Ctalk Language Reference

Object Oriented Extensions for C

Robert Kiesling

Table of Contents

1 Introduction to Ctalk

Ctalk provides an extensions to the C programming language that allow programmers to use object oriented language features, like class objects, methods, operator overloading, and inheritance, in C programs. This manual describes Ctalk, version 0.0.66.

If you are not familiar with object oriented programming concepts, you might like to read an introductory text on the subject. Using Ctalk requires that you are familiar with the concepts of object oriented programming, as well as the C language. This manual uses object oriented programming terms extensively.

The next chapter describes how to use Ctalk to create programs and use the Ctalk compiler's command line interface. The following chapters describe the class hierarchy, method application programming interface, the interface with C and its run-time libraries, and finally, some simple example programs.

1.1 C Language Conformance

Ctalk conforms as closely as possible to the C language defined by *International Standard ISO/IEC 9899*, commonly known as C99.

1.2 Compiler Compatibility

Ctalk is compatible with GNU GCC, version 2.95 and later. Ctalk recognizes GCC extensions like function and data attributes, and the preprocessor can use directives like `#include_next`. Ctalk recognizes but does not process deprecated language extensions and preprocessor directives like `#sccs` and `#unassert`.

To adapt Ctalk to another compiler, you may need to provide compiler-specific definitions. You will also need to add your compiler's definition file and initialization code, if necessary, to the file, `src/ccompat.c`, which contains the `COMPAT_INCLUDE` definition and functions for specific compilers.

The `ctpp` preprocessor also contains system-specific definitions. Consult its documentation for details.

Some Document Conventions

When the term, "Ctalk," appears in this manual, it refers to the programming language, and when `ctalk` appears, it refers to the Ctalk compiler and libraries.

`identifier`
: Names of classes, variables, methods and messages, file names, and source code examples appear in monospaced type.

variable
: Metasyntactic variables and citations appear in italic or oblique type.

2 Using Ctalk

Ctalk has two parts: a preprocessor and interpreter that translates Ctalk code into C, which a compiler can use to create an executable program, and a run-time library, which the compiler links with the executable.

Ctalk has its own C99 compatible preprocessor, `ctpp`, which is described in its Texinfo manual, `ctpp.info`.

To use `ctalk`, you must provide at least the name of the input file. Normally, you also provide the name of the output file with the '`-o`' option. If you use the conventions followed by GNU C, the output file is normally the base name of the input file with the extension '`.i`', as in this example.

```
$ ctalk myprog.c -o myprog.i
```

The `ctalk` program preprocesses `myprog.c` and translates it into standard C. After `ctalk` has finished, you can compile and link the output of `ctalk` to produce an executable program.

```
$ gcc myprog.i -o myprog -lctalk
```

More conveniently, the `ctcc` command combines these operations, with the appropriate command line options, to build an executable program.

```
$ ctcc myprog.c -o myprog
```

If you need to build a program for debugging, the `ctdb` command builds executables that you can debug with `gdb`. See ⟨undefined⟩ [Debugging], page ⟨undefined⟩.

For more information, refer to the *ctalk(1)*, *ctcc(1)*, *ctdb(1)*, *gcc(1)*, and *gdb(1)* manual pages.

2.1 Command Line Options

`--clearpreload`
: Clear preloaded methods so they can be rewritten.

`-E`
: Preprocess the input and exit.

`-h, --help`
: Print a help message and exit.

`-I` *`dir`*
: Add *dir* to the `ctalk` include search path.

`--keeppragmas`
: Write pragmas untranslated to the output.

`--nolibinc`
: Do not include Ctalk's system headers in the output.

`--nopreload`
: Do not use preloaded methods.

`-o` *`file`*
: Write the `ctalk` output to *file*.

`--printlibdirs`
: Print the library directories and exit.

`--printtemplates`
: Print the templates that Ctalk loads and caches (but does not necessarily send to the output).

`--progress`
: Print dots to indicate Ctalk's progress.

`-P`
: Do not output line number information.

`-s dir`
: Add *dir* to the compiler system include search path.

`-V`
: Print the Ctalk version number and exit.

`-v`

`--verbose`
: Print verbose warnings. This option also sets the `--warnextension`, `--warnduplicatenames` and `--warnunresolvedselexpr` options.

`--warnclasslibs`
: Print the names of class libraries as they are loaded.

