R Internals - with modifications for pqR

R Version 2.15.0 (2012-03-30), pqR version 2.15.1 (2016-10-24)

R Development Core Team, pqR modifications by Radford Neal

ISBN 3-900051-14-3

Table of Contents

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, first begun for R 2.4.0, and should be checked against the current version of the source code.

1.1 SEXPs

The values of R variableso and other objects 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 pointing to `VECTOR_SEXPREC` structures). So the basic building blocks of R objects are often called *nodes*, meaning `SEXPREC`s or `VECTOR_SEXPREC`s.

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.

Node structures of all types have as their first four fields a 64-bit `sxpinfo` header, a pointer to the attributes, and pointers to the previous and next node in a doubly-linked list (for garbage collection purposes). Additional fields vary by type (a `SEXPTYPE`), which is specfied by a five-bit field in the `sxpinfo` header.

1.1.1 SEXPTYPEs

Currently `SEXPTYPE`s 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 Values 11 and 12 were used for internal factors and ordered factors and have since been withdrawn.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.

no	SEXPTYPE	Description
0	NILSXP	R_NilValue
1	SYMSXP	symbols
2	LISTSXP	pairlists
3	CLOSXP	closures
4	ENVSXP	environments
5	PROMSXP	promises
6	LANGSXP	language objects
7	SPECIALSXP	special functions
8	BUILTINSXP	builtin functions
9	CHARSXP	internal character strings
10	LGLSXP	logical vectors
13	INTSXP	integer vectors
14	REALSXP	numeric vectors
15	CPLXSXP	complex vectors
16	STRSXP	character vectors
17	DOTSXP	dot-dot-dot object
18	ANYSXP	make “any” args work
19	VECSXP	list (generic vector)

20	EXPRSXP	expression vector
21	BCODESXP	byte code
22	EXTPTRSXP	external pointer
23	WEAKREFSXP	weak reference
24	RAWSXP	raw vector
25	S4SXP	S4 classes not of simple type

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[1] 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 (though note that taking the `CAR` or `CDR` of `R_NilValue` gives `R_NilValue` without error, so disaster may be avoided).

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 `SEXPTYPE`s 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. Prior to R 2.4.0 these were represented as empty lists.

1.1.2 Rest of header

The `sxpinfo` header is defined as the 64-bit C structure below:

```
struct sxpinfo_struct {
    /* Type and namedcnt in first byte */
    unsigned int nmcnt : 3;   /* count of "names" referring to object */
    unsigned int type : 5;    /* discussed above */
    /* Garbage collector stuff - keep in one byte to maybe speed up access */
    unsigned int gccls : 3;   /* node class for garbage collector */
    unsigned int gcgen : 1;   /* old generation number - may be best first */
    unsigned int mark : 1;    /* marks object as in use in garbage collector */
    /* Object flag */
    unsigned int obj : 1;     /* set if this is an S3 or S4 object */
    /* Flags to synchronize with helper threads */
    unsigned int in_use: 1;   /* whether contents may be in use by a helper */
    unsigned int being_computed : 1;  /* whether helper may be computing this */
    /* "general purpose" field, used for miscellaneous purposes */
    unsigned int gp : 16;     /* The "general purpose" field */
```

[1] 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.

```
    union {
      struct {               /* field below is for vectors only */
        R_len_t truelength;      /* for old stuff - may someday be defunct... */
      } vec;
      struct {               /* fields below are for non-vectors only */
        /* Debugging */
        unsigned int debug : 1;  /* function/environment is being debugged */
        unsigned int rstep : 1;  /* function is to be debugged, but only once */
        unsigned int trace : 1;  /* function is being traced */
        /* Symbol binding */
        unsigned int unused : 1;      /* not yet used */
        unsigned int spec_sym : 1;    /* this is a "special" symbol */
        unsigned int no_spec_sym : 1; /* environment has no special symbols */
        unsigned int base_env : 1;    /* this is R_BaseEnv or R_BaseNamespace */
        unsigned int basec : 1;       /* sym has base binding in global cache */
        /* Primitive operations */
        unsigned char pending_ok;/* whether args can have computation pending */
        unsigned short var1;     /* variant for eval of unary primitive arg */
      } nonvec;
    } u;
};
```

Note that one field is for vectors only (including `CHARSXP`), while some other fields are for non-vectors only.

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 `rstep` bit is used only for closures, to indicate "debugonce".

The `trace` bit is used for functions for `trace()`.

For symbols, the `basec` bit is used for the `BASE_CACHE` flag, that indicates that the symbol is in the global cache with a binding in the base environment, allowing its value to be obtained directly without looking in the hash table. (Currently done only for non-active bindings that are functions.)

The `spec_sym` bit is used as the `SPEC_SYM` flag for symbols, indicating that it is one of a set of special symbols (eg, `+` and `if`) for which an attempt is made to have function lookup be faster.

The `base-env` bit, accessed by the `IS_BASE` macro, is used to slightly speed up the check for whether an environment is either `R_BaseEnd` or `R_BaseNamespace`, which is done in time-critical code.

The `no_spec_sym` bit is used as the `NO_SPEC_SYM` flag for environments to indicate that the environment has no symbols with the `SPEC_SYM` flag set (and has never had such a symbol), so that it can be bypassed in searches for such symbols. Currently, this bit is maintained only for unhashed environments (always set to zero for hashed environments).

The field called `nmcnt` in pqR replaces the two-bit `named` field in R-2.15.0. The `nmcnt` field is accessed and set by the `NAMEDCNT` and `SET_NAMEDCNT` macros, by other macros for

testing, incrementing, and decrementing the field, and also by the `NAMED` and `SET_NAMED` macros provided for compatibility with the previous scheme using `named`.

The `nmcnt` field is three bits in size, allowing for values from 0 to 7, which are interpreted as the number of bindings to variables or other references to the object, though in many cases, this is simply an upper bound on the number of references, since the count is not always decremented when a reference disappears.

