTransparentBg**: can the background be fully or semi-transparent?
- **haveRaster**: is there support for rendering raster images?
- **haveCapture**: is there support for grid::grid.cap?
- **haveLocator**: is there an interactive locator?

The last three can often be deduced to be false from the presence of NULL entries instead of the corresponding functions.

6.1.3 Handling text

Handling text is probably the hardest task for a graphics device, and the design allows for the device to optionally indicate that it has additional capabilities. (If the device does not, these will if possible be handled in the graphics engine.)

The three callbacks for handling text that must be in all graphics devices are text, strWidth and metricInfo with declarations

```c
void text(double x, double y, const char *str, double rot, double hadj, pGcontext gc, pDevDesc dd);

double strWidth(const char *str, pGEcontext gc, pDevDesc dd);

void metricInfo(int c, pGEcontext gc, double* ascent, double* descent, double* width, pDevDesc dd);
```

The ‘gc’ parameter provides the graphics context, most importantly the current font and fontsize, and ‘dd’ is a pointer to the active device’s structure.

The **text** callback should plot ‘str’ at ‘(x, y)’\(^5\) with an anti-clockwise rotation of ‘rot’ degrees. (For ‘hadj’ see below.) The interpretation for horizontal text is that the baseline is at y and the start is a x, so any left bearing for the first character will start at x.

\(^5\) in device coordinates
The `strWidth` callback computes the width of the string which it would occupy if plotted horizontally in the current font. (Width here is expected to include both (preferably) or neither of left and right bearings.)

The `metricInfo` callback computes the size of a single character: `ascent` is the distance it extends above the baseline and `descent` how far it extends below the baseline. `width` is the amount by which the cursor should be advanced when the character is placed. For `ascent` and `descent` this is intended to be the bounding box of the ‘ink’ put down by the glyph and not the box which might be used when assembling a line of conventional text (it needs to be for e.g. `hat(b)` to work correctly). However, the `width` is used in `plotmath` to advance to the next character, and so needs to include left and right bearings.

The interpretation of ‘c’ depends on the locale. In a single-byte locale values 32...255 indicate the corresponding character in the locale (if present). For the symbol font (as used by `graphics::par(font=5)`, `grid::gpar(fontface=5)` and by ‘plotmath’), values 32...126, 161...239, 241...254 indicate glyphs in the Adobe Symbol encoding. In a multibyte locale, `c` represents a Unicode point (except in the symbol font). So the function needs to include code like

```r
Rboolean Unicode = mbcslocale && (gc->fontface != 5);
if (c < 0) { Unicode = TRUE; c = -c; }
if(Unicode) UniCharMetric(c, ...); else CharMetric(c, ...);
```

In addition, if device capability `hasTextUTF8` (see below) is true, Unicode points will be passed as negative values: the code snippet above shows how to handle this. (This applies to the symbol font only if device capability `wantSymbolUTF8` is true.)

If possible, the graphics device should handle clipping of text. It indicates this by the structure element `canClip` which if true will result in calls to the callback `clip` to set the clipping region. If this is not done, the engine will clip very crudely (by omitting any text that does not appear to be wholly inside the clipping region).

The device structure has an integer element `canHadj`, which indicates if the device can do horizontal alignment of text. If this is one, argument ‘hadj’ to `text` will be called as 0, 0.5, 1 to indicate left-, centre- and right-alignment at the indicated position. If it is two, continuous values in the range [0, 1] are assumed to be supported.

Capability `hasTextUTF8` if true, it has two consequences. First, there are callbacks `textUTF8` and `strWidthUTF8` that should behave identically to `text` and `strWidth` except that ‘str’ is assumed to be in UTF-8 rather than the current locale’s encoding. The graphics engine will call these for all text except in the symbol font. Second, Unicode points will be passed to the `metricInfo` callback as negative integers. If your device would prefer to have UTF-8-encoded symbols, define `wantSymbolUTF8` as well as `hasTextUTF8`. In that case text in the symbol font is sent to `textUTF8` and `strWidthUTF8`.

Some devices can produce high-quality rotated text, but those based on bitmaps often cannot. Those which can should set `useRotatedTextInContour` to be true from graphics API version 4.

Several other elements relate to the precise placement of text by the graphics engine:

```r
double xCharOffset;
double yCharOffset;
double yLineBias;
double cra[2];
```

These are more than a little mysterious. Element `cra` provides an indication of the character size, `par("cra")` in base graphics, in device units. The mystery is what is meant by ‘character size’: which character, which font at which size? Some help can be obtained by looking at what this is used for. The first element, ‘width’, is not used by R except to set the graphical parameters. The second, ‘height’, is use to set the line spacing, that is the relationship between `par("mai")` and `par("mai")` and so on. It is suggested that a good choice is
Chapter 6: Graphics

\[ dd->cra[0] = 0.9 \times \text{fnsize}; \]
\[ dd->cra[1] = 1.2 \times \text{fnsize}; \]

where ‘\text{fnsize}’ is the ‘size’ of the standard font (\text{cex}=1) on the device, in device units. So for
a 12-point font (the usual default for graphics devices), ‘\text{fnsize}’ should be 12 points in device units.

The remaining elements are yet more mysterious. The \text{postscript}() device says

\[
\begin{verbatim}
/* Character Addressing Offsets */
/* These offsets should center a single */
/* plotting character over the plotting point. */
/* Pure guesswork and eyeballing ... */

dd->xCharOffset = 0.4900;
dd->yCharOffset = 0.3333;
dd->yLineBias = 0.2;
\end{verbatim}
\]

It seems that \text{xCCharOffset} is not currently used, and \text{yCharOffset} is used by the base graphics
system to set vertical alignment in \text{text()} when \text{pos} is specified, and in \text{identify}(). It is
occasionally used by the graphic engine when attempting exact centring of text, such as character
string values of \text{pch} in \text{points()} or \text{grid.points()}—however, it is only used when precise
character metric information is not available or for multi-line strings.

\text{yLineBias} is used in the base graphics system in \text{axis()} and \text{mtext()} to provide a default
for their ‘\text{padj}’ argument.

6.1.4 Conventions

The aim is to make the (default) output from graphics devices as similar as possible. Generally
people follow the model of the \text{postscript} and \text{pdf} devices (which share most of their internal
code).

The following conventions have become established:

- The default size of a device should be 7 inches square.
- There should be a ‘\text{pointsize}’ argument which defaults to 12, and it should give the
  pointsize in big points (1/72 inch). How exactly this is interpreted is font-specific, but it
  should use a font which works with lines packed 1/6 inch apart, and looks good with lines
  1/5 inch apart (that is with 2pt leading).
- The default font family should be a sans serif font, e.g Helvetica or similar (e.g. Arial on
  Windows).
- \text{lwd = 1} should correspond to a line width of 1/96 inch. This will be a problem with pixel-
  based devices, and generally there is a minimum line width of 1 pixel (although this may
  not be appropriate where anti-aliasing of lines is used, and \text{cairo} prefers a minimum of 2
  pixels).
- Even very small circles should be visible, e.g. by using a minimum radius of 1 pixel or
  replacing very small circles by a single filled pixel.
- How RGB colour values will be interpreted should be documented, and preferably be sRGB.
- The help page should describe its policy on these conventions.

