FramepaC User's Reference

Version 1.10

(Document Version 39a)

CMU-CMT-98-MEMO

Ralf D. Brown
6 July 2001

Abstract

FramepaC is a frame manipulation and utility package for C++ programmers which has been optimized for speed over ultimate generality. In addition to frame manipulation, it provides nearly all of the data types supported by Common Lisp. Despite the emphasis on raw speed, it still provides a rich set of features for frames, including inheritance, a Lisp-like reader, demand-loaded frames using either a disk file or a network server as backing store, and optionally demons.

1. Introduction

FramepaC is a highly efficient frame manipulation package for C++ programmers which was originally designed for use in the MikroKARAT knowledge acquisition environment [1]. Although designed primarily for the author's needs in various NLP programs, FramepaC is still quite general and should be valuable in a variety of other applications.

FramepaC's core frame function set has been patterened after FrameKit [2], a Common Lisp package used in a number of previous projects, including the ONTOS knowledge acquisition system [3, 4] which MikroKARAT was designed to supplant. FrameKit's primary drawback has been its extreme slowness, due to handling the most general cases imaginable; even fairly simple modifications have on more than one occasion resulted in a system-wide speedup by a factor of three or more. FramepaC omits one of the most expensive features in FrameKit (automatic copying of all retrieved data--which FrameKit does to permit destructive modification of the result, something that happens rather infrequently), and makes another one (demons) a compile-time option. It also omits views, which have never found much use (ONTOS does not use them), and always stores certain slots and facets in a frame even when they have not been explicitly used in order to allow direct lookups in place of searches when accessing those most-common fillers.

The remainder of this document contains the following chapters:

- an overview of the data types available in FramepaC
- a brief tutorial on programming with FramepaC
- an overview of demons
- an overview of FramepaC memory-allocation functions
- an overview of Motif-related user-interface functions
- a complete reference of all the functions provided by FramepaC
- a reference of the various configuration-related preprocessor symbols
- a trouble-shooting Q&A
- instructions for using the test/demo program supplied with FramepaC
- a review of benchmark results
- instructions for compiling FramepaC

```
FrSymbolTable                                        Public Types
FrReadTable
FrReader
FrConfiguration
FrEventList
FrTimer
FrMemoryPool
   |
   +------ FrAllocator

FrObject
   |
   +------ FrArray
   |         |
   |         +----- FrSparseArray
   |
   +------ FrAtom
   |         |
   |         +----- FrSymbol
   |         +----- FrString
   |         +----- FrNumber
   |                  |
   |                  +----- FrFloat
   |                  +----- FrInteger
   +------ FrBitVector
   +------ FrCons
   |         |
   |         +----- FrList
   |
   +------ FrHashTable
   +------ FrHashEntry
   |         |
   |         +----- FrHashEntryObject
   |
   +------ FrFrame
   |         |
   |         +----- VFrame
   |
   +------ FrQueue
   +------ FrStruct
   +------ FrWidget ---- (see Figure 2-2)
   |
   +------ FrISockStream \
   +------ FrOSockStream / FrSockStream
```

Table 2-1: The FramepaC Data-Type Hierarchy

2. The FramepaC Type System

FramepaC provides a number of data types, largely patterned after the Lisp types, which allow flexible storage of information. The three primary types used by FramepaC functions are the `FrObject`, the `FrSymbol`, and the `FrList`. The FramepaC types are arranged in a class hierarchy (described in the next section), so that a more specific type of object may be passed to a function anywhere a general type is expected. In general, a pointer to the object is used rather than the object itself.

2.1. Class Hierarchy

```
FrObject
   |
   +------ FrWidget
             |
             +----- FrWButtonBar
             +----- FrWForm
             +----- FrWFrame
             +----- FrWLabel
             +----- FrWList
             +----- FrWMainWindow
             +----- FrWOptionMenu
             +----- FrWRadioBox
             +----- FrWRowColumn
             +----- FrWScrollBar
             +----- FrWScrollWindow
             +----- FrWSelectionBox
             +----- FrWSeparator
             +----- FrWSlider
             +----- FrWArrow
             |       |
             |       +----- FrWArrowG
             |
             +----- FrWPopupMenu
             |       |
             |       +----- FrWPulldownMenu
             |       |
             |       +----- FrWPromptPopup
             |               |
             |               +----- FrWDialogPopup
             |               +----- FrWFramePrompt
             |
             +----- FrWCascadeButton
             +----- FrWPushButton
             |       |
             |       +----- FrWPushButtonG
             |       +----- FrWToggleButton
             |
             +----- FrWText
             |       |
             |       +----- FrWShadowText
             |       +----- FrWFrameCompleter
             |
             +----- FrWProgressIndicator
                     |
                     +----- FrWProgressPopup
```

Table 2-2: The FramepaC Data-Types (Motif Widgets)

A `FrObject`, as its name implies, is a generalized object which can be of any of the Lisp-like types shown in Table 2-1, including `FrSymbol` and `FrList`. This is the most general of the data types used in the FramepaC interface.

The `FrFrame` is the main data type, which most of the other types were created to support in some fashion. A frame contains one or more slots, each with one or more facets containing some number of fillers. A `VFrame` is a virtualized frame, which need not exist in memory until explicitly accessed.

A `FrSymbol` is the exact equivalent of a Lisp symbol. For any given sequence of characters, there is exactly one `FrSymbol`[1], so a `FrSymbol` may always be retrieved by name, no matter where that name is actually stored.

[1]More accurately, one per symbol table, since the program may explicitly create additional symbol tables after it starts.

```
(FrObject)
   |
   +------ VFrameInfo
            |
            +----- VFrameInfoFile
            +----- VFrameInfoServer
(FrHashEntry)
   |
   +------ HashEntryVFrame
   |
   +------ HashEntryServer

FrSlot
FrFacet

FrServer
FrClient

FrSymHashTable
```

Table 2-3: FramepaC-internal Types

Each `FrSymbol` may have a `FrFrame` associated with it, in which case the `FrSymbol` is the name of that frame.

A `FrList` is the basic data type used to store values in a frame. It contains a sequence of `FrObject`s, which may naturally be `FrLists` themselves. The type `FrCons` is similar to `FrList`, joining two arbitrary objects, whereas `FrList` joins an arbitrary object to the front of a shorter list.

Numbers may be included in lists and frames by creating an object of the appropriate subclass of type `FrNumber`: an `FrInteger` for any value that may be held in a `signed long`, or a `FrFloat` for any other number that may be held in a `double`.

2.2. Enumerated Types

In addition to the class hierarchy described in the previous section, FramepaC exports a number of enumerated types.

Most predicates return a boolean value of type `FrBool`, which may be:

```
False
True
```

The type of an object may not be known until run time, so each object can return its own type as a value of `ObjectType` using the `objType()` method:

```
OT_FrObject,
OT_Frame,
OT_VFrame,
OT_VFrameInfo,
OT_FrAtom,
OT_FrCons,
OT_FrNumber,
OT_FrString,
OT_FrSymbol,
OT_FrList,
OT_FrStruct,
OT_FrFloat,
OT_FrInteger,
```

```
   OT_FrQueue,
   OT_FrArray,
   OT_FrBitVector,
   OT_FrHashTable,
   OT_FrHashEntry,
   OT_FrStream,
   OT_FrISockStream,
   OT_FrOSockStream,
   OT_FrSockStream,
   OT_FrWidget,
   OT_FrWSeparator,
   OT_FrWFrame,
   OT_FrWArrow, OT_FrWArrowG,
   OT_FrWLabel,
   OT_FrWRowColumn,
   OT_FrWList,
   OT_FrWOptionMenu,
   OT_FrWPopupMenu,
   OT_FrWPulldownMenu,
   OT_FrWPromptPopup,
   OT_FrWDialogPopup,
   OT_FrWPushButton,
   OT_FrWPushButtonG,
   OT_FrWToggleButton,
   OT_FrWCascadeButton,
   OT_FrWForm,
   OT_FrWSlider,
   OT_FrWButtonBar,
   OT_FrWText,
   OT_FrWShadowText,
   OT_FrWFramePrompt,
   OT_FrWFrameCompleter,
   OT_FrWTextWindow,
   OT_FrWMainWindow,
   OT_FrWScrollWindow,
   OT_FrWScrollBar,
   OT_FrWRadioBox,
   OT_FrWSelectionBox,
   OT_FrWProgressIndicator,
   OT_FrWProgressPopup,
```

When a value is not present in the frame being accessed, it may optionally be retrieved from another frame according to the current inheritance method, which is one of the values in `InheritanceType`:

```
   NoInherit,         // no inheritance
   InheritSimple,     // only look at first filler of IS-A/INSTANCE-OF slots
   InheritDFS,        // depth-first search on IS-A
   InheritBFS,        // breadth-first search on IS-A
   InheritPartDFS,    // depth-first search on PART-OF
   InheritPartBFS,    // breadth-first search on PART-OF
   InheritUser,       // call user-provided function for inheritance
and optionally
   InheritLocalDFS,   // follow slot’s INHERITS facet before doing DFS
   InheritLocalBFS,   // follow slot’s INHERITS facet before doing BFS
```

The optional support for demons defines an enumerated type `DemonType` to specify which of multiple different kinds of demons is to be added or removed. The different demon types are

```
DT_IfCreated         // the indicated slot/facet was newly created
DT_IfAdded           // the indicated filler is about to be added
DT_IfRetrieved       // the indicated slot/facet has been retrieved
DT_IfMissing         // the indicated slot/facet needs inheritance
DT_IfDeleted         // the indicated filler has just been deleted
```

2.3. Miscellaneous Types

When accessing `FrString`s, the characters in the string may be 8, 16, or 32 bits wide. The type `FrChar_t` will hold any character in any `FrString`.

Many functions take an optional `FrCompareFunc` to determine whether two `FrObject`s are equal. Such a comparison function is defined as

```
FrBool func(const FrObject *obj1, const FrObject *obj2) ;
```

Many of the iteration functions take a function argument of type `FrIteratorFunc` which is called on each iteration. This iteration function is defined as

```
FrBool func(const FrObject *, va_list) ;
```

The `mapcar` and `mapcan` functions require a mapping function of type `FrListMapFunc`, which is

```
FrObject *func(const FrObject *obj, va_list args) ;
```

Demon functions have a particular signature, which is defined as type `DemonFunc`, and is

```
FrBool func(const FrSymbol *frame, const FrSymbol *slot,
             const FrSymbol *facet, const FrObject *filler,
             va_list args) ;
```

The `filler` argument is used only by the IfAdded and IfDeleted types of demon, and will be 0 for all other types of demons.

When calling `synchronize_VFrames`, you may optionally provide a function to be called as each frame is stored; such a function has the following signature:

```
void frame_update_hookfunc(FrSymbol *framename) ;
```

3. Programming with FramepaC Functions

This chapter provides an introduction to the use of FramepaC functions; a complete listing of the available functions is in Chapter 7. FramepaC has both a procedural interface and an object-oriented one; with a few exceptions, the procedural interface provides a subset of the object-oriented interface's functionality (in fact, most of the procedural interface is a simple veneer or "shim" that converts the procedural calls into object method calls). This chapter will concentrate on the procedural interface.

The very first FramepaC function a program must call is `initialize_FramepaC`, which prepares FramepaC for operation. Any attempt to call other functions before this one is considered an error, with consequences ranging from none to a system crash.

3.1. Starting Out: Symbols, Lists, and Frames

In order to work with `FrSymbol`s, one must first generate or find some. Consider the following code fragment:

```
FrSymbol *sym1 = makeSymbol("TEST") ;
FrSymbol *sym2 = findSymbol("FOO") ;
```

The first line in the example finds or creates the symbol whose name is TEST, and sets `sym1` to point at that symbol. The second line attempts to find the symbol whose name is FOO; if no such symbol has been created yet, it sets `sym2` to 0. Note that, unlike the FramepaC reader (discussed below), neither `makeSymbol` nor `findSymbol` convert the string to uppercase--the exact string specified, including any special characters, becomes the symbol's name.

Two other methods of generating a symbol are to read one into FramepaC with either the overloaded `>>` operator (discussed in more detail in Section 3.3) or `string_to_FrObject`, or to create a unique symbol with `gensym`. For example,

```
FrObject *object ;
FrSymbol *symbol ;
char *charptr = "SYMBOL1 DATA DATA DATA" ;

cin >> object ;
object = string_to_FrObject(charptr) ;
                        symbol = gensym("BASENAME") ;
```

Here, the first two functions return a `FrObject` which happens to be a symbol (which can be tested using the function `SYMBOLP`), and the last line returns a symbol whose name is of the form `BASENAME1234`.

The three main methods of creating a list are the functions `pushlist` and `makelist` and the `FrList` constructor. `pushlist` constructs a list one element at a time, while the other two methods require that all elements in the list be given at once (`makelist` takes an arbitrary zero-terminated sequence of `FrObject`s, while the `FrList` constructor takes a maximum of four items). The following code fragment builds three lists, each consisting of the objects pointed at by `obj1`, `obj2`, and `obj3`, in that order.

```
FrList *l1, *l2, *l3 ;
FrObject *obj1, *obj2, *obj3 ;

...
// set obj1, obj2, and obj3
...
l1 = 0 ;
pushlist(obj3,l1) ;     // each pushlist() adds one item to
pushlist(obj2,l1) ;     // the front of the list, so the calls
pushlist(obj1,l1) ;     // must be in the opposite order
l2 = makelist(obj1,obj2,obj3,(FrObject*)0) ;
l3 = new FrList(obj1,obj2,obj3) ;
```

Frames are rather complex structures, so building one from scratch programmatically takes a number of function calls. They may also be read into memory from a stream or character string, using either a native textual representation or a FrameKit-compatible `MAKE-FRAME` or `MAKE-FRAME-OLD` representation.

To build a frame, one must first create it, and then add the various fillers it contains (the necessary slots and facets to hold the fillers are created automatically as required). Creating a frame is done with a call such as

```
FrFrame *fr = create_frame(makeSymbol("TEST-FRAME")) ;
```

after which fillers may be added either singly or an entire list at a time:

```
add_filler(fr,makeSymbol("IS-A"),makeSymbol("VALUE"),
                makeSymbol("OTHER-FRAME")) ;
add_fillers(fr,makeSymbol("DATA"),makeSymbol("VALUE"),list_of_data) ;
```

Reading in a frame is done using the same functions used for reading in other types of objects; the functions described in Section 3.2 may be used to determine what kind of an object was read, i.e.

```
FrObject *object ;

cin >> object ;
if (FRAMEP(object)) ...
object = string_to_FrObject(charptr) ;
if (FRAMEP(object)) ...
```

if the frame is in native format. FramepaC can also manipulate frames which were printed in FrameKit's `MAKE-FRAME` or `MAKE-FRAME-OLD` formats. In this case, the read function will return a list which may be converted to an actual frame with the function `FrameKit_to_FramepaC`. As a code fragment, this operation would be

```
FrObject *object ;
FrFrame *frame ;

cin >> object ;
if (CONSP(object) && (listhead(object)==makeSymbol("MAKE-FRAME") ||
                      listhead(object)==makeSymbol("MAKE-FRAME-OLD")))
   {
   frame = FrameKit_to_FramepaC(object) ;
   free_object(object) ;
   }
```

The overloaded operator `<<` for streams may be used to output any FrObject. This facility is somewhat simplistic in the current version, and performs a minimum of pretty-printing and no circularity checks. Therefore, lists will not wrap to fit on screen and any FrCons which ultimately points at itself will result in an infinite loop.

3.2. Dealing with Polymorphism

Since FramepaC uses C++'s polymorphism to allow an object of a more general type to actually contain a more specific type of object, we need a way to determine the exact type of an object. For this purpose, FramepaC provides a number of type determination functions.

`ATOMP()`, `CONSP()`, `SYMBOLP()`, `STRINGP()`, `NUMBERP()`, and `FRAMEP()` return `True` if the `FrObject` they are passed is, respectively, a FrAtom (symbol, string, or number), a FrCons (cons cell or list), a symbol, a string, a number (either FrInteger or FrFloat), or a frame located in memory. Additionally, the function `is_frame` determines whether the `FrSymbol` it is given is the name of a frame, and the method `objType` returns a value indicating the exact type of the object.

Consider the case where the user is asked to enter an object, and one function (requiring a symbol) is called if the user enters a symbol, and another one (able to accept any type of object) is called if it is not a symbol:

```
FrObject *object ;

cin >> object ;
if (SYMBOLP(object))
   functionA((FrSymbol *)object) ;
else
   functionB(object) ;
```

or

```
FrObject *object ;

cin >> object ;
if (object && object->objType() == OT_Symbol)
   functionA((FrSymbol *)object) ;
else
   functionB(object) ;
```

3.3. The FramepaC Reader

Because reading and writing Lisp-style objects is a common operation, FramepaC provides a set of functions to do just that. Note, however, that the FramepaC reader is not a complete implementation of a Lisp reader, so some constructs will not work or produce unexpected results when given data intended for a Lisp reader.

The first character of an object is used to determine its type. A left parenthesis indicates a `FrList`; a digit indicates a `FrNumber`, as does a plus or minus sign (unless the following character is a non-digit; whether the number is an `FrInteger` or a `FrFloat` is determined by examining the number for a decimal point and/or exponent); a left bracket indicates a `FrFrame`; a double, single, or back- quote indicates a `FrString` consisting of eight-, sixteen-, or 32-bit characters, respectively; a hash mark ('#') indicates the start of a Lisp-style form; a semicolon indicates the start of a comment running to the end of the line; and an alphabetic character, vertical bar, or one of a number of special symbols indicates a `FrSymbol`. Any other non-whitespace character will be returned as a one-character `FrString`.

The native format for a frame is

```
[<framename> <numslots>
   [<slotname>  [<facetname> <filler> ...]
                [<facetname> <filler> ...]
                ...]
   [<slotname>  [<facetname> <filler> ...]
                [<facetname> <filler> ...]
                ...]
   ...]
```

where <numslots> is the number of slots listed in the remainder of the frame representation. This number is optional and may be omitted when writing frames, but is generated on output to make reading the frame back into FramepaC more efficient. In the special case of zero slots, i.e.

```
[FOO 0]
```

or

```
[FOO]
```

the frame FOO is created if it does not already exist, but not modifed if it does exist.

FramepaC can also process frames in FrameKit format, though it reads them as lists rather than directly into frames. The function FrameKit_to_FramepaC will convert a list in FrameKit "make-frame" or "make-frame-old" format into a FramepaC frame. The two FrameKit formats are

```
(MAKE-FRAME <framename>
    (<slot1> (<facet1> (<view1> <filler> ...) (<view2> <filler> ...))
             (<facet2> (<view1> <filler> ...) ...))
    (<slot2> ...)
    ...)
```

and

```
(MAKE-FRAME-OLD <framename>
    (<slot1> (<facet1> <filler> ...) (<facet2> <filler> ...) ...)
    (<slot2> (<facet1> <filler> ...) ...)
    ...)
```

where the view for MAKE-FRAME-OLD is assumed to be COMMON.

When reading in symbols, FramepaC converts their names to uppercase by default. This may be overridden (and characters not normally considered to be part of a symbol's name may be embedded) by surrounding the symbol's name with 'pipe' or vertical bar characters ('|'). If the name is quoted in this way, it may contain any character except NUL or vertical bar, including whitespace and newlines. Any symbols whose names contain lowercase letters or characters not normally considered part of a symbol name will be quoted in this manner when they are printed. Note that the functions makeSymbol and findSymbol, used to access symbols by name, do not perform any uppercasing on the names they are given, and will include any vertical bar characters as part of the name if present (this is the only way to create a symbol name containing a vertical bar, which makes them useful for internal symbols which are never printed out). Examples:

```
CAR             ; symbol with name "CAR"
car             ; symbol with name "CAR" -- same as previous
|CAR|           ; symbol with name "CAR" -- same as first
|car|           ; symbol with name "car"
car one         ; two symbols: "CAR" and "ONE"
|car one|       ; symbol with name "car one"
+               ; symbol with name "+"
+1              ; number--not a symbol
```

Strings may be fed to FramepaC using either single or double quotes or backquotes. The type of quotes used indicates the size of the characters in the string (though the printed representation of the string always breaks everything into eight-bit bytes): a double quote is a string consisting of normal eight-bit characters, a single quote is a string of 16-bit characters (e.g. Unicode), and a backquote is a string of 32-bit characters (e.g. ISO 10646). A backslash is used to quote the following character (allowing quotes to be embedded); thus, to enter a literal backslash into the string, it must be doubled. One special case is recognized so that NUL characters may be embedded in a string--if the backslash is followed by the character '0' (zero), the two-character sequence is replaced by an ASCII NUL (the character whose value is binary zero). Examples:

```
"abc"           ; 8-bit characters, 'a', 'b', and 'c'
"abc\0"         ; 8-bit characters, 'a', 'b', 'c', and NUL
'\0a\0b'        ; 16-bit characters, 'a' and 'b'
`\0\0\0a`       ; 32-bit character 'a'
"\"\\"          ; 8-bit characters, double quote and backslash
```

Numbers are automatically stored in the proper type of `FrNumber` object, either a `FrInteger` or a `FrFloat` depending on the number's format. If the number contains no decimal point and no exponent, it is stored as an integer; if it contains either or both, it is stored as a floating-point number. Examples:

```
1               ; integer, read as an object of type FrInteger
1.0             ; floating-point number, read as a FrFloat
+1.5            ; positive one point five
-1              ; integer, minus one
1.5e6           ; floating-point 1,500,000
-2E-3           ; floating-point -0.002
```

The hash character ('#') introduces a Lisp-style form. FramepaC can interpret the most common forms, although several are only approximated (e.g. a #A() array is read as a simple list, since FramepaC does not have an array type). Within a Lisp-style form, only eight-bit strings can be read, since the single quote and backquote are interpreted as Lisp would, namely as abbreviations of (QUOTE x) and (BACKQUOTE x).

FramepaC supports the following Lisp and Lisp-style forms:

- **#()**
  Vector. This form is read as a simple `FrList`. As in Lisp, an infix argument may be given to specify the length of the vector; in this case, the list is truncated to the specified length if longer, or extended by duplicating the last element if shorter than specified.
- **#‘**
  Function. This form returns the `FrList` (FUNCTION x), where **x** is the object immediately following the #‘. For example, #‘TEST would return (FUNCTION TEST).
- **#**
  Character. Returns a `FrSymbol` whose one-character name is the character immediately following the backslash, regardless of what that character may be (whitespace, control character, etc.), and without converting the character to uppercase. For example, #\a returns the `FrSymbol` |a|.
- **#:**
  Uninterned Symbol. Since FramepaC has no notion of interned vs. uninterned, this form simply returns the symbol immediately following the *#:*.
- **#|**
  Balanced Comment. Ignore all further input until a |# sequence is encountered. Unlike Lisp, FramepaC does not support nested balanced comments; the first |# encountered ends the comment.
- **#*n*#**
  Shared Object Reference. Returns a pointer to a *copy* of the object defined by a `#n=` earlier in the same list or Lisp form. Since FramepaC does not have garbage collection, the structure sharing implied by #= and ## is not implemented to avoid serious problems and complications in deallocating the object returned by a read operation.
- **#A()**
  Array. This form is read as a simple `FrList`.
- **#B***nnn*
  Binary rational number. If the string of binary digits following the #B contains a forward slash, the return value is a `FrFloat` containing the ratio of numerator and denominator; otherwise, the return value is a `FrInteger`.
- **#H()**
  Hash Table. The items between the parentheses are interpreted as the entries in a `FrHashTable`.

- **#HEntryObj()**
  Hash Table Entry. The value within the parentheses is interpreted as the key for a `FrHashEntryObject`.
- **#O***nnn*
  Octal rational number. If the string of octal digits following the #B contains a forward slash, the return value is a `FrFloat` containing the ratio of numerator and denominator; otherwise, the return value is a `FrInteger`.
- **#Q()**
  Queue. The items between the parentheses are interpreted as the contents of a `FrQueue`, from head to tail.
- **#***m***R***nnn*
  Variable-radix rational number. 'm' is a number between 2 and 36 indicating the radix in which to read the digits 'nnn' following the #R. The digit sequence may include a forward slash, in which case the returned value is the ratio of the indicated numerator and denominator.
- **#S()**
  Structure. The contents between the parentheses are interpreted as field-name and field-value pairs, as in Lisp. The return value is a `FrStruct`.
- **#X***nnn*
  Hexadecimal rational number. If the string of hexadecimal digits following the #X contains a forward slash, the return value is a `FrFloat` containing the ratio of numerator and denominator; otherwise, the return value is a `FrInteger`.

Any sequence introduced by a hash ('#') which is not listed above, or for which the code implementing the type of the return value has not been linked into the executable, is returned as a `FrSymbol` with the appropriate name (including the leading hash character). By only linking in the readers for types which are actually used by the program, the size of the executable is minimized and the application will not receive objects which might confuse it because they are unknown to the application's code.

It is important to remember that the FramepaC reader is not a full Lisp reader, so various constructs such as #A() for arrays will not be read properly (though in most cases, an approximation will be made, e.g. arrays are read as lists and rationals as floats). Since FramepaC does not have anything corresponding to Lisp's packages (which allow multiple namespaces which are still simultaneously accessible), symbols of the form "PACKAGE:SYMBOLNAME" will be stored with precisely that name in the current symbol table.

Like Lisp's reader, FramepaC's is designed to be user-extensible, either at compile-time or at run-time. Compile-time extension may be achieved by instantiating objects of the `FrReader` class; run-time extension is achieved by explicitly setting the functions to be called when specified characters are encountered or by creating additional instances of `FrReadTable` (see the appropriate sections of Chapter 7 for details). For example, to transparently convert FrameKit frames into FramepaC frames when reading, one could use

```
FrReadStreamFunc list_reader ;
FrObject *list_or_frame(istream &in, const char *digits)
{
   FrObject *list = list_reader(in,digits) ;
   if (list && && list->consp() && ((FrList*)list)->first())
      {
      FrObject *head = ((FrList*)list)->first() ;
      if (head == makeSymbol("MAKE-FRAME") ||
          head == makeSymbol("MAKE-FRAME-OLD"))
         {
         FrFrame *frame = FrameKit_to_FramepaC(list) ;
         list->freeObject() ;
         return frame ;
         }
      }
   return list ;
}

....
list_reader = FramepaC_readtable->getStreamReader('(') ;
FramepaC_readtable->setReader('(',list_or_frame) ;
....
```

3.4. Using Virtual Frames

Virtual frames allow you to store your frames in a file or on a server and retrieve them on demand. Using virtual frames requires a few changes from using regular frames, but can mostly be treated identically with regular frames (for examples, regular and virtual frames may be mixed freely).

First, the virtual-frame functions do not take actual frames as arguments, but the *names* of the frames. Thus, all variables which would be of type `FrFrame*` for regular frames are instead of type `FrSymbol*`. Only in exceptional cases would an actual pointer to a frame be used, in which case its type can be either `VFrame*` or `FrFrame*` (since `VFrame` is a subclass of `FrFrame`).

Second, the virtual-frame system must be initialized. Thus, the program must call either `initialize_VFrames_memory`, `initialize_VFrames_disk`, or `initialize_VFrames_server` before using virtual frames. These initialization functions set up the backing store and load in an index of the existing frames so that FramepaC knows which frames should be fetched when accessed and which need to be newly created. Before the program exits, it must call `shutdown_VFrames` to ensure that all necessary updates to the backing store are performed.

Finally, there should be periodic calls to `synchronize_VFrames` or `commit_frame` to ensure that changes are stored back in the file or on the server in a timely manner. Although any modified frames will be stored when `shutdown_VFrames` is called, regular calls to enforce updates of the backing store will minimize the number of updates lost in case of a crash, and may be required in multi-user situations when others are accessing the same backing store.

3.5. Manipulating Lists

Section 3.1 showed how to create a list, but did not go into details on how to use lists. This section will do just that.

As in Lisp, a list consists of a series of FrCons cells connecting two other items. The left half of the FrCons is the first item in the list, while the right half is the remainder of the list. Not surprisingly, then, FramepaC provides functions to get both the first item in the list and the remainder after removing the first item, called `listhead` and `listtail`. The empty list is a simple null pointer, i.e. 0, and both head and tail of the empty list are defined to be 0 (the empty list) for convenience.

The code fragment

```
FrList l = new FrList(new FrString("first"),
                      new FrString("second"),
                      new FrString("third")) ;
cout << "The first item in 'l' is " << listhead(l) << endl ;
cout << "and the remainder is " << listtail(l) << endl ;
```

will produce the output

```
The first item in 'l' is "first"
and the remainder is ("second" "third")
```

There are two main ways of processing each element of a list, depending on whether the list is to be discarded as it is processed. If the list is a temporary list or a copy which is not pointed at by any other variables, the function `poplist` will simultaneously return the first element of the list and remove it from the list, deallocating the list's first FrCons cell. To print out each element of a list, one per line, you would use a loop such as

```
while (lst)
   {
   FrObject *item = poplist(lst) ;
        ...
   }
```

On the other hand, if the list must be preserved, the usual method would be

```
while (lst)
   {
   FrObject *item = listhead(lst) ;
        ...
   lst = listtail(lst) ;
   }
```

or

```
for (FrList *lst = somelist ; lst ; lst = listtail(lst))
   {
   FrObject *item = listhead(lst) ;
        ...
   }
```

The functions which return entire facets from a frame (`get_fillers`, `get_values`, and `get_sems`) return the actual lists stored in the frame. Thus, the returns lists must not be destructively modified. If it is necessary to destructively modify such a list of fillers, you must first copy the list, and then operate on the copy. The two functions provided for copying lists are `copylist` (`FrList::copy`) and `copytree` (`FrList::deepcopy`), which perform shallow and deep copies, respectively. `copylist` will only copy the top-most level of the list, so the elements pointed at in the copy will be the actual elements from the original list; in contrast, `copytree` also copies the elements in the list, so the returned copy will be completely separate from the original list. In general, `copytree` is the preferred function; `copylist` is typically only useful when the copy will be destroyed with `poplist`, which deletes the nodes of the list but not the elements they point at.

!!!

Other useful functions include `FrList::listlength`, `FrList::member`, and `listreverse`. The first function, `FrList::listlength`, returns the number of items at the top level of the list (skipping any items in embedded lists). Determining whether some item is present in the list is done with `FrList::member`, which returns a pointer to the end-portion of the list with the desired item as its head, or 0 if the item is not in the list. Finally, `listreverse` destructively reverses the items in the list, returning a pointer to the new beginning of the list.

```
FrList *l = new FrList(symbolONE, symbolTWO, symbolTHREE, symbolFOUR);
cout << "The list's length is " << l->listlength() << " items" <<endl;
cout << "The tail from THREE is " << l->member(symbolTHREE)
     << endl ;
l = listreverse(l) ;  /* must store new head, or lose most of list */
cout << "The reversed list is " << l << endl ;
l->freeObject() ;
```

would produce the following output:

```
The list's length is 4 items
The tail from THREE is (THREE FOUR)
The reversed list is (FOUR THREE TWO ONE)
```

Note that if all you want to do is determine whether a list is empty or not, it is easier and much more efficient to simply check whether the pointer is NULL rather than to call `listlength`, i.e.

```
if (list == 0)
   ...
```

or

```
if (!list)
   ...
```

instead of

```
if (list->listlength() == 0)
   ...
```

3.6. Manipulating Frames

Obviously, once a frame has been created as described in Section 3.1, one must be able to do more with it than merely print it out. This section covers a variety of ways to access and modify frames.