This count of references allows pqR to more efficiently maintain the appearence that when an object is assigned to a variable, passed as the argument of a function, or stored as an element of a list, it is duplicated, so that subsequent changes to the original object do not modify the copy, and vice versa. This duplication can be deferred and sometimes avoided by instead simply incrementing the `nmcnt` field for the object.

See Section 1.8 [Reference counts and the nmcnt field], page 16, for details.

The `gp` bits are called that because they are supposedly 'general purpose'. This term is misleading, however, since at least one (bit 4) is not at all general purpose, but is instead reserved in all objects. The only meaning of 'general purpose' seems to have been that people using them felt no obligation to document their usage.

We label these bit 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 `SEXPTYPE`s 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. Bit 15 is used for an environment to indicate if it participates in the global cache. See the next section on Section 1.2 [Environments and variable lookup], page 7, for further details.

The macros `MISSING` and `SET_MISSING` are used for pairlists of arguments. Four bits are reserved, but only two are used. 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. Note that this information is not captured by the use of `R_MissingArg` for the bound value in completely missing (no default) arguments, so this use is not redundant (though it could be reduced to one bit).

Bit 0 is used by macros `DDVAL` and `SET_DDVAL`. This indicates that a `SYMSXP` is one of the symbols `..n` which are implicitly created when `...` 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.

Bits 5 to 15 of the gp field are used for the table offset in a BUILTINSXP or SPECIALSXP. This offset is found again from the name when restoring saved data.

1.1.3 The 'data'

A SEXPREC is a C structure containing the 64-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 listsxp_struct listsxp;
    struct envsxp_struct envsxp;
    struct closxp_struct closxp;
    struct promsxp_struct promsxp;
} u;
```

All of these alternatives apart from the first are three pointers, and the first should be no larger, so the union should occupy three words.

The vector types are RAWSXP, CHARSXP, LGLSXP, INTSXP, REALSXP, CPLXSXP, STRSXP, VECSXP, EXPRSXP and WEAKREFSXP. Remember that such types are a VECTOR_SEXPREC, which again consists of the header and the same three pointers, but followed by an integer giving the length of the vector, and then followed by the data. The data is aligned as required for a double and/or a pointer (even when the actual data is integer, or some other type not requiring this alignment).

A symsxp structure is also overlaid on top of these other structures (see below).

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. In particular, the garbage collector assumes that all the "three-pointer" types have pointers that can be acessed as CAR, CDR, and TAG, even when they aren't in a LISTSXP object, and that pointers in all vector types can be accessed with STRING_ELT even when they are not in a STRSXP.

NILSXP
: There is only one object of type NILSXP, R_NilValue, with no data. However, asking for the CAR, CDR, or TAG of R_NilValue will actually return R_NilValue without error. Note that since there is only one NILSXP, the tests s==R_NilValue and isNull(s) should be equivalent.

SYMSXP
: Pointers to three nodes, the name, value and internal, accessed by PRINTNAME (a CHARSXP), SYMVALUE and INTERNAL. (If the symbol's value is a .Internal function, the last is a pointer to the appropriate SEXPREC.) Many symbols have SYMVALUE R_UnboundValue.

  A fourth pointer follows this, pointing to the next symbol in the global hash table of symbols. It is not used outside routines for maintaining this table.

Note that currently symbols are never removed from this table, and hence never garbage collected. This is sometimes relied upon in C code, making it difficult to change the scheme so that space for unused symbols can be recovered.

There are then three further pointers, used to record the last binding of the symbol that was found in an unhashed environment, and the last environment in which the symbol was not found (in a lookup for a function). See the next section on Section 1.2 [Environments and variable lookup], page 7.

`CLOSXP` Pointers to the formals (a pairlist), the body and the environment.

`ENVSXP` Pointers to the frame, enclosing environment and hash table (`R_NilValue` or a `VECSXP`). A frame is a tagged pairlist with tag the symbol and CAR the bound value.

`PROMSXP` Pointers to the value, expression and environment (in which to evaluate the expression). Once an promise has been evaluated, the environment is set to `R_NilValue`.

`LISTSXP` Pointers to the CAR, CDR (usually a `LISTSXP` or `R_NilValue`) and TAG (a `SYMSXP` or `R_NilValue`).

`LANGSXP` 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 `LANGSXP`s, but can be symbols (`SYMSXP`s) or expression vectors (`EXPRSXP`s).

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

`SPECIALSXP`
`BUILTINSXP`
An integer (in the gp field) giving the offset into the table of primitives/`.Internal`s, plus various information copied from that table for fast access. This extra information is set up when the offset is set with `SET_PRIMOFFSET`.

`CHARSXP` `length` followed by a block of bytes (allowing for the `nul` terminator). The `truelength` field holds a hash value.

`LGLSXP`
`INTSXP` `length` followed by a block of C `int`s (which are 32 bits on all R platforms).

`REALSXP` `length` followed by a block of C `double`s

`CPLXSXP` `length` followed by a block of C99 `double complex`s.

`RAWSXP` `length` followed by a block of bytes.

`STRSXP` `length` followed by a block of pointers (`SEXP`s pointing to `CHARSXP`s).

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

`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 `R_NilValue`, an environment or an external pointer, and the '`finalizer`' is a function or `R_NilValue`.

`S4SXP` two unused pointers and a tag.

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

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. The sizes of the small node classes (all except class 7) are determined by code in memory.c; one will always be large enough for the non-vector nodes. Nodes in these classes are allocated from pages of about 2000 bytes. Vectors that do not fit in any small node class are given node class 7, and allocated individually.

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.

Some bindings of variables are 'active bindings', for which fetching and storing activate user-supplied functions. They are created by `makeActiveBinding`. See the help on that function and comments in file `src/main/envir.c` for details.

Bindings and environments can be locked against change. See the help for `lockBinding` and `lockEnvironment`.

Finally, 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 recorded in the `SYMVALUE` field of a symbol (which is `R_UnboundValue` if it has no value in the base environment). This environment contains heavily-used symbols such as `+` and `for`. Note that the `INTERNAL` field of a symbol may also contain values, but these are not technically bindings in any environment. When R is started the primitive and internal functions are installed (by C code) in these fields of symbols. 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, other than local environments of functions, are normally hashed for faster access (including insertion and deletion). The frame field is unused for hashed environments.

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[2] which have been `attach`ed. When an environment is either `attach`ed or `detach`ed, the names of its symbols are flushed from the cache. The cache is (usually) used when searching for variables when the global environment is reached, but not when directly searching some other environment that participates in the cache (since then bindings in the environment masked by others in the global cache should be visible).

Variables in the global cache whose binding within it is its binding in the base environment are marked so that they can be accessed quickly without consulting the hash table.

Local environments for functions may be marked (by `NO_SPEC_SYM`) as known not to contain certain 'special' symbols, such as `+` and `for`, so that lookup of such symbols can quickly skip these environments. The list of such symbols, marked by `SPEC_SYM`, is in `src/main/names.c`.

Every symbol contains a pointer to the environment in which the last unhashed binding for the symbol was found, along with a pointer to the binding cell that was found for that lookup. If a binding for the symbol is looked for again in this environment, the binding cell can be returned without a search (unless it has been set to `R_UnboundValue` due to deletion of the variable). Similarly, every symbol contains a pointer the the last unhashed environment in which a lookup for the symbol failed, provided that lookup was for a function, and either the lookup later succeeded, or the lookup was for an S3 method (for which lookups often fail).

These fields are cleared to the C NULL pointer (rather than being followed) by the garbage collector, so that they will not result in memory being occupied after the environment or the object bound is no longer used. Note that it is possible that although these pointers are set only for unhashed environments, such an environment may have become hashed later.

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

[2] Remember that attaching a list or a saved image actually creates and populates an environment and attaches that.

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 `attach`ed 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 `VECSXP` is part of the header of a `SEXP` of type `ENVSXP`, and this points to `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 `src/main/envir.c`. Unless set otherwise (e.g. by the `size` argument of `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 `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 `R_UnboundValue`.

1.3 Attributes

As we have seen, every `SEXPREC` has a pointer to the attributes of the node (default `R_NilValue`). The attributes can be accessed/set by the macros/functions `ATTRIB` and `SET_ATTRIB`, but such direct access is normally only used to check if the attributes are `R_NilValue` or to reset them. Otherwise access goes through the functions `getAttrib` and `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 `DUPLICATE_ATTRIB` to automate this.

Note that the 'attributes' of a `CHARSXP` are used as part of the management of the `CHARSXP` cache: of course `CHARSXP`'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 `R_NilValue` or a pairlist of non-zero length (and this is checked by `SET_ATTRIB`). The attributes are named (via tags on the pairlist). The replacement function `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 `R_NilValue`).

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 `REALSXP`, 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 `NILSXP`, `R_NilValue`, and that should never have attributes (and this is enforced in `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 `OBJECT` and S4 bits). Binary operations normally call `copyMostAttributes` to copy most attributes from the longer argument (and if they are of the same length from both, preferring the values on the 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:

```
> 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`. The exact declaration as below is used by RStudio, which breaks (when debugging) if it is changed:

```
typedef struct RCNTXT {
    struct RCNTXT *nextcontext; /* The next context up the chain */
    int callflag; /* The context "type" */
    JMP_BUF cjmpbuf; /* 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 */
    void *vmax;          /* top of R_alloc stack */
    int intsusp;                /* interrupts are suspended */
    SEXP handlerstack;          /* condition handler stack */
    SEXP restartstack;          /* stack of available restarts */
    struct RPRSTACK *prstack;   /* stack of pending promises */
    SEXP *nodestack;
#ifdef BC_INT_STACK
```

```
    IStackval *intstack;          /* BC_INT_STACK seems to never be defined */
#endif
    SEXP srcref;          /* The source line in effect */
    const struct R_local_protect *local_pr;  /* linked list of protected vars */
} RCNTXT, *context;
```

The 'types' are from

```
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 - or .C, etc. */
};
```

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.

Calls to closures set a context, as do calls of builtins when profiling is being done. Calls of the foreign functions (`.C`, `.Fortran`, `.External`, and `.Call`) also set a context, with type `CTXT_BUILTIN` (even though in pqR these are now special primitives).

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 `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 `UseMethod`) the evaluation process is the same until the `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 *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[3] 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 `.Internal`: `quote` is a special primitive, `+` is a builtin primitive, `cbind` is a special `.Internal` and `grep` is a builtin `.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 `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 `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 `trunc` allows one or more but the default method only accepts one.

[3] There is currently one other difference: when profiling builtin functions are counted as function calls but specials are not.

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

```
argument "foobar" is missing, with no default
```

Internally missingness is handled by two mechanisms. The object `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 `R_MissingArg` with the first `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 `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 `missing()` function. An argument is clearly missing if its missing bit is set or if the value is `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` for evaluation of constants and variables, and also on entry to every primitive or internal function, with subsequent behaviour depending on the setting in the `eval` column of the function's specification (obtained with the macro `PRIMPRINT`). For primitive functions, `R_Visible` should be modified if necessary in the code for that function, though if `PRIMPRINT` is 0 it will be set back to `TRUE` in any case. For internal functions, it may be modified, and this modification will be allowed to stand if `PRIMPRINT` is 2, but not if it is either 0 (set to TRUE) or 1 (set to FALSE).

The R primitive function `invisible` makes use of this mechanism: it's `PRIMPRINT` value is 1, and it just sets `R_Visible = FALSE` and returns its argument.

Primitive or internal functions may call other functions which may change the visibility flag. For example, left brace (a primitive) will leave `R_Visible` as it was set by the call of `eval` for the last expression in the braces. Note that since `eval` is called when evaluating promises, even object lookup can change `R_Visible`.

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 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 eval function

The `eval` function is the central routine in the interpreter. It takes as arguments an expression to evaluate and an environment (both of which must be protected by the caller) and returns the value that the expression evaluates to.