These conventions are less clear-cut for bitmap devices, especially where the bitmap format
does not have a design resolution.

The interpretation of the line texture (\text{par("lty")}) is described in the header
\text{GraphicsEngine.h} and in the help for \text{par}: note that the ‘scale’ of the pattern should be
proportional to the line width (at least for widths above the default).
6.1.5 ‘Mode’

One of the device callbacks is a function `mode`, documented in the header as

* device_Mode is called whenever the graphics engine
* starts drawing (mode=1) or stops drawing (mode=0)
* GMode (in graphics.c) also says that
* mode = 2 (graphical input on) exists.
* The device is not required to do anything

Since mode = 2 has only recently been documented at device level. It could be used to change the graphics cursor, but devices currently do that in the locator callback. (In base graphics the mode is set for the duration of a locator call, but if type != "n" is switched back for each point whilst annotation is being done.)

Many devices do indeed do nothing on this call, but some screen devices ensure that drawing is flushed to the screen when called with mode = 0. It is tempting to use it for some sort of buffering, but note that ‘drawing’ is interpreted at quite a low level and a typical single figure will stop and start drawing many times. The buffering introduced in the X11() device makes use of mode = 0 to indicate activity: it updates the screen after ca 100ms of inactivity.

This callback need not be supplied if it does nothing.

6.1.6 Graphics events

Graphics devices may be designed to handle user interaction: not all are.

Users may use `grDevices::setGraphicsEventEnv` to set the eventEnv environment in the device driver to hold event handlers. When the user calls `grDevices::getGraphicsEvent`, R will take three steps. First, it sets the device driver member `gettingEvent` to true for each device with a non-NULL eventEnv entry, and calls `initEvent(dd, true)` if the callback is defined. It then enters an event loop. Each time through the loop R will process events once, then check whether any device has set the result member of eventEnv to a non-NULL value, and will save the first such value found to be returned. C functions `doMouseEvent` and `doKeybd` are provided to call the R event handlers `onMouseDown`, `onMouseMove`, `onMouseUp`, and `onKeybd` and set eventEnv$result during this step. Finally, `initEvent` is called again with `init=false` to inform the the devices that the loop is done, and the result is returned to the user.

6.1.7 Specific devices

Specific devices are mostly documented by comments in their sources, although for devices of many years’ standing those comments can be in need of updating. This subsection is a repository of notes on design decisions.

6.1.7.1 X11()

The X11(type="Xlib") device dates back to the mid 1990’s and was written then in Xlib, the most basic X11 toolkit. It has since optionally made use of a few features from other toolkits: libXt is used to read X11 resources, and libXmu is used in the handling of clipboard selections.

Using basic Xlib code makes drawing fast, but is limiting. There is no support of translucent colours (that came in the Xrender toolkit of 2000) nor for rotated text (which R implements by rendering text to a bitmap and rotating the latter).

The hinting for the X11 window asks for backing store to be used, and some windows managers may use it to handle repaints, but it seems that most repainting is done by replaying the display list (and here the fast drawing is very helpful).

There are perennial problems with finding fonts. Many users fail to realize that fonts are a function of the X server and not of the machine that R is running on. After many difficulties, R tries first to find the nearest size match in the sizes provided for Adobe fonts in the standard
75dpi and 100dpi X11 font packages—even that will fail to work when users of near-100dpi screens have only the 75dpi set installed. The 75dpi set allows sizes down to 6 points on a 100dpi screen, but some users do try to use smaller sizes and even 6 and 8 point bitmapped fonts do not look good.

Introduction of UTF-8 locales has caused another wave of difficulties. X11 has very few genuine UTF-8 fonts, and produces composite fontsets for the iso10646-1 encoding. Unfortunately these seem to have low coverage apart from a few monospaced fonts in a few sizes (which are not suitable for graph annotation), and where glyphs are missing what is plotted is often quite unsatisfactory.

The current approach is to make use of more modern toolkits, namely cairo for rendering and Pango for font management—because these are associated with Gtk+2 they are widely available. Cairo supports translucent colours and alpha-blending (via Xrender), and anti-aliasing for the display of lines and text. Pango’s font management is based on fontconfig and somewhat mysterious, but it seems mainly to use Type 1 and TrueType fonts on the machine running R and send grayscale bitmaps to cairo.

### 6.1.7.2 windows()

The windows() device is a family of devices: it supports plotting to Windows (enhanced) metafiles, BMP, JPEG, PNG and TIFF files as well as to Windows printers.

In most of these cases the primary plotting is to a bitmap: this is used for the (default) buffering of the screen device, which also enables the current plot to be saved to BMP, JPEG, PNG or TIFF (it is the internal bitmap which is copied to the file in the appropriate format).

The device units are pixels (logical ones on a metafile device).

The code was originally written by Guido Masarotto with extensive use of macros, which can make it hard to disentangle.

For a screen device, xd->gawin is the canvas of the screen, and xd->bm is the off-screen bitmap. So macro DRAW arranges to plot to xd->bm, and if buffering is off, also to xd->gawin. For all other device, xd->gawin is the canvas, a bitmap for the jpeg() and png() device, and an internal representation of a Windows metafile for the win.metafile() and win.print device. Since ‘plotting’ is done by Windows GDI calls to the appropriate canvas, its precise nature is hidden by the GDI system.

Buffering on the screen device is achieved by running a timer, which when it fires copies the internal bitmap to the screen. This is set to fire every 500ms (by default) and is reset to 100ms after plotting activity.

Repaint events are handled by copying the internal bitmap to the screen canvas (and then reinitializing the timer), unless there has been a resize. Resizes are handled by replaying the display list: this might not be necessary if a fixed canvas with scrollbars is being used, but that is the least popular of the three forms of resizing.

Text on the device has moved to ‘Unicode’ (UCS-2) in recent years. UTF-8 is requested (hasTextUTF8 = TRUE) for standard text, and converted to UCS-2 in the plotting functions in file src/extra/graphapp/gdraw.c. However, GDI has no support for Unicode symbol fonts, and symbols are handled in Adobe Symbol encoding.

There is support for translucent colours (with alpha channel between 0 and 255) was introduced on the screen device and bitmap devices. This is done by drawing on a further internal bitmap, xd->bm2, in the opaque version of the colour then alpha-blending that bitmap to xd->bm. The alpha-blending routine is in a separate DLL, msimg32.d11, which is loaded on first use. As small a rectangular region as reasonably possible is alpha-blended (this is rectangle

---

6 It is technically possible to use alpha-blending on metafile devices such as printers, but it seems few drivers have support for this.
r in the code), but things like mitre joins make estimation of a tight bounding box too much work for lines and polygonal boundaries. Translucent-coloured lines are not common, and the performance seems acceptable.