A frame would not be of much use if it were not possible to retrieve previously-stored information from the frame. The main function to do so is `get_fillers`, which returns the list of all items stored in a particular facet within a given slot. For efficiency, the `VALUE` and `SEM` facets may be retrieved directly with `get_values` and `get_sems`, respectively. Since one is commonly interested only in the first filler in a facet, the variants `first_filler`, `get_value`, and `get_sem` are available to return the first filler, and are functionally equivalent to `listhead(get_fillers(...))`, etc. For example,

```
cout << "The list of SEM fillers in the frame's TEST slot is "
     << get_sems(frame,makeSymbol("TEST")) << endl ;
cout << "The first VALUE filler in the frame's TIME slot is "
     << get_value(frame,makeSymbol("TIME")) << endl ;
cout << "The DEFAULT facet in the frame's DATA slot is "
     << get_fillers(frame,makeSymbol("DATA"),makeSymbol("DEFAULT"))
     << endl ;
```

The primary means of changing a frame is to delete one or more fillers and then optionally add new ones. The set of functions to erase portions of a frame includes `erase_filler`, `erase_value`, `erase_sem`, `erase_facet`, and `erase_slot`. The first three functions remove a specific item from an arbitrary facet, the `VALUE` facet, and the `SEM` facet, respectively. The fourth, `erase_facet` removes all fillers from a particular facet, while the last, `erase_slot`, removes all fillers from all facets of the indicated slot.

```
/* remove XYZZY from the VALUE facet of the TEST slot */
erase_value(frame,makeSymbol("TEST"),makeSymbol("XYZZY")) ;
/* remove all fillers from the SEM facet of the DATA slot */
erase_facet(frame,makeSymbol("DATA"),makeSymbol("SEM")) ;
/* remove the entire TIME slot and all its facets */
erase_slot(frame,makeSymbol("TIME")) ;
```

Three functions are provided to process portions of a frame with no prior knowledge of the slots and facets in the frame, as well as two functions to retrieve a list of the slots and facets in the frame. These are `do_facets`, which iterates over the facets of a slot; `do_slots`, which iterates over the slots in a frame; `do_all_facets`, which iterates over all facets of each and every slot in the frame; `slots_in_frame`, which returns a list of the slots in the frame; and `facets_in_slot`, which returns a list of the facets in a particular slot . Each of these takes a pointer to a function as one of its arguments, and invokes the specified function for each slot or facet it iterates over. Additional parameters may be given to the `do_*` functions, which are passed through unchanged to the user function in the form of a variable argument list (`va_list`), which the standard macro `va_arg` can decompose into the individual parameters. For example, consider the following brief function to print all of the slot and facet names in a frame to a specified stream:

```
FrBool show_facet_name(const FrFrame *frame, const FrSymbol *slot,
                     const FrSymbol *facet, va_list args)
{
   ostream *out = va_arg(args,ostream*) ;
   *out << slot << " " << facet << endl ;
   return True ;
}

        ...
        do_all_facets(frame,show_facet_name,&cout) ;
        ...
```

3.7. Socket Streams

For convenience in creating distributed applications, FramepaC provides three classes which extend the standard I/O streams to functions over BSD Unix-style network sockets: `FrISockStream`, `FrOSockStream`, and `FrSockStream`. These are derived from, respectively, `istream`, `ostream`, and `iostream`.

Once opened, socket streams may be used in exactly the same manner as any other simple stream. Thus -- other than a few lines of setup -- the identical source code may be used whether your program reads/writes standard input/output or is acting as a network server.

!!!

3.8. A More Complex Example

!!!

3.9. Efficiency Considerations

If you have a constant symbol which will be used repeatedly in the course of execution (i.e. `VALUE` or `SEM`), you can save the overhead of looking up the symbol each time by storing the result of a `makeSymbol()` or `findSymbol()` in a variable at the beginning of the program run, such as in an initialization function. Then, use the value of that variable whenever you need that particular symbol--symbols are guaranteed never to change while the symbol table containing them remains in existence. FramepaC uses this approach internally, with variables such as `symbolISA` and `symbolVALUE` (which are not accessible outside FramepaC, however).

If you switch symbol tables, you will need to refresh the variables you use to store symbols, since symbols with the same names will differ between symbol tables, and can not validly be compared.

Virtual frames are somewhat slower than regular frames even after they have been loaded into memory, because there is an extra level of lookups on each frame access. Thus, it can be useful in time-critical code to force a virtual frame which is accessed multiple times in a row into memory, and then operate on the frame as if it were a regular frame. In a multi-user environment, where other users might access the same backing store, the frame should be locked while these manipulations are in progress. The function `lock_frame` will simultaneously lock the frame and force it into memory, returning a pointer to the actual frame.

4. Demons

FramepaC can optionally support daemon functions which are activated whenever a particular slot is created, deleted, accessed, modified, or found to be absent. Because the support for demons slows down the entire system (by about 10 percent even when no demons have been defined, more when demons are active[2]), a compile-time option specifies whether to include the daemon support. A user program may determine whether demons are supported by testing for the existence of the preprocessor variable `FrDEMONS`.

Using demons is fairly simple; using demons well is an art form.

!!!

[2] In FrameKit, the overhead imposed by checking for demons was far larger, on the order of 300% for the faster functions such as `get-fillers`; FramepaC uses a different and much faster method of storing and checking for demons.

5. Memory Allocation

Virtually all memory allocation in FramepaC is performed through its own internal memory allocation routines, which are also made available to applications. There are two separate but closely-related sets of routines: a standard, variable-size memory allocator equivalent to `malloc`/`free`, and an extremely fast allocator for fixed-size objects. To avoid excessive memory fragmentation, it is advisable to use FramepaC's routines instead of the standard `malloc`/`free` or `new`/`delete`.

5.1. Standard Memory Allocations

The following convenience macros are available for memory allocations:

`type`* *FrNew(*`type`*)*

Allocate an object of the indicated type, and return a pointer to the newly-allocated object.

`type`* *FrNewC(*`type`*,int)*

Allocate an array of objects of the indicated type, clear the allocated memory to all zero bytes, and return a pointer to the first object in the newly-allocated array.

`type`* *FrNewN(*`type`*,int)*

Allocate an array of objects of the indicated type, and return a pointer to the first object in the newly-allocated array.

`type`* *FrNewR(*`type`*,void *blk,int newsize)*

Resize a previously-allocated block of memory to the specified new size in bytes. Returns a pointer to the resized block (which may differ from the original pointer), or 0 if unable to resize the block (in which case the original remains untouched).

Additional functions, such as `FrMalloc` and the like on which the above macros are based, related to memory allocation are listed in Section 7.59.3.

5.2. Memory Pools

Sometimes, it is useful to have a separate pool of memory from which allocations are made for a specific purpose. To this end, FramepaC provides the class `FrMemoryPool`, which essentially provides a wrapper around the standard memory allocation functions in such a way that the allocations made through an instance of `FrMemoryPool` are made from different regions of memory than those allocations made through any other instance of `FrMemoryPool` or through `FrMalloc` or the `FrNew`X convenience macros.

In addition to providing a separate region of memory for allocation, instances of `FrMemoryPool` may be set to maintain a reserve of memory which is allocated immediately and never used until a request for memory cannot be satisfied either from existing blocks in the memory pool or by reqeusting more memory from the system. This allows critical operations to complete successfully even when memory has otherwise been exhausted.

*FrMemoryPool::FrMemoryPool(const char *name, int reserve = 0,*
int maxblocks = 0)

Create a new memory pool, specifying a name for identification purposes, how many blocks of memory to hold in reserve, and the maximum number of blocks to allocate from the system before attempting to reclaim memory which has already been allocated. Each block used by the memory pool contains

FrBLOCKING_SIZE bytes; allocation request for more than that amount (less internal overhead) are passed directly to the system for fulfillment, and will not be affected by either the reserve or the limit settings.

The reserve parameter indicates how many blocks to allocate immediately but not use until requests for additional memory cannot be satisfied by system memory allocation calls.

The maxblocks parameter, if nonzero, indicates how many blocks may be allocated for the memory pool before attempting to reclaim memory. If the memory pool contains at least maxblocks allocation units and a request for memory cannot be satisfied from the existing pool, then memory compaction and/or garbage collection (depending on capabilities added by subclasses of FrMemoryPool such as FrAllocator) will be attempted and the system will be asked for additional memory only if the memory reclamation did not make sufficient memory available.

FrMemoryPool::~FrMemoryPool()

Delete the memory pool and return any memory it has allocated to the available global memory for the program.

*void *FrMemoryPool::allocate(size_t size)*

Request a block containing at least size bytes of memory from the memory pool.

*void FrMemoryPool::release(void *item)*

Return the specified item (previously allocated with allocate) to the memory pool for re-use.

*void *FrMemoryPool::reallocate(void *item, size_t newsize)*

Resize the specified memory block item to contain at least newsize bytes. If it is not possible to expand a memory block in place, a new allocation will be made and the contents of the existing memory block copied into the new one. This function returns the address of the resulting resized block.

void FrMemoryPool::set_max_blocks(int maxblocks)

Set the limit to the number of allocation units the memory pool may contain before this instance of FrMemoryPool attempts to reclaim memory prior to requesting additional memory from the system.

void FrMemoryPool::reserve_blocks(int reserved)

Set the number of blocks to hold in reserve against the possibility of the system memory being exhausted. If the new reserve is larger than the existing reserve, the necessary additional memory blocks will be allocated from the system immediately; if smaller, any extraneous blocks will be returned to the system.

*void FrMemoryPool::setCallback(FrMemoryPoolCallback *cb)*

Specify a function to be called whenever the status of the memory reserve changes. The three occurrences are FrMPCR_UsingReserve (a block from the reserve pool was used to satisfy a request), FrMPCR_ReservingBlock (a block was freed, but was added to the reserve pool rather than being returned to the system), and FrMPCR_IncreasingReserve (another block was added to the reserve pool).

int FrMemoryPool::blocks_allocated() const

Determine how many allocation units are currently in use by the memory pool (not counting any reserve).

long FrMemoryPool::bytes_allocated() const

Determine the total number of bytes currently in use by the memory pool, including any free memory being managed by the pool, but not any reserve.

int FrMemoryPool::max_blocks() const

Determine how many allocation units are allowed before the memory pool attempts to reclaim memory prior to requesting additional memory from the system.

int FrMemoryPool::reserved_blocks() const

Determine how many allocation units are to be kept in reserve for use by the memory pool when system memory is exhausted.

int FrMemoryPool::reserve_available() const

Determine the number of allocation units currently available as a reserve.

*const char *FrMemoryPool::typeName() const*

Retrieve the name for the instance of the memory pool, which was set at the time it was created.

*FrMemoryPool *FrMemoryPool::nextPool() const*

Return the next memory pool in the internal list of instances of `FrMemoryPool`. This function is not intended to be useful to applications.

*FrMemoryPoolCallback *FrMemoryPool::getCallback() const*

Retrieve the currently-active callback function for reporting changes to the memory pool's reserve.

5.3. The Suballocator

FramepaC includes its own very fast memory suballocator, and makes this suballocator available to applications for use with their own objects. The class **FrAllocator** implements a suballocation strategy within fixed-size blocks. What this means is that a block of memory is allocated from the system and treated as an array of smaller objects, each individually allocatable and stored on a free list when not actually in use by the application. This organization permits the smaller units to be allocated and deallocated with a handful of inline machine instructions, only rarely calling a function to allocate another block of memory.

The typical use of `FrAllocator` for allocating objects of a particular class requires three lines in the class' declaration and a fourth line elsewhere. In the class definition, one should declare a static variable of type `FrAllocator` and define `new` and `delete` in terms of `allocate` and `release`, as in:

```
class Foo
{
private:
   static FrAllocator allocator ;
   ...
public:
   void *operator new(size_t) { return allocator.allocate() ; }
   void operator delete(void *blk) { allocator.release(blk) ; }
   ...
}

FrAllocator Foo::allocator("Foo",sizeof(Foo)) ;
```

Note that `FrAllocator` is not recommended for objects of more than 1000 bytes or so, due to excessive internal fragmentation in the blocks from which the objects are suballocated.

*FrAllocator::FrAllocator(const char *name, int objsize)*

Prepare for suballocations of objects which are each `objsize` bytes. The indicated name is used to identify the suballocator instance in `show_memory_usage`'s output, and should ordinarily indicate the type of objects being allocated.

FrAllocator::~FrAllocator()

Release all memory still allocated by the instance of the suballocator.

*void *FrAllocator:allocate()*

Retrieve one object from the suballocator's memory pool and return it to the caller. If no more objects are left in the pool, a new block of memory is allocated from system memory and divided into objects, and one of the new objects is returned.

void FrAllocator:release()

Place the indicated object back into the suballocator's memory pool.

int FrAllocator::blocks_allocated()

Determine how many memory blocks the given instance of the suballocator is controlling.

int FrAllocator:bytes_allocated()

Determine the total number of bytes contained in the memory the suballocator is controlling.

int FrAllocator:objects_allocated()

Determine the total number of objects contained in the memory the suballocator is controlling.

int FrAllocator:freelist_length()

Determine how many objects are currently available for allocation without requesting further memory from the system.

*const char *FrAllocator:typeName()*

Retrieve the object type name specified at instantiation time.

int FrAllocator:objectSize()

Determine the size, in bytes, of one object, as specified at instantiation time.

*FrAllocator *FrAllocator:nextAllocator()*

Follow an internal list of all suballocator instances which are currently instantiated. This function is for internal use, and there is no means of retrieving the first instance on the internal list.

int FrAllocator:async_gc()

(not yet implemented, meant for internal use) !!!

int FrAllocator:gc()

(not yet fully implemented)
At this time, `gc` performs a memory compaction, returning any entirely unused suballocator blocks to the global memory pool.

!!!

int FrAllocator:compact()

Release any currently-unused memory blocks controlled by the given instance of the suballocator.

6. User-Interface Functions

When compiled under Unix with X Windows, FramepaC can provide a set of classes for manipulating Motif user-interface objects. When compiled with the `FrMOTIF` symbol *#define*d in `frconfig.h`, the `FrWidget` class and all of its subclasses (as shown in Figure 2-2) are available, but will only be linked into the executable if actually used by the application.

Using `FrWidget` and its subclasses instead of direct Xlib and Motif calls can not only simplify programming and potentially improve portability, it will also speed up compilations by avoiding the many thousands of lines of X and Motif header files.

!!!

It is hoped that a future revision of FramepaC may provide support for Microsoft Windows through the `FrWidget` heirarchy, thus allowing applications to be ported from X to Microsoft Windows with no source-code changes other than the initialization call.

7. Function Reference

The functions provided by FramepaC are declared in the include file `FramepaC.h`. This file includes various other files, so the directory containing it must be placed on the compiler's include-file path (i.e. with the `-I` switch).

7.1. Initialization Functions

FramepaC must be initialized before any of the remaining functions described in this document may be used. This is accomplished by calling one of the following functions:

void initialize_FramepaC(int max_symbols = 0)

Initialize the FramepaC system and create a symbol table for the specified number of symbols (which will be expanded automatically if needed--the only detrimental effect of too small a size will be increased run-time). If 'max_symbols' is omitted or 0, the default size set at compile-time will be used.

*void FrInitializeMotif(const char *window_name, Widget parent,*
int max_symbols = 0)

Initialize the FramepaC sytem to use the Motif user interface. This function should be used if the Motif system will be explicitly initialized before FramepaC is initialized; `parent` is the Widget for the toplevel application window, and `window_name` is the title which should be given to the FramepaC message window (which displays all `FrError` and `FrMessage` output). The next function may be used if you do not wish to concern yourself with Motif initialization.

*Widget FrInitializeMotif(int *orig_argc, char **orig_argv,*
*const char *maintitle, const char *msgtitle,*
*const char *icon_name,*
FrBool allow_resize = True)

Initialize both FramepaC and the Motif user interface. This function requires both the original `argc` and `argv` passed to `main`, which it will update to remove any Motif-specific commandline arguments. The other three strings give the titles for the full toplevel window, the FramepaC message window, and the application's icon, respectively. Finally, `allow_resize` indicates whether the user should be allowed to resize the application's main window using the X window manager.

If operation with virtual frames is desired, one of the following functions must be called before the first attempt to use virtual frames. **Failure to call one of these functions before attempting to create or load virtual frames is an error and can cause a variety of undesired behavior**.

*void *initialize_VFrames_memory(int symtabsize)*

Set up the VFrames system to operate only in memory, without any kind of backing store. After being set up with this functions, `VFrame` calls will operate identically to the corresponding `FrFrame` calls. This function returns a newly-created symbol table of the specified size (default size if 0 specified) if successful or 0 on error.

Each call to `initialize_VFrames_*()` creates a new symbol table and makes it the current symbol table, permitting multiple sets of virtual frames each with a different kind of backing store. It is up to the user to switch between symbol tables as appropriate before attempting to access frames.

This function should be called after `initialize_FramepaC` but before any use of virtual frames functions.

*void *initialize_VFrames_disk(char *filename, int symtabsize, FrBool transactions, FrBool force_create = True)*

Set up the VFrames system to use a disk file as backing store for the frames. This function returns a newly-created symbol table of the specified size (default size if 0) if successful or 0 on error. If FramepaC has been compiled without disk storage support (preprocessor symbol FrNO_DATABASE is defined), this function always indicates an error. If `force_create` is `True`, the backing store file will be created if it does not exist; if `False`, the call will fail. The flag `transactions` indicates whether or not to maintain a transaction log file which may be used to roll back incomplete operations in the event of a crash.

This function should be called after `initialize_FramepaC` but before any use of virtual frames functions.

*void *initialize_VFrames_server(char *server, int port, char *database, int symtabsize, FrBool transactions, FrBool force_create = True)*

Set up the VFrames system to use a database server for retrieving and storing frames. This function returns a newly-created symbol table of the specified size if successful or 0 on error. If FramepaC has been compiled without network server support (preprocessor symbol FrNO_SERVER is defined), this function always indicates an error. If `force_create` is `True`, the database will be created if it does not yet exist.

This function should be called after `initialize_FramepaC` but before any use of virtual frames functions.

A program which uses virtual frames must also call the `shutdown_VFrames` function once for each `initialize_VFrames_XX` call before it terminates. Failure to do so will likely result in lost updates, and can potentially corrupt the backing store.

int shutdown_VFrames()

For the frames associated with the current symbol table, commit any dirty frames to backing store, close access to the backing store, and destroy the symbol table, making the default symbol table the current symbol table. Returns 0 if successful or a nonzero error code.

7.2. Input/Output Functions

ostream << FrObject

Send a printed representation of the given object to the output stream in the same manner as normal C++ streams operation. If it is known that the output will always be to a string, it will probably be preferable to use the much faster function `FrObject::print(char*)` instead of a `strstream`.

*ostream << FrObject**

Send a printed representation of the object pointed at by the argument to the output stream in the same manner as normal C++ streams operation. If it is known that output will always be to a string, it will probably be preferable to use the much faster function `FrObject::print(char*)` instead of a `strstream`.

istream >> FrObject&*

Read the printed representation of a `FrObject` (FrString, FrList, FrSymbol, FrFrame, FrNumber, FrQueue, FrArray, FrStruct, etc.) from the input stream and store it in the specified pointer. For converting a string into an object, it will usually be preferable to use the function `string_to_FrObject` instead of creating a `strstream` consisting of the string (the latter takes four to five times as long to execute).

See Section 3.3 for details on the formats of items acceptable to the FramepaC reader.

istream >> FrSymbol&*

Read the printed representation of a FrSymbol from the input stream and store a pointer to the symbol in the specified pointer. For converting a string into a symbol, it will usually be preferable to use either `makeSymbol` or `string_to_Symbol` rather than creating a `strstream` consisting of the string (the latter takes much longer to execute than `string_to_Symbol`).

*int FrObject_string_length(const FrObject *obj)*

Return the length of the printed representation for the given object. `FrObject::print(char*)` requires a buffer one longer than this (for the terminating NUL). When `obj` is not 0, this function is identical to the `displayLength` method.

*char *FrObject::print(char *buffer) const*

Place the printed representation of the given object into the user-supplied buffer (which must be large enough for the entire representation). This function returns a pointer to the character immediately following the last character in the printed representation (which will be the terminating NUL). When `obj` is not 0, this method is identical to the `displayValue` method.

*char *FrObject::print() const*

Place the printed representation of the given object into a newly-allocated buffer. This function returns a pointer to the buffer containing the NUL-terminated printed representation; this buffer must be deallocated with `FrFree` when no longer required.

*FrObject *string_to_FrObject(char *&input)*

Read the printed representation of a `FrObject` (FrString, FrList, FrSymbol, FrFrame, FrNumber, FrArray, FrStruct, etc.) from the supplied input string and return a pointer to the object. The pointer to the input string is updated to point at the character immediately following the last one consumed in reading the printed representation of the object.

See Section 3.3 for details on the formats of items acceptable to the FramepaC reader.

*FrSymbol *string_to_Symbol(char *&input)*

Read the printed representation of a FrSymbol from the supplied input string and return a pointer to it. The pointer to the input string is updated to point at the character immediately following the last one consumed in reading the symbol.

FrBool import_FrameKit_frames(istream &in, ostream &echo)

Read a file containing textual representations of frames (in either FrameKit or FramepaC format) from the stream `in` into the current symbol table as virtual frames. If `echo` is nonzero, a one-line progress report is written to the stream for each frame read.

Returns `True` if the file was successfully read, or `False` if an error occurred.

*FrBool export_FrameKit_frames(ostream &output, FrList *frames)*

Writes each of the frames named in `frames` to the output stream in FrameKit `MAKE-FRAME` format.

Returns `True` if successful, or `False` if any of the elements of the list is not a symbol naming a frame.

7.3. Type Determination Functions

int ARRAYP(const FrObject)*

virtual FrBool FrObject::arrayp() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrArray` or some subclass thereof. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `ATOMP` does).

int ATOMP(const FrObject)*

virtual FrBool FrObject::atomp() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrAtom` or some subclass of it (`FrSymbol`, `FrString`, `FrNumber`, etc.). The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `ATOMP` does).

int CONSP(const FrObject)*

virtual FrBool FrObject::consp() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrCons` or some subclass of `FrCons` (only `FrList` at this time). The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `CONSP` does).

int FRAMEP(const FrObject)*

virtual FrBool FrObject::framep() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrFrame` or its subclass `VFrame`. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `FRAMEP` does). Use `is_frame` or `FrSymbol::isFrame` to determine whether a particular symbol is the name of a frame.

int HASHP(const FrObject)*

virtual FrBool FrObject::hashp() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrHashTable`. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `HASHP` does).

int NUMBERP(const FrObject)*

virtual FrBool FrObject::numberp() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrNumber` or one of its subclasses (`FrFloat` or `FrInteger`). The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `NUMBERP` does).

virtual ObjectType FrObject::objType() const

Return a value identifying the exact object type, such as `OT_Frame`, `OT_FrQueue`, etc.

int QUEUEP(const FrObject)*

virtual FrBool FrObject::queuep() const

Return `True` if the object pointed at by the given pointer is an instance of the class `FrQueue`. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `QUEUEP` does).

int STRINGP(const FrObject)*

virtual FrBool FrObject::stringp()

Return `True` if the object pointed at by the given pointer is an instance of the class `FrString`. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `STRINGP` does).

int STRUCTP(const FrObject)*

virtual FrBool FrObject::structp()

Return a true value if the object pointed at by the given pointer is an instance of the class `FrStruct`. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `STRUCTP` does).

int SYMBOLP(const FrObject)*

virtual FrBool FrObject::symbolp()

Return a true value if the object pointed at by the given pointer is an instance of the class `FrSymbol`. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `SYMBOLP` does).

int VECTORP(const FrObject)*

virtual FrBool FrObject::vectorp()

Return a true value if the object pointed at by the given pointer is an instance of the class `FrBitVector` or some subclass thereof. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `VECTORP` does).

int WIDGETP(const FrObject)*

virtual FrBool FrObject::widgetp()

Return a true value if the object pointed at by the given pointer is an instance of the class `FrWidget` or some subclass thereof. The second form is the object-oriented function underlying the first form, and should only be called if the pointer is already known to be non-NULL (which is what `WIDGETP` does).

*FrBool is_frame(const FrSymbol *name)*

FrBool FrSymbol::isFrame()

Return `True` if the specified symbol is the name of a frame, `False` if not. If the specified frame is a virtual frame which is not currently in memory, it will not be fetched from the backing store.

7.4. Common Object Functions

Rather than `delete`, you should always use `free_object(FrObject *)` or `object->freeObject()`, since some object types (i.e. `FrSymbol`) may never be freed, and some (i.e. `FrFrame`) should not normally be freed.

7.4.1. Procedural Interface

*void free_object(FrObject *) ;*

Deallocate the item pointed at by its argument, if appropriate for the type of object actually being referenced.

FrBool eql(const FrObject, const FrObject*)*

Determine whether the two objects are identical or are both numbers and have the same value.

FrBool equal(const FrObject, const FrObject*)*

Determine whether the two objects are identical or are both of the same type and have the same value. This is the most general of the object comparison functions, and thus the slowest; if the types of the two numbers are known, it is usually preferable to use either `eql()` or `==` instead. The `==` operator is valid between two `FrSymbol`s, two `FrNumber`s, and two `FrString`s.

7.4.2. Object-Oriented Interface

The functions described in this section are all virtual functions, which allows you to use them on a `FrObject*` without the need to worry about what type of object is actually present. However, you do need to ensure that the pointer you are using is not NULL before calling the function.

virtual ObjectType objType() const

Determine what type of object is actually present.

*virtual const char *objTypeName() const*

Return a printed string containing the name of the type of object, for example, "FrObject", "FrFrame", or "FrSymbol".

virtual ObjectType objSuperclass() const

Determine the type of object from which the object instance is derived, if any.

virtual void freeObject()

Erase or otherwise deallocate the object, as appropriate for its type.

*virtual FrObject *copy() const*

Make a shallow copy of the object. Only the top-level structure of the object will be copied, if appropriate. For example, a shallow copy of a list will copy only the cons cells forming the list, not any of the items in the list. For most objects, `copy` and `deepcopy` are the same.

*virtual FrObject *deepcopy() const*

Make a copy of the complete structure of an object, if appropriate. For example, a deep copy of a list will copy the items in the list as well as the cons cells forming the list. For most objects, `copy` and `deepcopy` are the same.

*virtual FrObject *subseq(size_t start, size_t stop)*

Make a copy of the indicated portion of the object, if appropriate. Both *start* and *stop* are zero-based indices into the object, indicating which elements (inclusively) are to be placed in the result. For example, **subseq(1,2)** of the list **(A B C D)** will return the new list **(B C)**.

*virtual FrObject *reverse()*

For compound objects whose component objects are ordered, destructively reverse the components. Returns an object pointer which should be used on subsequent references instead of the original pointer (since e.g. the original pointer to a reversed list points at the last item in the reversed list).

virtual ostream &printValue(ostream &output) const

Output a printed representation of the object's value to the supplied stream. This is the function underlying the overloading of the `<<` operator for streams to FramepaC objects.

virtual size_t displayLength() const

Determine how many bytes the printed representation of the object will require. This function allows you to allocate the correct amount of buffer space when creating a string containing the object's value with `displayValue`.

*virtual char *displayValue(char *buffer) const*

Store a printed representation of the object's value in the supplied buffer. The required size of the buffer may be determined with `displayLength`.

virtual int length() const

If appropriate for the object, return a measure of its size. Returns the length of a list, number of characters in a string, number of items in a queue or hash table, etc.

FrBool iterate(FrIteratorFunc func, ...) const

If appropriate for the object, call **func** once for each subobject of the given object; i.e. once for each item stored in a hashtable, array, queue, etc. This function returns *True* if the provided function returns *True* for each and every call, *False* otherwise. If `func` ever returns *False*, the iteration is terminated immediately.

For objects for which the operation is not meaningful, this function returns *True* (the operation was trivially successful).

virtual FrBool iterateVA(FrIteratorFunc func, va_list args) const

This function is the same as **iterate**, but accepts a variable-argument list rather than a list of arguments.

virtual unsigned long hashValue() const

Return a hashing key value which is used by `FrHashTable` to locate the object.

*virtual FrObject *car() const*

If appropriate for the object, return the first subobject it contains. Returns the first item in a list, or the item at the head of a queue.

For objects for which the operation is not meaningful, this function returns 0. If the object is known to be a `FrCons` or `FrList`, the method `first` may be used instead of `car` to avoid the overhead of a virtual function call.

*virtual FrObject *cdr() const*

If appropriate for the object, return the remainder after removing the first subobject it contains. Returns the remainder of a list.

For objects for which the operation is not meaningful, this function returns 0. If the object is known to be a `FrList`, the method `rest` may be used instead of `cdr` to avoid the overhead of a virtual function call.

virtual FrBool equal(const FrObject) const*

Determine whether the indicated object is equivalent to the object to which the method is applied.

*virtual int compare(const FrObject *) const*

Determine the sort-order relationship between the indicated object and the object to which the method is applied. Returns -1 if the applied object should sort before the indicated object, +1 if the reverse, and 0 if either order is acceptable (generally because the two items are equivalent).

*virtual FrObject *getNth(size_t N) const*

If appropriate for the object, return a copy of the `N`th component (which must be released with `freeObject` or `freeObject` when no longer needed). `N` may range from 0 to one less than the logical size of the object.

This method returns 0 if the specified index is out of range or the operation is not meaningful for the object. It is currently useful for `FrCons` (for N=0,1), `FrList`, `FrQueue`, and `FrString` (where it returns a one-character FrString).

*virtual FrBool setNth(size_t N, const FrObject *newelt) const*

If appropriate for the object, destructively modify the `N`th component, setting it to a copy of `newelt`. Returns `True` if the object was modified, `False` otherwise. This method returns `False` if the operation is not meaningful for the object, the specified index is out of range, or the given `newelt` is of the wrong type.

`setNth` is currently useful for `FrArray`, `FrCons` (for N=0,1), `FrList`, `FrQueue`, and `FrString` (which requires a `FrString` for `newelt`, and uses its first character).

*virtual size_t locate(const FrObject *item, size_t start = (size_t)-1) const*

*virtual size_t locate(const FrObject *item, FrCompareFunc func, size_t start = (size_t)-1) const*

Locate a component or subsequence of a sequence object (array, list, queue, or string). This method is currently effective on `FrList`, `FrQueue`, and `FrString`. The valid types for `item` vary slightly between classes, so see the description of `locate` under the appropriate class for more details. `start` indicates where in the sequence to start searching: if `-1`, search the entire object, else search the portion of the sequence *after* index `start`. This permits all matches in an object to be iterated over by repeatedly calling this method with the return value of the previous call as the value of `start`.

This method returns `(size_t)-1` if no match was found, or the object is not of an appropriate type. If a comparison function is provided, it is called repeatedly to determine matching elements of the sequence object; if no function is provided, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*virtual FrObject *insert(const FrObject *item, size_t pos)*

Destructively insert a new item or list of items into a sequence object (array, list, queue, or string) at the indicated position, which may range from 0 (insert prior to first element) to `length()` (insert after last element). A pointer to the modified object is returned; it will normally be identical to the pointer to the object on which the method was called, but will differ for `FrList`s if `pos` is 0.

The object will not be modified if `pos` is greater than `length()` of the object, or if the object is not of an appropriate type.

*virtual FrObject *elide(size_t start, size_t end)*

Destructively remove the indicated subsequence of the object, if appropriate, from the `start`h component to the `end`th component, inclusive. A pointer to the modified object is returned; it will normally be identical to the pointer to the object on which the method was called, but will differ for `FrList`s if `start` is 0.

If `start` is greater than or equal to `length()` of the object, the object will not be modified. If `end` is greater than the size of the object, all elements after `start` are removed.

virtual FrBool expand(size_t increment)

If appropriate for the object, expand its size by `increment` elements. Returns `True` if the object was expanded, `False` if not.

virtual FrBool expandTo(size_t newsize)

If appropriate for the object, and the object's current size is less than `newsize` elements, expand its size to `newsize` elements. Returns `True` if the object was expanded, `False` if not.

*istream &operator >> (istream &input, FrObject *&obj)*

Read an object from the given `istream`, and set the supplied `FrObject*` to point at the object read.