In pqR, `eval` calls `evalv` (or is a macro expanding to a call of `evalv` internally), which takes one additional argument, that may specify that a "variant" result is allowed. This argument should be 0 if no variant is allowed, as for plain `eval`. Symbols for other variants are defined in `Defn.h`.

For example, a caller of `evalv` might specify (by `VARIANT_NULL`) that the value will not be used (evaluation being done only for side effects), and may therefore be returned as `R_NilValue`, or (by `VARIANT_SUM`) that if the value is a numeric vector, only the sum of vector elements is needed, so that this sum may be returned rather than the vector (which will then not need to have space allocated for it). However, the caller of `evalv` must always be prepared to receive an ordinary value, rather than the variant asked for.

For some variants (eg, `VARIANT_SUM`), there is no need to specially identify that the value returned is the variant. For others (eg, `VARIANT_TRANS`, which may return the transpose of the result), a variant result is identified by the global variable `R_variant_result` being set to a non-zero value after the return of `evalv`. The caller of `evalv` should note the contents of `R_variant_result` and then reset it to zero before it could taken as indicating that a later return is of a variant. (But note that `evalv` will itself start by setting `R_variant_result` to zero, so there is no need to set it just before calling `evalv`.)

If the expression evaluated is a call of a primitive funciton, the variant argument of `evalv` will be passed on to the C function implementing this primitive if the appropriate flag is set in the primitive's specification (see below). Variant arguments may be passed

onwards from the expression passed to `evalv` — for example, from an "if" statement to the evaluation of the chosen branch, or from a function call to the evaluation of the body of the function — with any variant result propagated back, as controlled by the `VARIANT_PASS_ON` macro.

Primitive functions (objects of type BUILTINSXP and SPECIALSXP) are implemented using C functions that by convention have names of the form `do_XXX`. These functions are associated with symbols by tables in the source files that define the functions, which are copied to a single table for all such functions by initialization code in `names.c`. This table also has other information, such as whether the primitive is BUILTIN or SPECIAL, and whether the variant argument of evalv is passed on to the `do_XXX` function.

Arguments are passed to the `do_XXX` function as a pairlist. This is inefficient, since it requires that a CONS cell be allocated for every argument passed to a primitive. In pqR, a faster interface is provided for simple cases of unary primitives, in which the argument is unnamed, and are not S3 or S4 objects. This interface calls a `do_fast_XXX` function to handle such simple cases, which receives the argument of the primitive directly as a C argument. These functions are also set up with tables in the various source files and initialization code in `names.c`. For details, see `names.c`, the documentation in the code for `evalv`, and the definitions for PRIMFUN_FAST and related macros in `Rinternals.h`.

1.8 Reference counts and the nmcnt field

There is evidence in the code that R at one point used a single-bit `named` field in an object, to indicate whether the object was 'named' — that is, was the value of some variable. Later, this field was expanded to two bits, though only values 0, 1, and 2 were possible. A value of 2 for `named` indicated that the object is (or at least might be) the value of more than one variable (or otherwise could be referenced in more than one way), and therefore would need to be duplicated before being modified when accessed via any of these references. The value of `named` was never decremented in this scheme, leading to many unnecessary duplications.

In pqR, a true reference counting scheme is implemented, using a `nmcnt` field (currently three bits in size) that records how many variables, function arguments, list elements, or other references an object has. If this number is greater than the maximum that is possible (`MAX_NAMEDCNT`, currently 7), the maximum value is stored. Currently, `nmcnt` is not always decremented when a reference ceases to exist, so `nmcnt` is merely an upper bound on the number of references. In particular, note that once `nmcnt` reaches `MAX_NAMEDCNT`, it can never be decremented, since it is not known how many references beyond this number actually exist.

For compatibility with C code written for use with the previous scheme, the `NAMED` and `SET_NAMED` macros are still defined. `SET_NAMED` sets `nmcnt` to the value given if it is 0 or 1, and to `MAX_NAMEDCNT` if the value given to `SET_NAMED` is 2. `NAMED` delivers the value of `nmcnt` if it is 0 or 1, and the value 2 if `nmcnt` is any value greater than 1. Furthermore, if `NAMED` delivers the value 2, it changes `nmcnt` to `MAX_NAMEDCNT`. This is necessary because existing code may assume that if `NAMED` is 2, it will never decrease, and hence the object will always be duplicated before being modified, regardless of subsequent operations.

The `nmcnt` field can be accessed by the macro `NAMEDCNT`, but when appropriate, macros for testing its value called `NAMEDCNT_EQ_0`, `NAMEDCNT_GT_0`, and `NAMEDCNT_GT_1` should

be used, since they may be more efficient than a comparison with the value returned by NAMEDCNT.

When use of helper threads or task merging is enabled, an object that is being used by a task that has not yet completed must not be modified, even if its NAMEDCNT field is such that it could otherwise be modified. This constraint is implemented by having the NAMEDCNT macro wait until no task is using the object before returning the value. See Section 1.20 [Helper threads and task merging], page 30, for details.

The nmcnt field can be set with the SET_NAMEDCNT macro, but when possible, it is better to use the SET_NAMEDCNT_0, SET_NAMEDCNT_1, SET_NAMEDCNT_MAX, INC_NAMEDCNT, DEC_NAMEDCNT, and INC_NAMEDCNT_0_AS_1 macros, since they may be more efficient. The INC_NAMEDCNT and DEC_NAMEDCNT macros increment and decrement nmcnt, except that they do not change its value if it is MAX_NAMEDCNT, and DEC_NAMEDCNT does not change a value of zero. The INC_NAMEDCNT_0_AS_1 macro first changes nmcnt to 1 if it is 0, and then increments the value (unless it is MAX_NAMEDCNT).

In detail, pqR aims to follow (but may not yet fully follow) the (tentative) conventions below regarding how NAMEDCNT should be set, and when an object may be modified:

- Pairlists, language constructs, and closures (LISTSXP, LANGSXP, DOTSXP, CLOSXP, and BCODESXP objects) should never be modified — all changes must be implemented by duplicating all or part of the object — except for internal uses of such objects that are never visible to user R code (eg, lists of bindings in environments and lists of attributes), and when such objects are in the process of being constructed. This immutability is the easiest way of, for example, ensuring that expressions being evaluated are not changing during the course of their evaluation. Accordingly, NAMEDCNT is not meaningful for such objects.