The support for a transparent background in `png()` predates full alpha-channel support in `libpng` (let alone in PNG viewers), so makes use of the limited transparency support in earlier versions of PNG. Where 24-bit colour is used, this is done by marking a single colour to be rendered as transparent. R chose "#fdfefd", and uses this as the background colour (in `GA_NewPage` if the specified background colour is transparent (and all non-opaque background colours are treated as transparent). So this works by marking that colour in the PNG file, and viewers without transparency support see a slightly-off-white background, as if there were a near-white canvas. Where a palette is used in the PNG file (if less than 256 colours were used) then this colour is recorded with full transparency and the remaining colours as opaque. If 32-bit colour were available then we could add a full alpha channel, but this is dependent on the graphics hardware and undocumented properties of GDI.

### 6.2 Colours

Devices receive colours as a `typedef rcolor` (an `unsigned int`) defined in the header `R_ext/GraphicsEngine.h`). The 4 bytes are R, G, B and alpha from least to most significant. So each of RGB has 256 levels of luminosity from 0 to 255. The alpha byte represents opacity, so value 255 is fully opaque and 0 fully transparent: many but not all devices handle semi-transparent colours.

Colors can be created in C via the macro `R_RGBA`, and a set of macros are defined in `R_ext/GraphicsDevice.h` to extract the various components.

Colours in the base graphics system were originally adopted from S (and before that the GRZ library from Bell Labs), with the concept of a (variable-sized) palette of colours referenced by numbers ‘1...N’ plus ‘0’ (the background colour of the current device). R introduced the idea of referring to colours by character strings, either in the forms ‘#RRGGBB’ or ‘#RRGGBBAA’ (representing the bytes in hex) as given by function `rgb()` or via names: the 657 known names are given in the character vector `colors` and in a table in file `colors.c` in package `grDevices`. Note that semi-transparent colours are not 'premultiplied', so 50% transparent white is ‘#ffffff80’.

Integer or character `NA` colours are mapped internally to transparent white, as is the character string "NA".

The handling of negative colour numbers was undefined (and inconsistent) prior to R 3.0.0, which made them an error. Colours greater than ‘N’ are wrapped around, so that for example with the default palette of size 8, colour ‘10’ is colour ‘2’ in the palette.

Integer colours have been used more widely than the base graphics sub-system, as they are supported by package `grid` and hence by `lattice` and `ggplot2`. (They are also used by package `rgl`.) `grid` did re-define colour ‘0’ to be transparent white, but `rgl` used `col2rgb` and hence the background colour of base graphics.

Note that positive integer colours refer to the current palette and colour ‘0’ to the current device (and a device is opened if needs be). These are mapped to type `rcolor` at the time of use: this matters when re-playing the display list, e.g. when a device is resized or `dev.copy` is used. The palette should be thought of as per-session: it is stored in package `grDevices`.

The convention is that devices use the colorspace ‘sRGB’. This is an industry standard: it is used by Web browsers and JPEGs from all but high-end digital cameras. The interpretation is a matter for graphics devices and for code that manipulates colours, but not for the graphics engine or subsystems.

R uses a painting model similar to PostScript and PDF. This means that where shapes (circles, rectangles and polygons) can both be filled and have a stroked border, the fill should
be painted first and then the border (or otherwise only half the border will be visible). Where both the fill and the border are semi-transparent there is some room for interpretation of the intention. Most devices first paint the fill and then the border, alpha-blending at each step. However, PDF does some automatic grouping of objects, and when the fill and the border have the same alpha, they are painted onto the same layer and then alpha-blended in one step. (See p. 569 of the PDF Reference Sixth Edition, version 1.7. Unfortunately, although this is what the PDF standard says should happen, it is not correctly implemented by some viewers.)

The mapping from colour numbers to type rcolor is primarily done by function RGBpar3: this is exported from the R binary but linked to code in package grDevices. The first argument is a SEXP pointing to a character, integer or double vector, and the second is the rcolor value for colour 0 (or "0"). C entry point RGBpar is a wrapper that takes 0 to be transparent white: it is often used to set colour defaults for devices. The R-level wrapper is col2rgb.

There is also R_GE_str2col which takes a C string and converts to type rcolor: "0" is converted to transparent white.

There is a R-level conversion of colours to '##RRGGBBAA' by image.default(useRaster = TRUE).

The other color-conversion entry point in the API is name2col which takes a colour name (a C string) and returns a value of type rcolor. This handles "NA", "transparent" and the 657 colours known to the R function colors().

6.3 Base graphics

The base graphics system was migrated to package graphics in R 3.0.0: it was previously implemented in files in src/main.

For historical reasons it is largely implemented in two layers. Files plot.c, plot3d.c and par.c contain the code for the around 30 .External calls that implement the basic graphics operations. This code then calls functions with names starting with G and declared in header Rgraphics.h in file graphics.c, which in turn call the graphics engine (whose functions almost all have names starting with GE).

A large part of the infrastructure of the base graphics subsystem are the graphics parameters (as set/read by par()). These are stored in a GPar structure declared in the private header Graphics.h. This structure has two variables (state and valid) tracking the state of the base subsystem on the device, and many variables recording the graphics parameters and functions of them.

The base system state is contained in baseSystemState structure defined in R_ext/GraphicsBase.h. This contains three GPar structures and a Boolean variable used to record if plot.new() (or persp) has been used successfully on the device.

The three copies of the GPar structure are used to store the current parameters (accessed via gpptr), the ‘device copy’ (accessed via dppttr) and space for a saved copy of the ‘device copy’ parameters. The current parameters are, clearly, those currently in use and are copied from the ‘device copy’ whenever plot.new() is called (whether or not that advances to the next ‘page’). The saved copy keeps the state when the device was last completely cleared (e.g. when plot.new() was called with par(new=TRUE)), and is used to replay the display list.

The separation is not completely clean: the ‘device copy’ is altered if a plot with log scale(s) is set up via plot.window().

There is yet another copy of most of the graphics parameters in static variables in graphics.c which are used to preserve the current parameters across the processing of inline parameters in high-level graphics calls (handled by ProcessInlinePars).

Snapshots of the base subsystem record the ‘saved device copy’ of the GPar structure.
6.3.1 Arguments and parameters

There is an unfortunate confusion between some of the graphical parameters (as set by \texttt{par}) and arguments to base graphic functions of the same name. This description may help set the record straight.

Most of the high-level plotting functions accept graphical parameters as additional arguments, which are then often passed to lower-level functions if not already named arguments (which is the main source of confusion).

Graphical parameter \texttt{bg} is the background colour of the plot. Argument \texttt{bg} refers to the fill colour for the filled symbols 21 to 25. It is an argument to the function \texttt{plot.xy}, but normally passed by the default method of \texttt{points}, often from a \texttt{plot} method.

Graphics parameters \texttt{cex}, \texttt{col}, \texttt{lty}, \texttt{lwd} and \texttt{pch} also appear as arguments of \texttt{plot.xy} and so are often passed as arguments from higher-level plot functions such as \texttt{lines}, \texttt{points} and \texttt{plot} methods. They appear as arguments of \texttt{legend}, \texttt{col}, \texttt{lty} and \texttt{lwd} are arguments of \texttt{arrows} and \texttt{segments}. When used as arguments they can be vectors, recycled to control the various lines, points and segments. When set a graphical parameters they set the default rendering: in addition \texttt{par(cex=)} sets the overall character expansion which subsequent calls (as arguments or on-line graphical parameters) multiply.