*ostream &operator << (ostream &output, const FrObject *object)*

Print the value of an object to the given output stream. This is equivalent to `printValue` when `object` is not 0, and outputs "()" when `object` is 0.

ostream &operator << (ostream &output, const FrObject &object)

Print the value of the object to the given output stream.

7.5. FrArray Functions

`FrArray` is an extensible array class which stores `FrObject`s as array elements.

*FrArray::FrArray(size_t size, const FrList *init = 0)*

*FrArray::FrArray(size_t size, const FrObject **init, FrBool copyitems = True)*

Create a new array capable of storing `size` items. For the first variant of the constructor, the elements of `init` are copied into the array; if `init` contains fewer than `size` elements, the final element of the list is copied into all remaining locations of the array. If `init` is 0, all elements of the array are initialized to 0.

For the second variant of the constructor, `init` is an array of pointers to `FrObject`s which must contain at least `size` elements. If `copyitems` is `False`, only the pointers in `init` are copied into the array; otherwise, the objects pointed at by `init` are copied prior to being placed into the new `FrArray`.

FrArray::~FrArray()

Deallocate the array.

virtual FrBool FrArray::arrayp() const

Returns `True` to indicate that the object is an array.

virtual size_t FrArray::length() const

Return the number of elements that the array currently contains, including any entries set to the NULL pointer (0).

*virtual FrObject *FrArray::car() const*

Returns `array[0]`.

*virtual FrObject *FrArray::reverse()*

Reverse the order of the elements in the array. Returns a pointer to the reversed `FrArray`.

*FrObject *&FrArray::operator[] (size_t N)*

Create a reference to the `N`th element of the array, which may then be retrieved or modified as with `getNth` or `setNth`.

virtual FrBool FrArray:expand(size_t increment)

Expand the `FrArray`'s size by `increment` elements, setting the new elements to 0. Returns `True` if the array was expanded, `False` if not.

The array may be shrunk using `elide`. To shrink an array to size `N`, use

```
array->elide(N, array->length()) ;
```

virtual FrBool FrArray::expandTo(size_t newsize)

If the `FrArray`'s current size is less than `newsize` elements, expand its size to `newsize` elements and set the new elements to 0. Returns `True` if the array was expanded, `False` if not.

*virtual FrObject *FrArray::subseq(size_t start, size_t end)*

Return a new `FrArray` containing only elements `start` through `end` (inclusive) of the original array. A complete deep copy of the array elements is made.

*virtual FrObject *FrArray::insert(const FrObject *newelt, size_t location, FrBool copyitem = True)*

Insert the object or objects specified by `newelt` into the array prior to element `location`, expanding the array as needed and shifting all elements from `location` to the end of the array to accomodate the new element(s). `location` may range from 0 to `length()`; location 0 prepends the new element(s) to be beginning of the array, while location `length()` appends the new element(s) to the end of the array.

If `newelt` is `consp()`, then it is assumed to be a list of elements to be inserted; otherwise, it is assumed to be a single new element to be inserted. If you wish to insert a single element which is itself a list, you must wrap it inside a single-element list. If `copyitem` is `True`, a copy of the item(s) is made prior to insertion into the array; otherwise, the object itself is used and may not be deleted as long as it remains in the array (an exception is that the cons cells making up the top-level list for `newelt` may be freed), nor may it be referenced after the array is deleted.

*virtual FrObject *FrArray::elide(size_t start, size_t end)*

Remove elements `start` through `end` (inclusive) from the array, and shift all remaining elements down to close the gap created by the removal. This method reduces the size of the array by (`end-start`+1) elements. A pointer to the modified `FrArray` is returned.

*virtual size_t FrArray::locate(const FrObject *item, size_t start = (size_t)-1) const*

*virtual size_t FrArray::locate(const FrObject *item, FrCompareFunc func, size_t start = (size_t)-1) const*

Locate an element or sequence of elements in the array. If `item` is `consp()`, this method locates the first occurrence of sequential elements in the array after `start` matching the list pointed at by `item`; otherwise, it locates the single element matching `item`. Note that in order to locate a single element which happens to be a list, you must wrap it in another, one-element list. The search begins following the `start`h element of the array, which permits the result of a `locate` to be passed to a subsequent call in order to search for additional matches. The return value is the index into the array of the first match after `start`, or `(size_t)-1` if no match could be found.

If provided, the comparison function `func` is called repeately to determine which element(s) of the array match `item`; otherwise, a simple pointer comparison (equivalent to Lisp's `eq`) is used.

7.6. FrBitVector Functions

`FrBitVector` is an extensible array class which stores a string of bits as array elements.

Whenever a boolean value is required as input to be stored in the `FrBitVector`, a variety of different `FrObject`s may be supplied. The input value is converted to a `FrBool` by the function `FrBooleanValue`, which accepts as `True` the following:

1. the symbol `T`
2. any symbol whose printed name begins with either 'T' or 'Y' (upper- or lower-case)
3. any `FrString` whose first character is 'T' or 'Y' (either case)
4. any `FrNumber` whose integer portion is non-zero

and as `False`

the NULL pointer

the symbol `NIL`

any other `FrSymbol` not mentioned already

any `FrString` whose first character is neither 'T' nor 'Y'

any `FrNumber` whose integer portion is zero

any other type of object

FrBitVector::FrBitVector(size_t size = 0)

*FrBitVector::FrBitVector(size_t size, const FrList *init)*

Create a new bit vector capable of storing `size` bits. For the first variant of the constructor, the array is cleared to all zero bits. For the second variant, the elements of `init` are passed through `FrBooleanValue` in turn, and placed in successive locations of the bit vector; if `size` is greater than the length of `init`, any values between the end of `init` and the specified size of the array remain cleared to zero.

FrBitVector::FrBitVector(const FrBitVector &vect)

Create a new copy of the bit vector `vect`.

FrVector::~FrBitVector()

Deallocate the bit vector.

virtual FrBool FrBitVector::vectorp() const

Returns `True` to indicate that the object is a bit vector.

virtual size_t FrBitVector::length() const

Return the number of elements that the bit vector currently contains.

*virtual FrObject *FrBitVector::reverse()*

Reverse the order of the elements in the vector. Returns a pointer to the reversed `FrBitVector`.

virtual FrBool FrBitVector:expand(size_t increment)

Expand the `FrBitVector`'s size by `increment` bits, setting the new elements to 0. Returns `True` if the vector was expanded, `False` if not.

The vector may be shrunk using `elide`. To shrink a vector to size `N`, use

```
vector->elide(N, vector->length()) ;
```

virtual FrBool FrBitVector::expandTo(size_t newsize)

If the `FrVector`'s current size is less than `newsize` bits, expand its size to `newsize` bits and set the new elements to 0. Returns `True` if the vector was expanded, `False` if not.

*virtual FrObject *FrBitVector::subseq(size_t start, size_t end)*

Return a new `FrBitVector` containing only bits `start` through `end` (inclusive) of the original vector.

*virtual FrObject *FrBitVector::insert(const FrObject *newelt, size_t location, FrBool copyitem = True)*

Insert the bit or bits specified by `newelt` into the bit vector prior to bit number `location`, expanding the vector as needed and shifting all bits from `location` to the end of the vector to accomodate the new bit(s). `location` may range from 0 to `length()`; location 0 prepends the new bit(s) to be beginning of the vector, while location `length()` appends the new bit(s) to the end of the vector.

If `newelt` is `consp()`, then it is assumed to be a list of bits to be inserted; otherwise, it is assumed to be a single new bit to be inserted.

*virtual FrObject *FrBitVector::elide(size_t start, size_t end)*

Remove bits `start` through `end` (inclusive) from the vector, and shift all remaining bits down to close the gap created by the removal. This method reduces the size of the vector by (`end-start`+1) bits. A pointer to the modified `FrBitVector` is returned.

*virtual size_t FrArray::locate(const FrObject *item, size_t start = (size_t)-1) const*

*virtual size_t FrArray::locate(const FrObject *item, FrCompareFunc func, size_t start = (size_t)-1) const*

Locate a bit or sequence of bits in the array. If `item` is `consp()`, this method locates the first occurrence of sequential bits in the array after `start` matching the list pointed at by `item`; otherwise, it locates the next occurrence of the single bit specified by `item`. The search begins following the `start`h bit of the array, which permits the result of a `locate` to be passed to a subsequent call in order to search for additional matches. The return value is the index into the vector of the first match after `start`, or `(size_t)-1` if no match could be found.

For bit vectors, the comparison function is ignored, making the second form equivalent to the first.

virtual FrBool FrBitVector::iterateVA(FrIteratorFunc func, va_list args) const

This method invokes `func` once for each bit in the vector, passing as the function's first argument an `FrInteger` with value 0 or 1, depending on the value of the bit. This method may also be invoked indirectly by calling `FrObject::iterate` when the object is a bit vector.

FrBool FrBitVector::getBit(size_t N) const

Return the value of the `N`th bit in the vector. If `N` is greater than or equal to the length of the vector, this method return `False`.

FrBool FrBitVector::setBit(size_t N, const FrBool value) const

Set the value of the `N`th bit to `value`. This method returns `True` if the bit was successfully updated, `False` if unable to make the change (i.e. `N` is out of range).

size_t FrBitVector::vectorlength() const

Return the number of bits currently stored in the vector. This is a non-virtual version of `length()` for greater efficiency when the object is known to be a `FrBitVector`.

*FrBitVector *FrBitVector::intersection(const FrBitVector *vect) const*

*FrBitVector *FrBitVector::operator * (const FrBitVector &vect) const*

Return a newly-constructed bit vector which contain the logical AND of corresponding bits in the two given vectors. The length of the returned vector is the lesser of the lengths of the two input vectors, since all bits which are present in only one vector are by definition 0 after ANDing.

*FrBitVector *FrBitVector::vectorunion(const FrBitVector *vect) const*

*FrBitVector *FrBitVector::operator + (const FrBitVector &vect) const*

Return a newly-constructed bit vector which contain the logical OR of corresponding bits in the two given vectors. The length of the returned vector is the larger of the lengths of the two given vectors.

*FrBitVector *FrBitVector::difference(const FrBitVector *vect) const*

*FrBitVector *FrBitVector::operator - (const FrBitVector &vect) const*

Return a newly-constructed bit vector which contain the 1 bits in each location containing a 1 bit in this vector and a 0 bit in `vect`, and a 0 bit in every other location. This method is equivalent to a set difference operation.

void FrBitVector::negate()

Invert all of the bits in the vector.

7.7. FrAtom Functions

`FrAtom` is an abstract class, which should never be instantiated directly. Use an object of the appropriate subclass (`FrSymbol`, `FrString`, or one of `FrNumber`'s subclasses) instead.

7.8. FrSymbol Functions

Most slot fillers, as well as frame, slot, and facet names, will be symbols rather than strings or numbers. The following functions are provided for manipulating symbols.

FramepaC internally maintains a collection of all symbols ever created during the lifetime of a symbol table, and re-uses symbols whenever possible. Thus there is always a unique symbol for any given string of characters unless you switch symbol tables (a capability intended for use primarily by database servers, which may need to keep multiple conflicting frames under a single name).

Because symbols are internally managed and only destroyed when the symbol table containing them is destroyed, it is an error to call `new` or `delete` for a `FrSymbol`, and the program will be aborted if you attempt to do so.

7.8.1. Construction Functions

*FrSymbol *makeSymbol(const char *name)*

*FrSymbol *FrSymbol::makeSymbol(const char *name) const*

Return a pointer to the symbol with the specified name in the currently active symbol table, creating it if it is not already present in the system. `makeSymbol` does not modify the name in any way; in particular, it will not convert it to uppercase as the FramepaC reader (see Section 3.3) does by default.

*FrSymbol *gensym(char *basename) ;*

Return a newly-created symbol which is guaranteed to be unique. The symbol's name will be the given string followed by one or more digits. If the argument is 0, the default base name of "GENSYM" will be used.

Unlike most Lisp implementations, FramepaC actually ensures that the created symbol does not yet exist.

*FrSymbol *gensym(FrSymbol *basename) ;*

Return a newly-created symbol which is guaranteed to be unique. The new symbol's name will be the given symbol's name followed by one or more digits.

7.8.2. Basic Functions

*FrSymbol *findSymbol(char *name)*

*FrSymbol *FrSymbol::findSymbol(char *name) const*

Return a pointer to the symbol with the specified name, or 0 if the symbol has never been created, in the currently active symbol table. `findSymbol` does not modify the name in any way; in particular, it will not convert it to uppercase as the FramepaC reader (see Section 3.3) does by default.

*const char *symbol_name(FrSymbol *symbol)*

*const char *FrSymbol::symbolName() const*

Return a pointer to the character string representing the symbol's name. Because only a single symbol is ever created for a given name, `makeSymbol(symbol_name(S)) == S` for all strings S which do not exceed the maximum length of a symbol's name.

*FrSymbol == char**

For ease of writing, the above expression is equivalent to `FrSymbol* == findSymbol(char*)`. Note that due to a limitation imposed by C++, you will need to dereference the FrSymbol* you actually have in order to use this feature, i.e.

```
*sym == "IS-A"
```

For maximum execution speed, it is usually preferable to use either `findSymbol` or `makeSymbol` once early in the program and compare against a variable containing the result rather than using the implicit `findSymbol` of this overloaded operator.

*FrSymbol != char**

As above, but testing for inequality. For maximum execution speed, it is usually preferable to use either `findSymbol` or `makeSymbol` once early in the program and compare against a variable containing the result rather than using the implicit `findSymbol` of this overloaded operator.

*static FrBool FrSymbol::nameNeedsQuoting(const char *name)*

Determine whether the printed representation of the given symbol name requires quoting with vertical bars when printed.

7.9. FrSymbolTable Functions

The following functions are primarily intended for use by database servers, which may need to store multiple frames on the same symbol name. By allowing multiple symbol tables, each database can thus have its own name space. However, symbols will no longer be unique program-wide, but only within a particular symbol table.

Note that for purposes of reading in symbols, certain symbols (mainly those with standard uses in frames) are preloaded into each symbol table and will thus appear to be the same from symbol table to symbol table, though they in fact are not and will therefore not compare as equal.

*FrSymbolTable *create_symbol_table(int size)*

FrSymbolTable::FrSymbolTable(int size = 0)

Create a new symbol table of the specified initial size (or the default size, if 0), and return a pointer to a record describing the symbol table. The caller is responsible for remembering any and all additional symbol tables created in this manner.

*FrSymbolTable *default_symbol_table() ;*

static FrSymbolTable FrSymbolTable::selectDefault()

Select the default symbol table created at FramepaC initialization as the one to use for subsequent FramepaC calls for the purposes of symbol manipulation, and return the symbol table which was active before this call.

*void destroy_symbol_table(FrSymbolTable *symboltable)*

FrSymbolTable::~FrSymbolTable()

Delete the specified symbol table and all frames associated with it. Before using this function, you must be absolutely certain that you have freed any objects other than frames which reference symbols from the named table; failure to do so will leave FramepaC very confused and could even crash your program.

*FrSymbolTable *select_symbol_table(FrSymbolTable *newtable)*

*FrSymbolTable *FrSymbolTable::select()*

Make the specified symbol table the one which will be used by subsequent FramepaC calls for the purposes of symbol manipulation, and return the current symbol table.

*FrSymbol *makeSymbol(const FrSymbol *sym)*

Make a new symbol with the same name as the given symbol in the current symbol table. This function allows you to copy symbols from one symbol table to another in order to correctly make comparisons between symbols in different symbol tables.

FrBool FrSymbolTable::expand(int increment)

Expand the symbol table's capacity by the indicated number of symbols. The return value indicates whether the expansion was successful.

FrBool FrSymbolTable::expandTo(int newsize)

Expand the symbol table's capacity to the indicated number of symbols. The return value indicates whether the expansion was successful. If the symbol table is already at least the indicated size, this function does nothing and returns successfully.

*FrSymbolTable *current_symbol_table()*

*static FrSymbolTable *FrSymbolTable::current()*

Return the currently active symbol table.

*static FrSymbol *FrSymbolTable::add(const char *name)*

Insert a new symbol with the indicated name into the symbol table if it does not already exist. Return a pointer to the symbol in the symbol table, whether newly created or pre-existing. Unlike the FramepaC reader, no conversions are performed on the name, so the symbol's name will be the exact contents of the supplied string, even if the string contains lowercase letters or special symbols.

*static FrSymbol *FrSymbolTable::add(const char *name,*
*const FrObject *value)*

Insert a new symbol with the indicated name into the symbol table if it does not already exist, or find the symbol if already present. Set the symbol's associated value to `value`, and return a pointer to the symbol.

This function is only available if `FrSYMBOL_VALUE` is #define'd.

*FrSymbol *FrSymbolTable::gensym(const char *name)*

Generate a unique symbol, whose name begins with the indicated characters. This function generates a new symbol by appending an incrementing number to the end of the supplied name. Unlike most Lisp implementations, FramepaC's `gensym` guarantees that the symbol did not already exist in the symbol table (by repeated attempts with differing distinguishing numbers).

*FrSymbol *FrSymbolTable::lookup(const char *name)*

Find the symbol with the specified name in the symbol table, and return a pointer to it, or 0 if the symbol is not present in the symbol table.

void FrSymbolTable::setNotify(VFrameNotifyType type,
*VFrameNotifyFunc *func)*

Set the function which should be called when a database server notification of the indicated type of event (e.g. frame creation/deletion/update) is received for the symbol table.

*VFrameNotifyFunc *FrSymbolTable::getNotify(VFrameNotifyType type)*

Retrieve the function which is currently being called when a database server notification of the indicated type is received for the symbol table.

void FrSymbolTable::setProxy(VFrameNotifyType type,
*VFrameProxyFunc *func)*

(network backing-store support function)
This function is used to specify the function which will be called when the database server requires that the calling client perform some frame-update action on behalf of another client.

!!!

*VFrameProxyFunc *FrSymbolTable::getProxy(VFrameNotifyType type)*

(network backing-store support function)
Determine which function will be called when the database server requires this client to perform a frame update on behalf of another client.

!!!

*void FrSymbolTable::setDeleteHook(FrSymTabDeleteFunc *delhook)*

Specify the function (if any) to be called when the symbol table is deleted. The `delhook` function, if not 0, will be called with the symbol table as its sole argument when the symbol table is deleted, before the frames and symbols associated with the symbol table are deleted.

*FrSymTabDeleteFunc *FrSymbolTable::getDeleteHook() const*

Determine which function (if any) will be called when the symbol table is deleted.

*void FrSymbolTable::setShutdown(VFrameShutdownFunc *func)*

Specify which function should be called when the database server providing the backing store for the symbol table indicates that it is about to shut down.

*VFrameShutdownFunc *FrSymbolTable::getShutdown()*

Retrieve the function which will be called when the database server for the symbol table indicates that it is shutting down.

FrBool FrSymbolTable::isReadOnly() const

Determine whether the symbol table is associated with a read-only backing store. If this function returns `True`, then any changes made to frames will be lost when the symbol table is closed with `shutdown_VFrames` or a modified frame is discarded from memory. Frames are still retrieved from the backing store on demand, but modifications will be as fleeting as they are for a symbol table without backing store.

FrBool FrSymbolTable::iterateFrame(FrIteratorFunc func, ...) const

Iterate through the symbol table in the same manner as `iterate`, but call the indicated function once for each frame currently loaded in memory which is named by a symbol in the symbol table. Returns `True` if every call to the specified function returned `True`, and `False` otherwise. The method `iterate` calls the specified function once for each symbol in the symbol table, regardless of whether a frame is associated with the symbol.

FrBool FrSymbolTable::iterateFrameVA(FrIteratorFunc func, va_list args) const

Same as `iterateFrame`, but this function accepts a variable argument list rather than a list of arguments, which may be useful when called from a function which itself takes a variable number of arguments.

*FrList *FrSymbolTable::listRelations() const*

Return a list of all of the bidirectional relations which have been defined for the current symbol table; FramepaC automatically maintains inverse links for the `VALUE` facets of any slots whose name is contained in the list returned by this function. The returned list consists of a list of two-element lists, each of which contains the two symbols naming the inverse links related to each other (note that a link can be its own inverse, in which case the two elements of the corresponding sublist are the same). The returned list has been newly created and must be explicitly freed (e.g. with `freeObject` once no longer required. See also `FrSymbol::defineRelation` and `FrSymbol::undefineRelation`.

*void do_all_symtabs(DoAllSymtabsFunc *func, ...)*

Execute the indicated function once for every symbol table currently in existence. Any additional arguments given to `do_all_symtabs` are passed directly to `func` as a variable-argument list.

7.10. FrString Functions

The `FrString` class provides storage for character strings, where the character size may be one, two, or four bytes (8-bit, 16-bit, or 32-bit character sets, such as ASCII/ANSI, Unicode, and ISO 10646).

7.10.1. Constructors

FrString::FrString()

Create an empty string.

*FrString::FrString(const char *value)*

Create a new string consisting of the characters pointed at by `value`, up to but not including the NUL character terminating the C-style string. The new string is assigned the default character size of one byte.

FrString::FrString(FrChar_t value)

Create a new string consisting of the single character specified by `value`. The new string's character width is set to the smallest size which can contain all of the non-zero bits of `value`.

*FrString::FrString(const char *value, int length, int width = 1)*

Create a new string consisting of the `length` characters pointed at by `value`, each character consisting of `width` bytes (which may be either 1, 2, or 4). This string may contain NUL characters. If the specified width is other than one byte per character, the supplied string must be a byte-wise representation of the characters in the string, with big-endian (most-significant byte first) byte ordering.

FrString + FrString

*FrString + FrString**

FrString FrString::concatenate(FrString*)*

FrString FrString::concatenate(FrString&)*

Create a new string consisting of the contents of the two supplied strings. The characters in the resulting string will be of the same width as the wider character width in the two original strings.

7.10.2. String Manipulation

FramepaC strings may be manipulated without being concerned with the character width used by the strings, be it 8-bit, 16-bit, or 32-bit characters. Character widths are converted as necessary.

*FrString += char**

FrString += FrChar_t

FrString FrString::append(char*)*

FrString FrString::append(FrChar_t)*

Destructively append the C-style character string or the `FrChar_t` to the end of the supplied FramepaC string (the C-style character string remains unchanged). The character string is assumed to consist of eight-bit characters, which will be widened as necessary to match the character size already present in the `FrString`. The `FrChar_t` is assumed to be of the smallest width necessary to contain all of its non-zero bits, and the string's character width is increased if necessary.

FrString += FrString

*FrString += FrString**

FrString FrString::append(FrString*)*

FrString FrString::append(FrString&)*

Destructively append the second string to the end of the first (the second string remains unchanged). The characters in the resulting string will be of the same width as the wider character width in the two original strings.

FrString::charWidth()

Determine how many bytes are used to store each character in the specified string.

size_t FrString::stringLength()

Determine the number of characters stored in the specified string. This is a non-virtual function, and is thus faster than the equivalent virtual function `length()` if the object is known to be a `FrString`.

size_t FrString::stringSize() const

Determine the number of bytes required to store the string's value, excluding any terminating NUL character.

*const char *FrString::stringValue()*

Returns a pointer to the buffer used to store the string's value. This buffer is `charWidth()*stringLength()` or `stringSize()` bytes in length.

(char)FrString*

Like `stringValue`, this operator returns a pointer to the buffer used to store the string's value.

FrChar_t FrString::operator[size_t index]

Returns the character (a value of type `FrChar_t`) at the indicated position (counting from 0) in the string. This operator may only be used as an rvalue; to change the character at a particular position, use `setChar` or `setNth`.

FrString::setChar(size_t index, FrChar_t newchar)

Destructively modify the string by replacing the indicated character in the string by the new value. If a `FrString` is to be used as the new character, call `setNth` instead of this method.

7.10.3. Comparison

FrString == FrString

FrString != FrString

FrString < FrString

FrString <= FrString

FrString > FrString

FrString >= FrString

Perform the appropriate comparison between the values of the two strings. Because of a C++ limitation, you will need to dereference the `FrString*` which is what is normally available, i.e. use

```
*str1 == *str2
```

*FrString == char**

*FrString != char**

*FrString < char**

*FrString <= char**

*FrString > char**

*FrString >= char**

Perform the appropriate comparison between the value of the `FrString` and the character string. As before, you will need to dereference `FrString*` variables.

*virtual size_t FrString::locate(const FrObject *item, size_t start = (size_t)-1) const*

*virtual size_t FrString::locate(const FrObject *item, FrCompareFunc func, size_t start = (size_t)-1) const*

Locate a character or substring of the given string. The `item` may be either a `FrString` or a `FrList` of `FrString`s and/or `FrSymbol`s (which is treated as a single string for the purposes of searching the given string). The search begins following the `start`h element of the list, which permits the result of a `locate` to be passed to a subsequent call in order to search for additional matches. The return value is the index into the list of the first match after `start`, or `(size_t)-1` if no match could be found.

If provided, the comparison function `func` is called repeately to determine which characters or substrings of the string match; otherwise, a simple character-wise comparison is used.

7.10.4. Substrings

```
 !!!
FrString *FrFirstWord(const FrString *words) ;
FrString *FrLastWord(const FrString *words) ;
FrString *FrButLastWord(const FrString *words) ;
```

7.11. FrNumber Functions

The class `FrNumber` is the base class for the two classes which actually store numbers, `FrInteger` and `FrFloat`. An `FrInteger` stores a long integer, while `FrFloat` stores a double-precision floating-point number. You should never instantiate `FrNumber`, only its subclasses.

7.11.1. Constructors

FrInteger(long value)

FrFloat(double value)

Create a new `FrNumber` object with the specified value.

7.11.2. Manipulation

double floatValue() const

Determine the number's value as a floating-point number.

double fraction() const

Determine the fractional portion of the number's value.

long intValue() const

Determine the whole-number portion of the number's value.

double imag() const

Return the imaginary portion of a number. *This method is intended to support a future complex-number class.*

double real() const

Return the real portion of a number. *This method is intended to support a future complex-number class; it is currently identical to* `floatValue()`.

7.11.3. Comparison

FrNumber == FrNumber

FrNumber != FrNumber

FrNumber < FrNumber

FrNumber <= FrNumber

FrNumber > FrNumber

FrNumber >= FrNumber

Perform the appropriate comparison, after coercing both `FrNumber`s to be `FrFloat`s if either one is a `FrFloat`. As before, you'll need to dereference the `FrNumber*` which is normally what you'll have to get C++ to properly recognize the overloaded operator; for consistency, the right side is thus also a `FrNumber` rather than a `FrNumber*`.

FrNumber == long

FrNumber != long

FrNumber < long

FrNumber <= long

FrNumber > long

FrNumber >= long

Perform the appropriate comparison, coercing the `FrNumber` to be an `FrInteger` to match the right-hand side. As before, you'll need to dereference the `FrNumber*` which is normally what you'll have to get C++ to properly recognize the overloaded operator.

FrNumber == double

FrNumber != double

FrNumber < double

FrNumber <= double

FrNumber > double

FrNumber >= double

Perform the appropriate comparison, coercing the `FrNumber` to be `FrFloat` to match the right-hand side. As before, you'll need to dereference the `FrNumber*` which is normally what you'll have to get C++ to properly recognize the overloaded operator.

7.11.4. Other Operators

(double)FrNumber

Convert the `FrNumber`'s value into a floating-point number.

(long)FrNumber

Convert the `FrNumber`'s value into an integer, truncating it if the value is a floating-point number.

7.12. FrList Functions

Lists are *the* major data structure following frames from the point of view of FramepaC. Lists are used to store the fillers in a frame, and are thus used to specify arguments and return values for many frame manipulation functions. To assist in their use, FramepaC provides a set of Lisp-like list-manipulation functions.

7.12.1. Constructing Lists

*FrList(const FrObject *elt1)*

*FrList(const FrObject *elt1, const FrObject *elt2)*

*FrList(const FrObject *elt1, const FrObject *elt2, const FrObject *elt3)*

*FrList(const FrObject *elt1, const FrObject *elt2, const FrObject *elt3, const FrObject *elt4)*

Create a new list of one to four elements. These constructors may be used as

```
FrList *l = new FrList(new FrInteger(1),makeSymbol("FOO")) ;
```

Since a program would not use a `FrList` object as such (only pointers to them), use of the constructors as

```
FrList l(new FrInteger(1),makeSymbol("FOO")) ;
```

is inappropriate.

*FrList *copylist(FrList *source) ;*

Create a new list whose elements are identical to the elements of the specified list. This function is equivalent to the virtual method `copy()`.

*FrList *copytree(FrList *source) ;*

Create a new list whose elements are themselves copies of the elements of the specified list. This function is equivalent to the virtual method `deepcopy()`.

*FrList *makelist(FrObject *obj1, ...) ;*

Create a list consisting of the specified objects. The list of objects to be placed in the list is terminated with a (FrObject*)0.

*FrList *nconc(FrList *l1, FrList *l2)*

Destructively add the list `l2` to the end of list `l1`, returning a pointer to the combined list (in case `l1` is 0).

*FrList *pushlist(FrObject *newval, FrList *&list) ;*

Add a new element to the beginning of the list pointed at by the given pointer 'list' and update the pointer.

7.12.2. FrList Manipulation

*FrCons *listassoc(const FrList *l, const FrObject *item)*

*FrCons *listassoc(const FrList *l, const FrObject *item, FrCompareFunc cmp)*

Find the element of the given association list whose head matches the specified item. If given, the function `cmp` is called to determine when a matching item is found; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrList *listdifference(const FrList *l1, const FrList *l2)*

*FrList *listdifference(const FrList *l1, const FrList *l2, FrCompareFunc cmp)*

Return a list containing those elements of `l1` not contained in `l2`. The returned list is always a copy of the values in `l1`, and may thus be destructively modified (including destructive modification of the list's elements) without affecting `l1`. If given, the function `cmp` is called to determine whether an item in `l1` is present in `l2`; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrBool listequiv(const FrList *l1, const FrList *l2)*

Determine whether the two lists contain equal elements, ignoring any differences in order. I.e.

```
(A B "C") is equivalent to (A B "C")
(A C 2.5 "Efg" (A C)) is equivalent to ((A C) "Efg" 2.5 A C)
(1.5 42 17) is not equivalent to (17 42)
(1.5 42 17) is not equivalent to (1.5 42 17.1)
(A C 2.5 "Efg" (A C)) is not equivalent to (A C 2.5 "Efg" (C A))
```

*FrObject *listhead(FrList *list) ;*

Return the first element in the specified list.

*FrList *listintersection(const FrList *l1, const FrList *l2)*

*FrList *listintersection(const FrList *l1, const FrList *l2, FrCompareFunc cmp)*

Return a list containing all elements present in both `l1` and `l2`. The returned list is always a copy of the values in the two original lists, and may thus be destructively modified (including destructive modification of the list's elements) without affecting either `l1` or `l2`. If given, the function `cmp` is called to determine whether an item is present in both lists; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*int listlength(FrList *list)*

Return the length of the specified list.

*FrList *listmember(FrList *list, FrObject *item)*

*FrList *listmember(FrList *list, FrObject *item, FrCompareFunc cmp)*

Determine whether the given object is an element of the specified list, returning the sublist starting at the desired element (if present) or 0 if not present. If given, the function `cmp` is called to determine when a matching item is found; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrObject *listnth(FrList *list, size_t n)*

Retrieve the *n*th element of the given list, much as if the list were in fact an array subscripted beginning with zero. `listnth(l,0)` returns the first element of the list, `listnth(l,1)` returns the second, etc.

*FrList *listremove(const FrList *l, const FrObject *item)*

*FrList *listremove(const FrList *l, const FrObject *item,FrCompareFunc cmp)*

Destructively remove the first element of the given list which matches the specified item, returning a pointer to the modified list. This function should not be used if there are other pointers to the list besides the one passed in, as the first element of the list may be removed from the list and freed.

*FrList *listreverse(FrList *l)*

Destructively reverse the given list, returning a pointer to the resulting reversed list. Note that any other pointers to the original list will end up pointing at a single-element list consisting of only the first element of the original list.