  An atomic or non-atomic vector that is referenced from an object of one of these types (eg, a numeric constant in a language construct) should have its NAMEDCNT set to MAX_NAMEDCNT whenever it becomes visible elsewhere (eg, as the result of eval) to ensure that it is never changed.
- Environments (ENVSXP objects) may always be modified (ie, bindings within them created, deleted, or changed), since their specified semantics is that the same environment may be referenced in several ways, with changes to the environment seen via all references. Accordingly, NAMEDCNT is not meaningful for environments. (It is possible that in future NAMEDCNT will be maintained for environemnts so that when it is decremented to zero the NAMEDCNT in all values bound in the environment can be decremented, but this is not attempted at present.)
- Promises (PROMSXP objects) should have NAMEDCNT set to the number of bindings of the promise in an environment. This count will usually be one, but may be greater as a result of method dispatch. (This count is not yet properly maintained in pqR, so at present it must always be assumed to be greater than one.)
- NAMEDCNT for atomic and non-atomic vectors (objects of type LGLSXP, INTSXP, REALSXP, CPLXSXP, STRSX, RAWSXP, VECSXP, and EXPRSXP) and for objects of type S4SXP should be set to the count how many references exist to the object in any of the following ways:
  - as the value bound to a name in an environment.
  - as the value of an attibute of an object for which NAMEDCNT is non-zero.

- as an element in a list (`VECSXP` or `EXPRSXP`) for which `NAMEDCNT` is non-zero.
- as the value of a promise for which `NAMEDCNT` is non-zero.

However, `NAMEDCNT` for a vector or `S4SXP` objects is allowed to be zero if there is only a single reference to it as an attribute or a list element. If such a vector is made visible otherwise (eg, as the value returned by `eval`), and `NAMEDCNT` for the object referencing it is still non-zero, its `NAMEDCNT` must be set to 1, so that it will be correctly seen and maintained later. This convention is convenient because it sometimes avoids the need to set `NAMEDCNT` to 1 for all elements when creating a list, and because it sometimes automatically results in `NAMEDCNT` being zero for an element of a list that has `NAMEDCNT` of zero (either because it was always zero, or because it became zero after `NAMEDCNT` was decremented, for example in `get_rm`).

Note that this convention applies recursively. For example, when a list with a non-zero `NAMEDCNT` contains a list with zero `NAMEDCNT` which in turn contains a real vector with zero `NAMEDCNT`, the list and real vector with zero `NAMEDCNT` are considered to really have `NAMEDCNT` of 1, and must have `NAMEDCNT` set to 1 if they become visible elsewhere.

Note also that an object may reference another object more than once, in which case `NAMEDCNT` for the referenced object must account for all such references. For example, an object that appears twice in a list, and is otherwise unreferenced, should have its `NAMEDCNT` field set to 2.

- `NAMEDCNT` is not meaningful for the remaining types of objects (`NILSXP`, `SYMSXP`, `SPECIALSXP`, `BUILTINSXP`, `CHARSXP`, `EXTPTRSXP`, and `WEAKREFSXP`). These objects are either never modified, or modifications are intended to be visible everywhere.

It is always permissible (though undesirable) to set `NAMEDCNT` to a value greater than it needs to be according to the above conventions. Also, note that setting and accessing `NAMEDCNT` is permitted for all objects, even those for which it is not meaningful,

1.9 The write barrier and the garbage collector

R has since version 1.2.0 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`s). 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.)

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

1.10 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 has been the default (except for ASCII saves) since R 2.4.0. Compression is applied to the whole file stream, including the headers, so serialized files can be uncompressed or re-compressed by external programs. Since R 2.10.0 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 R 1.4.0, 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`. In the current version of pqR, objects stored as shared constants (probably in read-only memory) are written with a flag bit saying they are constants. This flag will be ignored by earlier versions of pqR or R that didn't have such constants, but will cause the current version of pqR to unserialize them as the same constants. (A check is done that other aspects of the object match the constant, for compatibility with future versions that might have a wider variety of such constants.)

For all `SEXPTYPE`s except `NILSXP`, `SYMSXP` and `ENVSXP` serialization starts with an integer with the `SEXPTYPE` in bits 0:7[4] 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[5] in bits 12:27. Pairlist-like objects write their attributes (if any), tag (if any), CAR and then CDR (avoiding tail recursion on CDR): 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.

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.

1.11 Encodings for CHARSXPs

Character data in R are stored in the sexptype `CHARSXP`. Until R 2.1.0 it was assumed that the data were in the platform's native 8-bit encoding, and furthermore it was quite often assumed that the encoding was ISO Latin-1 or a near-superset (such as Windows' CP1252 or Latin-9).

As from R 2.1.0 there was support for other encodings, in particular UTF-8 and the multi-byte encodings used on Windows for CJK languages. However, there was no way of indicating which encoding had been used, even if this was known (and e.g. `scan` would not

[4] only bits 0:4 are currently used for `SEXPTYPE`s but values 241:255 are used for pseudo-`SEXPTYPE`s.

[5] Currently the only relevant bits are 0:1, 4, 14:15.

know the encoding of the file it was reading). This lead to packages with data in French encoded in Latin-1 in `.rda` files which could not be read in other locales (and they would be able to be displayed in a French UTF-8 locale, if not in non-UTF-8 Japanese locales).

R 2.5.0 introduced a limited means to indicate the encoding of a `CHARSXP` 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), which is to a large extent done as from R 2.7.0.

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 is 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.

This was taken further in R 2.7.0. There 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[6] on Windows.) From R 2.10.0 it is enabled for all platforms with MBCS support, 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 be '`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` (as from R 2.13.1).