The handling of missing values differs in the two classes of uses. Generally these are errors when used in \texttt{par} but cause the corresponding element of the plot to be omitted when used as an element of a vector argument. Originally the interpretation of arguments was mainly left to the device, but as from R 3.0.0 some of this is pre-empted in the graphics engine (but for example the handling of \texttt{lwd = 0} remains device-specific, with some interpreting it as a ‘thinnest possible’ line).

6.4 Grid graphics

[At least pointers to documentation.]
Chapter 7: GUI consoles

The standard R front-ends are programs which run in a terminal, but there are several ways to provide a GUI console.

This can be done by a package which is loaded from terminal-based R and launches a console as part of its startup code or by the user running a specific function: package Rcmdr is a well-known example with a Tk-based GUI.

There used to be a Gtk-based console invoked by R --gui=GNOME: this relied on special-casing in the front-end shell script to launch a different executable. There still is R --gui=Tk, which starts terminal-based R and runs tcltk::tkStartGui() as part of the modified startup sequence.

However, the main way to run a GUI console is to launch a separate program which runs embedded R: this is done by Rgui.exe on Windows and R.app on OS X. The first is an integral part of R and the code for the console is currently in R.dll.

7.1 R.app

R.app is a OS X application which provides a console. Its sources are a separate project\(^1\), and its binaries link to an R installation which it runs as a dynamic library libR.dylib. The standard CRAN distribution of R for OS X bundles the GUI and R itself, but installing the GUI is optional and either component can be updated separately.

R.app relies on libR.dylib being in a specific place, and hence on R having been built and installed as a Mac OS X ‘framework’. Specifically, it uses /Library/Frameworks/R.framework/R. This is a symbolic link, as frameworks can contain multiple versions of R. It eventually resolves to /Library/Frameworks/R.framework/Versions/Current/Resources/lib/libR.dylib, which is (in the CRAN distribution) a ‘fat’ binary containing multiple sub-architectures.

OS X applications are directory trees: each R.app contains a front-end written in Objective-C for one sub-architecture: in the standard distribution there are separate applications for 32- and 64-bit Intel architectures.

Originally the R sources contained quite a lot of code used only by the OS X GUI, but by R 3.0.0 this was been migrated to the R.app sources.

R.app starts R as an embedded application with a command-line which includes --gui=aqua (see below). It uses most of the interface pointers defined in the header Rinterface.h, plus a private interface pointer in file src/main/sysutils.c. It adds an environment it names tools:RGUI to the second position in the search path. This contains a number of utility functions used to support the menu items, for example package.manager(), plus functions q() and quit() which mask those in package base—the custom versions save the history in a way specific to R.app.

There is a configure option --with-aqua for R which customizes the way R is built: this is distinct from the --enable-R-framework option which causes make install to install R as the framework needed for use with R.app. (The option --with-aqua is the default on OS X.) It sets the macro HAVE_AQUA in config.h and the make variable BUILD_AQUA_TRUE. These have several consequences:

- The quartz() device is built (other than as a stub) in package grDevices: this needs an Objective-C compiler. Then quartz() can be used with terminal R provided the latter has access to the OS X screen.
- File src/unix/aqua.c is compiled. This now only contains an interface pointer for the quartz() device(s).

\(^1\) an Xcode project, in SVN at https://svn.r-project.org/R-packages/trunk/Mac-GUI
• capabilities("aqua") is set to TRUE.
• The default path for a personal library directory is set as "~/Library/R/x.y/library."
• There is support for setting a ‘busy’ indicator whilst waiting for system() to return.
• R_ProcessEvents is inhibited in a forked child from package parallel. The associated callback in R.app does things which should not be done in a child, and forking forks the whole process including the console.
• There is support for starting the embedded R with the option --gui=aqua: when this is done the global C variable useaqua is set to a true value. This has consequences:
  • The R session is asserted to be interactive via R_Interactive.
  • .Platform$GUI is set to "AQUA". That has consequences:
    • The environment variable DISPLAY is set to ‘:0’ if not already set.
    • /usr/local/bin is appended to PATH since that is where gfortran is installed.
    • The default HTML browser is switched to the one in R.app.
    • Various widgets are switched to the versions provided in R.app: these include graphical menus, the data editor (but not the data viewer used by View()) and the workspace browser invoked by browseEnv().
    • The grDevices package when loaded knows that it is being run under R.app and so informs any quartz devices that a Quartz event loop is already running.
  • The use of the OS’s system function (including by system() and system2(), and to launch editors and pagers) is replaced by a version in R.app (which by default just calls the OS’s system with various signal handlers reset).
• If either R was started by --gui=aqua or R is running in a terminal which is not of type ‘dumb’, the standard output to files stdout and stderr is directed through the C function Rstd_WriteConsoleEx. This uses ANSI terminal escapes to render lines sent to stderr as bold on stdout.
• For historical reasons the startup option -psn is allowed but ignored. (It seems that in 2003, ‘r27492’, this was added by Finder.)


8 Tools

The behavior of R CMD check can be controlled through a variety of command line arguments and environment variables.

There is an internal --install=value command line argument not shown by R CMD check --help, with possible values:

check:file
Assume that installation was already performed with stdout/stderr to file, the contents of which need to be checked (without repeating the installation). This is useful for checks applied by repository maintainers: it reduces the check time by the installation time given that the package has already been installed. In this case, one also needs to specify where the package was installed to using command line option --library.

fake
Fake installation, and turn off the run-time tests.

skip
Skip installation, e.g., when testing recommended packages bundled with R.

no
The same as --no-install: turns off installation and the tests which require the package to be installed.

The following environment variables can be used to customize the operation of check: a convenient place to set these is the check environment file (default, ~/.R/check.Renviron).

_/R_CHECK_ALL_NON_ISO_C_/  
If true, do not ignore compiler (typically GCC) warnings about non ISO C code in system headers. Note that this may also show additional ISO C++ warnings. Default: false.

_/R_CHECK_FORCE_SUGGESTS_/  
If true, give an error if suggested packages are not available. Default: true (but false for CRAN submission checks).

_/R_CHECK_RD_CONTENTS_/  
If true, check Rd files for auto-generated content which needs editing, and missing argument documentation. Default: true.

_/R_CHECK_RD_LINE_WIDTHS_/  
If true, check Rd line widths in usage and examples sections. Default: false (but true for CRAN submission checks).

_/R_CHECK_RD_STYLE_/  
If true, check whether Rd usage entries for S3 methods use the full function name rather than the appropriate \method markup. Default: true.

_/R_CHECK_RD_XREFS_/  
If true, check the cross-references in .Rd files. Default: true.

_/R_CHECK_SUBDIRS_NOCASE_/  
If true, check the case of directories such as R and man. Default: true.

_/R_CHECK_SUBDIRS STRICT_/  
Initial setting for --check-subdirs. Default: ‘default’ (which checks only tarballs, and checks in the src only if there is no configure file).