*FrList *listsort(FrList *list,int (*cmp)(const FrObject*, const FrObject*)*

Destructively sort the specified list, using the given function to determine the desired ordering of elements, and return a pointer to the new beginning of the list. The `cmp` function should return zero if the two objects are equivalent, a negative value if the first object should precede the second, and a positive value if the first object should follow the second in the sorted list.

Since this function is destructive, any other pointers to the list which is sorted will end up pointing at arbitrary positions within the resulting sorted list.

*FrList *listtail(FrList *list) ;*

Return the remainder of the list, not including the first element.

*FrList *listunion(const FrList *l1, const FrList *l2)*

*FrList *listunion(const FrList *l1, const FrList *l2, FrCompareFunc cmp)*

Return a list containing the elements of `l1` plus those elements of `l2` not already contained in `l1`. The returned list is always a copy of the values in the two original lists, and may thus be destructively modified (including destructive modification of the list's elements) without affecting either `l1` or `l2`. If given, the function `cmp` is called to determine whether an item in `l2` is present in `l1`; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrList *mapcan(FrList *list, ListMapFunc mapfunc, ...)*

Apply the given function to each element of the list, and combine the returned lists with `nconc`. The mapping function takes two arguments (a `FrObject` and a `va_list` for the optional arguments) and returns a list.

*FrList *mapcar(FrList *list, ListMapFunc mapfunc, ...)*

Apply the given function to each element of the list, and return a list containing the values returned by the mapping function. The mapping function takes two arguments (a `FrObject` and a `va_list` for the optional arguments) and returns an arbitrary `FrObject`.

*FrObject *poplist(FrList *&list)*

*FrObject *poplist(FrList *&list, FrBool destructive)*

Remove the first element from the beginning of the specified list, updating the list pointer 'list' and returning the removed object. The first form of the call and the second form with a second argument of True will free the first cons cell in the list; use the second form with `False` if you have other pointers to the list besides the one passed to `poplist()`--but in that case it may be just as easy to simply do

```
var = list->car() ;
list = list->cdr() ;
```

instead of var=poplist(list,False).

7.12.3. Object-Oriented Interface

*int FrList::assoc(const FrObject *item)*

*int FrList::assoc(const FrObject *item, FrCompareFunc cmp)*

Find the element of the given association list whose head matches the specified item. If given, the function `cmp` is called to determine when a matching item is found; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrObject *FrList::car()*

Retrieve the first item in the list. This is a virtual function which may be used on any object of type `FrObject` or subclasses thereof. If the object is known to be a `FrCons` or a list, it is more efficient to use the non-virtual function `first`.

*FrObject *FrList::cdr()*

Retrieve the second item in a cons cell; in the case of a list, this is the remainder of the list after removing the first element (which is the null pointer 0 if the end of the list has been reached). This is a virtual function which may be used on any object of type `FrObject` or subclasses thereof. If the object is known to be a list, it is more efficient to use the non-virtual function `rest`.

*FrList *FrList::difference(const FrList *l2)*

*FrList *FrList::difference(const FrList *l2, FrCompareFunc cmp)*

Return a list containing those elements of the called object which are not contained in `l2`. The returned list is always a copy of the values in the called object's list, and may thus be destructively modified (including destructive modification of the list's elements) without affecting the original object. If given, the function `cmp` is called to determine whether an item in the object's list is present in `l2`; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrBool FrList::equal(const FrList *l2)*

Determine whether the other list contains equal elements in the same order as the given list.

*FrBool FrList::equiv(const FrList *l2)*

Determine whether the other list contains equal elements to those in the given list, disregarding any differences in the order of the elements. See also `listequiv`.

*virtual size_t FrList::locate(const FrObject *item, size_t start = (size_t)-1) const*

*virtual size_t FrList::locate(const FrObject *item, FrCompareFunc func,
size_t start = (size_t)-1) const*

Locate an element or subsequence of the list. If `item` is `consp()`, this method locates the first sublist after `start` matching the list pointed at by `item`; otherwise, it locates the single element matching `item`. Note that in order to locate a single element which happens to be a list, you must wrap it in another, one-element list. The search begins following the `start`h element of the list, which permits the result of a `locate` to be passed to a subsequent call in order to search for additional matches. The return value is the index into the list of the first match after `start`, or `(size_t)-1` if no match could be found.

If provided, the comparison function `func` is called repeatedly to determine which element(s) of the list match; otherwise, a simple pointer comparison (equivalent to Lisp's `eq`) is used.

*FrObject *FrList::first()*

Retrieve the first element of the list.

*FrBool FrList::intersect(const FrList *l2) const*

*FrBool FrList::intersect(const FrList *l2, FrCompareFunc cmp) const*

Determine whether the two lists have any members in common. If a comparison function is provided, it is called repeatedly to determine whether an equivalent item is present in each list; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrList *FrList::intersection(const FrList *l2) const*

*FrList *FrList::intersection(const FrList *l2, FrCompareFunc cmp) const*

Return a list containing all elements present in both the given list and `l2`. The returned list is always a copy of the values in the two original lists, and may thus be destructively modified (including destructive modification of the list's elements) without affecting either list. If given, the function `cmp` is called to determine whether an item is present in both lists; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrList *FrList::last()*

Retrieve the one-element list containing the last element in the given list.

int FrList::listlength()

Return the length of the specified list, including an empty list. Unlike the generic function `length`, this method is not a virtual method and may safely be called on any list, even the empty list (which is represented as a null pointer).

*FrList *FrList::listunion(const FrList *l2) const*

*FrList *FrList::listunion(const FrList *l2,FrCompareFunc cmp) const*

Return a list containing the elements of the given list and all elements of `l2` not already contained in the given list. All elements of the result are copies of the items in the two original list.

If given, the function `cmp` is called to determine whether an item of `l2` is present in the given list; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*int FrList::member(const FrObject *item)*

*int FrList::member(const FrObject *item, FrCompareFunc cmp)*

Determine whether the given object is an element of the specified list, returning the sublist starting at the desired element (if present) or 0 if not present. If present, the function `cmp` is called to determine whether a matching item has been found in the list; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrObject *FrList::nconc(FrList *newtail)*

Destructively append the list `newtail` to the end of the original list.

*FrObject *FrList::nth(size_t n)*

Return the *n*th item in the list, counting from zero.

*FrList *FrList::nthcdr(size_t n)*

Return the result of performing N calls to the `rest` method.

*FrObject *FrList::second()*

Returns the second element of the list, or 0 if the list has fewer than two elements.

*FrObject *FrList::third()*

Returns the third element of the list, or 0 if the list has fewer than three elements.

*FrList *FrList::operator - (const FrList *l2) const*

Return a new list containing all elements of the given list which are not contained in the second list. A simple pointer equality test (equivalent to Lisp's `eq`) is used. The returned list has been newly created and must be explicitly freed once no longer required.

*FrList *FrList::operator + (const FrList *l2) const*

Return a new list containing the union of the elments in the two given lists. A simple pointer equality test (equivalent to Lisp's `eq`) is used to test membership.

*FrList *FrList::operator * (const FrList *l2) const*

Return a new list containing the intersection of the elements in the two given lists. A simple pointer equality test is used to determine common membership.

*FrObject *&FrList::operator [] (size_t position) const*

Return a reference to the element at the `position`th location in the list, counting from zero. Thus, `FrList[0]` is the list's head element, `FrList[1]` is the second element, etc. Modifying the returned value is equivalent to calling `replaca` on the appropriate element of the list; as with `replaca`, you must first free the object before replacing it in order to avoid a memory leak.

*int FrList::position(const FrObject *item) const*

*int FrList::position(const FrObject *item, FrCompareFunc cmp) const*

*int FrList::position(const FrObject *item, FrBool from_end) const*

*int FrList::position(const FrObject *item, FrCompareFunc cmp,*
FrBool from_end) const

Determine the relative position of `item` within the list (0 = first element of list, 1 = second, etc.); returns -1 if no matching item is present in the list. If `from_end` is omitted or is `False`, this method returns the position of the first occurrence of a matching item; if present and `True`, this method returns the position of hte last occurrence. If `cmp` is specified, it is called for each item of the list to determine whether a matching item is present in the list; otherwise, a simple pointer comparison (equivalent to Lisp's `eq`) is used.

*FrObject *FrList::replaca()*

Destructively replace the first element in the original list. You must first free the object being replaced in order to avoid a memory leak (unless there is another pointer to that object).

*FrObject *FrList::replacd()*

Destructively replace the tail of the given list. If there is no other pointer to the tail, you must first free it before replacing it in order to avoid a memory leak.

*FrObject *FrList::rest()*

Retrieve the remainder of the list, i.e. the list that would be created by removing the first element of the original list. If the specified list has only one element, the null pointer 0 is returned. Any destructive modifications to the returned list will also affect the original list.

*FrList *FrList::sort(int (*cmpfunc)(const FrObject *,const FrObject *))*

Destructively sort the specified list, using the given function to determine the desired ordering of elements, and return a pointer to the new beginning of the list. The `cmp` function should return zero if the two objects are equivalent, a negative value if the first object should precede the second, and a positive value if the first object should follow the second in the sorted list.

Since this function is destructive, any other pointers to the list which is sorted will end up pointing at unpredictable positions within the resulting sorted list.

*FrBool FrList::subsetOf(const FrList *l2) const*

*FrBool FrList::subsetOf(const FrList *l2, FrCompareFunc cmp) const*

Determine whether the list is a subset of the indicated list (contains only items which are also in `l2`). If a comparison function is provided, it is called repeatedly to determine whether an equivalent item is present in each list; if omitted, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

7.13. Structure Functions

FramepaC's `FrStruct` type is the equivalent of Lisp's structure, and may be read from a file or written to a file in a Lisp-compatible format. Unlike Lisp's `(defstruct)` however, `FrStruct` does not have a statically-declared set of fields -- arbitrary fields may be added to any `FrStruct` instance at any time. In this respect, `FrStruct` is more akin to an associative array than to a record.

*FrStruct::FrStruct(const FrSymbol *type)*

Create a new structure with the specified type tag. The structure is initially empty.

FrStruct::~FrStruct()

Destroy the indicated structure, removing all items stored in it.

*void FrStruct::put(const FrSymbol *fieldname, const FrObject *value)*

Add the specified field to the structure if it does not already exist, and set the field's value to be a copy of the indicated object. If the field already exists and it not empty, the existing value will be discarded via `freeObject()`.

*FrObject *FrStruct::get(const FrSymbol *fieldname)*

Retrieve the value associated with the indicated field, if the field exists. Returns 0 if the field does not exist in the structure or the field is empty. The returned pointer points at the actual object stored in the structure, so any destructive modification of the return value will be reflected in any future retrievals (such an action is not recommended and not guaranteed to work in the future).

*FrSymbol *FrStruct::typeName() const*

Get the type tag associated with the structure when it was created.

*FrList *FrStruct::fieldNames() const*

Return a list consisting of the names of the fields currently present in the structure. This list is newly-created and must be deleted (e.g. with `freeObject`) when no longer required.

virtual FrBool FrStruct::iterateVA(FrIteratorFunc func, va_list args) const

Call the specified function `func` once for each field in the structure. The function is called with a `FrCons` of the field name and field value and the variable-argument list `args`. If any invocation of `func` returns `False`, the iteration is terminated immediately and `iterateVA` returns `False`; otherwise, this method returns `True`.

DO NOT MODIFY THE CONTENTS OF THE FrCONS PASSED TO THE ITERATION FUNCTION. You should also make a copy of the `FrCons` if you need to refer to it again after modifying the structure with `put`.

7.14. FrQueue Functions

Since queues are a useful data structure for many algorithms, FramepaC includes a `FrQueue` data type providing a single-ended queue.

FrQueue::FrQueue()

Create an empty queue.

*FrQueue::FrQueue(const FrList *items)*

Create a queue containing the specified items. The first element of the list becomes the queue's head (first item to be returned on reading) and the last element of the list becomes the tail of the queue.

FrQueue::~FrQueue()

Destroy the queue.

*void FrQueue::add(const FrObject *item, FrBool copy = True)*

Add the specified item to the tail end of the queue. By default, a complete copy of the item is made and that copy actually placed on the queue; if the supplied object was dynamically allocated and will never again be modified by the calling program, this extra copying step may be bypassed by supplying `False` as the second argument.

*void FrQueue::addFront(const FrObject *item, FrBool copy = True)*

Add the specified item to the head end of the queue, where it becomes the next item to be returned by `pop` or `peek`. By default, a complete copy of the item is made and that copy actually placed on the queue; if the supplied object was dynamically allocated and will never again be modified by the calling program, this extra copying step may be bypassed by supplying `False` as the second argument.

*FrBool FrQueue::remove(const FrObject *item)*

Remove the first item in the queue which is identical to (pointer equality) the specified item from the queue. This function returns `True` if anything was removed, `False` if not.

*FrBool FrQueue::remove(const FrObject *item, FrCompareFunc cmp)*

Remove the first item in the queue which compares as equal to the specified item (via a call to the supplied function `cmp`) from the queue. This function returns `True` if anything was removed, `False` if not.

*FrObject *FrQueue::find(const FrObject *item, FrCompareFunc cmp) const*

Find the first item in the queue which compares as equal to the specified item (via a call to the supplied function `cmp`), and return a pointer to that item, or 0 if no matching item is contained in the queue.

*virtual size_t FrQueue::locate(const FrObject *item, size_t start = (size_t)-1) const*

*virtual size_t FrQueue::locate(const FrObject *item, FrCompareFunc func, size_t start = (size_t)-1) const*

Locate an element or subsequence of the queue. If `item` is `consp()`, this method locates the first subsequence in the queue after `start` matching the list pointed at by `item`; otherwise, it locates the single element matching `item`. Note that in order to locate a single element which happens to be a list, you must wrap it in another, one-element list. The search begins following the `start`h element of the queue, which permits the result of a `locate` to be passed to a subsequent call in order to search for additional matches. The return value is the index into the queue of the first match after `start`, or `(size_t)-1` if no match could be found.

If provided, the comparison function `func` is called repeately to determine which element(s) of the queue match; otherwise, a simple pointer comparison (equivalent to Lisp's `eq`) is used.

*FrObject *FrQueue::pop()*

Remove the first item in the queue and return a pointer to it. The returned item must be deallocated with `freeObject` or `free_object` once it is no longer needed. This method returns 0 if the queue is already empty.

*FrObject *FrQueue::peek() const*

Return the first item in the queue without removing it, or 0 if the queue is currently empty.

void FrQueue::clear()

Remove all items from the queue.

size_t FrQueue::queueLength()

Return the number of items currently in the queue.

7.15. Frame Functions

Since frame manipulation is the main purpose of FramepaC, it provides an extensive set of functions which operate on instances of the class `FrFrame`. Each frame is associated with a symbol, which is its name. Since frames are associated with symbols, they are also associated with a symbol table, and will thus automatically be deleted when their symbol table is destroyed. Unless otherwise specified, all frame-related functions operate using symbols from the current symbol table, which allows multiple sets of frames to coexist even if some of their names clash.

7.15.1. Creating and Loading Frames

*FrFrame *create_frame(FrSymbol *name)*

Create a new frame if one with the given name does not exist, or return the existing frame by that name.

*FrFrame *create_vframe(FrSymbol *name)*

Create a new virtual frame if no frame with the given name exists, or return the existing frame by that name (fetching it from the server or backing store if necessary).

*FrFrame *find_frame(FrSymbol *name)*

Return a pointer to the frame associated with the given symbol (i.e. the frame with the given name) or 0 if there is no frame by that name currently in memory.

*VFrame *find_vframe(FrSymbol *name)*

Return a pointer to the frame associated with the given symbol (i.e. the frame with the given name) or 0 if there is no frame by that name. If necessary, the frame will be fetched from the server or backing store.

*FrFrame *get_old_frame(FrSymbol *frame,int generation)*

If the symbol is the name of a virtual frame with backing store, this function reloads the requested prior version of the frame from the backing store (generation 0 is the most recently saved copy of the frame, generation 1 is the next most-recent copy, generation 2 is the version prior to that, etc.). This function has no effect if there is no backing store for the desired frame.

*void create_slot(FrFrame *frame, FrSymbol *slotname)*

*void create_slot(FrSymbol *frame, FrSymbol *slotname)*

Create a slot with the specified name if it does not already exist in the frame. Note that it is generally not necessary to call this function, since all the `add_XX` functions create slots as needed.

A slot always contains either two or three facets (depending on compile-time options), even when newly-created, though they are initially empty; additional facets will be created as fillers for them are added. The facets which always exist are `VALUE`, `SEM`, and optionally `INHERITS` (used for local or "sideways" inheritance).

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

7.15.2. Testing Frames

*FrBool frame_is_empty(const FrFrame *frame)*

*FrBool frame_is_empty(const FrSymbol *name)*

Determine whether there are any fillers in the frame at all. This will normally return True only for frames which have just been created or erased.

*FrBool is_a_p(FrFrame *frame, FrFrame *possible_parent)*

*FrBool is_a_p(FrSymbol *frame, FrSymbol *possible_parent)*

Determine whether one frame is an ancestor of another by following the `INSTANCE-OF` and `IS-A` links.

Returns `True` if `possible_parent` can be reached from `frame` through some chain of `IS-A` links, possibly with an initial `INSTANCE-OF` link.

The former variant operates only on in-memory frames, while the latter variant is for virtual frames and will fetch the frame as necessary.

*FrBool part_of_p(FrFrame *frame, FrFrame *possible_container)*

*FrBool part_of_p(FrSymbol *frame, FrSymbol *possible_container)*

Determine whether one entity described by a frame is included as a part of another by following the `PART-OF` link. Returns `True` if `possible_container` can be reached from `frame` through some chain of `PART-OF` links.

The former variant operates only on in-memory frames, while the latter variant is for virtual frames and will fetch the frame as necessary.

7.15.3. Manipulating Frames

*void add_filler(FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet,FrObject *filler)*

*void add_filler(FrSymbol *frame, const FrSymbol *slot, const FrSymbol *facet,FrObject *filler)*

Append the specified filler to the end of the list of fillers for the indicated slot and facet if it is not already present. The caller may discard or otherwise destructively modify the specified filler once this function returns, as the value actually added to the frame is a copy.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void add_value(FrFrame *frame, const FrSymbol *slot,FrObject *filler)*

*void add_value(FrSymbol *frame, const FrSymbol *slot,FrObject *filler)*

Append the specified filler to the end of the list of fillers for the indicated slot's `VALUE` facet if it is not already present. The caller may discard or otherwise destructively modify the specified filler once this function returns, as the value actually added to the frame is a copy.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void add_sem(FrFrame *frame, const FrSymbol *slot,FrObject *filler)*

*void add_sem(FrSymbol *frame, const FrSymbol *slot,FrObject *filler)*

Append the specified filler to the end of the list of fillers for the indicated slot's `SEM` facet if it is not already present. The caller may discard or otherwise destructively modify the specified filler once this function returns, as the value actually added to the frame is a copy.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void add_fillers(FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet,FrList *fillers)*

*void add_fillers(FrSymbol *frame, const FrSymbol *slot, const FrSymbol *facet,FrList *fillers)*

Append each item in the given list of fillers to the specified slot and facet. The caller may discard or otherwise destructively modify the specified fillers once this function returns, as the values actually added to the frame are copies of the supplied values.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void add_values(FrFrame *frame, const FrSymbol *slot,FrList *fillers)*

*void add_values(FrSymbol *frame, const FrSymbol *slot,FrList *fillers)*

Append each item in the given list of fillers to the specified slot's `VALUE` facet. The caller may discard or otherwise destructively modify the specified fillers once this function returns, as the values actually added to the frame are copies of the supplied values.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void add_sems(FrFrame *frame, const FrSymbol *slot,FrList *fillers)*

*void add_sems(FrSymbol *frame, const FrSymbol *slot,FrList *fillers)*

Append each item in the given list of fillers to the specified slot's `SEM` facet. The caller may discard or otherwise destructively modify the specified fillers once this function returns, as the values actually added to the frame are copies of the supplied values.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void erase_facet(FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet)*

*void erase_facet(FrSymbol *frame, const FrSymbol *slot, const FrSymbol *facet)*

Delete all fillers from the specified facet.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void erase_filler(FrFrame *frame, FrSymbol *slot, FrSymbol *facet, FrObject *filler)*

*void erase_filler(FrFrame *frame, FrSymbol *slot, FrSymbol *facet, FrObject *filler, FrCompareFunc cmp)*

*void erase_filler(FrSymbol *frame, FrSymbol *slot, FrSymbol *facet, FrObject *filler)*

*void erase_filler(FrSymbol *frame, FrSymbol *slot, FrSymbol *facet, FrObject *filler, FrCompareFunc cmp)*

Delete the specified filler in the indicated slot's facet, using the specified comparison function (currently `eql` or `equal`, see Section 7.4.1) for the second form. The first form always uses a simple pointer comparison, the equivalent of Lisp's `eq`.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*void erase_frame(FrFrame *frame)*

*void erase_frame(FrSymbol *frame)*

Delete all fillers for the entire frame, but do not deallocate the frame proper.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void erase_sem(FrFrame *frame, const FrSymbol *slot, FrObject *filler)*

*void erase_sem(FrFrame *frame, const FrSymbol *slot, FrObject *filler, FrCompareFunc cmp)*

*void erase_sem(FrSymbol *frame, const FrSymbol *slot, FrObject *filler)*

*void erase_sem(FrSymbol *frame, const FrSymbol *slot,FrObject *filler, FrCompareFunc cmp)*

Delete the specified filler from the indicated slot's `SEM` facet, using the specified comparison function for the second form. The two comparison functions provided standard with FramepaC are `eql` and `equal` (see Section 7.4.1); if no comparison function is specified, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*void erase_slot(FrFrame *frame, const FrSymbol *slot)*

*void erase_slot(FrSymbol *frame, const FrSymbol *slot)*

Delete all fillers from all facets of the specified slot.

The first variant operates only on in-memory frames, while the second variant is for virtual frames and will fetch the frame if necessary.

*void erase_value(FrFrame *frame, const FrSymbol *slot, FrObject *filler)*

*void erase_value(FrFrame *frame, const FrSymbol *slot, FrObject *filler, FrCompareFunc cmp)*

*void erase_value(FrSymbol *frame, const FrSymbol *slot, FrObject *filler)*

*void erase_value(FrSymbol *frame, const FrSymbol *slot,FrObject *filler, FrCompareFunc cmp)*

Delete the specified filler from the indicated slot's `VALUE` facet, using the specified comparison function for the second form. The two comparison functions provided standard with FramepaC are `eql` and `equal` (see Section 7.4.1); if no comparison function is specified, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*FrObject *first_filler(FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet)*

*FrObject *first_filler(FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet, FrBool inherit)*

*FrObject *first_filler(FrSymbol *frame, const FrSymbol *slot, const FrSymbol *facet)*

*FrObject *first_filler(FrSymbol *frame, const FrSymbol *slot, const FrSymbol *facet, FrBool inherit)*

Retrieve the first filler for the given slot and facet. DO NOT DESTRUCTIVELY MODIFY THE RETURNED FILLER. For the second and fourth forms, if `inherit` is `False`, the value (FrList *)0 will be returned if there are no fillers for the facet in the given frame; otherwise, if there are no fillers, FramepaC attempts to retrieve the first inherited filler.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*FrList *get_fillers(FrFrame *frame, FrSymbol *slot, FrSymbol *facet)*

*FrList *get_fillers(FrFrame *frame, FrSymbol *slot, FrSymbol *facet,FrBool inherit)*

*FrList *get_fillers(FrSymbol *frame, FrSymbol *slot, FrSymbol *facet)*

*FrList *get_fillers(FrSymbol *frame, FrSymbol *slot, FrSymbol *facet, FrBool inherit)*

Retrieve the list of fillers for the given slot and facet. DO NOT MODIFY THE RETURNED LIST. For the second form and fourth forms, if `inherit` is `False`, the value (FrList *)0 will be returned if there are no fillers for that facet in the frame; otherwise, if there are no fillers, FramepaC attempts to retrieve inherited fillers by following the INSTANCE-OF and IS-A links.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*FrList *get_sem(FrFrame *frame, FrSymbol *slot)*

*FrList *get_sem(FrFrame *frame, FrSymbol *slot,FrBool inherit)*

*FrList *get_sem(FrSymbol *frame, FrSymbol *slot)*

*FrList *get_sem(FrSymbol *frame, FrSymbol *slot,FrBool inherit)*

Retrieve the list of fillers for the given slot's `SEM` facet. DO NOT MODIFY THE RETURNED LIST. For the second form and fourth forms, if `inherit` is `False`, the value (FrList *)0 will be returned if there are no fillers for that facet; otherwise, if there are no fillers, FramepaC attempts to retrieve inherited fillers by following the INSTANCE-OF and IS-A links.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*FrList *get_values(FrFrame *frame, FrSymbol *slot)*

*FrList *get_values(FrFrame *frame, FrSymbol *slot,FrBool inherit)*

*FrList *get_values(FrSymbol *frame, FrSymbol *slot)*

*FrList *get_values(FrSymbol *frame, FrSymbol *slot,FrBool inherit)*

Retrieve the list of fillers for the given slot's `VALUE` facet. DO NOT MODIFY THE RETURNED LIST. For the second form and fourth forms, if 'inherit' is `False`, the value (FrList *)0 will be returned if there are no fillers for that facet; otherwise, if there are no fillers, FramepaC attempts to retrieve inherited fillers by following the INSTANCE-OF and IS-A links.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

*FrObject *get_value(FrFrame *frame, const FrSymbol *slot)*

*FrObject *get_value(FrFrame *frame, const FrSymbol *slot, FrBool inherit)*

*FrObject *get_value(FrSymbol *frame, const FrSymbol *slot)*

*FrObject *get_value(FrSymbol *frame, const FrSymbol *slot, FrBool inherit)*

Retrieve the first filler for the given slot's `VALUE` facet. DO NOT DESTRUCTIVELY MODIFY THE RETURNED FILLER.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

7.15.4. Processing Entire Frames

*FrBool do_slots(FrFrame *frame, FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, va_list args), ...)*

*FrBool do_slots(FrSymbol *frame, FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, va_list args), ...)*

Call the indicated function once for each slot in the given frame. Returns `True` if all invocations of the function returned `True`, `False` if any of them returned `False` (in which case no further slots are processed). The final parameter passed to the indicated function is a variable argument list consisting of all the remaining arguments passed to `do_slots()`; these may be accessed with `va_start`, `va_arg`, and `va_end`.

The former variant operates only on in-memory frames, while the latter variant is for virtual frames and will fetch the frame as necessary.

*FrBool do_facets(FrFrame *frame, FrSymbol *slot,*
*FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet, va_list args), ...)*

*FrBool do_facets(FrSymbol *frame, FrSymbol *slot,*
*FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet, va_list args), ...)*

Call the indicated function once for each nonempty facet in the given slot. Returns `True` if all invocations of the function returned `True`, `False` if any of them returned `False` (in which case no further facets are processed). The final parameter passed to the indicated function is a variable argument list consisting of all the remaining arguments passed to `do_facets()`; these may be accessed with `va_start`, `va_arg`, and `va_end`.

The former variant operates only on in-memory frames, while the latter variant is for virtual frames and will fetch the frame as necessary.

*FrBool do_all_facets(FrFrame *frame,*
*FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet, va_list args), ...)*

*FrBool do_all_facets(FrSymbol *frame,*
*FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet, va_list args), ...)*

Call the indicated function once for each nonempty facet in the given frame. Returns `True` if all invocations of the function returned `True`, `False` if any of them returned `False` (in which case no further facets are processed). The final parameter passed to the indicated function is a variable argument list consisting of all the remaining arguments passed to `do_all_facets()`; these may be accessed with `va_start`, `va_arg`, and `va_end`.

The former variant operates only on in-memory frames, while the latter variant is for virtual frames and will fetch the frame as necessary.

*void doAllFrames(void (*func)(const FrFrame *frame, va_list args), ...)*

Invoke the specified function for each frame which is currently instantiated for the active symbol table (see Section 7.9). Note that when using virtual frames, only those frames which are currently in memory will be processed; frames which are still in the backing store will be skipped. The indicated function is called with a pointer to the frame and any additional arguments provided to `doAllFrames()`.

virtual FrBool FrFrame::iterateVA(FrIteratorFunc func, va_list args) const

Call the specified function `func` once for each facet currently in the frame which have fillers. The function is called with a `FrList` of the frame's name, the slot name, the facet name, and the filler list for the facet, as well as the variable-argument list `args`. If any invocation of `func` returns `False`, the iteration is terminated immediately and `iterateVA` returns `False`; otherwise, this method returns `True`.

DO NOT MODIFY THE CONTENTS OF THE FrList PASSED TO THE ITERATION FUNCTION. You should also make a deep copy of the `FrList` if you need to refer to it again after modifying the frame with any of the other methods.

*FrList *FrFrame::allSlots() const*

*FrList *FrSymbol::allSlots()*

Returns a list consisting of the names of all slots within the given frame, whether or not the slot has any fillers at all (generally, only the predefined slots will ever exist without having any fillers). The returned list has been newly created and must be explicitly freed once no longer required.

*FrList *facets_in_slot(const FrFrame *frame, const FrSymbol *slotname)*

*FrList *facets_in_slot(FrSymbol *framename, const FrSymbol *slotname)*

Returns a list consisting of the names of all facets within the indicated slot, whether or not the facet has any fillers at all (generally, only the predefined slots--`VALUE`, `SEM`, and optionally `INHERITS`--will ever exist without having any fillers). The returned list has been newly created and must be explicitly freed once no longer required.

*FrList *slots_in_frame(const FrFrame *frame)*

*FrList *slots_in_frame(FrSymbol *framename)*

Returns a list consisting of the names of all slots within the given frame, whether or not the slot has any fillers at all (generally, only the predefined slots will ever exist without having any fillers). The returned list has been newly created and must be explicitly freed once no longer required.

*FrList *FrFrame::slotFacets(const FrSymbol *slotname) const*

*FrList *FrSymbol::slotFacets(const FrSymbol *slotname)*

Returns a list consisting of the names of all facets within the indicated slot, whether or not the facet has any fillers at all (generally, only the predefined slots--`VALUE`, `SEM`, and optionally `INHERITS`--will ever exist without having any fillers). The returned list has been newly created and must be explicitly freed once no longer required.

7.15.5. Inheritance

Inheritance is a powerful capability afforded by frames. When a value is not know for the current frame, the system can search for a value in other frames specified by some inheritance link and search mechanism. If a value is found in this way, it is returned as if it had been in the frame from the beginning.

void set_inheritance_type(InheritanceType inherit)

Specify how FramepaC should search for values for a slot which has no values in the current frame. The currently supported values are `NoInherit`, `InheritSimple` (only follow first filler of INSTANCE-OF and IS-A slots), `InheritDFS` (depth-first search on IS-A), `InheritBFS` (breadth-first search on IS-A), `InheritPartDFS` (depth-first search on PART-OF), and `InheritLocalDFS` (first try to follow slot's `INHERITS` facet, then do regular depth-first search). In addition, the symbols `InheritPartBFS` and `InheritLocalBFS` are defined but not yet supported.

InheritanceType get_inheritance_type()

Determine how values are inherited for slots with no values of their own.

*FrList *inheritable_slots(const FrFrame *frame, InheritanceType inherit)*

*FrList *inheritable_slots(FrSymbol *frame, InheritanceType inherit)*

Determine which slots and facets the given frame may inherit from one or more of its ancestors using the indicated inheritance method. The result is returned as an association list of slots with the facets available for the slot. For example,

```
((IS-A VALUE SEM)
 (SUBCLASSES VALUE SEM)
 (INSTANCE-OF VALUE SEM)
 (INSTANCES VALUE SEM)
 (WEIGHT VALUE SEM M-UNIT DEFAULT)
 )
```

The former variant operates only on in-memory frames, while the latter variant is for virtual frames and will fetch the frame as necessary. The returned list has been newly created and must be explicitly freed once no longer required.