Windows has no UTF-8 locales, but rather expects to work with UCS-2[7] strings. R (being written in standard C) would not work internally with UCS-2 without extensive

[6] See define `USE_UTF8_IF_POSSIBLE` in file `src/main/gram.c`.

[7] 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.

changes. The `Rgui` console[8] 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.

1.12 The CHARSXP cache

A global cache for `CHARSXP`s created by `mkChar` was introduced in R 2.6.0 – the cache ensures that most `CHARSXP`s with the same contents share storage ('contents' including any declared encoding). Not all `CHARSXP`s are part of the cache – notably '`NA_STRING`' is not. `CHARSXP`s reloaded from the `save` formats of R prior to 0.99.0 are not cached (since the code used is frozen and few examples still exist).

The cache records the encoding of the string as well as the bytes: all requests to create a `CHARSXP` 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.

1.13 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` it is 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 will be of the form (from R < 2.6.0)

```
> 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.)

[8] but not the GraphApp toolkit.

1.14 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.]

1.14.1 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`. Note that some code tests `IS_S4_OBJECT` without first checking that it is an object of any sort, so the 'S4 bit' is reserved even for things like primitive functions.

S4 objects are created via `new()`[9] 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.

1.14.2 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`).

1.14.3 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`, `.SigArgs` 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.

[9] This can also create non-S4 objects, as in `new("integer")`.

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.

1.14.4 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`[10] 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: 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

[10] although this is not recommended as it is less future-proof.

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`.

1.15 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 `SEXPREC`s 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 `src/main/memory.c`, which are also the methods recommended in Section "Memory allocation" in *Writing R Extensions* for use in R packages, namely the use of `R_alloc`, `Calloc`, `Realloc` and `Free`. Memory allocated by `R_alloc` is freed by the garbage collector once the 'watermark' has been reset by calling `vmaxset`. This is done automatically by the wrapper code calling primitives and `.Internal` functions (and also by the wrapper code to `.Call` and `.External`), but `vmaxget` and `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 `alloca`, and in almost all cases[11] this is managed by the compiler, allocates memory on the C stack and is very efficient.

[11] but apparently not on Windows.

There are two disadvantages in using `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

```
R_CheckStack();
```

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). (`do_makeunique` in file `src/main/unique.c` provides an example of both points.)

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 `src/main/RBufferUtils.h`. This uses a structure which contains a buffer, the current size and the default size. A call to

```
R_AllocStringBuffer(size_t blen, R_StringBuffer *buf);
```

sets `buf->data` to a memory area of at least `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 `R_FreeStringBufferL` releases memory if more than the default has been allocated whereas a call to `R_FreeStringBuffer` frees any memory allocated.

The `R_StringBuffer` structure needs to be initialized, for example by

```
static R_StringBuffer ex_buff = {NULL, 0, MAXELTSIZE};
```

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

```
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);
```

1.15.1 Internals of R_alloc

The memory used by `R_alloc` is allocated as R vectors, of type `RAWSXP` for 'small' allocations (less than 2^31 - 1 bytes) and of type `REALSXP` for allocations up to 2^34 - 1 bytes on 64-bit machines. Thus the allocation is in units of 8 bytes, and is rounded up. A request for zero bytes currently returns `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 `R_VStack`, as the garbage collector marks everything that can be reached from that location. When a vector is `R_allocated`, its

ATTRIB pointer is set to the current R_VStack, and R_VStack is set to the latest allocation. Thus R_VStack is a single-linked chain of vectors currently allocated via R_alloc. Function vmaxset resets the location R_VStack, and should be to a value that has previously be obtained *via* vmaxget: allocations after the value was obtained will no longer be protected and hence available for garbage collection.

1.16 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.

1.16.1 Base environment

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

There appears to be a variable .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 .Last.value which appears as a variable in the base environment.

Errors can give rise to objects .Traceback and last.warning in the base environment.

1.16.2 Global environment

The seed for the random number generator is stored in object .Random.seed in the global environment.

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

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

1.17 Modules

R makes use of a number of shared objects/DLLs stored in the 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 a few of these (e.g. vfonts) the issue is size: the database for the Hershey fonts is included in the C code of the module and was at one time an appreciable part of the codebase for a rarely used feature. However, for most of the modules the motivation has been the amount of (often optional) code they will bring in via libraries to which they are linked.

internet The internal HTTP and FTP clients and socket support, which link to system-specific support libraries.

lapack The code which makes use of the LAPACK library, and is linked to libRlapack or an external LAPACK library.

vfonts The Hershey font databases and the code to draw with them.

`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.

`internet2.dll`
(Windows only.) An alternative version of the internet access routines, compiled against Internet Explorer internals (and so loads `wininet.dll` and `wsock32.dll`).

1.18 Visibility

1.18.1 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/Maintainters.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

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

1.18.2 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.

1.19 Lazy loading

Lazy loading was introduced in R 2.0.0, for code in packages and for datasets in packages (the latter is still optional, but code is always lazy-loaded as from R 2.14.0). 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. R 2.10.0 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 stores 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).

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`.

1.20 Helper threads and task merging

The pqR facility for doing computations in helper threads and/or deferred evaluation for task merging is supported by the "helpers" library in `src/extra/helpers`, where documentation on it may be found. The `helpers-app.h` and `helpers-app.c` files in that directory define how the helpers facility interfaces to the pqR interpreter.

At present, computation in helper threads (or in other deferred tasks) is confined to numeric vectors. Two bits in the headers of such objects (actually all objects) allow checks on how computations that may be done by helper threads or otherwise are deferred are progressing. These bits are accessed by the `helpers_is_being_computed` and `helpers_is_in_use` macros. An object is being computed by a task if computation of its data part was scheduled to be done by a task that has not been completed (or perhaps not even started). An object is in use by a task if it is an input to a scheduled task for which it is not also the output. (However, a object with maximum `NAMEDCNT` may not be flagged as in use even if it is, since such an object will never be modified anyway.) Note that these bits are set only by the master thread (whenever the master thread looks to see what tasks performed by helpers have completed), so access to them does not raise any synchronization issues.

The `WAIT_UNTIL_COMPUTED` and `WAIT_UNTIL_COMPUTED_2` macros wait until an object is not being computed, or until two objects are both not being computed.

Objects that may not have been computed yet should be visible only in code that has explicitly asked to see such "pending" objects. For example, the result of calling `eval` will never be an object that is still being computed. Code that wishes to receive a pending object as the result of evaluating an expression must instead call `evalv`, passing a variant argument with the `VARIANT_PENDING_OK` bit set. Similarly, `PRVALUE` will never return a pending object, but `PRVALUE_PENDING_OK` may.

Objects that are being used by a task that has not completed may be seen anywhere within the code for the interpeter, or in user-written C code. Data in such objects may be accessed without regard to their possibly being used by a helper thread, but this data must not be changed. This is ensured by making `NAMEDCNT` (and hence also `NAMED`) check whether the object is in use, and if so wait until it is not in use before returning the appropriate value (unless it is `MAX_NAMEDCNT`) — in effect, `NAMEDCNT` is temporarily increased while the object is in use by a task. Correct code that only alters objects after verifying that `NAMEDCNT` is zero will therefore work properly with objects that must not be altered until a task (or tasks) has completed. However, the macro `NAMEDCNT_GT_0` is written to return `TRUE` if `NAMEDCNT` is greater than zero, without waiting until the object is not in use; similarly, `NAMEDCNT_EQ_0` and `NAMEDCNT_GT_1` return immediately when the comparison does not depend on whether the object is in use by a task.

To avoid problems with the use of multiple threads by the helpers facility, the Unix/Linux `fork` function, used by package `parallel` and others, is redefined to `Rf_fork` in `Rinternals.h`. The `Rf_fork` function waits for all helper threads to become idle before forking, and then disables use of helper threads in the child process.

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, `axis` is defined as