_/R_CHECK_USE_CODETOOLS_/  
If true, make use of the codetools package, which provides a detailed analysis of visibility of objects (but may give false positives). Default: true (if recommended packages are installed).
_R_CHECK_USE_INSTALL_LOG_
If true, record the output from installing a package as part of its check to a log file (00install.out by default), even when running interactively. Default: true.

_R_CHECK_VIGNETTES_NLINES_
Maximum number of lines to show at the bottom of the output when reporting errors in running vignettes. Default: 10.

_R_CHECK_CODOC_S4_METHODS_
Control whether codoc() testing is also performed on S4 methods. Default: true.

_R_CHECK_DOT_INTERNAL_
Control whether the package code is scanned for .Internal calls, which should only be used by base (and occasionally by recommended) packages. Default: true.

_R_CHECK_EXECUTABLES_
Control checking for executable (binary) files. Default: true.

_R_CHECK_EXECUTABLES_EXCLUSIONS_
Control whether checking for executable (binary) files ignores files listed in the package’s BinaryFiles file. Default: true (but false for CRAN submission checks). However, most likely this package-level override mechanism will be removed eventually.

_R_CHECK_PERMISSIONS_
Control whether permissions of files should be checked. Default: true iff .Platform$OS.type == "unix".

_R_CHECK_FF_CALLS_
Allows turning off checkFF() testing. If set to ‘registration’, checks the registration information (number of arguments, correct choice of .C/.Fortran/.Call/.External) for such calls provided the package is installed. Default: true.

_R_CHECK_FF_DUP_
Controls checkFF(check_DUP) Default: true (and forced to be true for CRAN submission checks).

_R_CHECK_LICENSE_
Control whether/how license checks are performed. A possible value is ‘maybe’ (warn in case of problems, but not about standardizable non-standard license specs). Default: true.

_R_CHECK_RD_EXAMPLES_T_AND_F_
Control whether check_T_and_F() also looks for “bad” (global) ‘T’/’F’ uses in examples. Off by default because this can result in false positives.

_R_CHECK_RD_CHECKRD_MINLEVEL_
Controls the minimum level for reporting warnings from checkRd. Default: -1.

_R_CHECK_XREFS_REPOSITORIES_
If set to a non-empty value, a space-separated list of repositories to use to determine known packages. Default: empty, when the CRAN, Omegahat and Bioconductor repositories known to R is used.

_R_CHECK_SRC_MINUS_W_IMPLICIT_
Control whether installation output is checked for compilation warnings about implicit function declarations (as spotted by GCC with command line option -Wimplicit-function-declaration, which is implied by -Wall). Default: false.
_R_CHECK_SRC_MINUS_W_UNUSED_
Control whether installation output is checked for compilation warnings about unused code constituents (as spotted by GCC with command line option `-Wunused`, which is implied by `-Wall`). Default: true.

_R_CHECK_WALL_FORTRAN_
Control whether gfortran 4.0 or later `-Wall` warnings are used in the analysis of installation output. Default: false, even though the warnings are justifiable.

_R_CHECK_ASCII_CODE_
If true, check R code for non-ascii characters. Default: true.

_R_CHECK_ASCII_DATA_
If true, check data for non-ascii characters. Default: true.

_R_CHECK_COMPACT_DATA_
If true, check data for ascii and uncompressed saves, and also check if using bzip2 or xz compression would be significantly better. Default: true.

_R_CHECK_SKIP_ARCH_
Comma-separated list of architectures that will be omitted from checking in a multi-arch setup. Default: none.

_R_CHECK_SKIP_TESTS_ARCH_
Comma-separated list of architectures that will be omitted from running tests in a multi-arch setup. Default: none.

_R_CHECK_SKIP_EXAMPLES_ARCH_
Comma-separated list of architectures that will be omitted from running examples in a multi-arch setup. Default: none.

_R_CHECK_VC_DIRS_
Should the unpacked package directory be checked for version-control directories (.CVS, .svn . . .)? Default: true for tarballs.

_R_CHECK_PKG_SIZES_
Should du be used to find the installed sizes of packages? R CMD check does check for the availability of du, but this option allows the check to be overruled if an unsuitable command is found (including one that does not respect the `-k` flag to report in units of 1Kb, or reports in a different format – the GNU, OS X and Solaris du commands have been tested). Default: true if du is found.

_R_CHECK_DOC_SIZES_
Should qpdf be used to check the installed sizes of PDFs? Default: true if qpdf is found.

_R_CHECK_DOC_SIZES2_
Should gs be used to check the installed sizes of PDFs? This is slower than (and in addition to) the previous check, but does detect figures with excessive detail (often hidden by over-plotting) or bitmap figures with too high a resolution. Requires that R_GSCMD is set to a valid program, or gs (or on Windows, gswin32.exe or gswin64c.exe) is on the path. Default: false (but true for CRAN submission checks).

_R_CHECK_ALWAYS_LOG_VIGNETTE_OUTPUT_
By default the output from running the R code in the vignettes is kept only if there is an error. Default: false.

_R_CHECK_CLEAN_VIGN_TEST_
Should the vign_test directory be removed if the test is successful? Default: true.
_R_CHECK_REPLACING_IMPORTS_

Should warnings about replacing imports be reported? These sometimes come from auto-generated NAMESPACE files in other packages, but most often from importing the whole of a namespace rather than using importFrom. Default: false (but true for CRAN submission checks).

_R_CHECK_UNSAFE_CALLS_

Check for calls that appear to tamper with (or allow tampering with) already loaded code not from the current package: such calls may well contravene CRAN policies. Default: true.

_R_CHECK_TIMINGS_

Optionally report timings for installation, examples, tests and running/re-building vignettes as part of the check log. The format is ‘[as/bs]’ for the total CPU time (including child processes) ‘a’ and elapsed time ‘b’, except on Windows, when it is ‘[bs]’. In most cases timings are only given for ‘OK’ checks. Times with an elapsed component over 10 mins are reported in minutes (with abbreviation ‘m’). The value is the smallest numerical value in elapsed seconds that should be reported: non-numerical values indicate that no report is required, a value of ‘0’ that a report is always required. Default: "". (10 for CRAN checks.)

_R_CHECK_INSTALL_DEPENDS_

If set to a true value and a test installation is to be done, this is done with .libPaths() containing just a temporary library directory and .Library. The temporary library is populated by symbolic links¹ to the installed copies of all the Depends/Imports/LinkingTo packages which are not in .Library. Default: false (but true for CRAN submission checks).

Note that this is actually implemented in R CMD INSTALL, so it is available to those who first install recording to a log, then call R CMD check.

_R_CHECK_DEPENDS_ONLY_
_R_CHECK_SUGGESTS_ONLY_

If set to a true value, running examples, tests and vignettes is done with .libPaths() containing just a temporary library directory and .Library. The temporary library is populated by symbolic links² to the installed copies of all the Depends/Imports and (for the second only) Suggests packages which are not in .Library. (As an exception, packages in a ‘VignetteBuilder’ field are always made available.) Default: false (but _R_CHECK_SUGGESTS_ONLY_ is true for CRAN checks).