*void inherit_all_fillers(const FrFrame *frame)*

*void inherit_all_fillers(const FrFrame *frame, InheritanceType inherit)*

*void inherit_all_fillers(FrSymbol *frame)*

*void inherit_all_fillers(FrSymbol *frame, InheritanceType inherit)*

Find all fillers which the given frame can possibly inherit using specified (or the current) inheritance method, and place those fillers directly in the given frame. The one-argument variants use the current inheritance method to retrieve the fillers.

The first two variants operate only on in-memory frames, while the final two variants are for virtual frames and will fetch the frame if necessary.

7.15.6. Object-Oriented Interface

As has been mentioned previously, FramepaC provides both a procedural and an object-orient interface; this section describes the object methods to which most of the preceding procedural calls expand.

Most of the frame-related methods exist in both class `FrFrame` and class `FrSymbol`. The former operate only on in-memory frames, while the latter can also function with virtual frames, fetching them as required. The trade-off is naturally somewhat slower execution due to the overhead of finding and possibly fetching the frame corresponding to the symbol and then calling the equivalent method in class `FrFrame`.

*FrFrame::FrFrame(const char *framename)*

*FrFrame::FrFrame(const FrSymbol *framename)*

*VFrame::VFrame(const char *framename)*

*VFrame::VFrame(const FrSymbol *framename)*

Create a new frame with the given name. It is an error to create a frame by the same name as one which already exists in the current symbol table; therefore, it is usually preferable to call either `create_frame`, `create_vframe`, `FrSymbol::createFrame`, or `FrSymbol::createVFrame`. These functions will return an existing frame rather than create a duplicate.

FrFrame::~FrFrame

VFrame::~VFrame

Destroy the frame and deallocate its storage.

*FrFrame *FrSymbol::createFrame()*

*FrFrame *FrSymbol::createVFrame()*

Create a new frame or virtual frame with the given name if it does not yet exist, or return a pointer to the existing frame by that name.

*void FrFrame::addFiller(const FrSymbol *slotname, const FrSymbol *facetname, const FrObject *filler)*

*void FrSymbol::addFiller(const FrSymbol *slotname, const FrSymbol *facetname, const FrObject *filler)*

Append the specified filler to the end of the list of fillers for the indicated slot and facet if it is not already present. The caller may discard or otherwise destructively modify the specified filler once this method call returns, as the value actually added to the frame is a copy.

*void FrFrame::addFillers(const FrSymbol *slotname, const FrSymbol *facetname, const FrList *fillers)*

*void FrSymbol::addFillers(const FrSymbol *slotname, const FrSymbol *facetname, const FrList *fillers)*

Append each item in the given list of fillers to the specified slot and facet. The caller may discard or otherwise destructively modify the specified fillers once this method call returns, as the values actually added to the frame are copies of the supplied values.

*void FrFrame::addSem(const FrSymbol *slotname, const FrObject *filler)*

*void FrSymbol::addSem(const FrSymbol *slotname, const FrObject *filler)*

Append the specified filler to the end of the list of fillers for the indicated slot's `SEM` facet if it is not already present. The caller may discard or otherwise destructively modify the specified filler once this function returns, as the value actually added to the frame is a copy.

*void FrFrame::addSems(const FrSymbol *slotname, const FrList *fillers)*

*void FrSymbol::addSems(const FrSymbol *slotname, const FrList *fillers)*

Append each item in the given list of fillers to the specified slot's `SEM` facet. The caller may discard or otherwise destructively modify the specified fillers once this function returns, as the values actually added to the frame are copies of the supplied values.

*void FrFrame::addValue(const FrSymbol *slotname, const FrObject *filler)*

*void FrSymbol::addValue(const FrSymbol *slotname, const FrObject *filler)*

Append the specified filler to the end of the list of fillers for the indicated slot's `VALUE` facet if it is not already present. The caller may discard or otherwise destructively modify the specified filler once this function returns, as the value actually added to the frame is a copy.

*void FrFrame::addValues(const FrSymbol *slotname, const FrList *fillers)*

*void FrSymbol::addValues(const FrSymbol *slotname, const FrList *fillers)*

Append each item in the given list of fillers to the specified slot's `VALUE` facet. The caller may discard or otherwise destructively modify the specified fillers once this function returns, as the values actually added to the frame are copies of the supplied values.

*FrList *FrFrame::collectSlots(InheritanceType inherit, FrList *allslots=0)*

*FrList *FrSymbol::collectSlots(InheritanceType inherit, FrList *allslots=0)*

Determine which slots and facets the given frame may inherit from one or more of its ancestors using the indicated inheritance method. The result is returned as an association list of slots with the facets available for the slot. For example,

```
((IS-A VALUE SEM)
 (SUBCLASSES VALUE SEM)
 (INSTANCE-OF VALUE SEM)
 (INSTANCES VALUE SEM)
 (WEIGHT VALUE SEM M-UNIT DEFAULT)
 )
```

The returned list has been newly created and must be explicitly freed once no longer required.

int FrFrame::commitFrame()

int FrSymbol::commitFrame()

Store the frame to the backing store if its "dirty" flag indicates that the backing store needs to be updated.

*FrFrame *Frame::copyFrame(FrSymbol *newname)*

*FrFrame *FrSymbol::copyFrame(FrSymbol *newname)*

Make an exact copy of the given old frame into the specified new frame. Links will not be updated, as the purpose of this function is to maintain backup copies of a frame which is changing. Because the links are not updated, the copy of the frame should not be modified, as that will cause internal data to become inconsistent.

*FrSlot *createSlot(const FrSymbol *slotname)*

Create a slot with the given name if it does not already exist in the frame. The return value is used internally and should be ignored--it is an error to modify the `FrSlot` structure.

This function is rarely required, since all functions which add fillers to the frame will create slots as needed.

int FrSymbol::deleteFrame()

Remove the specified frame from main memory and from the backing store. The frame will be lost for all time unless the backing store preserves old versions of frames.

FrBool FrFrame::dirtyFrame()

FrBool FrSymbol::dirtyFrame()

Determine whether the frame has been modified since it was created, loaded, or last committed to the backing store.

*int delete_frame(FrSymbol *frame)*

Remove the specified frame from main memory and from the backing store. The frame will be lost for all time unless the backing store preserves old versions of frames.

int FrSymbol::discardFrame()

*int discard_frame(FrSymbol *frame)*

Remove the specified frame from main memory without first updating it in the backing store. If the frame is not a virtual frame or the VFrame system is being operated without backing store, the frame is lost for all time.

size_t FrFrame::displayLength()

Determine how many bytes will be required to store the printed representation generated by `displayValue`.

*char *FrFrame::displayValue(char *buffer)*

Store a printed representation of the frame in the indicated buffer, and return a pointer to the byte following the last one stored there.

*FrBool doAllFacets(FrBool (*func)(const FrFrame *frame,*
*const FrSymbol *slot, const FrSymbol *facet, va_list args),*
va_list args)

Call the indicated function once for each nonempty facet in the given frame. Returns `True` if all invocations of the function returned `True`, `False` if any of them returned `False` (in which case no further facets are processed. The final parameter passed to the indicated function is the same variable argument list passed to `doAllFacets`; the arguments in the list may be accessed with `va_start`, `va_arg`, and `va_end`.

*FrBool doFacets(const FrSymbol *slotname,*
*FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, const FrSymbol *facet, va_list args),*
va_list args)

Call the indicated function once for each nonempty facet in the given slot. Returns `True` if all invocations of the function returned `True`, `False` if any of them returned `False` (in which case no further facets are processed). The final parameter passed to the indicated function is the same variable argument list passed to `doFacets`; the arguments in the list may be accessed with `va_start`, `va_arg`, and `va_end`.

*FrBool doSlots(FrBool (*func)(const FrFrame *frame, const FrSymbol *slot, va_list args), va_list args)*

Call the indicated function once for each slot in the given frame. Returns `True` if all invocations of the function returned `True`, `False` if any of them returned `False` (in which case no further slots are processed). The final parameter passed to the indicated function is the same variable argument list passed to `doSlots`; the arguments in the list may be accessed with `va_start`, `va_arg`, and `va_end`.

FrBool FrFrame::emptyFrame()

FrBool FrSymbol::emptyFrame()

Determine whether there are any non-empty slots in the frame. Returns `True` if the frame is completely empty, `False` if it contains any fillers. This check does not test for inherited fillers.

*void FrFrame::eraseFacet(const FrSymbol *slotname, const FrSymbol *facetname)*

*void FrSymbol::eraseFacet(const FrSymbol *slotname, const FrSymbol *facetname)*

Delete all fillers from the specified facet of the given slot.

*void FrFrame::eraseFiller(const FrSymbol *slotname, const FrSymbol *facetname, const FrObject *filler)*

*void FrFrame::eraseFiller(const FrSymbol *slotname, const FrSymbol *facetname, const FrObject *filler,*
FrCompareFunc cmp)

*void FrSymbol::eraseFiller(const FrSymbol *slotname, const FrSymbol *facetname, const FrObject *filler)*

*void FrSymbol::eraseFiller(const FrSymbol *slotname, const FrSymbol *facetname, const FrObject *filler,*
FrCompareFunc cmp)

Delete the specified filler in the indicated slot's facet, using the given comparison function if supplied. The two comparison functions provided standard with FramepaC are `eql` and `equal` (see Section 7.4.1); if no comparison function is specified, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

void FrFrame::eraseFrame()

void FrSymbol::eraseFrame()

Delete all fillers for the entire frame, but do not delete the frame itself (for that, use either `FrSymbol::deleteFrame`, `delete_frame(FrSymbol*)`, or `delete FrFrame*`).

*void FrFrame::eraseSlot(const char *slotname)*

*void FrFrame::eraseSlot(const FrSymbol *slotname)*

*void FrSymbol::eraseSlot(const char *slotname)*

*void FrSymbol::eraseSlot(const FrSymbol *slotname)*

Delete all fillers from all facets of the specified slot.

*void FrFrame::eraseSem(const FrSymbol *slotname, const FrObject *filler)*

*void FrFrame::eraseSem(const FrSymbol *slotname, const FrObject *filler, FrCompareFunc cmp)*

*void FrSymbol::eraseSem(const FrSymbol *slotname, const FrObject *filler)*

*void FrSymbol::eraseSem(const FrSymbol *slotname, const FrObject *filler, FrCompareFunc cmp)*

Delete the specified filler from the indicated slot's `SEM` facet, using the specified comparison function if supplied. The two comparison functions provided standard with FramepaC are `eql` and `equal` (see Section 7.4.1); if no comparison function is specified, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*void eraseValue(const FrSymbol *slotname, const FrObject *filler)*

*void eraseValue(const FrSymbol *slotname, const FrObject *filler, FrCompareFunc cmp)*

Delete the specified filler from the indicated slot's `VALUE` facet, using the specified comparison function if supplied. The two comparison functions provided standard with FramepaC are `eql` and `equal` (see Section 7.4.1); if no comparison function is specified, a simple pointer equality test (equivalent to Lisp's `eq`) is used.

*FrObject *FrFrame::firstFiller(const FrSymbol *slotname, const FrSymbol *facet, FrBool inherit = True)*

*FrObject *FrSymbol::firstFiller(const FrSymbol *slotname, const FrSymbol *facet, FrBool inherit = True)*

Retrieve the first filler for the given slot and facet, performing the current inheritance method if the facet is absent or empty and `inherit` is `True`. DO NOT DESTRUCTIVELY MODIFY THE RETURNED FILLER.

*FrSymbol *FrFrame::frameName()*

Get the frame's name.

*FrList *FrFrame::getFillers(const FrSymbol *slotname, const FrSymbol *facet, FrBool inherit = True)*

*FrList *FrSymbol::getFillers(const FrSymbol *slotname, const FrSymbol *facet, FrBool inherit = True)*

Retrieve the list of fillers present in the given slot and facet, performing the current inheritance method if the facet is absent or empty in the specified frame and `inherit` is `True`. **DO NOT MODIFY THE RETURNED LIST**.

*FrList *FrFrame::getImmedFillers(const FrSymbol *slotname, const FrSymbol *facet)*

*FrList *FrSymbol::getImmedFillers(const FrSymbol *slotname, const FrSymbol *facet)*

Retrieve the list of fillers present in the given slot and facet, without attempting to inherit the fillers if the facet is absent or empty. **DO NOT MODIFY THE RETURNED LIST**.

*FrList *FrFrame::getSem(const FrSymbol *slotname,FrBool inherit = True)*

*FrList *FrSymbol::getSem(const FrSymbol *slotname,FrBool inherit = True)*

Retrieve the list of fillers for the given slot's `SEM` facet, performing the current inheritance method if the facet is empty or the slot does not exist and `inherit` is `True`. **DO NOT MODIFY THE RETURNED LIST**.

*FrObject *FrFrame::getValue(const FrSymbol *slotname,FrBool inherit = True)*

*FrObject *FrSymbol::getValue(const FrSymbol *slotname,FrBool inherit = True)*

Retrieve the first filler for the given slot's `VALUE` facet, performing the current inheritance method if the facet is empty or the slot does not exist and `inherit` is `True`. **DO NOT DESTRUCTIVELY MODIFY THE RETURNED FILLER**.

*FrList *FrFrame::getValues(const FrSymbol *slotname,FrBool inherit = True)*

*FrList *FrSymbol::getValues(const FrSymbol *slotname,FrBool inherit = True)*

Retrieve the list of fillers for the given slot's `VALUE` facet, performing the current inheritance method if the facet is empty or the slot does not exist and `inherit` is `True`. **DO NOT MODIFY THE RETURNED LIST**.

void FrFrame::inheritAll()

void FrFrame::inheritAll(InheritanceType inherit)

void FrSymbol::inheritAll()

void FrSymbol::inheritAll(InheritanceType inherit)

Find all fillers which the given frame can possibly inherit using specified (or the current) inheritance method, and place those fillers directly in the given frame. The parameterless variants use the current inheritance method to retrieve the fillers.

*FrBool FrFrame::isA_p(const FrFrame *possible_parent)*

*FrBool FrSymbol::isA_p(const FrSymbol *possible_parent)*

Determine whether one frame is an ancestor of another by following the `INSTANCE-OF` and `IS-A` links. Returns `True` if `possible_parent` can be reached from `frame` through some chain of `IS-A` links, possibly with an initial `INSTANCE-OF` link.

FrBool FrSymbol::isFrame()

Determine whether the symbol is the name of a frame. If the frame is virtual, it is not loaded into memory if it was not already there.

void FrFrame::markDirty(FrBool dirty = True)

Modify the flag indicating whether the frame needs to be written out to the backing store.

int FrFrame::numberOfSlots()

Determine how many slots are present in the frame.

ObjectType objType()

Determine whether the frame is a regular or virtual frame. Returns `OT_Frame` for regular, in-memory, frames and `OT_VFrame` for virtual frames. Symbols always return `OT_Symbol`, regardless of whether they name a frame.

*char *objTypeName()*

Return a string containing the printable name of the type of frame. Returns "FrFrame" for regular frames and "VFrame" for virtual frames (and "FrSymbol" for symbols regardless of whether they name a frame).

*FrFrame *FrSymbol::oldFrame(int generation)*

If the symbol is the name of a virtual frame with backing store, this function reloads the requested prior version of the frame from the backing store (generation 0 is the most recently saved copy of the frame, generation 1 is the next most-recent copy, generation 2 is the version prior to that, etc.). This function has no effect if there is no backing store for the desired frame.

*FrBool FrFrame::partOf_p(const FrFrame *possible_container)*

*FrBool FrSymbol::partOf_p(const FrSymbol *possible_container)*

Determine whether one entity described by a frame is included as a part of another by following the `PART-OF` link. Returns `True` if `possible_container` can be reached from the given frame through some chain of `PART-OF` links.

ostream &FrFrame::printValue(ostream &output)

Output a printed representation of the frame to the indicated stream.

*void FrFrame::replaceFiller(const FrSymbol *slot, const FrSymbol *facet,*
*const FrObject *old, const FrObject *newfiller)*

*void FrFrame::replaceFiller(const FrSymbol *slot, const FrSymbol *facet,*
*const FrObject *old, const FrObject *newfiller,*
FrCompareFunc cmp)

If the specified filler `old` exists in the indicated slot's facet, remove it and add the new filler. This method has no effect if the old filler is not present in the indicated location.

The first variation requires that the supplied old filler be identical to one in the frame (in practice, this will only be available for symbols and when processing the results of a getFillers); the second variation uses the indicated function to determine whether a filler in the frame is equal to the supplied filler.

*void FrFrame::replaceSem(const FrSymbol *slot,*
*const FrObject *old, const FrObject *newfiller)*

*void FrFrame::replaceSem(const FrSymbol *slot,*
*const FrObject *old, const FrObject *newfiller,*
FrCompareFunc cmp)

If the specified filler `old` exists in the indicated slot's `SEM` facet, remove it and add the new filler. This method has no effect if the old filler is not present in the indicated location.

The first variation requires that the supplied old filler be identical to one in the frame (in practice, this will only be available for symbols and when processing the results of a getFillers); the second variation uses the indicated function to determine whether a filler in the frame is equal to the supplied filler.

*void FrFrame::replaceValue(const FrSymbol *slot,
const FrObject *old, const FrObject *newfiller)*

*void FrFrame::replaceValue(const FrSymbol *slot,
const FrObject *old, const FrObject *newfiller,
FrCompareFunc cmp)*

If the specified filler `old` exists in the indicated slot's `VALUE` facet, remove it and add the new filler. This method has no effect if the old filler is not present in the indicated location.

The first variation requires that the supplied old filler be identical to one in the frame (in practice, this will only be available for symbols and when processing the results of a getFillers); the second variation uses the indicated function to determine whether a filler in the frame is equal to the supplied filler.

7.15.7. Advanced Functions

*FrList *collect_prefix_matching_frames(char *prefix, char *longest, int buflen)*

Return a list of the names of all frames in the current symbol table whose names start with the given prefix. If `longest` is nonzero, also return the longest prefix all of the returned frames have in common, up to a maximum of `buflen`-1 characters (the final character is reserved for the terminating NUL). The returned list has been newly created and must be explicitly freed once no longer required.

For example, if the current symbol table contains frames named ABACUS, COMPUTING, COMPUTER, and COMPUTER-REPAIR, then a search for the prefix "COM" would yield (COMPUTING COMPUTER COMPUTER-REPAIR) and the longest prefix would be "COMPUT".

*char *complete_frame_name(const char *prefix)*

Return a string containing the longest prefix in common among all frames for the current symbol table having the indicated string as the beginning of their names, or 0 if there are no matching frames. The returned string is allocated from the heap, and must be released with `FrFree` (see Section 7.59.3).

*void copy_frame(const FrSymbol *oldframe, const FrSymbol *newframe)*

Make an exact copy of the given old frame into the specified new frame. Links will not be updated, as the purpose of this function is to maintain backup copies of a frame which is changing. Because the links are not updated, the copy of the frame should not be modified, as that will cause internal data to become inconsistent. This function does nothing if `oldframe` is not present in memory already; use `copy_vframe` instead if the frame must be retrieved from backing store.

*void copy_vframe(const FrSymbol *oldframe, const FrSymbol *newframe)*

Make an exact duplicate (except for the frame name) of the given old frame into the specified new frame. Links are not updated, as the purpose of this function is to make backup copies of a frame which is changing. Because the links are not updated, the copy of the frame should not be modified, as that will cause internal data to become inconsistent. Unlike `copy_frame`, this frame will fetch the frame from backing store if it is not currently in memory.

*void define_relation(char *relation, char *inverse)*

*void FrSymbol::defineRelation(const FrSymbol *inverse)*

Define a link and its inverse which FramepaC is to maintain automatically. Whenever a filler which is the name of a frame is added to or deleted from the VALUE facet of a relation slot, FramepaC automatically adds/deletes the current frame to the VALUE facet of the inverse slot in that other frame. Links are symmetrical, so operating on the inverse slot will affect the relation slot in the other frame.

By default, the relations IS-A/SUBCLASSES, INSTANCE-OF/INSTANCES, and PART-OF/HAS-PARTS are defined.

*void undefine_relation(char *relation, char *inverse)*

*void FrSymbol::undefineRelation(const FrSymbol *inverse)*

Specify that FramepaC should no longer maintain the specified link automatically. DO NOT UNDEFINE THE STANDARD RELATIONS (IS-A/SUBCLASSES, INSTANCE-OF/INSTANCES, and PART-OF/HAS-PARTS) or various functions will no longer work correctly because the data consistency necessary to their operation will not be maintained.

*FrSymbol *FrSymbol::inverseRelation()*

Determine the name of the frame link which is the inverse of the one with the indicated name, or 0 if the given name is not that of a relation with an inverse (either predefined or specified with the `define_relation` function). For example,

```
makeSymbol("IS-A")->inverseRelation() == makeSymbol("SUBCLASSES")
```

*FrList *FramepaC_to_FrameKit(FrFrame *frame)*

Return a MAKE-FRAME list corresponding to the given frame. When printed to a file, the returned list can be read in by FrameKit.

*FrFrame *FrameKit_to_FramepaC(FrList *framespec)*

Given a list containing a FrameKit-style MAKE-FRAME or MAKE-FRAME-OLD frame definition, return the FramepaC frame with the equivalent contents.

7.15.8. Virtual Frame Control Functions

int abort_transaction(int transaction_handle)

Indicate that some update of the backing store for the currently active symbol table since the indicated transaction was begun failed, and that all updates in the transaction must be undone. Any subtransactions within the indicated transaction which have not yet been terminated are closed and undone as well. Virtual frames currently in memory will also revert to their state at the beginning of the failed transaction (they may either be restored directly or discarded from memory and reloaded on the next access).

This function has no effect if the current symbol table is not using backing store, and is thus operating only on in-memory frames.

int commit_all_frames()

Force all modified frames to be written out to the backing store. Returns 0 if successful or -1 if any frames could not be successfully committed.

*int commit_frame(FrSymbol *frame)*

Force the indicated frame to be written out to the backing store if it is dirty.

*int discard_frame(FrSymbol *frame)*

Throw out the named frame, forcing it to be re-read from backing store on the next access. This function does nothing when operating without backing store.

void discard_LRU_frames()

Throw out the least-recently used virtual frames in every symbol table, preferentially discarding unmodified frames and writing out any dirty frames it is forced to discard. This function has no effect unless the preprocessor symbol `FrLRU_DISCARD` is defined.

int end_transaction(int transaction_handle)

Indicate that all updates of the backing store for the currently active symbol table since the indicated transaction was begun with `start_transaction` have been successful. Any subtransactions within the indicated transactions which have not yet been ended are also terminated successfully.

This function has no effect if the current symbol table is not using backing store, and is thus operating only on in-memory frames.

*FrBool frame_is_dirty(FrSymbol *frame)*

FrBool FrSymbol::dirtyFrame()

Determine whether the frame has been changed since it was last read into memory from the backing store.

*FrBool frame_locked(const FrSymbol *frame)*

FrBool FrSymbol::isLocked()

Determine whether the indicated frame is currently locked.

*FrFrame *lock_frame(FrSymbol *frname)*

*FrFrame *FrSymbol::lockFrame()*

Ensure that no one else modifies the indicated frame until it is unlocked. This function also forces the frame to be loaded into memory if it was not already there, and prevents it from being swapped out of memory until it is unlocked. The frame should be unlocked as soon as practical, and the number of simultaneously locked frames should be kept to a minimum to avoid interfering with other users. Locking a frame can also be useful in time-critical code, since it allows the use of a pointer to the frame instead of the frame's name, eliminating one level of indirection from each frame access. For example:

```
FrFrame *fr = lock_frame(name) ;
...
add_value(fr,symbolXYZ,filler1) ;
add_value(fr,symbolYZW,filler2) ;
...
unlock_frame(name) ;
```

int start_transaction(int transaction handle)

Indicate that any updates of the backing store for the currently active symbol table from this point forward should be treated as a single atomic operation. That is, when the transaction ends, either all updates will have been successful, or all updates will have been undone (so effectively, they never happened). The transaction continues until either `end_transaction` (successful) or `abort_transaction` (failure, undo changes) is called with the handle returned by this function.

This function has no effect if the current symbol table is not using backing store, and is thus operating only on in-memory frames.

*int synchronize_VFrames(frame_update_hookfunc *update_hook = 0)*

Ensure that the backing store for the current symbol table has an up-to-date copy of all frames in the symbol table. If `update_hook` is nonzero, the function will be called for each frame processed.

*FrFrame *unlock_frame(FrSymbol *frname)*

*FrFrame *FrSymbol::unlockFrame()*

If the named frame is currently locked, unlock it and allow other users access to the frame. This function also allows the frame to be swapped out of memory if it is a virtual frame and some memory must be freed to satisfy another memory allocation request.

7.16. Hash Table Functions

Another data type which is useful on many occasions is the hash table. FramepaC provides the `FrHashTable` class for performing hash table operations with `FrObject`s.

FrHashTable::FrHashTable()

Create a new hash table with the default fill factor (see below).

FrHashTable::FrHashTable(int fill_factor)

Create a new hash table, initially with a maximum size of zero elements. The `fill_factor` specifies the average number of elements per hash bin before the table is automatically expanded. Before any elements are added, you must call either `expand` or `expandTo`.

FrHashEntryType FrHashTable::entryType() const

Determine the type of hash table entry this instance of a hash table expects to store.

void FrHashTable::expand(int increment)

Expand the hash table so that it is able to store an additional `increment` items. All items presently in the hash table are re-hashed.

void FrHashTable::expandTo(int newsize)

Expand the hash table until it is able to store `newsize` items. If the new size is less than or equal to the present size, the hash table will not be altered. If the hash table was expanded, all items presently in the table are re-hashed.

*FrHashEntry *FrHashTable::add(const FrHashEntry *entry)*

Insert a *copy* of the given hash table entry into the hash table. The hash table can hold objects of any subclass of `FrHashEntry`, so a `FrHashTable` can be used to hash any data type. Since a copy is inserted, the provided `entry` may be destroyed or modified after it is added to the hash table.

*int FrHashTable::remove(const FrHashEntry *entry)*

Remove the entry currently in the hash table with the same key as the given entry. Returns 0 if a matching item was removed, -1 if there was no matching item.

*FrHashEntry *FrHashTable::lookup(const FrHashEntry *entry)*

Find the item in the hash table with the same key as the given hash entry. Returns a pointer to the item or 0 if no matching item is present in the hash table.

*FrList *FrHashTable::prefixMatches(const char *prefix) const*

Return a list of the hash keys for the items in the hash table which in some sense match the specified prefix. This function is used in frame name completion, for instance, by returning a list of symbols stored in the table whose names begin with the indicated string. The returned list has been newly created and must be explicitly freed once no longer required.

This function will only return the hash keys for those hash table entries for which the entry's `prefixMatch` method returns `True`, unless the prefix is the empty string, in which case **all** hash keys are returned. The default `FrHashEntryObject` entry type returns `True` if the key is an `FrSymbol` or `FrFrame` and the prefix matches the beginning of the object's printing name, `False` in all other cases.

*char *FrHashTable::completionFor(const char *prefix) const*

Retrieve the longest string starting with `prefix` which all entries in the hash table have in common, or 0 if there are no entries matching `prefix`. As a special case, if `prefix` is 0 (rather than an empty string), the longest common prefix is returned even if the entry would otherwise have indicated that the prefix does not match.

This function will only return those hash table entries for which the entry's `prefixMatch` method returns `True` and `entryName` is not 0. The default `FrHashEntryObject` entry type always returns `False` for `prefixMatch`, and only returns an entry name if the entry's key is a `FrSymbol` or `FrFrame`.

*FrBool FrHashTable::doHashEntries(DoHashEntriesFunc *func, ...)*

Call the indicated function `func` once for each entry in the hash table. The function is called with the hash table entry as its first argument and a variable-argument list of any additional arguments given to `doHashEntries` as its second argument.

As indicated in the descriptions of various `FrHashTable` methods, the elements stored in the hash table are instances of some subclass of `FrHashEntry` (which itself is intended as an abstract base class, since it contains no fields for storing data). The predefined subclass `FrHashEntryObject` will suffices for most uses, since it associates an `FrObject` key with an arbitrary pointer, which may be used to specify another `FrObject` or some non-FramepaC data.

virtual FrBool FrHashEntry::hashp() const

Returns `True` to indicate that this object is associated with a hash table.

virtual FrHashEntryType FrHashEntry::entryType() const

Overridden by subclasses to indicate what type of hash table entry the object represents. This method returns `HE_base` for `FrHashEntry` and `HE_FrObject` for `FrHashEntryObject`.

*virtual FrSymbol *FrHashEntry::entryName() const*

Returns the `FrSymbol` corresponding to the hash key for the hash table entry. `FrHashEntry` always returns `0`; `FrHashEntryObject` returns the key if it is a symbol, the frame name if the key is a frame, or `0` for other object types used as a key.

virtual int FrHashEntry::sizeOf() const

Return the size of the object in bytes.

virtual unsigned int FrHashEntry::hashValue(int size) const

Compute the hash table index for the entry. (used internally)

*virtual int FrHashEntry::keycmp(const FrHashEntry *entry) const*

Determine the ordering relation between the hash keys of the given hash table entry and `entry`. Returns -1 if the given entry orders before `entry`, +1 if after, or 0 if the two entries have equal keys.

*virtual FrBool FrHashEntry::prefixMatch(const char *prefix,int length) const*

Determine whether the hash key or entry name begins with the specified `length`-byte character sequence. Always returns `False` for `FrHashEntry` and `FrHashEntryObject`.

*virtual FrObject *FrHashEntry::copy() const*

Make a duplicate of the specified hash table entry.

FrHashEntryObject()

Construct a new hash table entry whose key and associated data are both `0`.

*FrHashEntryObject(const FrObject *key)*

Construct a new hash table entry with the specified key and a NULL associated data pointer. The key is copied as needed whenever the entry is copied (such as while placing the entry into the hash table).

*FrHashEntryObject(const FrObject *key, void *data)*

Construct a new hash table entry with the specified key and associated data. The key is copied as needed, but the associated data is not (since it could be anything, including non-FramepaC types).

*const FrObject *FrHashEntryObject::getObject()*

Retrieve the key value for the hash table entry.

*void *FrHashEntryObject::getUserData() const*

Retrieve the associated data for the hash table entry. Since the return value is a `void*`, it must be cast appropriately; i.e., if the hash table is constructed using `FrNumber`s as the associated data, one would use

```
FrNumber *num = (FrNumber*)entry->getUserData() ;
```

*void FrHashEntryObject::setUserData(void *data)*

Modify the associated data for the hash table entry. This function merely replaces the pointer stored in the entry; if the previous associated data must be deallocated (to avoid memory leaks), the calling code must do so itself, by first retrieving the previous value with `getUserData()`.

7.17. FrSockStream Functions

FramepaC provides three `iostream` derivative classes which operate over Unix-style network sockets rather than connecting to a file or character device: `FrSockStream`, `FrISockStream`, and `FrOSockStream`.

FrISockStream::FrISockStream(FrSocket s)

Open an input stream which will use the socket `s` as the source from which data is read. For portability, `s` should not be a file descriptor, since only Unix treats file descriptors and socket descriptors interchangeably.

If `s` is `INVALID_SOCKET`, then the constructed stream is in a closed state; you must use `FrISockStream::setSocket` before attempting input from the stream.

FrISockStream::FrISockStream(FrSocket s, int port)

Open a socket listening on `port` for connections from a remote process. If `s` is `INVALID_SOCKET`, then a new socket is created and prepared to listen on the specified port; otherwise, the given socket (assumed to be unbound) is used. To check for connection requests -- and actually start listening for the requests -- use `FrISockStream::awaitConnection`. Prior to the first call to `awaitConnection`, remote processes will receive a "Connection Refused" error because nobody is listening to the port.

*FrISockStream::FrISockStream(const char *hostname, int port)*

Open a connection to the specified port on `hostname`, and set it to read-only mode.