```
axis <- function(side, at = NULL, labels = NULL, ...)
    .Internal(axis(side, at, labels, ...))
```

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 `R_NilValue` 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:

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

2. Language elements and basic *operator*s (i.e., functions usually *not* called as `foo(a, b, ...)`) for subsetting, assignment, arithmetic and logic (but `@<-` is currently not primitive):

   ```
   [    [[   $    @     [<-  [[<-  $<-
   <-   <<-  =    ->    ->>

   +    -    *    /     ^    %%   %*%  %/%
   <    <=   ==   !=    >=   >
   |    ||   &    &&    !
   ```

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.,

      ```
      abs     sign    sqrt
      floor   ceiling
      ```

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

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 functions), 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  pnamedcnt
```

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<-
```

Note that optimizing NAMED = 1 is only effective within a primitive (as the closure wrapper of a .Internal will set NAMED = 2 when the promise to the argument is evaluated) and 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:

```
:           ~           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

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

The multi-argument primitives

```
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 (partially) matched the formal argument name. On the other hand,

```
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 from R 2.11.0 argument matching for primitives intended for end-user use 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 is special as it evaluates some of its 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, args and package checking (e.g. by tools::checkS3methods and by package **codetools** (http://CRAN.R-project.org/package=codetools)). There are two environments in the **base** package (and namespace), '.GenericArgsEnv' for those primitives which are internal S3 generics, and '.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 UseMethod or R_NilValue and environment the base namespace.

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

The QC function 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

```
$  $<-  &&  (  :  @  [  [[  [[<-  [<-  {  ||  ~  <-  <<-  =  ->  ->>
break  for function  if  next  repeat  return  while
```

). 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 codoc and 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 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 Ops and that defines their arguments to be e1 and e2: although it would be very unusual, an operator could be called as e.g. "+"(e1=a, e2=b) and if method dispatch occurred to a closure, there would be an argument name mismatch. So the definitions in environment .GenericArgsEnv have to use argument names e1 and e2 even though the traditional documentation is in terms of x and y: codoc makes the appropriate adjustment via tools:::.make_S3_primitive_generic_env. The second discrepancy is with the Math group generics, where the group generic is defined with argument list (x, ...), but most of the members only allow one argument when used as the default method (and round and signif allow two as default methods): again fix-ups are used.

Those primitives which are in .GenericArgsEnv are checked (via tests/primitives.R) to be generic *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 .Internal or primitive functions are not internally generic except by reading the source code.

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

```
#ifdef 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/appl`; 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 hard-coded 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 Mac 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.

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`.

How (and if) compiled code is loaded is down to the package's startup code such as `.First.lib` or `.onLoad`, although a `useDynlib` directive in a namespace provides an alternative. 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. Prior to R 2.12.0 the contents of the directory could be moved to a zip file `Rdata.zip` and a listing written in file `filelist`: this was principally done on Windows.

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 (character) columns are '`File`' (the file name with extension), '`Name`' (the '`\name`' section), '`Type`' (from the optional '`\docType`' section), '`Title`', '`Encoding`', '`Aliases`', '`Concepts`' and '`Keywords`'. The last three are character strings with zero or more entries separated by '`, `'.

File `hsearch.rds` records the information to be used by '`help.search`'. This is a list of four unnamed elements which are character vectors 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 (the remaining column of the last three elements) to a particular help file. The first element has further columns '`LibPath`' (empty since R 2.3.0), '`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' (`.[RS]nw` 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 Mac 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 Mac 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 revised for version 1.4.0 (and tidied up for 2.7.0). This was 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**.

Some notes on the changes for 1.4.0 can be found at `http://www.stat.auckland.ac.nz/~paul/R/basegraph.html` and `http://www.stat.auckland.ac.nz/~paul/R/graphicsChanges.html`.

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.

[1] The C code is in files `base.c`, `graphics.c`, `par.c`, `plot.c` and `plot3d.c` in directory `src/main`.

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 `pch` values `0...26` to sets of lines/circles, centring (and 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 subsystems, after 6 years 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 unregistered 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 concerned 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 characteris-

[2] although that needs to be handled carefully, as for example the `xspline` functions used prior to R 2.7.0 to depend on the aspect ratio of the pixels, and the `circle` callback is given a radius (and that should be interpreted as in the x units).

tics 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.

Prior to R 2.14.0 all of the callback functions need to be set, to dummy functions if the capability (such as a locator) is not available. In devices which will only be used in R 2.14.0 or later 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 `DevDesc` structure: a '`double`' array of length 2.

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;
    SEXP DLlastElt;
    SEXP savedSnapshot;
    Rboolean dirty;
    Rboolean recordGraphics;
    GESystemDesc *gesd[MAX_GRAPHICS_SYSTEMS];
    Rboolean ask;
}
```

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 `pGEDevDesc` 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

```
pGEDevDesc gdd;

R_GE_checkVersionOrDie(R_GE_version);
R_CheckDeviceAvailable();
```

[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.

```
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

```
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.