_R_CHECK_NO_RECOMMENDED_

If set to a true value, augment the previous checks to make recommended packages unavailable unless declared. Default: false (but true for CRAN submission checks). This may give false positives on code which uses grDevices::densCols and stats:::asSparse as these invoke KernSmooth and Matrix respectively.

_R_CHECK_CODETOOLS_PROFILE_

A string with comma-separated name=value pairs (with value a logical constant) giving additional arguments for the codetools functions used for analyzing package code. E.g., use _R_CHECK_CODETOOLS_PROFILE_="suppressLocalUnused=FALSE" to turn off suppressing warnings about unused local variables. Default: no additional

¹ under Windows, junction points, or copies if environment variable R_WIN_NO_JUNCTIONS has a non-empty value.
² see the previous footnote.
arguments, corresponding to using `skipWith = TRUE, suppressPartialMatchArgs = FALSE` and `suppressLocalUnused = TRUE`.

**R_CHECK_CRAN_INCOMING**
Check whether package is suitable for publication on CRAN. Default: false, except for CRAN submission checks.

**R_CHECK_XREFS_USE_ALIASES_FROM_CRAN**
When checking anchored Rd xrefs, use Rd aliases from the CRAN package web areas in addition to those in the packages installed locally. Default: false.

**R_SHLIB_BUILD_OBJECTS_SYMBOL_TABLES**
Make the checks of compiled code more accurate by recording the symbol tables for objects (.o files) at installation in a file `symbols.rds`. (Only currently supported on Linux, Solaris, OS X, Windows and FreeBSD.) Default: true.

**R_CHECK_CODE_ASSIGN_TO_GLOBALENV**
Should the package code be checked for assignments to the global environment? Default: false (but true for CRAN submission checks).

**R_CHECK_CODE.Attach**
Should the package code be checked for calls to `attach()`? Default: false (but true for CRAN submission checks).

**R_CHECK_CODE_DATA_INTO_GLOBALENV**
Should the package code be checked for calls to `data()` which load into the global environment? Default: false (but true for CRAN submission checks).

**R_CHECK_DOT_FIRSTLIB**
Should the package code be checked for the presence of the obsolete function `.First.lib()`? Default: false (but true for CRAN submission checks).

**R_CHECK_DEPRECATED_DEFUNCT**
Should the package code be checked for the presence of recently deprecated or de-funct functions (including completely removed functions). Also for platform-specific graphics devices. Default: false (but true for CRAN submission checks).

**R_CHECK_SCREEN_DEVICE**
If set to `‘warn’`, give a warning if examples etc open a screen device. If set to `‘stop’`, give an error. Default: empty (but `‘stop’` for CRAN submission checks).

**R_CHECK_WINDOWS_DEVICE**
If set to `‘stop’`, give an error if a Windows-only device is used in example etc. This is only useful on Windows: the devices do not exist elsewhere. Default: empty (but ‘stop’ for CRAN submission checks).

**R_CHECK_TOPOLEVEL_FILES**
Report on top-level files in the package sources that are not described in `Writing R Extensions` nor are commonly understood (like `ChangeLog`). Variations on standard names (e.g. `COPYRIGHT`) are also reported. Default: false (but true for CRAN submission checks).

**R_CHECK_GCT_N**
Should the `--use-gct` use `gctorture2(n)` rather than `gctorture(TRUE)`? Use to a positive integer to enable this. Default: 0.

**R_CHECK_LIMIT_CORES**
If set, check the usage of too many cores in package `parallel`. If set to `‘warn’` gives a warning, to `‘false’` or `‘FALSE’` the check is skipped, and any other non-empty value
gives an error when more than 2 children are spawned. Default: unset (but ‘TRUE’ for CRAN submission checks).

_R_CHECK_CODE_USAGE_VIA_NAMESPACES_
If set, check code usage (via codetools) directly on the package namespace without loading and attaching the package and its suggests and enhances. Default: true (and true for CRAN submission checks).

_R_CHECK_EXIT_ON_FIRST_ERROR_
If set to a true value, the check will exit on the first error. Default: false.

_R_CHECK_S3_METHODS_NOT_REGISTERED_
If set to a true value, report (apparent) S3 methods exported but not registered. Default: false (but true for CRAN submission checks).

_R_CHECK_OVERWRITE_REGISTERED_S3_METHODS_
If set to a true value, report already registered S3 methods in base/recommended packages which are overwritten when this package’s namespace is loaded. Default: false (but true for CRAN submission checks).

CRAN’s submission checks use something like
_R_CHECK_CRAN_INCOMING_=TRUE
_R_CHECK_VC_DIRS_=TRUE
_R_CHECK_TIMINGS_=10
_R_CHECK_INSTALL_DEPENDS_=TRUE
_R_CHECK_SUGGESTS_ONLY_=TRUE
_R_CHECK_NO_RECOMMENDED_=TRUE
_R_CHECK_EXECUTABLES_EXCLUSIONS_=FALSE
_R_CHECK_DOC_SIZES2_=TRUE
_R_CHECK_CODE_ASSIGN_TO_GLOBALENV_=TRUE
_R_CHECK_CODE_ATTACH_=TRUE
_R_CHECK_CODE_DATA_INTO_GLOBALENV_=TRUE
_R_CHECK_CODE_USAGE_VIA_NAMESPACES_=TRUE
_R_CHECK_DOT_FIRSTLIB_=TRUE
_R_CHECK_DEPRECATED_DEFUNCT_=TRUE
_R_CHECK_REPLACING_IMPORTS_=TRUE
_R_CHECK_SCREEN_DEVICE_=stop
_R_CHECK_TOPLEVEL_FILES_=TRUE
_R_CHECK_S3_METHODS_NOT_REGISTERED_=TRUE
_R_CHECK_OVERWRITE_REGISTERED_S3_METHODS_=TRUE

These are turned on by R CMD check --as-cran: the incoming checks also use
_R_CHECK_FORCE_SUGGESTS_=FALSE

since some packages do suggest other packages not available on CRAN or other commonly-used repositories.


Chapter 9: R coding standards

R is meant to run on a wide variety of platforms, including Linux and most variants of Unix as well as Windows and OS X. Therefore, when extending R by either adding to the R base distribution or by providing an add-on package, one should not rely on features specific to only a few supported platforms, if this can be avoided. In particular, although most R developers use GNU tools, they should not employ the GNU extensions to standard tools. Whereas some other software packages explicitly rely on e.g. GNU make or the GNU C++ compiler, R does not. Nevertheless, R is a GNU project, and the spirit of the GNU Coding Standards should be followed if possible.

The following tools can “safely be assumed” for R extensions.

- An ISO C99 C compiler. Note that extensions such as POSIX 1003.1 must be tested for, typically using Autoconf unless you are sure they are supported on all mainstream R platforms (including Windows and OS X).
- A FORTRAN 77 compiler (but not Fortran 9x, although it is nowadays widely available).
- A simple make, considering the features of make in 4.2 BSD systems as a baseline. GNU or other extensions, including pattern rules using ‘%’, the automatic variable ‘$^’, the ‘+=’ syntax to append to the value of a variable, the (“safe”) inclusion of makefiles with no error, conditional execution, and many more, must not be used (see Chapter “Features” in the GNU Make Manual for more information). On the other hand, building R in a separate directory (not containing the sources) should work provided that make supports the VPATH mechanism.
  