FrSocket FrISockStream::socketNumber() const

Determine the socket descriptor being used by the stream.

FrSocket FrISockStream::connectionDied() const

Determine whether the socket connection is still alive.

FrSocket FrISockStream::inputAvailable() const

Determine whether it is possible to read at least one byte from the stream without blocking on the socket.

*FrSocket FrISockStream::awaitConnection(int timeout, ostream &err, ostream *out = 0)*

Start a stream created with `FrISockStream(socket,port)` listening on its port (if it is not already listening), and await a connection request from a remote process for up to `timeout` seconds. Returns the socket which was created to accept the request, or `INVALID_SOCKET` if no request was pending or arrived during the timeout period.

FrSocket FrISockStream::reset()

FrOSockStream::FrOSockStream(FrSocket s)

*FrOSockStream::FrOSockStream(const char *hostname, int port)*

FrSocket FrOSockStream::socketNumber() const

FrSocket FrOSockStream::connectionDied() const

FrSockStream::FrSockStream(FrSocket s)

*FrSockStream::FrSockStream(const char *hostname, int port)*

Open a read/write stream to the indicated port on host `hostname`. !!!

FrSocket FrSockStream::socketNumber() const

FrSocket FrSockStream::connectionDied() const

FrSocket FrSockStream::inputAvailable() const

*void await_socket_conection(int port, istream *&in, ostream *&out, ostream *&err)*

FrSocket await_socket_conection(int port, int timeout, ostream &err)

FrSocket listening_socket()

*void close_sockets(istream *&in, ostream *&out, ostream *&err)*

*FrSocket connect_to_port(const char *hostname, int port_number)*

int disconnect_port(FrSocket s)

!!!

int input_available(FrSocket s)

Determine whether any input is available for reading from the socket. If this function returns `True`, at least one character may be read from the socket without blocking.

*int await_activity(FrSocket *fds, int numfds, int timeout)*

Pause until any of the `numfds` sockets in the array `fds` have input available or a pending exception condition, or `timeout` seconds have elapsed. Returns `True` if any of the sockets are active, `False` if the timeout elapsed. If `timeout` is 0, the function returns immediately; if `timeout` is negative, it is forced to "forever" (eight hours).

7.18. FrReadTable Functions

Like Lisp, FramepaC uses a read-table which defines the actions to take when specific characters are encountered in the input. One default read table is always defined, and is used by `read_FrObject` and `string_to_FrObject`; additional tables may be created as desired.

FrReadTable::FrReadTable()

Create a new read table. The read table is initialized to a minimal set of functions for both strings and streams: semicolons are treated as comment introducers, valid characters for symbols cause a symbol to be read, and all other characters are treated as whitespace.

FrReadTable::~FrReadTable()

*typedef FrObject *FrReadStringFunc(char *&input, const char *digits)*

This is the type of a function which will be called by the FramepaC reader when reading from a string. The pointer `input` may be advanced to consume any additional input required to read the object; it initially points at the first character of the object, or the first character following a '`#`' which specifies the Lisp macro dispatch function desired. The second argument, `digits`, is 0 unless the function was called for a Lisp macro dispatch, in which case it points at a NUL-terminated string containing the digits of any infix numeric argument following the '#' of the macro dispatch.

*typedef FrObject *FrReadStreamFunc(istream &input, const char *digits)*

This is the type of a function which will be called by the FramepaC reader when reading from a stream. The stream `input` may be read as needed to get any additional input required to read the object; it is initially located at the first character of the object, or the first character following a '#' which specifies the Lisp macro dispatch function desired. The second argument, `digits`, is 0 unless the function was called for a Lisp macro dispatch, in which case it points at a NUL-terminated string containing the digits of any infix numeric argument following the '#' of the macro dispatch.

*void FrReadTable::setReader(char c, FrReadStringFunc *)*

*void FrReadTable::setReader(char c, FrReadStreamFunc *)*

*void FrReadTable::setReader(char c, FrReadStringFunc *, FrReadStreamFunc)*

Set the function to be called when the character `c` is encountered while reading an object from a string, a stream, or either (respectively). The application should normally set both types of read function to perform equivalent actions to avoid inconsistencies between reads from a string and reads from a stream.

*FrReadStringFunc *FrReadTable::getStringReader(char c)*

Determine the function which will be called when character `c` is encountered while reading from a string.

*FrReadStreamFunc *FrReadTable::getStreamReader(char c)*

Determine the function which will be called when character `c` is encountered while reading from a stream.

*FrObject *FrReadTable::readString(char *&input)*

Read the given string and convert the first object found. This method returns the symbol `*EOF*` if the end of the string is encountered before any objects, and advances the pointer `input` to point at the string's terminating NUL or to the character immediately following the last one forming a part of representation of the returned object.

*FrObject *FrReadTable::readStream(istream &input)*

Read the given stream and convert the first object found. This method returns the symbol `*EOF*` if an end-of-file condition is encountered before any objects, and advances the location of `input` to the character immediately following the last one forming a part of representation of the returned object.

7.19. FrReader Functions

Global instances of the `FrReader` class are used to extend the default read table at compile time. The constructor for `FrReader` updates the default read table as specified by its arguments when the program's global initialization is performed, before `main` is executed.

An advantage of this type of read-table setup is that only those classes which are actually used by the program need to be linked in, which considerably reduces code bloat.

*FrReader::FrReader(FrReadStringFunc *stringfunc, FrReadStreamFunc *streamfunc,*
*int leadin_char, char *leadin_string)*

Add the two read functions specified to the default FramepaC read table as follows:

- If `leadin_char` is `FrREADER_LEADIN_LISPFORM`, this instance defines a Lisp macro-dispatcher string `leadin_string`, which (unlike Lisp) may consist of multiple characters (unless it begins with 'R' or 'X'). Thus, `FrReader(f1,f2,FrREADER_LEADIN_LISPFORM,"Example")` defines the functions to be called when `#Example` is encountered. `leadin_string` is not case-sensitive.
- If `leadin_char` is `FrREADER_LEADIN_CHARSET`, this instance defines the read functions for each and every character contained in `leadin_string`. Thus, `FrReader(f1,f2,FrREADER_LEADIN_CHARSET,"ABCde")` defines the functions to be called when 'A', 'B', 'C', 'd', or 'e' are encountered (in this case, `leadin_string` is case-sensitive).
- If `leadin_char` is any other character, `leadin_string` is ignored and this instance defines the read functions for the specified character.

FrReader::~FrReader()

Destroy the instance of `FrReader`. Destroying a `FrReader` does **not** restore the entries in the read table which it overrode when it was constructed, but will remove a Lisp macro-dispatcher definition if originally constructed with `FrREADER_LEADIN_LISPFORM`.

*const char *FrReader::lispFormName() const*

Retrieve the Lisp macro-dispatcher lead-in string defined if this instance was constructed with `FrREADER_LEADIN_LISPFORM`, 0 otherwise.

*FrObject *FrReader::readString(char *&input, const char *digits)*

Read an object from the given input string if this instance was constructed with `FrREADER_LEADIN_LISPFORM`; returns 0 otherwise. This method is called internally by the FramepaC reader's Lisp macro-dispatcher function.

*FrObject *FrReader::readStream(istream &input, const char *digits)*

Read an object from the given input stream if this instance was constructed with `FrREADER_LEADIN_LISPFORM`; returns 0 otherwise. This method is called internally by the FramepaC reader's Lisp macro-dispatcher function.

7.20. Demon Functions

It is possible to have an arbitrary function invoked automatically whenever a certain action is performed. Once installed, such a demon function will be invoked again and again until explicitly removed, allowing it to monitor (and even affect) operations on frames.

The functions described in this section are optional, and will only be available if the preprocessor macro `FrDEMONS` is defined. Your installation may have disabled demons because of the additional overhead required to support demons even when none are active.

Five types of action may be monitored through the use of demons. These actions are creating a slot or facet, adding a filler, removing a filler, retrieving a value from a frame, and a missing filler triggering inheritance.

Having a demon for missing fillers can be especially useful, because the demon can add values to the frame before inheritance is attempted. If the demon adds fillers to the facet for which it was triggered, FramepaC will use those fillers instead of performing inheritance.

!!!

*FrBool add_demon(FrSymbol *slotname, DemonType type, DemonFunc *func, va_list args = 0)*

Insert a new demon of the specified type to be called whenever the indicated action is performed on a slot of the given name. For example,

```
add_demon(makeSymbol("TEST"),DT_IfAdded,added_func,0) ;
```

would call `added_func` any time a filler is added to any slot named `TEST`. Note that the installed demon function only affects frames created while the current symbol table is active; thus, each symbol table has an independent set of demons.

*FrBool remove_demon(FrSymbol *slotname, DemonType type, DemonFunc *func)*

Remove the specified demon from the list of demons to be invoked on the indicated action on any slot with the given name.

*FrBool FrSymbol::addDemon(DemonType type, DemonFunc *func, va_list args = 0)*

Insert a new demon of the specified type to be called whenever the indicated action is performed on a slot of the given name. Note that the installed demon function only affects frames created while the current symbol table is active; thus, each symbol table has an independent set of demons.

*FrBool FrSymbol::removeDemon(DemonType type, DemonFunc *func)*

Remove the specified demon from the list of demons to be invoked on the indicated action on any slot with the given name.

7.21. Configuration-File Functions

FramepaC provides the necessary mechanisms to read and interpret configuration files which affect the operation of the program. The basic method for implementing a custom configuration file is to derive a subclass of `FrConfiguration` with one or more `FrConfigurationTable`s defining the actual keywords and data types present in the configuration file.

7.21.1. FrConfiguration

The `FrConfiguration` class is an abstract type from which you can easily derive classes to parse configuration files containing default settings for your program.

In order to create a tailored configuration-file parser, you must derive a subclass from `FrConfiguration` which overrides at least the methods `init` and `resetState`. The former method clears all data members added by the derived class to their default values, and the latter indicates which `FrConfigurationTable` to use when parsing begins (the configuration table itself can cause a switch to another table, creating a state machine). The derived class may optionally override `maxline` and `dump`, and may add additional parsing functions.

FrConfiguration::FrConfiguration()

Derived classes will normally not require an explicit constructor unless you wish to e.g. combine the construction and `load` operations in one.

FrConfiguration::~FrConfiguration()

If a derived class contains any `char*` or `FrList*` members, it will require a destructor which frees those members to avoid a memory leak.

virtual void FrConfiguration::init()

This method must be overridden by any derived classes, and must reset all added data members to their default values (e.g. setting all `char*` and `FrList*` to 0).

virtual void FrConfiguration::resetState()

This method must be overridden by any derived classes, and must set the `curr_state` pointer to the `FrConfigurationTable` on which the configuration-file parsing is to begin. The configuration table may indicate other configuration tables to which the parsing should switch on encountering particular keywords, allowing you to create a state machine whose initial state is the one indicated by `resetState`.

virtual size_t FrConfiguration::maxline() const

By default, `FrConfiguration` allows individual lines to be up to 2048 characters in length, but permits logical lines to continue across several lines in the file. By overriding this method, a derived class may change the maximum length of an individual line (though logical lines remain unlimited due to continuations).

*FrBool load(const char *filename, FrBool reset = True)*

If the indicated file exists and is readable, it is opened and parsed, with the results of the parsing placed into the object's data members. This method returns `True` if the file was successfully read, `False` otherwise.

FrBool load(istream &instream, FrBool reset = True)

The indicated stream is parsed, with the results of the parsing placed into the object's data members. This method returns `True` if the file was successfully read, `False` otherwise.

FrBool good() const

Determine whether the contents of the object are valid, i.e. has a configuration file been parsed into its data members?

int lineNumber() const

Return the current line number within the configuration file. This method is intended for use in an overloaded `warn` method to report where in the file an error occurred.

*ostream &dumpFlags(long flags, FrCommandBit *bits, ostream &output) const*

Convert the specified `flags` (as created by the `bitflags` method) back into a list of the flag names written to the specified stream.

virtual ostream &dump(ostream &out) const

Print the parsed contents of the configuration file to the specified stream. This method must be overloaded by subclasses of `FrConfiguration` if you wish to have a dump of the configuration for debugging purposes.

The following functions may be referenced from the `FrConfigurationTable`(s) for a derived class:

*FrBool integer(const char *line, void *location, void *extra)*

The data value for a keyword is expected to be an integer, which will be converted into a `long int` stored at the address pointed at by `location`.

*FrBool real(const char *line, void *location, void *extra)*

The data value for the associated keyword is expected to be a floating-point number, which will be converted into a `double` stored at the address pointed at by `location`.

*FrBool basedir(const char *line, void *location, void *extra)*

The data value for the associated keyword is expected to be a quoted string, which will be converted into a `char*` and stored internally, as well asoptionally stored at `location` (which, unlike for any other conversion function described here, may be `0`). The base directory specified by the keyword's data value is used internally to expand the "+" convention for the `filename` conversion function.

If `location` is non-NULL, the string pointed at by the `char*` must be freed with `FrFree`, usually in the configuration object's destructor.

*FrBool filename(const char *line, void *location, void *extra)*

The data value for the associated keyword is expected to be a quoted string, which will be converted into a `char*` stored at `location`. Should the string begin with a plus sign ("+"), the plus sign is replaced with the entire contents of the base directory string specified with a keyword using the **basedir** conversion function. Since the configuration file is only read once, a single keyword with the `basedir` conversion may be specified multiple times, changing the value of the "+" expansion each time.

The string pointed at by the `char*` must be freed with `FrFree`, usually in the configuration object's destructor.

*FrBool string(const char *line, void *location, void *extra)*

The data value for a keyword is expected to be a quoted string, which will be converted into a `char*` stored at the address pointed at by `location`. The string pointed at by the `char*` must be freed with `FrFree`, usually in the configuration object's destructor.

*FrBool bitflags(const char *line, void *location, void *extra)*

The data value for the associated keyword is assumed to be a list of symbols from the array of `FrCommandBit`s pointed at by `extra`. The result placed in `location` is an `long`, each of whose bits corresponds to the presence or absence of one of the symbols specified in the `FrCommandBit` array.

*FrBool list(const char *line, void *location, void *extra)*

The data value for the associated keyword is assumed to be a list of arbitrary values. !!!

*FrBool assoclist(const char *line, void *location, void *extra)*

The data value for the associated keyword is assumed to be a list of lists, where the first element of each sublist is used as a key value. !!!

*FrBool symbol(const char *line, void *location, void *extra)*

The data value for a keyword is expected to be a symbol, which will be converted into a `FrSymbol*` stored at the address pointed at by `location`.

*FrBool symlist(const char *line, void *location, void *extra)*

The data value for the associated keyword is expected to be a list of symbols. !!!

*FrBool yesno(const char *line, void *location, void *extra)*

Sets a variable of type `FrBool` at `location` to indicate whether the setting is affirmative or negative. If the first non-whitespace character following the keyword is 'Y', 'y', 'T', 't', '1', or '+', the variable is set to `True`; if the character is 'N', 'n', 'F', 'f', '0', or '-', the variable is set to `False`; otherwise, an error message is displayed.

*FrBool invalid(const char *line, void *location, void *extra)*

Warn the user that an invalid keyword was encountered. Always returns `True`.

7.21.2. FrCommandBit

The `FrCommandBit` structure is used to define bit flags for the `FrConfiguration` class. It consists of two data members: a `char*` for the name, which is used to read the list of flags and in writing them out again with `FrConfiguration::dumpFlags`, and a `long` containing the bit mask for the associated flag. *Exactly one* bit should be set in each bit mask.

`FrCommandBit` structures are always used in an array. The terminating element of the array is a `FrCommandBit` with both name and bit mask set to 0.

A sample command-bit array might be:

```
static FrCommandBit options[] =
   {
     { "VERBOSE",       0x00000001 },
     { "SHORTCUTS",     0x00000002 },
     { "NETWORK_MODE",  0x00000004 },
     { 0,               0 }
   } ;
```

7.22. Widget class FrWidget

This is the base class for all of the remaining `FrW*` classes described in the following sections. It is basically a wrapper for the X-Windows `Widget` object.

!!!

FrWidget::FrWidget(Widget w)

Create a FramepaC widget object from the indicated X-Windows widget.

FrWidget::FrWidget(Widget w, FrBool m)

virtual FrWidget::~FrWidget()

virtual FrObjectType FrWidget::objType() const

*virtual const char *FrWidget::objTypeName() const*

virtual FrObjectType FrWidget::objSuperclass() const *virtual ostream &FrWidget::printValue(ostream &output) const*

*virtual char *FrWidget::displayValue(char *buffer) const*

virtual size_t FrWidget::displayLength() const

virtual FrBool FrWidget::widgetp() const

*Widget FrWidget::operator * () const*

void FrWidget::destroy()

void FrWidget::keep()

void FrWidget::retain()

void FrWidget::manage()

void FrWidget::unmanage()

FrBool FrWidget::isManaged() const

void FrWidget::setManaged(FrBool m)

void FrWidget::map()

void FrWidget::unmap()

FrBool FrWidget::isMapped() const

void FrWidget::mapWhenManaged(FrBool do_map) const

void FrWidget::setValue(XtPointer value) const

void FrWidget::setUserData(XtPointer data) const

XtPointer FrWidget::getValue() const

XtPointer FrWidget::getUserData() const

*char *FrWidget::getLabel() const*

int FrWidget::getWidth(FrBool include_border = False) const

int FrWidget::getHeight(FrBool include_border = False) const

*void FrWidget::getSize(int *width, int *height, FrBool include_border = False) const* *void FrWidget::getPosition(int *x, int *y) const*

void FrWidget::setSensitive(FrBool sensitive) const

*void FrWidget::setLabel(const char *label) const*

*void FrWidget::setTextString(const char *text) const*

void FrWidget::setAlignment(int align) const

void FrWidget::setEnds(Widget from, Widget to) const

void FrWidget::setOrientation(int orientation) const

void FrWidget::setVertical(FrBool vertical) const

void FrWidget::setShadow(int thickness) const

*void FrWidget::setTitle(const char *title) const*

*void FrWidget::setIconName(const char *title) const*

void FrWidget::setPosition(int x, int y) const

void FrWidget::setWidth(int width) const

void FrWidget::setHeight(int height) const

void FrWidget::allowResize(FrBool state) const

void FrWidget::invertVideo() const

void FrWidget::setTraversal(FrBool state) const

void FrWidget::traverseCurrent() const

void FrWidget::traverseNext() const

void FrWidget::traversePrev() const

void FrWidget::traverseNextGroup() const

void FrWidget::traversePrevGroup() const

void FrWidget::traverseUp() const

void FrWidget::traverseDown() const

void FrWidget::navigationTabGroup() const

*void FrWidget::removeCallbacks(char *type) const*

*void FrWidget::addCallback(char *type, XtCallbackProc cb_func, XtPointer cb_data)*

*void FrWidget::removeCallback(char *type, XtCallbackProc cb_func, XtPointer cb_data) const*

void FrWidget::armCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::disarmCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::activateCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::activateCallback(XtCallbackProc cb_func)

void FrWidget::okCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::cancelCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::helpCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::nomatchCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old= False)

void FrWidget::changedCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::verifyCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::focusCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::loseFocusCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::resizeCallback(XtCallbackProc cb_func, XtPointer cb_data, FrBool remove_old = False)

void FrWidget::destroyCallback(XtCallbackProc cb_func, FrBool remove_old = False)

void FrWidget::destroyCallback(FrBool remove_old = False)

*void FrWidget::setFont(const char *fontname)*

void FrWidget::attach(Widget top,Widget bottom,Widget left,Widget right)

*void FrWidget::attach(FrWidget *top,FrWidget *bottom,FrWidget *left,*
*FrWidget *right)*

void FrWidget::attachOpposite(Widget top,Widget bottom,Widget left,Widget right)

*void FrWidget::attachOpposite(FrWidget *top,FrWidget *bottom,FrWidget *left,* *FrWidget *right)*

void FrWidget::attachOffsets(int top, int bottom, int left, int right)

void FrWidget::attachPosition(int top, int bottom, int left, int right)

void FrWidget::detach(FrBool top, FrBool bottom, FrBool left, FrBool right)

void FrWidget::warpPointer(int x, int y) const

void FrWidget::warpPointerCenter() const

*void FrWidget::getPointer(int *x, int *y) const*

Widget FrWidget::parentWidget() const

Widget FrWidget::grandparent() const

7.23. Widget class FrWArrow

This class encapsulates the arrow-head widget, used in scrollbars and similar visual elements on the screen.

In addition to the methods inherited from `FrWidget` (not all of which apply to arrows), `FrWArrow` provides the following methods of its own:

FrWArrow::FrWArrow(Widget parent, ArrowDirection direction)

!!!

*FrWArrow::FrWArrow(FrWidget *parent, ArrowDirection direction)*

!!!

7.24. Widget class FrWArrowG

This class encaspulates the arrow-head gadget, which is logically equivalent to the arrow-head widget for `FrWArrow`, but slightly more restricted due to the limitations of X-Windows gadgets vs. widgets.

In addition to the methods inherited from `FrWidget` (not all of which apply to arrows) and `FrWArrow`, `FrWArrowG` provides the following methods of its own:

FrWArrowG::FrWArrowG(Widget parent, ArrowDirection direction)

!!!

*FrWArrowG::FrWArrowG(FrWidget *parent, ArrowDirection direction)*

!!!

7.25. Widget class FrWButtonBar

This class encapsulates a linear bar containing a series of buttons on which the user may click to select various actions.

In addition to the methods inherited from `FrWidget` (not all of which apply to button bars), `FrWButtonBar` provides the following methods of its own:

*FrWButtonBar::FrWButtonBar(Widget parent, const FrButtonsAndCommands *buttons,*
int bcount, FrBool managed, XtPointer default_data)

*FrWButtonBar::FrWButtonBar(FrWidget *parent, const FrButtonsAndCommands *buttons,*
int bcount, FrBool managed, XtPointer default_data)

virtual FrWButtonBar::~FrWButtonBar()

*FrWPushButton *FrWButtonBar::nthButton(int n)*

7.26. Widget class FrWDialogPopup

This class encapsulates a popup window containing a user dialog.

In addition to the methods inherited from `FrWidget` (not all of which apply to popups), `FrWDialogPopup` provides the following methods of its own:

FrWDialogPopup::FrWDialogPopup(Widget w)

*FrWDialogPopup::FrWDialogPopup(Widget parent, const char *name, const char *title = 0,*
*const char *icon = 0, FrBool auto_unmanage = False)*

*FrWDialogPopup::FrWDialogPopup(FrWidget *parent, const char *name, const char *title = 0,*
*const char *icon = 0, FrBool auto_unmanage = False)*

7.27. Widget class FrWForm

This class encapsulates a window containing a form.

In addition to the methods inherited from `FrWidget` (not all of which apply to forms), `FrWForm` provides the following methods of its own:

*FrWForm::FrWForm(Widget parent, const char *name, FrBool managed = False,*
*const char *icon_name = 0)*

*FrWForm::FrWForm(FrWidget *parent, const char *name, FrBool managed = False,*
*const char *icon_name = 0)*

7.28. Widget class FrWFrameCompleter

This class encapsulates a fill-in text field for which the TAB key performs frame-name completion. On pressing TAB when at least two characters of a frame's name have been entered, as many additional characters as are uniquely identified by the already-entered prefix are inserted into the fill-in field, and if the result is not a complete frame name, a completion dialog is popped up. The dialog contains either a list of all valid completions or a list of valid continuations (if there are too many matching names).

In addition to the methods inherited from `FrWidget` (not all of which apply to completion fields), `FrWFrameCompleter` provides the following methods of its own:

*FrWFrameCompleter::FrWFrameCompleter(Widget parent, const char *initial,*
int columns=30, XtCallbackProc done_cb=0, XtPointer done_data=0)

Create a frame-completion text fill-in field as a child of the indicated widget, setting its initial contents to `initial` and its width to `columns` characters. If `done_cb` is not NULL, it will be called with `done_data` when the user accepts the input value in the text field.

*FrWFrameCompleter::FrWFrameCompleter(FrWidget *parent, const char *initial,*
int columns=30, XtCallbackProc done_cb=0, XtPointer done_data=0)

Create a frame-completion text fill-in field as a child of the indicated `FrWidget`, setting its initial contents to `initial` and its width to `columns` characters. If `done_cb` is not NULL, it will be called with `done_data` when the user accepts the input value in the text field.

7.29. Widget class FrWFramePrompt

This class encapsulates a popup dialog containing a fill-in text field for which the TAB key performs frame-name completion as described for `FrWFrameCompleter`.

In addition to the methods inherited from `FrWidget` (not all of which apply to prompt popups), `FrWFramePrompt` provides the following methods of its own:

*FrWFramePrompt::FrWFramePrompt(Widget parent, const char *label, const char *def,*
XtCallbackProc ok_cb = 0, XtPointer ok_data = 0,
XtPointer help_data = 0,
FrBool auto_unmanage = True, FrBool modal = False,
FrBool managed = True)

*FrWFramePrompt::FrWFramePrompt(FrWidget *parent, const char *label, const char *def,*
XtCallbackProc ok_cb = 0, XtPointer ok_data = 0,
XtPointer help_data = 0,
FrBool auto_unmanage = True, FrBool modal = False,
FrBool managed = True)

*void FrWFramePrompt::setFrameName(_FrSymbol *framename) const*

7.30. Widget class FrWFrame

This class encapsulates a window-frame object which may be placed around another window object.

In addition to the methods inherited from `FrWidget` (not all of which apply to window frames), `FrWFrame` provides the following methods of its own:

FrWFrame::FrWFrame(Widget parent)

*FrWFrame::FrWFrame(FrWidget *parent)*

7.31. Widget class FrWLabel

This class encapsulates a Motif textual-label widget.

In addition to the methods inherited from `FrWidget` (not all of which apply to text labels), `FrWLabel` provides the following methods of its own:

*FrWLabel::FrWLabel(Widget parent, const char *label, FrBool managed = True)*

Create a new label as a child of the indicated widget, with the given text string as the contents of the new label. If `managed` is `False`, the new label widget is left unmanaged, and will not be displayed until explicitly managed.

*FrWLabel::FrWLabel(FrWidget *parent, const char *label, FrBool managed = True)*

Create a new label as a child of the indicated `FrWidget`, with the given text string as the contents of the new label. If `managed` is `False`, the new label widget is left unmanaged, and will not be displayed until explicitly managed.

7.32. Widget class FrWList

This class encapsulates a Motif list box.

In addition to the methods inherited from `FrWidget` (not all of which apply to list boxes), `FrWList` provides the following methods of its own:

*FrWList::FrWList(Widget parent, const char *name, int visible, int max = 0)*

Create a new list box with the indicated name as a child of the given widget, showing a maximum of `visible` items on-screen at a time (allowing scrolling to any additional items). If `max` is non-zero, then the total number of items in the list box will be limited to `max` items at any time by removing the top-most item each time a new item is added at the bottom of the list.

*FrWList::FrWList(FrWidget *parent, const char *name, int visible, int max = 0)*

Create a new list box with the indicated name as a child of the given `FrWidget`, showing a maximum of `visible` items on-screen at a time (allowing scrolling to any additional items). If `max` is non-zero, then the total number of items in the list box will be limited to `max` items at any time by removing the top-most item each time a new item is added at the bottom of the list.

*void FrWList::add(const char *item)*

Add the indicated string as a new item at the bottom of the item list already contained by the list box. If the maximum number of items was limited at the time the list box was constructed and the list already contained the maximum number of items, the top-most item will be removed.

int FrWList::totalItems() const

Determine how many items are currently stored in the list box.

7.33. Widget class FrWMainWindow

This class encapsulates a Motif application's main window.

In addition to the methods inherited from `FrWidget` (not all of which apply to main windows), `FrWMainWindow` provides the following methods of its own:

*FrWMainWindow::FrWMainWindow(Widget parent, const char *name, const char *title,*
*const char *icon = 0,*
int height = 0, int width = 0, FrBool managed = False,
FrBool show_sep = False)

*FrWMainWindow::FrWMainWindow(FrWidget *parent, const char *name, const char *title,*
*const char *icon = 0,*
int height = 0, int width = 0, FrBool managed = False,
FrBool show_sep = False)

void FrWMainWindow::setWorkWindow(Widget) const

void FrWMainWindow::setMenuBar(Widget) const

7.34. Widget class FrWOptionMenu

In addition to the methods inherited from `FrWidget` (not all of which apply to option menus), `FrWOptionMenu` provides the following methods of its own:

*FrWOptionMenu::FrWOptionMenu(Widget parent, const char *name, FrList *options,*
*FrObject *def_option = 0, FrBool managed = False)*

*FrWOptionMenu::FrWOptionMenu(FrWidget *parent, const char *name, FrList *options,*
*FrObject *def_option = 0, FrBool managed = False)*

virtual FrWOptionMenu::~FrWOptionMenu()

*FrObject *FrWOptionMenu::currentSelection() const*

*void FrWOptionMenu::setSelection(const char *selection_text)*

void FrWOptionMenu::changeCallback(XtCallbackProc cb_func, XtPointer cb_data,
FrBool remove_old = False)

void FrWOptionMenu::changeCallback(XtCallbackProc cb_func)

void FrWOptionMenu::callChangeCallback(XtPointer call_data) const

7.35. Widget class FrWPopupMenu

In addition to the methods inherited from `FrWidget` (not all of which apply to popup menus), `FrWPopupMenu` provides the following methods of its own:

*FrWPopupMenu::FrWPopupMenu(Widget parent, const char *name, FrBool managed = False)*

*FrWPopupMenu::FrWPopupMenu(FrWidget *parent, const char *name, FrBool managed = False)*

virtual void FrWPopupMenu::popup()

void FrWPopupMenu::menuPosition(XtPointer call_data)

7.36. Widget class FrWProgressIndicator

This class encapsulates a progress indicator in which a horizontal bar progressively changes color as some "lengthy" process is performed.

In addition to the methods inherited from `FrWidget` (not all of which apply to progress indicators), `FrWProgressIndicator` provides the following methods of its own:

FrWProgressIndicator::FrWProgressIndicator(Widget parent)

FrWProgressIndicator::FrWProgressIndicator(Widget parent, FrBool managed,
*const char *piclass = 0)*

*FrWProgressIndicator::FrWProgressIndicator(FrWidget *parent, FrBool managed = True,*
*const char *piclass = 0)*

void FrWProgressIndicator::setProgress(int percentage) const

int FrWProgressIndicator::getProgress() const

7.37. Widget class FrWProgressPopup

This class encapsulates a progress indicator as described under `FrWProgressIndicator` within its own popup window.

In addition to the methods inherited from `FrWidget` (not all of which apply to progress indicators), `FrWProgressPopup` provides the following methods of its own:

*FrWProgressPopup::FrWProgressPopup(Widget parent, const char *label,*
*const char *ppclass = 0)*

*FrWProgressPopup::FrWProgressPopup(FrWidget *parent, const char *label,*
*const char *ppclass = 0)*

virtual FrWProgressPopup::~FrWProgressPopup()

7.38. Widget class FrWPromptPopup

This class encapsulates a popup window containing a user prompt dialog.