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`.

Several more were added at R 2.14.0 to support the `dev.capabilities` function: these are all small integers.

- `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

```
void text(double x, double y, const char *str, double rot, double hadj,
          pGgcontext 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`.

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(beta)` 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

[5] in device coordinates

multibyte locale, `c` represents a Unicode point (except in the symbol font). So the function needs to include code like

```
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.

A new capability in R 2.7.0 (graphics API version 4) was `hasTextUTF8`. If this is 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:

```
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

```
dd->cra[0] = 0.9 * fnsize;
dd->cra[1] = 1.2 * fnsize;
```

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

The remaining elements are yet more mysterious. The `postscript()` device says

```
/* 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;
```

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

`yLineBias` is used in the base graphics system in `axis()` and `mtext()` to provide a default for their '`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 `postscript` and `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 '`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).
- `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 `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 (`par("lty")` is described in the header `GraphicsEngine.h` and in the help for `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 is not surprising that was it not used by any device: prior to R 2.7.0 it was not set by `grid::grid.locator`. 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.

As from R 2.14.0 this callback need not be supplied if it does nothing.

6.1.6 Graphics events

Graphics devices may be designed to handle user interaction. The current model is similar to the one introduced in R 2.1.0 for the Windows screen device, but the design was changed in R 2.12.0 to be more open ended.

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 approach being taken as from R 2.7.0 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. As from R 2.7.0, 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.

Support for translucent colours (with alpha channel between 0 and 255) was introduced in R 2.6.0 for the screen device only, and extended to the bitmap devices in R 2.7.0.[6] 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.dll`, which is loaded on first use.[7] As small a rectangular region as reasonably possible is alpha-blended (this is rectangle `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 an `unsigned int` (in the `GPar` structure and some of the devices as the `typedef rcolor`): the comments in file `R_ext/GraphicsDevice.h` are the primary documentation. The 4 bytes in the `unsigned int` 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 (from R 2.0.0) 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.

[6] It is technically possible to use alpha-blending on metafile devices such as printers, but it seems few drivers have support for this.

[7] It is available on Windows 2000 or later, and so had to be optional in R 2.6.0.

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). 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`. Note that semi-transparent colours are not 'premultiplied', so 50% transparent white is '`#ffffff80`'.

What is not clear is how the RGB values are to be mapped to display colours in the graphics device. There was a proposal (`http://developer.r-project.org/sRGB-RFC.html`) to regard the mapping as the colorspace 'sRGB', which was adopted in R 2.13.0. The sRGB colorspace 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.)

6.3 Base graphics

The base graphics system is likely to move to package **graphics** at some stage, but it currently 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 `.Internal` 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 `dpptr`) 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.

There remain quite a number of anomalies. For example, function `GEcontourLines` is (despite its name) coded in file `plot3d.c` and used to support function `contourLines` in package **grDevices**.

6.4 Grid graphics

[At least pointers to documentation.]

7 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 file `~/.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_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** (`http://CRAN.R-project.org/package=codetools`) package, which provides a detailed analysis of visibility of objects (but may give false positives). Default: true.

`_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 of 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. Legacy mostly. Default: true.

`_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, Mac 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 mainly come from auto-generated `NAMESPACE` files in other packages. Default: false.

`_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/rebuilding 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 links[1] 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 links[2] to the installed copies of all

[1] under Windows, junction points, or copies if environment variable `R_WIN_NO_JUNCTIONS` has a non-empty value.

[2] see the previous footnote.

the Depends/Imports/LinkingTo and (for the second only) Suggests packages which are not in `.Library`. Default: false (but 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** (`http://CRAN.R-project.org/package=KernSmooth`) and **Matrix** (`http://CRAN.R-project.org/package=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** (`http://CRAN.R-project.org/package=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 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.

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
```

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.

8 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 Mac 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 Mac OS X). Packages will be more portable to R < 2.12.0 if written assuming only C89, but this should not be done where using C99 features will make for cleaner or more robust code.
- A FORTRAN 77 compiler (but not Fortran 9x).
- 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.)

```
;;; 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)))))
(setq 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.)

9 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.]

10 Use of TeX dialects

Various dialects of TeX and used for different purposes in R. The policy is that manuals be written in '`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 '`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 `tools::texi2dvi`. This is used for offline printing of help documents, preparing vignettes and for package manuals via `R CMD Rd2pdf`. It is not currently used for the R manuals created in directory `doc/manual`.

`tools::texi2dvi` makes use of a system command `texi2dvi` where available. On a Unix-alike this is usually part of '`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)latex`, `bibtex` and `makeindex` commands.

This process has been rather vulnerable to the versions of the external software used: particular issues have been `texi2dvi` and `texinfo.tex` updates, mismatches between the two[1], versions of the LaTeX package '`hyperref`' and quirks in index production. The licenses used for LaTeX and latterly '`texinfo`' prohibit us from including 'known good' versions in the R distribution.

On a Unix-alike `configure` looks for the executables for TeX and friends and if found records the absolute paths in the system `Renviron` file. This used to record '`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, TeXLive 2007.

[1] Linux distributions tend to unbundle `texinfo.tex` from '`texinfo`'.

Function and variable index

M

N

P

R

S

T

U

V

W

Concept index