Windows-specific makefiles can assume GNU make 3.79 or later, as no other make is viable on that platform.

- A Bourne shell and the “traditional” Unix programming tools, including grep, sed, and awk.
  
There are POSIX standards for these tools, but these may not be fully supported. Baseline features could be determined from a book such as The UNIX Programming Environment by Brian W. Kernighan & Rob Pike. Note in particular that ‘!‘ in a regexp is an extended regexp, and is not supported by all versions of grep or sed. The Open Group Base Specifications, Issue 7, which are technically identical to IEEE Std 1003.1 (POSIX), 2008, are available at http://pubs.opengroup.org/onlinepubs/9699919799/mindex.html.

Under Windows, most users will not have these tools installed, and you should not require their presence for the operation of your package. However, users who install your package from source will have them, as they can be assumed to have followed the instructions in “the Windows toolset” appendix of the “R Installation and Administration” manual to obtain them. Redirection cannot be assumed to be available via system as this does not use a standard shell (let alone a Bourne shell).

In addition, the following tools are needed for certain tasks.

- Perl version 5 is only needed for a few uncommonly-used tools: make install-info needs Perl installed if there is no command install-info on the system, and for the maintainer-only script tools/help2man.pl.
- Makeinfo version 4.7 or later is needed to build the Info files for the R manuals written in the GNU Texinfo system.

It is also important that code is written in a way that allows others to understand it. This is particularly helpful for fixing problems, and includes using self-descriptive variable names, commenting the code, and also formatting it properly. The R Core Team recommends to use a basic indentation of 4 for R and C (and most likely also Perl) code, and 2 for documentation.
in Rd format. Emacs (21 or later) users can implement this indentation style by putting the following in one of their startup files, and using customization to set the `c-default-style` to "bsd" and `c-basic-offset` to 4.)

```lisp
;; ESS
(add-hook 'ess-mode-hook
  (lambda ()
    (ess-set-style 'C++ 'quiet)
    ;; Because
    ;; DEF GNU BSD K&R C++
    ;; ess-indent-level 2 2 8 5 4
    ;; ess-continued-statement-offset 2 2 8 5 4
    ;; ess-brace-offset 0 0 -8 -5 -4
    ;; ess-arg-function-offset 2 4 0 0 0
    ;; ess-expression-offset 4 2 8 5 4
    ;; ess-else-offset 0 0 0 0 0
    ;; ess-close-brace-offset 0 0 0 0 0
    (add-hook 'local-write-file-hooks
      (lambda ()
        (ess-nuke-trailing-whitespace))))
  (ess-nuke-trailing-whitespace-p 'ask)
 ;; or even
 ;; (setq ess-nuke-trailing-whitespace-p t)
  ;; Perl
  (add-hook 'perl-mode-hook
    (lambda () (setq perl-indent-level 4))))
```

(The ‘GNU’ styles for Emacs’ C and R modes use a basic indentation of 2, which has been determined not to display the structure clearly enough when using narrow fonts.)
Chapter 10: Testing R code

10 Testing R code

When you (as R developer) add new functions to the R base (all the packages distributed with R), be careful to check if make test-Specific or particularly, cd tests; make no-segfault.Rout still works (without interactive user intervention, and on a standalone computer). If the new function, for example, accesses the Internet, or requires GUI interaction, please add its name to the “stop list” in tests/no-segfault.Rin.

[To be revised: use make check-devel, check the write barrier if you change internal structures.]
11 Use of TeX dialects

Various dialects of TeX and used for different purposes in R. The policy is that manuals be written in ‘\texttt{texinfo}’, and for convenience the main and Windows FAQs are also. This has the advantage that is is easy to produce HTML and plain text versions as well as typeset manuals.

\LaTeX is not used directly, but rather as an intermediate format for typeset help documents and for vignettes.

Care needs to be taken about the assumptions made about the R user’s system: it may not have either ‘\texttt{texinfo}’ or a TeX system installed. We have attempted to abstract out the cross-platform differences, and almost all the setting of typeset documents is done by \texttt{tools::texi2dvi}. This is used for offline printing of help documents, preparing vignettes and for package manuals via \texttt{R CMD Rd2pdf}. It is not currently used for the R manuals created in directory \texttt{doc/manual}.

\texttt{tools::texi2dvi} makes use of a system command \texttt{texi2dvi} where available. On a Unix-alike this is usually part of ‘\texttt{texinfo}’, whereas on Windows if it exists at all it would be an executable, part of MiKTeX. If none is available, the R code runs a sequence of (pdf)\texttt{latex}, \texttt{bibtex} and \texttt{makeindex} commands.

This process has been rather vulnerable to the versions of the external software used: particular issues have been \texttt{texi2dvi} and \texttt{texinfo.tex} updates, mismatches between the two\footnote{Linux distributions tend to unbundle \texttt{texinfo.tex} from ‘\texttt{texinfo}’.}, versions of the \LaTeX package ‘\texttt{hyperref}’ and quirks in index production. The licenses used for \LaTeX and latterly ‘\texttt{texinfo}’ prohibit us from including ‘known good’ versions in the R distribution.

On a Unix-alike \texttt{configure} looks for the executables for TeX and friends and if found records the absolute paths in the system \texttt{Renviron} file. This used to record ‘\texttt{false}’ if no command was found, but it nowadays records the name for looking up on the path at run time. The latter can be important for binary distributions: one does not want to be tied to, for example, TeX Live 2007.
12 Current and future directions

This chapter is for notes about possible in-progress and future changes to R: there is no commitment to release such changes, let alone to a timescale.

12.1 Long vectors

Vectors in R 2.x.y were limited to a length of $2^{31} - 1$ elements (about 2 billion), as the length is stored in the SEXPREC as a C int, and that type is used extensively to record lengths and element numbers, including in packages.

Note that longer vectors are effectively impossible under 32-bit platforms because of their address limit, so this section applies only on 64-bit platforms. The internals are unchanged on a 32-bit build of R.

A single object with $2^{31}$ or more elements will take up at least 8GB of memory if integer or logical and 16GB if numeric or character, so routine use of such objects is still some way off.

There is now some support for long vectors. This applies to raw, logical, integer, numeric and character vectors, and lists and expression vectors. (Elements of character vectors (CHARSXP) remain limited to $2^{31} - 1$ bytes.) Some considerations:

• This has been implemented by recording the length (and true length) as -1 and recording the actual length as a 64-bit field at the beginning of the header. Because a fair amount of code in R uses a signed type for the length, the ‘long length’ is recorded using the signed C99 type ptrdiff_t, which is typedef-ed to R_xlen_t.

• These can in theory have 63-bit lengths, but note that current 64-bit OSes do not even theoretically offer 64-bit address spaces and there is currently a 52-bit limit (which exceeds the theoretical limit of current OSes and ensures that such lengths can be stored exactly in doubles).