In addition to the methods inherited from `FrWidget` (not all of which apply to popups), `FrWPromptPopup` provides the following methods of its own:

FrWPromptPopup::FrWPromptPopup(Widget w)

*FrWPromptPopup::FrWPromptPopup(Widget parent, const char *label, const char *def,*
XtCallbackProc ok_cb = 0, XtPointer ok_data = 0,
XtPointer help_data = 0,
FrBool auto_unmanage = True, FrBool modal = False,
FrBool managed = True)

*FrWPromptPopup::FrWPromptPopup(FrWidget *parent, const char *label, const char *def,*
XtCallbackProc ok_cb = 0, XtPointer ok_data = 0,
XtPointer help_data = 0,
FrBool auto_unmanage = True, FrBool modal = False,
FrBool managed = True)

virtual void FrWPromptPopup::popup()

7.39. Widget class FrWPulldownMenu

In addition to the methods inherited from `FrWidget` (not all of which apply to pulldown menus), `FrWPulldownMenu` provides the following methods of its own:

*FrWPulldownMenu::FrWPulldownMenu(Widget parent, const char *name, FrBool managed = False)*

*FrWPulldownMenu::FrWPulldownMenu(FrWidget *parent, const char *name, FrBool managed=False)*

virtual void FrWPulldownMenu::popup()

void FrWPulldownMenupulldown(int button)

7.40. Widget class FrWPushButton

In addition to the methods inherited from `FrWidget` (not all of which apply to pushbuttons), `FrWPushButton` provides the following methods of its own:

FrWPushButton::FrWPushButton(Widget button)

*FrWPushButton::FrWPushButton(Widget parent, const char *label, FrBool managed = True,*
*FrBool centered = True, const char *pbclass = 0)*

*FrWPushButton::FrWPushButton(FrWidget *parent, const char *label, FrBool managed = True,*
*FrBool centered = True, const char *pbclass = 0)*

void FrWPushButton::setAlignment(int align) const

void FrWPushButton::setMnemonic(char mnem) const

void FrWPushButton::setDualMouseButton() const

void FrWPushButton::setTripleMouseButton() const

static int FrWPushButton::pressedButton(XtPointer call_data)

7.41. Widget class FrWPushButtonG

In addition to the methods inherited from `FrWidget` (not all of which apply to pushbuttons) and from `FrWPushbutton`, `FrWPushButtonG` provides the following methods of its own:

*FrWPushButtonG::FrWPushButtonG(Widget parent, const char *label, FrBool managed = True,
FrBool centered = True, const char *pbclass = 0)*

*FrWPushButtonG::FrWPushButtonG(FrWidget *parent, const char *label, FrBool managed = True,
FrBool centered = True, const char *pbclass = 0)*

7.42. Widget class FrWCascadeButton

In addition to the methods inherited from `FrWidget` (not all of which apply to buttons), `FrWCascadeButton` provides the following methods of its own:

*FrWCascadeButton::FrWCascadeButton(Widget parent, const char *label, char mnemonic = '\0',
Widget submenuID = 0)*

*FrWCascadeButton::FrWCascadeButton(FrWidget *parent, const char *label,
char mnemonic = '\0', FrWidget *submenuID = 0)*

7.43. Widget class FrWRadioBox

In addition to the methods inherited from `FrWidget` (not all of which apply to radio boxes), `FrWRadioBox` provides the following methods of its own:

*FrWRadioBox::FrWRadioBox(Widget parent, const char *title, const char *name = 0,
FrBool vertical = True, FrBool managed = True)*

*FrWRadioBox::FrWRadioBox(FrWidget *parent, const char *title, const char *name = 0,
FrBool vertical = True, FrBool managed = True)*

7.44. Widget class FrWRowColumn

In addition to the methods inherited from `FrWidget` (not all of which apply to matrix objects), `FrWRowColumn` provides the following methods of its own:

*FrWRowColumn::FrWRowColumn(Widget parent, const char *name, FrBool vertical=False)*

*FrWRowColumn::FrWRowColumn(FrWidget *parent, const char *name, FrBool vertical=False)*

void FrWRowColumn::setColumns(int columns) const

void FrWRowColumn::setPacked(int packed) const

void FrWRowColumn::setAlignment(int align) const

7.45. Widget class FrWScrollBar

In addition to the methods inherited from `FrWidget` (not all of which apply to scroll bars), `FrWScrollBar` provides the following methods of its own:

FrWScrollBar::FrWScrollBar(Widget scrollbar)

*FrWScrollBar::FrWScrollBar(Widget parent, FrBool vertical, int limit = 100,
FrBool managed = True, const char *sbclass = 0)*

*FrWScrollBar::FrWScrollBar(FrWidget *parent, FrBool vertical, int limit = 100,
FrBool managed = True, const char *sbclass = 0)*

void FrWScrollBar::setLimit(int limit) const

int FrWScrollBar::getLimit() const

void FrWScrollBar::setStart(int start) const

void FrWScrollBar::setLength(int length) const

void FrWScrollBar::setThumb(int start, int length) const

*void FrWScrollBar::getThumb(int *start, int *length) const*

7.46. Widget class FrWScrollWindow

In addition to the methods inherited from `FrWidget` (not all of which apply to scrollable windows), `FrWScrollWindow` provides the following methods of its own:

FrWScrollWindow::FrWScrollWindow(Widget w)

*FrWScrollWindow::FrWScrollWindow(Widget parent, const char *title,
FrBool autoscroll = True, FrBool managed = True)*

*FrWScrollWindow::FrWScrollWindow(FrWidget *parent, const char *title,
FrBool autoscroll = True, FrBool managed = True)*

*FrWidget *FrWScrollWindow::workWindow() const*

void FrWScrollWindow::scrollBorder(int width) const

void FrWScrollWindow::forceScrollBar(FrBool force) const

void FrWScrollWindow::scrollTo(Widget w, int horz_margin, int vert_margin) const

*void FrWScrollWindow::scrollTo(FrWidget *w, int horz_margin, int vert_margin) const*

Widget FrWScrollWindow::scrollBar(FrBool vertical) const

7.47. Widget class FrWSelectionBox

In addition to the methods inherited from `FrWidget` (not all of which apply to selection boxes), `FrWSelectionBox` provides the following methods of its own:

*FrWSelectionBox::FrWSelectionBox(Widget parent, FrList *items, char *label,*
XtCallbackProc ok_cb, XtPointer ok_data,
XtCallbackProc nomatch_cb,XtPointer nomatch_data,
XtCallbackProc apply_cb, XtPointer apply_data,
*const char **helptexts = 0,*
FrBool must_match = True,int visitems = 10,
FrBool managed = True)

*FrWSelectionBox::FrWSelectionBox(FrWidget *parent, FrList *items, char *label,*
XtCallbackProc ok_cb, XtPointer ok_data,
XtCallbackProc nomatch_cb,XtPointer nomatch_data,
XtCallbackProc apply_cb, XtPointer apply_data,
*const char **helptexts = 0,*
FrBool must_match = True,int visitems = 10,
FrBool managed = True)

void FrWSelectionBox::unmanageChild(FrWSelBoxChild child) const

!!!

*void FrWSelectionBox::buttonLable(FrWSelBoxChild button, const char *label) const*

!!!

*char *FrWSelectionBox::selectionText(XtPointer call_data)*

!!!

*FrObject *FrWSelectionBox::selectionFrObject(XtPointer call_data)*

!!!

7.48. Widget class FrWSeparator

In addition to the methods inherited from `FrWidget` (not all of which apply to separators), `FrWSeparator` provides the following methods of its own:

FrWSeparator::FrWSeparator(Widget parent, int linestyle = -1)

!!!

*FrWSeparator::FrWSeparator(FrWidget *parent, int linestyle = -1)*

!!!

7.49. Widget class FrWShadowText

In addition to the methods inherited from `FrWidget` (not all of which apply to shadowed-text fields), `FrWShadowText` provides the following methods of its own:

FrWShadowText::FrWShadowText(Widget w)

!!!

*FrWShadowText::FrWShadowText(Widget parent, const char *text, int columns = 30,
int rows = 1, FrBool editable = True,
FrBool traversal = True, FrBool wrap = True,
const char *name = 0)*

*FrWShadowText::FrWShadowText(FrWidget *parent, const char *text, int columns = 30,
int rows = 1, FrBool editable = True,
FrBool traversal = True, FrBool wrap = True,
const char *name = 0)*

virtual FrWShadowText::~FrWShadowText()

!!!

*char *FrWShadowText::getText() const*

*void FrWShadowText::getText(char *buf, int buflen) const*

*void FrWShadowText::setText(const char *t, FrBool to_end = True)*

*void FrWShadowText::verifyCallback(FrWText_VerifyFunc *verify)*

7.50. Widget class FrWSlider

In addition to the methods inherited from `FrWidget` (not all of which apply to sliders), `FrWSlider` provides the following methods of its own:

```
  !!!
FrWSlider(Widget parent, const char *name, int max, int min,
        int def_value, int precision, FrBool vertical = False)
FrWSlider(FrWidget *parent, const char *name, int max, int min,
        int def_value, int precision, FrBool vertical = False)
static int sliderValue(XtPointer call_data) ;
```

7.51. Widget class FrWText

In addition to the methods inherited from `FrWidget` (not all of which apply to text windows), `FrWText` provides the following methods of its own:

```
!!!
FrWText(Widget w) ;
FrWText(Widget parent, const char *text, int columns = 30,
     int rows = 1, FrBool editable = True,
     FrBool traversal = True, FrBool wrap = True,
     const char *name = 0) ;
FrWText(FrWidget *parent, const char *text, int columns = 30,
     int rows = 1, FrBool editable = True,
     FrBool traversal = True, FrBool wrap = True,
     const char *name = 0) ;
char *getText() const ;
void getText(char *buf, int buflen) const ;
void setText(const char *t,FrBool to_end = True) ;
void limitLength(int maxlen) ;
int lengthLimit() const ;
void verifyCallback(FrWText_VerifyFunc *verify) ;
void setSensitive(FrBool sensitive) const ;
FrWText_VerifyFunc *getVerifyCallback() const ;
```

7.52. Widget class FrWToggleButton

In addition to the methods inherited from `FrWidget` (not all of which apply to toggle buttons), `FrWToggleButton` provides the following methods of its own:

```
!!!
     FrWToggleButton(Widget parent, const char *label,
                     FrBool set = False,
                     FrBool managed = True) ;
     FrWToggleButton(FrWidget *parent, const char *label,
                     FrBool set = False,
                     FrBool managed = True) ;
     FrBool getState() const { return toggle_set ; }
     void setState(FrBool state) ;
     void alwaysVisible(FrBool vis) ;
```

7.53. Character Manipulation

FramepaC provides a number of 8-bit clean character manipulation functions equivalent to those found in the standard C library (which often fail on characters with the high bit set). It also provides a number of functions for dealing with 16-bit Unicode characters.

Note that unlike the ANSI C library's is...() functions, the results from the FramepaC equivalents are not defined when given EOF as input.

7.53.1. Eight-Bit Characters

macro int Fr_isupper(char c)

Determine whether the specified character is uppercase or not. `c` must be a character value in the range 0-255 (i.e. EOF is not allowed).

macro int Fr_islower(char c)

Determine whether the specified character is lowercase or not. `c` must be a character value in the range 0-255 (i.e. EOF is not allowed, and will result in undefined behavior).

macro int Fr_isdigit(char c)

Determine whether the specified character is a numeric digit. `c` must be a character value in the range 0-255 (i.e. EOF is not allowed, and will result in undefined behavior).

macro int Fr_ispunct(char c)

Determine whether the specified character is a punctuation mark.

macro int Fr_isspace(char c)

Determine whether the specified character is whitespace.

macro int Fr_isalpha(char c)

Determine whether the specified character is alphabetic.

macro char Fr_tolower(char c)

Convert the specified character to lowercase if it is not already lowercase. `c` must be a character value in the range 0-255 (i.e. EOF is not allowed, and will result in undefined behavior).

macro char Fr_toupper(char c)

Convert the specified character to uppercase if it is not already uppercase. `c` must be a character value in the range 0-255 (i.e. EOF is not allowed, and will result in undefined behavior).

macro char Fr_unaccent_Latin1(char c)

Convert an accented character in the Latin-1 encoding into the equivalent ASCII character without the accent, i.e. a-acute returns 'a', A-grave returns 'A', etc.

macro char Fr_unaccent_Latin2(char c)

Convert an accented character in the Latin-2 encoding into the equivalent ASCII character without the accent, i.e. a-acute returns 'a', A-grave returns 'A', etc.

macro char Fr_unaccent(char c)

Convert an accented character in either the Latin-1 or the Latin-2 encoding (depending on whether FrDEFAULT_CHARSET_Latin1 has been defined in frconfig.h) into the equivalent ASCII character without the accent, i.e. a-acute returns 'a', A-grave returns 'A', etc.

macro bool Fr_is8bit(int c)

Determine whether the indicated value is valid as an eight-bit character (which allows it to be used for the above functions).

*int Fr_stricmp(const char *s1, const char *s2)*

Determine whether two strings of eight-bit characters are equivalent, ignoring differences between uppercase and lowercase characters. The return value is 0 if the two strings are equivalent; a negative return value indicates that `s1` precedes `s2` in the collating sequence, while a positive return value indicates that `s2` precedes `s1`.

7.53.2. Eight-bit Strings

!!!

*char *FrCanonicalizeSentence(const char *string,FrBool force_uppercase = False,
const char *possible_delim /*[256]*/ = 0)*

!!!

*FrString *FrDecanonicalizeSentence(const char *string,
FrBool force_lowercase = False,
FrBool Unicode = False)*

*FrList *FrCvtSentence2Symbollist(char *sentence, FrBool force_uppercase = True)*

!!!

THIS FUNCTION DESTROYS ITS ARGUMENT!

*FrList *FrCvtSentence2Wordlist(const char *sentence)*

!!!

*FrList *FrCvtString2Wordlist(const char *sentence,
const char *possible_delim /*[256]*/ = 0)*

!!!

*FrList *FrCvtString2Symbollist(const char *sentence,
const char *possible_delim /*[256]*/ = 0)*

!!!

*FrSymbol *FrCvtString2Symbol(const char *word, FrBool force_uppercase = True)*

!!!

7.53.3. Sixteen-Bit Characters

Normally, ‘‘wide’’ characters use the `wchar_t` type, however, on some architectures, `wchar_t` is in fact 32 bits! FramepaC therefore uses its own `FrChar16` type, which is exactly 16 bits regardless of how big `wchar_t` is (in fact, it is usually defined as `wchar_t`).

macro char Fr_highbyte(FrChar16 c)

Retrieve the high half of the 16-bit character `c`.

macro char Fr_lowbyte(FrChar16 c)

Retrieve the low half of the 16-bit character `c`.

macro int Fr_isspace16(FrChar16 c)

Determine whether the specified 16-bit character is a whitespace character.

macro int Fr_isdigit16(FrChar16 c)

Determine whether the specified 16-bit character is a numeric digit.

FrChar16 Fr_quoteUnicode(FrChar16 c)

Transform the input character in such a manner that it can be stored into a string of eight-bit characters and be processed correctly. This function converts any wide characters for which either byte (or both) is NUL, Newline, or '|' into a character within the Unicode "private use" range which does not contain the offending byte value. Use `Fr_unquoteUnicode` to recover the original 16-bit character.

FrChar16 Fr_unquoteUnicode(FrChar16 quoted_c)

Unmangle a previously quoted Unicode character to recover the original value.

char Fr_quoteUnicode(char *buffer, size_t numchars)*

Transform the input string of wide characters in such a manner that it can be stored into a string of eight-bit characters and be processed correctly. This function converts any wide characters for which either byte (or both) is NUL, Newline, or '|' into a character within the Unicode "private use" range which does not contain the offending byte value. Use `Fr_unquoteUnicode` to recover the original 16-bit character.

This function destructively modifies the buffer it is given.

char Fr_unquoteUnicode(char *buffer, size_t numchars)*

Unmangle a string which has previously been processed by `Fr_quoteUnicode` to recover the original string of Unicode characters.

This function destructively modifies the buffer it is given.

*void Fr_quoteUnicode(char *dest, const char *buffer, size_t numchars)*

Copy the input string of wide characters, transforming the individual characters in such a manner that they can be stored into a string of eight-bit characters and be processed correctly. This function converts any wide characters for which either byte (or both) is NUL, Newline, or '|' into a character within the Unicode "private use" range which does not contain the offending byte value. Use `Fr_unquoteUnicode` to recover the original 16-bit character.

*void Fr_unquoteUnicode(char *dest, const char *buffer, size_t numchars)*

Copy the input string of quoted wide characters, un-transforming the individual characters to recover the original Unicode characters.

*FrChar16 Fr_ugetc(FILE *in, FrBool &byteswap)*

FrChar16 Fr_ugetc(istream &in, FrBool &byteswap)

!!!

*FrChar16 *Fr_ugets(FILE *in, FrChar16 *buffer, size_t maxline, FrBool &byteswap)*

*FrChar16 *Fr_ugets(istream &in, FrChar16 *buffer, size_t maxline, FrBool &byteswap)*

Read up to `maxline`-1 sixteen-bit characters from the input file or stream and store them in `buffer`. The read ends when a Newline or NUL character (extended to 16 bits) is encountered on the input or `maxline`-1 wide characters have been read. A sixteen-bit NUL is appended to the data read from the input source to terminate the string. The parameter `byteswap` indicates whether or not byte-swapping is required; if `True`, then the input source is assumed to be in little-endian byte order (low byte first) rather than big-endian byte order (high byte first); if a Unicode stream marker is encountered while reading input, `byteswap` is updated to reflect the actual state of byte swapping on the input source.

*size_t Fr_ustrlen(const FrChar16 *unicode_string)*

Return the number of characters (not bytes!) in the given Unicode string. This function is a Unicode equivalent of the standard runtime function `strlen`.

*FrChar16 *Fr_ustrchr(const FrChar16 *unicode_string, FrChar16 unicode_char)*

Search the given string for the desired character, returning a pointer to the first occurrence of the character within the string, or 0 if there are none. This function is a Unicode equivalent of the standard runtime function `strchr`.

7.53.4. Sixteen-bit Strings

!!!

*char *FrCanonicalizeUSentence(const FrChar16 *string, size_t length = 0,*
FrBool force_uppercase = False,
*const char *possible_delim /*[256]*/ = 0)*

!!!

*FrString *FrDecanonicalizeUSentence(const char *string,*
FrBool force_lowercase = False)

*char *FrASCII2Unicode(const char *string, FrBool canonicalize = False)*

*FrList *FrCvtUString2Wordlist(const FrChar16 *sentence, size_t length = 0,*
*const char *possible_delim /*[256]*/ = 0)*

!!!

*FrList *FrCvtUString2Symbollist(const FrChar16 *sentence,*
*const char *possible_delim /*[256]*/ = 0)*

FrCvt

7.53.5. Byte-Order Independence

To aid in interchanging binary data between different hardware platforms, FramepaC provides the following functions for storing and retrieving multi-byte values in a byte-order independent fashion.

short FrByteSwap16(short int value)

Byte-swap a 16-bit value on little-endian machines; on big-endian machines, this function leaves the value unchanged. This is useful for converting a value after loading it from a network-byte-order buffer or prior to storing a value into such a buffer; in general, however, it is preferable to use the FrLoad...() and FrStore...() functions instead of this function, since those functions also ensure consistent data sizes regardless of architecture.

long int FrByteSwap32(long int value)

Same as `FrByteSwap16`, but for 32-bit values: byte-swap on little-endian machines, leave `value` unchanged on big-endian machines. The notes for `FrByteSwap16` also apply here.

*void FrStoreByte(int value, void *buffer)*

Store the low eight bits of the indicated value into the buffer. (For architectures where the smallest addressable unit [char] is other than eight bits, this function differs from simply storing the character)

*void FrStoreShort(short int value, void *buffer)*

Store the indicated 16-bit value into the buffer in canonical big-endian byte order.

*void FrStoreAlignedShort(short int value, void *buffer)*

Store the indicated short integer into a properly-aligned buffer (for many architectures, `buffer` must be on a two-byte boundary). This function is faster than `FrStoreShort`, but can only be used if the buffer is known to be properly aligned.

*void FrStoreThreebyte(UINT32 value, void *buffer)*

Store the indicated 24-bit value into the buffer in canonical big-endian byte order.

*void FrStoreLong(UINT32 value, void *buffer)*

Store the indicated 32-bit value into the buffer in canonical big-endian byte order.

*void FrStoreAlignedLong(UINT32 value, void *buffer)*

Store the indicated 32-bit integer into a properly-aligned buffer (for many architectures, `buffer` must be on a four-byte boundary). This function is faster than `FrStoreLong`, but can only be used if the buffer is known to be properly aligned.

*void FrStoreFloat(float value, void *buffer)*

Store the indicated single-precision (32-bit) floating-point value into the buffer in canonical big-endian byte order.

*void FrStoreDouble(double value, void *buffer)*

Store the indicated double-precision (64-bit) floating-point value into the buffer in canonical big-endian byte order.

*int FrLoadByte(void *buffer)*

Load the low eight bits of the first character in the indicated buffer. (For architectures where the smallest addressable unit [char] is other than eight bits, this function differs from simply loading the character)

*short FrLoadShort(const void *buffer)*

Read a 16-bit value from the buffer where it is stored in canonical big-endian byte order into memory in machine byte order.

*int FrLoadAlignedShort(const void *buffer)*

Read a 16-bit value from the buffer where it is stored in canonical big-endian byte order into memory in machine byte order. The buffer must be properly aligned (i.e. on a two-byte boundary) to avoid generating a bus error or similar fatal exception on many architectures.

*long FrLoadThreebyte(const void *buffer)*

Read a 24-bit value from the buffer where it is stored in canonical big-endian byte order into memory in machine byte order.

*long FrLoadLong(const void *buffer)*

Read a 32-bit value from the buffer where it is stored in canonical big-endian byte order into memory in machine byte order.

*long FrLoadAlignedLong(const void *buffer)*

Read a 32-bit value from the buffer where it is stored in canonical big-endian byte order into memory in machine byte order. The buffer must be properly aligned (i.e. on a four-byte boundary) to avoid generating a bus error or similar fatal exception on many architectures.

*float FrLoadFloat(void *buffer)*

Load the indicated single-precision (32-bit) floating-point value from the buffer, where it is stored in canonical big-endian byte order.

*double FrLoadDouble(void *buffer)*

Load the indicated double-precision (64-bit) floating-point value from the buffer, where it is stored in canonical big-endian byte order.

7.53.6. Character Mappings

```
 !!!
char *FrMakeCharacterMap(const FrList *map) ;
void  FrDestroyCharacterMap(char mapping[]) ;
// destructively apply the indicated character mapping to the string
char *FrMapString(char *string, const char mapping[]) ;
```

7.54. Signal Handling

Because various flavors of Unix (not to mention non-Unix operating systems) provide slightly different variants on signal handling, FramepaC includes a class which hides the differences behind a uniform interface -- `FrSignalHandler`.

The typedef `FrSignalHandlerFunc` defines the type of the function which will handle the signal; it accepts an `int` argument and returns nothing.

*FrSignalHandler::FrSignalHandler(int signal, FrSignalHandlerFunc *handler)*

Create a new instance of the signal handler which will manage the specified signal (e.g. SIGINT, SIGHUP, etc.), and will call `handler` whenever the signal occurs. The handler may be 0 or `SIG_IGN` to ignore the signal, or `SIG_ERR` to generate an error exception.

FrSignalHandler::~FrSignalHandler()

Restore signal handling for the managed signal to its default state, and free the resources used by the signal handler.

*FrSignalHandlerFunc *FrSignalHandler::set(FrSignalHandlerFunc *new_handler)*

Specify a new function to be invoked whenever the signal occurs; returns the previous function. The new function may be 0 or `SIG_IGN` to ignore the signal, or `SIG_ERR` to throw an exception. A return value of 0 means that the signal was previously ignored.

void FrSignalHandler::raise(int arg) const

Invoke the signal-handling function with the specified integer argument, just as if the operating system had triggered the signal.

int FrSignalHandler::signalNumber() const

Return the number of the signal which the particular instance of `FrSignalHandler` is managing.

*FrSignalHandlerFunc *currentHandler() const*

Determine which function will be invoked when the managed signal occurs. A return value of 0 means that the signal is being ignored.

7.55. Timed Event Handling

!!!

typedef time_t FrEventFunc(void *client_data) ;

FrEventList::FrEventList()

!!!

FrEventList::~FrEventList()

*FrEvent *addEvent(time_t time, FrEventFunc *f, void *client_data, FrBool delta = False)*

Add a new event to the event list. The new event will be scheduled to execute at `time`, and will call function `f` with argument `client_data`. If `delta` is `True`, the specified time is an increment from the current time rather than an absolute time.

*FrBool FrEventList::removeEvent(FrEvent *event)*

Remove the specified event from the event list; returns `True` if the event was successfully removed, `False` if the event is not on the event list.

After this method is called and returns `True`, `event` will no longer be a valid pointer.

*void FrEventList::reschedule(FrEvent *event, time_t newtime)*

Change the execution time of the specified event to be `newtime`.

*void FrEventList::postpone(FrEvent *event, time_t delta)*

Delay the execution time of the specified event by `delta` seconds.

void FrEventList::executeEvents()

Each time this function is called, any events on the list whose scheduled time is less than or equal to the current time will be executed and then removed from the list (unless the event handler specifies a new time in the future).

This function must be called regularly.

7.56. Memory-Mapped File Functions

On platforms which support mapping disk files directly into a process' address space, the following functions provide a simple, consistent interface to the memory-mapping feature:

*FrFileMapping *FrMapFile(const char *filename, FrMapMode mode)*

Map the specified file into the caller's address space using the access mode specified by the enumeration `mode` (which may be `FrM_READONLY`, `FrM_READWRITE`, or `FrM_COPYONWRITE`). Returns a pointer to an instance of a private class used to track the file mapping, or 0 if the mapping failed or memory-mapped files are not supported by the operating system. If this function returns 0, the caller must arrange to read in data from the file with usual file-access functions; for a nonzero return, the next two functions indicate which portion of the caller's address space is being used by the file.

*void *FrMappedAddress(const FrFileMapping *fmap)*

Return the starting address of the region of memory into which the `FrMapFile` call producing the indicated `FrFileMapping` placed the file.

*size_t FrMappingSize(const FrFileMapping *fmap)*

Return the size (in bytes) of the region of memory into which the `FrMapFile` call producing the indicated `FrFileMapping` placed the file. The memory from `FrMappedAddress` to `FrMappedAddress`+`FrMappingSize`-1 may safely be accessed; no access should be attempted outside this range, nor should any writes be attempted even within the range if the mapping was set up with the `FrM_READONLY` option.

*FrBool FrUnmapFile(FrFileMapping *fmap)*

Clean up the specified mapping of file into address space. Returns `True` if successful and `False` if it should for some reason fail. After a successful `FrUnmapFile`, the file will no longer be accesssible in the caller's address space, and no accesses should be attempted in the range indicated by `FrMappedAddress` and `FrMappingSize`.

7.57. Performance Monitoring

FramepaC provides the `FrTimer` class for measuring how much time has been spent in an indicated section of the program. `FrTimer` records the CPU time, rather than "wall-clock" time (except under MS-DOS, where the two are equivalent anyway) to a resolution -- though not necessarily accuracy -- of 0.1 milliseconds.

FrTimer::FrTimer()

Create a new timer, and start it running. The elapsed CPU time may be retrieved via `read`, and is returned when the timer is stopped or paused. By default, the elapsed time includes the value of any nested timers (see `FrTimer(FrTimer*)` below), but this may be modified with `includeSubTimers`.

*FrTimer::FrTimer(FrTimer *parent)*

Create a new timer nested within the timing of the indicated parent timer, and start it running. If the parent timer's value does not include any nested timers (see `includeSubTimers` below), then the parent will be paused while this timer is running. The elapsed CPU time may be retrieved via `read`, and is returned when the timer is stopped or paused.

FrTimer::~FrTimer()

Destroy the timer, and restart the parent timer (if any) if it was paused by the running of the deleted timer.

void FrTimer::start()

Start a timer running. The elapsed time is reset to 0, even if the timer was already running, and the parent timer (if any) is paused if it was running and does not include the timings of nested timers.

UINT32 FrTimer::stop()

Stop a timer running, and return the total elapsed CPU time since it was last started (not including times during which it was explicitly or implicitly paused). The returned value increments at a rate of `FrTICKS_PER_SEC` counts each second. If the timer has a parent which was paused for this timer, the parent timer resumes running.

UINT32 FrTimer::pause()

Temporarily suspend the timer, and return the total elapsed CPU time since it was last started (not including times during which it was explicitly or implicitly paused). The returned value increments at a rate of `FrTICKS_PER_SEC` counts each second.

void FrTimer::resume()

Restart a previously suspended timer. If the timer was stopped, this method is equivalent to `start`; if the timer was already running, it has no effect.

UINT32 FrTimer::read()

Retrieve the current elapsed CPU time during which the timer was running since it was last started. The returned value increments at a rate of `FrTICKS_PER_SEC` counts each second.

void FrTimer::includeSubTimers(FrBool include)

Specify whether the timer's run-time value should include the values of any nested timers (those constructed with this timer as a parent) or not. If `include` is `True` (the default), then any nested timers are ignored for the purposes of computing the elapsed CPU time. If `include` is `False`, then this timer is suspended whenever a nested timer is running, and thus records the CPU time *only* when no nested timers are active. This may be used, for example, to determine how much time is spent in a particular function without including the time spent in other functions called by it.

FrBool FrTimer::includesSubTimers() const

Determine whether the reported time for the `FrTimer` includes the elapsed time during which subtimers were running or not.

FrBool FrTimer::isRunning() const

Determine whether the timer is currently running.

FrBool FrTimer::isPaused() const

Determine whether the timer is currently paused (either explicitly by `pause` or implicitly by a nested timer when `includeSubTimer(False)` has been called).

FrBool FrTimer::isStopped() const

Determine whether the timer is currently stopped.

7.58. Debugging Support

void FrObject::_() const

For convenience in `gbd`; this method is equivalent to `FrObject::print(cerr)`.

assert(condition)

Test the specified condition (which may be any valid rvalue), and print an error message showing the source file, line number, and failing condition if the value is zero. The program is normally terminated after the error message is printed (but see `FrAssertionFailureFatal` below).

Like the standard `assert`, the FramepaC version is a null macro if `NDEBUG` is defined prior to #include'ing `FramepaC.h`.

Source files which use assert() multiple times can save data space by adding the following lines after #include'ing `FramepaC.h` and before invoking assert() or assertq():

```
#ifndef NDEBUG
# undef _FrCURRENT_FILE
static const char _FrCURRENT_FILE[] = __FILE__ ;
#endif /* NDEBUG */
```

assertq(condition)

`assertq` is the same as `assert` except that it does not print the violated condition if the assertion fails.

int FrAssertionFailureFatal(int is_fatal)

By default, assertion failures (`assert` and `assertq`) cause the program to be terminated. This function allows the program to continue executing after an assertion failure if `is_fatal` is zero. The function returns the previous value of the `is_fatal` flag.

7.59. Miscellaneous Functions

This section covers those function which do not easily fit in any of the previous sections.

7.59.1. Messages

FramepaC internally uses a number of functions to provide a consistent method for displaying informative and error messages, which may also be redirected to other destinations. When `FrInitializeMotif` has been used, for example, these functions display their messages in a separate window, which can also be useful for programs incorporating FramepaC.

*void FrMessage(const char *message)*

Display the provided message on a separate line.

*void FrMessageVA(const char *message, ...)*

Display the result of applying `sprintf` to the provided message and any additional arguments on a separate line.

*void FrWarning(const char *message)*

Display an indication of a warning followed by the given message. This function is useful for informing the user of unusual but non-fatal conditions.

*void FrWarningVA(const char *message, ...)*

Display an indication of a warning followed by the given message. The message is composed by applying `sprintf()` to the message string and any additional arguments given to `FrWarningVA`. This function is useful for informing the user of unusual but non-fatal conditions.

*void FrNoMemory(const char *circumstance)*

If the application runs out of memory and is unable to recover (for example, `FrMalloc` returns 0), this message may be used to inform the user and then terminate the program. Its parameter is a description of the circumstance under which the program ran out of memory, such as "while allocating X" or "in X::Y". Note that the caller should be prepared to deal with a possible return from this function in case the default handler has been overridden.

*void FrNoMemoryVA(const char *circumstance, ...)*

If the application runs out of memory and is unable to recover (for example, `FrMalloc` returns 0), this message may be used to inform the user and then terminate the program. Its parameters are an `sprintf` format string and any additional arguments required by it, which together form a description of the circumstance under which the program ran out of memory, such as "while allocating %d bytes for X" or "in X::Y". Note that the caller should be prepared to deal with a possible return from this function in case the default handler has been overridden.