• The serialization format has been changed to accommodate longer lengths, but vectors of lengths up to $2^{31-1}$ are stored in the same way as before. Longer vectors have their length field set to -1 and followed by two 32-bit fields giving the upper and lower 32-bits of the actual length. There is currently a sanity check which limits lengths to $2^{48}$ on unserialization.

• The type R_xlen_t is made available to packages in C header Rinternals.h: this should be fine in C code since C99 is required. People do try to use R internals in C++, but C++98 compilers are not required to support these types.

• Indexing can be done via the use of doubles. The internal indexing code used to work with positive integer indices (and negative, logical and matrix indices were all converted to positive integers): it now works with either INTSXP or REALSXP indices.

• R function length was documented to currently return an integer, possibly NA. A lot of code has been written that assumes that, and even code which calls as.integer(length(x)) before passing to .C/.Fortran rarely checks for an NA result.

There is a new function xlength which works for long vectors and returns a double value if the length exceeds $2^{31-1}$. At present length returns NA for long vectors, but it may be safer to make that an error.

12.2 64-bit types

There is also some desire to be able to store larger integers in R, although the possibility of storing these as double is often overlooked (and e.g. file pointers as returned by seek are already stored as double).

Different routes have been proposed:
• Add a new type to R and use that for lengths and indices—most likely this would be a 64-bit signed type, say `longint`. R’s usual implicit coercion rules would ensure that supplying an `integer` vector for indexing or `length<-` would work.

• A more radical alternative is to change the existing `integer` type to be 64-bit on 64-bit platforms (which was the approach taken by S-PLUS for DEC/Compaq Alpha systems). Or even on all platforms.

• Allow either `integer` or `double` values for lengths and indices, and return `double` only when necessary.

The third has the advantages of minimal disruption to existing code and not increasing memory requirements. In the first and third scenarios both R’s own code and user code would have to be adapted for lengths that were not of type `integer`, and in the third code branches for long vectors would be tested rarely.

Most users of the `.C` and `.Fortran` interfaces use `as.integer` for lengths and element numbers, but a few omit these in the knowledge that these were of type `integer`. It may be reasonable to assume that these are never intended to be used with long vectors.

The remaining interfaces will need to cope with the changed `VECTOR_SSEXPREC` types. It seems likely that in most cases lengths are accessed by the `length` and `LENGTH` functions\(^1\) The current approach is to keep these returning 32-bit lengths and introduce ‘long’ versions `xlength` and `XLENGTH` which return `R_xlen_t` values.


### 12.3 Large matrices

Matrices are stored as vectors and so were also limited to $2^{31}-1$ elements. Now longer vectors are allowed on 64-bit platforms, matrices with more elements are supported provided that each of the dimensions is no more than $2^{31}-1$. However, not all applications can be supported.

The main problem is linear algebra done by FORTRAN code compiled with 32-bit `INTEGER`. Although not guaranteed, it seems that all the compilers currently used with R on a 64-bit platform allow matrices each of whose dimensions is less than $2^{31}$ but with more than $2^{31}$ elements, and index them correctly, and a substantial part of the support software (such as BLAS and LAPACK) also work.

There are exceptions: for example some complex LAPACK auxiliary routines do use a single `INTEGER` index and hence overflow silently and segfault or give incorrect results. One example is `svd()` on a complex matrix.

Since this is implementation-dependent, it is possible that optimized BLAS and LAPACK may have further restrictions, although none have yet been encountered. For matrix algebra on large matrices one almost certainly wants a machine with a lot of RAM (100s of gigabytes), many cores and a multi-threaded BLAS.

---

\(^1\) but `LENGTH` is a macro under some internal uses.
Function and variable index

A
alloca ........................................... 18
ARGSUSED ......................................... 3
ATTRIB ........................................... 7
attribute_hidden ................................. 20

C
Callon ........................................... 18
copyMostAttributes ......................... 7

d...

D
DDVAL .............................................. 3
dbog bit .......................................... 2
DispatchGeneric .............................. 10
DispatchOrEval ............................... 10
dump.frames .................................... 19
DUPLICATE_ATTRIB ............................ 7

E
eemacs ............................................ 52
error ............................................. 15
errorcall ........................................ 15

F
Free ................................................ 18

G
gp bits ......................................... 3

I
invisible ....................................... 11

L
last.warning .................................... 19
LEVELS .......................................... 3
Function and variable index

M
make ........................................ 51
makeinfo .................................... 51
MISSING ..................................... 3, 10
mkChar ...................................... 14
mkCharLenCE ................................ 15

N
NAMED ....................................... 2, 9, 23
named bit ................................... 2

P
Perl ........................................... 51
PRIMPRINT .................................. 11
PRSEEN ....................................... 3

R
R_alloc ..................................... 18
R_AllocStringBuffer ....................... 18
R_BaseNamespace ......................... 6
R_CheckStack .............................. 18
R_CheckStack2 .............................. 18
R_FreeStringBuffer ....................... 18
R_FreeStringBufferL ...................... 18
R_MissingArg .............................. 10
R_Visible .................................. 11
Rdll.hide ................................... 20
Realloc ..................................... 18

S
SET_ARGUSED ................................ 3
SET_ATTRIB ................................. 7
SET_DIVAL .................................. 3
SET_MISSING ............................... 3
SET_NAMED .................................. 2
SETLEVELS .................................. 3
spare bit ................................... 2

T
trace bit .................................... 2

U
UseMethod ................................... 9

V
vmaxget ..................................... 18
vmaxset ..................................... 18

W
warning .................................... 15
warningcall ................................ 15
Concept index

. argument ........................................... 3, 11
Internal function .................................. 10

A
allocation classes .................................. 5
argument evaluation ................................ 9
argument list ........................................ 2
atomic vector type .................................. 2
attributes .......................................... 7
attributes, preserving ............................... 7
autoprinting ........................................ 11

B
base environment .................................. 5, 19
base namespace .................................... 6
builtin function .................................... 10

C
coding standards .................................. 51
context .............................................. 8
copying semantics .................................. 2, 7

E
environment ......................................... 5
environment, base .................................. 5, 19
environment, global ................................ 19
expression .......................................... 2

F
function ............................................. 2

G
garbage collector ................................... 12
generic, generic ..................................... 10
generic, internal ................................... 10
global environment ................................. 19

L
language object ..................................... 2

M
method dispatch .................................... 9
missingness ......................................... 10
modules ............................................. 19

N
namespace ......................................... 6
namespace, base ..................................... 6
node ................................................. 1

P
preserving attributes ................................ 7
primitive function .................................. 10
promise ............................................. 3

S
S4 type .............................................. 2
search path ......................................... 6
serialization ....................................... 12
SEXP ............................................... 1
SEXPREC .......................................... 1
SEXPTYPE ......................................... 1
SEXPTYPE table ................................... 10
special function ....................................

U
user databases ...................................... 5

V
variable lookup .................................... 5
vector type ......................................... 4
visibility .......................................... 20

W
write barrier ....................................... 12