*void FrProgError(const char *message)*

Inform the user that a fatal internal error (such as a missing case in a `switch` statement) has been encountered, and then terminate the program. Note that the caller should be prepared to deal with a possible return from this function in case the display handler has been overridden.

*void FrProgErrorVA(const char *message, ...)*

Inform the user that a fatal internal error (such as a missing case in a `switch` statement) has been encountered, and then terminate the program. The error message is generated by applying `sprintf` to the message string and any additional arguments passed to `FrProgErrorVA`. Note that the caller should be prepared to deal with a possible return from this function in case the display handler has been overridden.

*void FrError(const char *message)*

Inform the user than an unrecoverable error (for which none of the above functions would be applicable) has occurred, and then terminate the program. Note that the caller should be prepared to deal with a possible return from this function in case the display handler has been overridden.

*void FrErrorVA(const char *message, ...)*

Inform the user than an unrecoverable error (for which none of the above functions would be applicable) has occurred, and then terminate the program. The text of the error message is generated by applying `sprintf` to the message string and any additional arguments passed to `FrErrorVA`. Note that the caller should be prepared to deal with a possible return from this function in case the display handler has been overridden.

*void FrUndefined(const char *func_name)*

Inform the user that the program has attempted to invoke an as-yet undefined or unimplemented function.

*void FrInvalidVirtualFunction(const char *class_and_method)*

Inform the user of the internal error of invoking an invalid virtual method. The argument is a string indicating the class name and method, such as "Foo::bar".

*void FrMissedCase(const char *function)*

Inform the user of the programming error of having missed a case in a `switch` statement. Its argument is the name of the function in which the switch statement is located.

This function is normally called from a `default` case in a switch statement, i.e.

```
void foo(int arg)
{
   switch (arg)
     {
     case 0:
         ...
     case 1:
         ...
     default:
         FrMissedCase("foo") ;
     }
```

will invoke the error message whenever `foo` is called with an argument other than 0 or 1.

In addition to the wrapper functions listed above, there is a hooking function for each of the above, which will set the actual function to be invoked and return the current handler:

FramepaC_error_handler set_message_handler(FramepaC_error_handler)

FramepaC_error_handler set_warning_handler(FramepaC_error_handler)

FramepaC_error_handler set_out_of_memory_handler(FramepaC_error_handler)

FramepaC_error_handler set_prog_error_handler(FramepaC_error_handler)

FramepaC_error_handler set_undef_function_handler(FramepaC_error_handler)

FramepaC_error_handler set_fatal_error_handler(FramepaC_error_handler)

FramepaC_error_handler set_invalid_function_handler(FramepaC_error_handler)

FramepaC_error_handler set_missed_case_handler(FramepaC_error_handler)

7.59.2. Database

*char *database_index_name(char *database_name, int indexnumber)*

Generate the filename of the indicated index for the database. The returned buffer containing the name has been dynamically allocated and must be explicitly freed with `FrFree`.

*char *VFrames_indexfile()*

Return the name of the main index file (listing frames by name) for the current symbol table, or 0 if the current symbol table is not using a file for backing store.

*fstream *VFrames_indexstream(int indextype)*

Return an open stream referencing the indicated auxiliary index file for the current symbol table, or 0 if the current symbol table is not using a file for backing store or the indicated index type is not being maintained for the symbol table. The valid index types are

```
INDEX_INVSLOTS          inverted index by slot and facet names
INDEX_INVFILLERS        inverted index by filler values
INDEX_INVWORDS          index by words within string fillers
```

7.59.3. Memory Allocation

A few of the FramepaC functions return a block of allocated memory which must be explicitly freed. This may be done with `FrFree`. If desired, `FrMalloc` may be used to allocate memory instead of `new`, particularly in overridden class-specific `new` and `delete` operators. `FrMalloc` can be much faster than `new char[]`, especially for small objects (under 2000 or so bytes, which covers the majority of all allocations). `FrMalloc` has the additional advantage that it minimizes memory fragmentation conflicting with FramepaC's own memory allocations. Even faster allocations for `new` are available for uniformly-size objects with the **FrAllocator** class, described in Chapter 5.3.

To use `FrMalloc` for all object allocations, add the following two lines to the object at the top of each inheritance hierarchy in your program:

```
void *operator new(size_t size) { return FrMalloc(size) ;}
void operator delete(void *block) { FrFree(block) ;}
```

void FramepaC_gc()

Return as much FramepaC-allocated memory as possible to the system. If enabled, FramepaC will also attempt to rearrange its internal memory allocations to create the largest possible blocks of free memory. This function is primarily intended for use in a `new_handler` (see `set_new_handler` in your C++ manual) to make additional memory available to allocations which do not use `FrMalloc`. Unrestrained use of this function can severely impact performance and fragment memory.

*void *FrCalloc(size_t nitems, size_t size)*

Allocate an array of `nitems` items, each of the indicated size, and return a pointer to the first item in the array. Returns 0 if unable to allocate the requested amount of memory even after compacting free space and optionally discarding the least-recently used virtual frames.

*void FrFree(void *block)*

Deallocate the indicated block, which was previously obtained from `FrMalloc`.

*void *FrMalloc(size_t numbytes)*

Allocate a block of `numbytes` bytes and return a pointer to its beginning, or 0 if unable to allocate the requested amount of memory even after compacting free space.

*void *FrRealloc(void *block, size_t newsize, FrBool copydata = True)*

Change the size of the indicated block of memory, and return a pointer to the resulting memory block (which need not be the same one passed in). If a new block must be allocated, `copydata` indicates whether or not to copy the contents of the original block into the new block; if the contents need not be preserved, you will save time by setting `copydata` to `False`.

If a new block must be allocated but `FrRealloc` is unable to obtain one of the required size, it returns 0 and leaves the original block unchanged. If the new allocation was successful, the original block will be freed.

void show_memory_usage(ostream &out)

Display a list of the various memory pools currently in use, including the number of suballocator blocks currently assigned to the pool, the total number of bytes in those blocks, the number of objects for the memory pool which are currently in use, and the number of objects on the pool's free list. In addition, statistics for the `FrMalloc` memory pool are printed, listing the number of blocks of various sizes which are currently in use and currently free (as well as the number of kilobytes in each category and the total number of blocks and kilobytes allocated/free over all sizes).

```
Memory Usage
============        Blocks          Bytes       Objs Used  Unallocated
Symbols                 1           16352
FrCons                  1           16352            6      1356
FrInteger               1           16352            3      2041
FrFloat                 1           16352            1      1361
FrFrame                 1           16352            1       193
FrSlot                  1           16352            1      1021
FrString                1           16352            2      1020
FrMalloc                1           16352
============
   Total                8          130816

Malloc blocks:   30    2    0    0    0    0    0    0    0    0    9 =   41
Malloc KB:        0    0    0    0    0    0    0    0    0    0    8 =    8
Free blocks:      0    0    0    0    0    0    0    0    0    0    1 =    1
Free KB:          0    0    0    0    0    0    0    0    0    0    8 =    8
```

The following convenience macros are also available:

`type`* *FrNew(*`type`*)*

Allocate an object of the indicated type, and return a pointer to the newly-allocated object. Unlike `new`, this macro does not execute any constructors which might be associated with `type`.

`type`* *FrNewC(*`type`*,int)*

Allocate an array of objects of the indicated type, clear the allocated memory to all zero bytes, and return a pointer to the first object in the newly-allocated array.

`type`* *FrNewN(*`type`*,int)*

Allocate an array of objects of the indicated type, and return a pointer to the first object in the newly-allocated array. Unlike `new`, this macro does not execute any constructors which might be associated with `type`.

`type`* *FrNewR(*`type`*,void *blk,int newsize)*

Resize a previously-allocated block of memory to the specified new size in bytes. Returns a pointer to the resized block (which may differ from the original pointer), or 0 if unable to resize the block (in which case the original remains untouched).

7.59.4. Motif Interface

FramepaC was intended for use in an application using the Motif user interface, and it is thus useful to have support for that interface.

*void FrInitializeMotif(char *window_name, Widget parent, int max_symbols)*

This performs the same function as `initialize_FramepaC`, and additionally prepares FramepaC to use a separate window for any error or warning messages it may generate (instead of writing them to the Xterm or similar window from which the application was started). It is not necessary to use `initialize_FramepaC` when using this function.

If `window_name` is 0, FramepaC supplies a default window name.

*Widget FrInitializeMotif(int *orig_argc, char **orig_argv,*
*const char *maintitle, const char *msgtitle,*
*const char *icon_name,*
FrBool allow_resize = True)

Initialize both FramepaC and the Motif user interface. This function requires both the original `argc` and `argv` passed to `main`, which it will update to remove any Motif-specific commandline arguments. The other three strings give the titles for the full toplevel window, the FramepaC message window, and the application's icon, respectively. Finally, `allow_resize` indicates whether the user should be allowed to resize the application's main window using the X window manager.

void FrShutdownMotif()

This performs the same function as `FrShutdown`, and additionally cleans up after FramepaC's use of Motif. It is not necessary to use `FrShutdown` when using this function.

In addition to the above functions, FramepaC also overrides the Xt memory allocations functions to use its own instead. Thus, `XtMalloc`, `XtCalloc`, `XtRealloc`, and `XtFree` are all integrated into the FramepaC memory management system and can take advantage of its greater speed, memory compaction, and automatic frame discarding to free memory. In order for the override to work properly, the FramepaC library must be linked in before the Xt library, i.e. (for Unix) `-lframepac` must precede `-lXt` on the linking commandline, and you must use `FrInitializeMotif`.

8. Preprocessor Symbols

The following configuration options may be set when FramepaC is compiled, and may be checked in applications using FramepaC:

FramepaC_Version
Specifies the version number of FramepaC in use. The value of this symbol is 100 times the actual version number, i.e. 90 is version 0.90, 111 is version 1.11, etc.

FramepaC_Version_string
Specifies the version number of FramepaC as a string, i.e. "1.11".

FrDEMONS
If defined, support for demon functions has been compiled into FramepaC.

FrREPLACE_MALLOC If defined, the standard library functions `malloc`, `calloc`, `realloc`, and `free` have been overridden to use `FrMalloc`, `FrCalloc`, `FrRealloc`, and `FrFree` instead.

FrMEMORY_CHECKS If defined, the internal housekeeping variables associated with each block of memory are checked for consistency on every `FrMalloc`, etc. function call. If not defined, much less checking is performed, which slightly improves execution speed but can cause memory to become much more corrupted before the error is detected. When defined, FramepaC can detect many memory overruns and most duplicate deallocations as soon as the affected memory block is deallocated.

FrEXTRA_INDEXES If defined, support for database indexes beyond the simply by-name index required for retrieving frames on demand has been compiled into FramepaC.

FrLRU_DISCARD If defined, virtual frames will be discarded when memory is exhausted, beginning with the least-recently accessed frame. When not defined, frames will not be discarded automatically, but only when `discard_frame` is called explicitly. The automatic discarding is an option because it increases the overhead of virtual frames by the time required to maintain access-time information.

FrMOTIF If defined, support for the Motif user interface has been compiled into FramepaC. If not defined, `FrInitializeMotif` and `FrShutdownMotif` become identical in operation to `initialize_FramepaC` and `FrShutdown`.

FrREPLACE_XTMALLOC If defined, the standard Xlib memory-allocation functions `XtMalloc`, `XtRealloc`, and `XtFree` have been overridden to use `FrMalloc`, etc. instead.

FrSEPARATE_XLIB_ALLOC If defined, Xlib memory allocation is performed from a different memory pool than the one used by `FrMalloc`, allowing `show_memory_usage` to distinguish between memory allocated by/for X/Motif functions and that allocated by/for FramepaC functions.

FrDATABASE
If defined (default), disk-based virtual frames were enabled at compile-time, and all of the associated functions and database-handling code are available.

FrSERVER
If defined, server-based virtual frames were enabled at compile-time, and all of the associated functions and network code are available.

FrSYMBOL_VALUE
If defined, each symbol may have an associated value (in Lisp terms, a binding).

FrLITTLEENDIAN Defined if the architecture on which FramepaC is running stores multi-byte values least-significant byte first.

FrBIGENDIAN Defined if the architecture on which FramepaC is running is known to store multi-byte values most-significant byte first.

FrMAX_NUMSTRING_LEN
This constant indicates the maximum number of characters the FramepaC reader will process when reading a number.

FrMAX_SYMBOLNAME_LEN
This constant indicates the maximum number of characters the FramepaC reader will process when reading a symbol's name.

FrMAX_ULONG_STRING This constant indicates the maximum number of characters required to represent a UINT32 in base 10.

FrMAX_DOUBLE_STRING This constant indicates the maximum number of characters required to represent a `double` in base 10.

Defined if the type `long` is known to contain exactly 32 bits.

9. Troubleshooting

9.1. Questions and Answers

Q: Why do I get ((NIL)(NIL)(NIL)(NIL)(NIL)(NIL)(NIL)(NIL)...) as output instead of the expected list?

A: You have freed the list at which your variable pointed. Unlike Lisp, with its automatic garbage collection that frees memory only when it is no longer referenced, FramepaC requires you to manually deallocate memory. If you had another pointer to the list, or re-used the original pointer after freeing the list without first changing it, you could see output similar to that above, because your pointer will be pointing somewhere inside FramepaC's internal list of free cells until FramepaC reuses the list cell to which the pointer is pointing. One method for avoiding this error is to set the pointer whose list you free to 0 if you are not immediately assigning a new value to it for other reasons.

Q: Why is my program's memory usage constantly increasing?

A: As above, FramepaC does not perform garbage collection, so it is up to you to ensure that every object which is created is deallocated when it is no longer required. The function `show_memory_usage(ostream&)` can give you an idea of what types of objects are not being deallocated. A prime candidate is usually forgetting to free lists which are the return value of some function which created the list specifically to be returned to the caller (as many of FramepaC's functions do).

Q: Why can't my program read objects of certain types? It gets one or more symbols and/or lists instead.

A: Your program makes no use whatsoever of that particular class of object, so it has linked in none of the code for the class -- including the functions to read in objects of that type. If you really need to read that type of object, add a dummy global variable of the appropriate type to one of your program's source files.

10. Test/Demo Program

The file `test.C` in the FramepaC source distribution contains a simple interactive loop with which you may enter objects and have them echoed back, or perform a number of commands. To enter a command instead of a symbol or other object, type "*" (asterisk) followed by a space, followed by the command name and any arguments. The following commands are supported at this time:

ALL-FRAMES

FrList all frames currently defined in the active symbol table. All of these frames will have been entered by the user during the current program run, or have been accessed and retrieved from the backing store.

ALL-SLOTS <frame>

FrList all slots (and their facets) which the named frame is capable of inheriting.

ALL-SYMBOLS

Display a listing of all of the symbols in the current symbol table (this can become extremely long if any data files have been loaded).

BENCH

Display a menu of benchmark tests (described in Chapter 11).

CHECKMEM

Perform a consistency check on the FramepaC memory chain, and indicate whether memory is OK or corrupted.

CLIENT

Display a menu for testing the client side of the networking code.

COMPLETE

FrList all frames in the current symbol table which have names beginning with the user-provided string. Also show the longest common prefix among the returned frames, which is what a name-completion function could provide if invoked on the original string.

DEMONS

(this command is only available if demon support has been compiled into FramepaC) !!!

EXPORT

Store all of the frames in the current symbol table in the user-selected file in FrameKit format.

EXPORT-NATIVE

Store all of the frames in the current symbol table in the user-selected file in FramepaC’s native format.

GC

Force a memory compaction (as FramepaC does not have true garbage collection) and return any blocks of memory which could be freed to the system or the global pool of unallocated memory.

GENSYM

Create a new, unique symbol and display it.

IMPORT

Load all of the FrameKit-format frames in the user-selected file into the current symbol table. This is useful for converting an old ONTOS ontology into FramepaC format; simply use the following sequence of commands to the test program:

```
* bench 8 <dbname> 4
* import <ontosdb>
save
nil 0
```

A 2000-frame ONTOS ontology can be converted into a FramepaC database in under two minutes in this way.

INHERIT <frame> <slot>

Show the fillers for the VALUE, SEM, and DEFAULT facets of the specified slot, as inherited from any ancestor frames entered previously in this program run.

INHERITANCE

Display a menu of the inheritance types, and select a new type.

IS-A-P <frame1> <frame2>

Determine whether the frame named by the symbol `frame1` can inherit from the frame named by the symbol `frame2` through a chain of `IS-A` links.

LOCK

Display a menu allowing you to lock or unlock frames.

LOGIN

Identify yourself as a particular user to the FramepaC access control system. You will be prompted for a username and password (which will both be echoed to the screen).

Display a table showing how much memory is being used by FramepaC for various purposes.

MEMBLOCKS

Display a list of the memory blocks under FramepaC's control (this can become quite long!), showing the size and free/allocated status of each block.

NEWUSER

Add a new user record to the FramepaC access control system.

PASSWD

Change a particular user's password in the FramepaC access control system. You will be prompted for the user's name, the old password, and the new password (which will all be echoed to the screen).

PART-OF-P <frame1> <frame2>

Determine whether the frame named by the symbol `frame1` can inherit from the frame named by the symbol `frame2` through a chain of `PART-OF` links.

RELATIONS

Display all of the relations for which FramepaC automatically maintains inverse links. This command prints a list of two-element lists; each sublist defines the two slots which are in an inverse relation with each other. A slot may be its own inverse.

RENAME

Change the name of a frame, and update any inverse links pointing at the frame. Any other links pointing at the frame's original name will **not** be updated.

REVERT

For a virtual frame with backing store, revert the frame to a specified prior version.

SYMTAB

Display a menu of symbol table manipulation options. You may create, delete, switch, and list symbol tables.

TRANSACT

Display a menu of database transaction operations. This allows you to test, for example, the capability to roll back the backing store after an error.

When not entering one of the above commands, the test program accepts any type of FramepaC object and echoes it back along with some pertinent information about the object, such as its type, how many bytes it takes to print its value, etc.

11. Benchmark Results

The following benchmark tests are available in the test program. Each allows you to specify a size `N` and a number of repetitions (to increase the total run time for more accurate timing):

1. **makeSymbol** (Test 1)
   Call makeSymbol on the string "RELATION" <repetitions> times.

2. **Creation** (Test 2)
   Create N frames and then delete all of them (note that the deletion actually frees the frame record proper, unlike `erase_frame` or `free_object(FrFrame*)`, which affect at most the lists of fillers for the frame).

3. **Relations** (Test 3)
   Create a "hub" frame, and then create N frames, adding the hub as a filler for the `IS-A`, `INSTANCE-OF`, and `PART-OF` slots in each frame; after creating all N frames, delete them as in Test 2.

4. **Inheritance** (Test 4)
   Create a top-level frame and then N frames, each of which `IS-A` the previous frame. Add a filler for the `TEST` slot in the top-level frame, and retrieve the fillers for the `TEST` slot in the bottom-most frame (with inheritance) 100 times. Finally, delete all frames as in Test 2.

5. **Output Speed** (Tests 5a to 5c)
   Repeatedly convert each of three objects to a string containing its printed representation. The three objects are a symbol; a list containing a symbol, a string, a cons of a symbol and a string, a floating point number, and an integer; and a frame with various slots filled.

6. **Input Speed** (Tests 6a to 6c)
   Repeatedly convert each of three objects from their string representations to their actual values and then free the created object. The three objects are the same as for the previous test, except that the list contains a sublist rather than a cons (the FramepaC reader did not handle dotted lists until very recently).

7. **Virtual Frames (memory)** (Tests 7-1 to 7-3)
   The three tests in this category are identical to Tests 2 through 4, but use objects of type `VFrame` instead of `FrFrame`. All frames are created and manipulated entirely in memory, to show the overhead of using `VFrame` instead of `FrFrame`.

8. **Virtual Frames (disk)**
   This item is the same as the previous, except that the frames are stored in a disk file. This benchmark was not tested because it is highly dependent on factors beyond the actual speed of the program, such as the location of files on disk and the contents of the disk cache.

9. **Virtual Frames (server)**
   This item is the same as "Virtual Frames (memory)", except that the frames are stored on a remote machine and accessed through the network interface. This benchmark also was not tested because of its dependence on outside factors.

10. **FrameKit/LOOM Benchmarks**
    A set of benchmarks which are equivalent to a set of benchmarks use to gauge FrameKit and LOOM speeds. Note that the original Lisp implemention of the benchmarks contains some bugs which cause performance to be overstated on some tests and understated on others.

11. **Memory Allocation Speed** (Tests 11a to 11i)
    This item causes repeated allocation and deallocation of variously sized blocks of memory in various orders. In the execution-time tables below, Tests 11a to 11c allocate blocks of 20,000 bytes each, Tests 11d to 11f use blocks of 200 bytes each, and Tests 11g to 11i use blocks of 20 bytes each (which often creates a working set small enough to fit entirely in the RAM cache). Tests 11a, 11d, and 11g allocate all memory blocks, then release them in the same order in which they were allocated. Tests 11b, 11e, and 11h allocate all memory blocks, then release them in the opposite order from which they were allocated. Finally, Tests 11c, 11f, and 11i allocate and release memory blocks in random order (freeing any still-allocated blocks once the specified number of memory blocks has been allocated).

Test	Size	Repetitions	Total Time	Time/Iteration (ms)
1	1	50,000	0.331 s	0.0066
2	1000	200	4.152 s	20.8
3	1000	200	14.109 s	70.5
4 Simp	1000	40	16.004 s	400.1
4 DFS	1000	40	18.087 s	452.2
4 BFS	1000	40	28.869 s	721.7
5a	1	50,000	0.456 s	0.0091
5b	1	50,000	2.299 s	0.046
5c	1	50,000	4.839 s	0.097
6a	1	50,000	0.998 s	0.020
6b	1	50,000	6.242 s	0.125
6c	1	50,000	22.086 s	0.442
7-1	1000	200	5.458 s	27.3
7-2	1000	200	15.807 s	79.0
7-3 S	1000	40	16.253 s	406.3
7-3 DF	1000	40	19.334 s	483.4
7-3 BF	1000	40	28.856 s	721.4

-- these timings are very old, and do not reflect current performance --

Table 11-1: Timings for Intel Pentium (16-bit code)

The test machines were

- an Intel Pentium running at 90 MHz, using the MS-DOS operating system and the Borland C++ v3.1 compiler in large memory model (Table 11-1) and the Watcom C++32 v10.0a compiler (32-bit code) in flat memory model (Tables 11-2 and 11-3).
- a Sun SPARCstation LX running SunOS 4.1.3, using GCC (Table 11-5)

Overhead for VFrame on the Pentium: creation 31.3% (6.5/20.8), relations 12.1% (8.5/70.5), inheritance 1.5% (6.2/400.1).

Overhead for VFrame on the Pentium: creation 2.7% (0.17/6.26), relations 8.0% (1.3/16.2). Inheritance is faster for VFrame, which appears to be an artifact due to a slightly different execution path when a backing-store is present.

Overhead for VFrame on the Sparc: creation 35%, relations 5.8%. Inheritance is faster for VFrame, which appears to be an artifact due to a slightly different execution path when a backing-store is present.

Test	Size	Repetitions	Total Time	Time/Iteration (ms)
1	1	2,000,000	3.535 s	0.00177
2	1000	300	1.869 s	6.23
3	1000	300	4.837 s	16.1
4 Simp	1000	100	21.698 s	217.0
4 DFS	1000	100	18.213 s	182.1
4 BFS	1000	100	21.998 s	220.0
5a	1	500,000	1.248 s	0.0025
5b	1	500,000	26.919 s	0.0538
5c	1	500,000	15.949 s	0.0319
6a	1	200,000	0.802 s	0.00401
6b	1	200,000	6.801 s	0.0340
6c	1	200,000	22.790 s	0.114
7-1	1000	300	1.930 s	6.43
7-2	1000	300	5.246 s	17.5
7-3 S	1000	100	20.498 s	205.0
7-3 DF	1000	100	16.966 s	169.7
7-3 BF	1000	100	21.112 s	211.1
11a	250	1000	0.779 s	0.78 (D4G = 27.062)
11b	250	1000	0.621 s	0.62 (D4G = 26.869)
11c	250	1000	0.524 s	0.52 (D4G = 24.907)
11d	2500	1000	7.109 s	7.11 (D4G = 22.434)
11e	2500	1000	7.034 s	7.03 (D4G = 22.185)
11f	2500	1000	6.268 s	6.27 (D4G = 23.812)
11g	5000	1000	8.728 s	8.73 (D4G = 17.786)
11h	5000	1000	8.961 s	8.96 (D4G = 17.984)
11i	5000	1000	9.194 s	9.19 (D4G = 32.314)

Table 11-2: Timings for Intel Pentium (Watcom 32-bit code, MS-DOS)

Test	Size	Repetitions	Total Time	Time/Iteration (ms)
1	1	2,000,000	3.50 s	0.00175
2	1000	300	2.06 s	6.87
3	1000	300	5.56 s	18.5
4 Simp	1000	100	32.10 s	321.0
4 DFS	1000	100	25.58 s	255.8
4 BFS	1000	100	27.56 s	275.6
5a	1	500,000	1.23 s	0.0025
5b	1	500,000	26.25 s	0.0525
5c	1	500,000	16.03 s	0.0321
6a	1	200,000	0.79 s	0.0040
6b	1	200,000	7.85 s	0.0393
6c	1	200,000	21.37 s	0.107
7-1	1000	300	2.19 s	7.30
7-2	1000	300	6.47 s	21.57
7-3 S	1000	100	30.87 s	308.7
7-3 DF	1000	100	23.06 s	230.6
7-3 BF	1000	100	25.28 s	252.8
11a	250	1000	40.68 s	40.68
11b	250	1000	43.31 s	43.31
11c	250	1000	19.68 s	19.68
11d	2500	1000	6.47 s	6.47
11e	2500	1000	6.60 s	6.60
11f	2500	1000	6.69 s	6.69
11g	5000	1000	9.985 s	10.55
11h	5000	1000	10.461 s	10.32
11i	5000	1000	10.103 s	10.92

Table 11-3: Timings for Intel Pentium (Watcom 32-bit code, Win32)

Test	Size	Repetitions	Total Time		Time/Iteration (ms)
1	1	2,000,000	3.90	s	0.00195
2	1000	300	2.20	s	7.33
3	1000	300	5.77	s	19.2
4 Simp	1000	100	32.46	s	324.6
4 DFS	1000	100	23.95	s	239.5
4 BFS	1000	100	28.67	s	286.7
5a	1	500,000	1.70	s	0.0034
5b	1	500,000	40.98	s	0.0820
5c	1	500,000	15.22	s	0.0304
6a	1	200,000	0.99	s	0.0049
6b	1	200,000	13.08	s	0.0654
6c	1	200,000	18.95	s	0.0948
7-1	1000	300	2.42	s	8.07
7-2	1000	300	6.15	s	20.5
7-3 S	1000	100	25.65	s	256.5
7-3 DF	1000	100	22.35	s	223.5
7-3 BF	1000	100	26.20	s	262.0
11a	250	1000	108.59	s	108.59 (Windows DPMI)
11b	250	1000	75.80	s	75.80 (Windows DPMI)
11c	250	1000	59.71	s	59.71 (Windows DPMI)
11d	2500	1000	5.71	s	5.71
11e	2500	1000	5.72	s	5.72
11f	2500	1000	6.15	s	6.15
11g	5000	1000	9.56	s	9.56
11h	5000	1000	9.01	s	9.01
11i	5000	1000	10.05	s	10.05

Table 11-4: Timings for Intel Pentium (Microsoft 32-bit code)

Test	Size	Repetitions	Total Time	Time/Iteration (ms)
1	1	2,000,000	19.06 s	0.0095
2	1000	300	3.12 s	10.4
	10,000	30	3.53 s	
	50,000	6	3.74 s	
	100,000	3	3.67 s	
3	1000	300	16.58 s	55.3
	10,000	30	16.55 s	
	50,000	6	16.92 s	
	100,000	3	16.62 s	
4 Sim	1000	100	49.99 s	499.9
4 DFS	1000	100	91.48 s	914.8
4 BFS	1000	100	74.94 s	749.4
5a	1	1,000,000	5.45 s	0.0055
5b*	1	200,000	94.35 s	0.472
5c	1	200,000	19.75 s	0.099
6a	1	200,000	3.21 s	0.0161
6b*	1	200,000	118.31 s	0.592
6c	1	200,000	129.81 s	0.649
7-1	1000	300	4.75 s	15.8
7-2	1000	300	18.39 s	61.3
7-3 S	1000	100	47.94 s	479.4
7-3 DFS	1000	100	78.57 s	785.7
7-3 BFS	1000	100	73.12 s	731.2
11a	250	1000	2.28 s	2.28 (GCC: 2.83)
11b	250	1000	2.32 s	2.32 (GCC: 2.46)
11c	250	1000	3.36 s	3.36 (GCC: 5.95)
11d	2500	1000	15.33 s	15.33 (GCC: 24.44)
11e	2500	1000	14.62 s	14.62 (GCC: 22.62)
11f	2500	1000	22.01 s	22.01 (GCC: 60.67)
11g	5000	1000	28.96 s	28.96 (GCC: 52.69)
11h	5000	1000	29.62 s	29.62 (GCC: 44.98)
11i	5000	1000	42.33 s	42.33 (GCC: 124.31)

[*] Note: The floating point number in the list distorts the timing because floating point is very slow on this machine due to the lack of a math coprocessor.

Table 11-5: Timings for SPARCstation LX

12. Compiling and Installing FramepaC

Installing FramepaC consists of the following general steps:

1. Obtain the FramepaC source code distribution
2. Create directories for the source code and compiled library
3. Unpack (if necessary) and copy the source code into the source code directory
4. Edit the Makefile to suit your environment
5. Edit **frconfig.h**, **frpcglbl.h**, and **frmem.h** to suit your needs and environment. Usually only **frconfig.h** will require changes.
6. Type ’make install’ to build and install the FramepaC library; ’make all’ to just build the library and the FramepaC test program.
7. (optional) Type ’make clean’ to remove unnecessary files created during compilation.

!!!

The Makefile contains a number of customization options at the beginning. The main options you will need to set are the locations of various directories and the compiler to use (there are various subfiles for different compilers and target environments; select the appropriate one). You may also wish to inspect the selected sub-makefile, as these also contain settable options.

The main configuration options in **frconfig.h** are whether to associate a value with each symbol, whether to enable ’sideways inheritance’, whether to include support for using a separate server and/or a disk database for backing store on virtual frames, and whether to completely replace the standard memory allocation functions. Secondary options specify the limits on various string sizes, such as the maximum length numbers require when printed out.

The final step, ’make install’, creates an archive file **framepac.a** (or **framepac.lib**) which may be linked with your program. It and all header files needed to use FramepaC are copied into the specified installation directory.

12.1. Portability

FramepaC was jointly developed under both MS-DOS on an Intel 386 (and later a Pentium) and SunOS Unix on a Sun SPARCstation. Since these machines differ in word size, endianness, and numerous other respects, the code should be portable to most systems with few, if any, problems.

13. Bibliography

[1] R.D. Brown.
The MikroKARAT Knowledge Acquisition Environment.
Technical Report CMU-CMT-94-MEMO, Carnegie Mellon University Center for Machine Translation, 1994.

[2] E. Nyberg.
FrameKit User's Guide.
Technical Memorandum, Carnegie Mellon University Center for Machine Translation, 1988.

[3] S. Nirenburg, I. Monarch, T. Kaufmann, I. Nirenburg and J. Carbonell.
Acquisition of Very Large Knowledge Bases: Methodology, Tools, and Applications.
Technical Report 88-108, Carnegie Mellon University Center for Machine Translation, 1988.

[4] I. Monarch.
ONTOS: Reference Manual
1989.

 (Last changed 26-Nov-96) </body> </html>]

Index

Table of Contents

List of Tables