`--warnduplicatenames`
: Print a warning when an object duplicates a C variable name. Because of the way Ctalk's just-in-time interpreter works, the front end prints warnings of any duplicate names. The variables and objects need not be in the same scope. Usually, though, Ctalk can make an intelligent decision about how to process objects and variables with duplicate names. This option does not affect errors caused by duplicate symbols or shadowing.

`--warnextension`
: Print warnings for some compiler extensions.

`--warnunresolvedselfexpr`
: Prints warnings if `self` appears in an argument block with either an instance variable label or an unresolvable method label following it. In expressions like these, the class of each element of a collection, when represented by `self` within the argument block, often can't be determined until run time.

  You can specify a class for `self` by placing a class cast expression (described in the section *Class casting*) before the `self` keyword. See ⟨undefined⟩ [Class casting], page ⟨undefined⟩.

  For example, if a program contains an expression like this:

```
List new textLines;

   ...The program adds items to textLines...

textLines map {

  if (self length > 0) {

    ...do something...
```

```
  }

}
```

Then the expression, `self length` would generate a warning due to the label, `length,` because the class membership of, `self,` which represents each successive element of the, `textLines,` list, normally isn't determined until run time, and so the receiver class of `length` is also undetermined.

However, if you know that `textLines` contains only `String` objects, then you can add a class cast expression in the argument block.

```
textLines map {

  if ((String *)self length > 0) {

    ...do something...

  }

}
```

This tells the program to treat `length's` receiver, `self,` as a String object, so it's possible to determine, before the program is actually executed, whether, `self length,` is a valid expression.

3 Classes

Class library files contain class and method definitions and are located in the `classes` subdirectory. After installation, the libraries are in a `ctalk` include subdirectory defined by the macro *CLASSLIBDIR* when the program is built.

3.1 Class Hierarchy

```
Object
 Bitmap
  DisplayFont
   X11Font
     X11Cursor
   X11FreeTypeFont
  X11Bitmap
 Boolean
 Collection
  Array
  List
   AssociativeArray
   SortedList
  Stream
   FileStream
    DirectoryStream
    ReadFileStream
    WriteFileStream
   TerminalStream
    ANSITerminalStream
    Win32TerminalStream
    X11TerminalStream
   NetworkStream
    TCPIPNetworkStream
     TCPIPNetworkStreamReader
     TCPIPNetworkStreamWriter
    TCPIPV6NetworkStream
     TCPIPV6NetworkStreamReader
     TCPIPV6NetworkStreamWriter
    UNIXNetworkStream
     UNIXNetworkStreamReader
     UNIXNetworkStreamWriter
 TreeNode
 Event
  Application
   ClassLibraryTree
   GLUTApplication
   ObjectInspector
   LibrarySearch
```

```
 Exception
  SystemErrnoException
 InputEvent
 SignalEvent
  SignalHandler
Expr
 CFunction
Magnitude
 Character
  String
 Float
 Integer
  CTime
    CalendarTime
 LongInteger
 Pen
 Point
  Line
   Rectangle
  Circle
Method
Pane
 ANSITerminalPane
  ANSIWidgetPane
   ANSIButtonPane
   ANSILabelPane
   ANSIListBoxPane
   ANSIMessageBoxPane
   ANSIProgressBarPane
   ANSIScrollingListBoxPane
   ANSIScrollPane
   ANSITextBoxPane
   ANSITextEntryPane
   ANSIYesNoBoxPane
 X11Pane
  GLXCanvasPane
  X11PaneDispatcher
   X11CanvasPane
    X11ButtonPane
    X11CheckBoxPane
    X11LabelPane
    X11ListPane
    X11MessageBoxPane
    X11ScrollBarPane
    X11TextEntryPane
    X11YesNoBoxPane
     X11TextEntryBox
```

```
          X11ListBox
           X11FileSelectDialog
        X11TextPane
         X11TextEditorPane
      Symbol
       Key
       Vector
```

3.2 `Object` class

Ctalk defines the `Object` class by default at run time. All other classes are subclasses of `Object`.

Instance Variables

`value` The value of the object. For classes that correspond to the C data types: `Character`, `Float`, `Integer`, `LongInteger`, and `String`, ctalk formats the value as it would appear as if output by the C `printf` function.

When you use one of these classes in a C expression, ctalk translates the object's value to its C data type.

Other classes may define `value` to return class specific data.

Null Objects

To help make Ctalk compatible with C, Ctalk defines the macro STR_IS_NULL, which checks whether the object's value evaluates to false in C. That means that objects that have the value '`(null)`', are an empty string, or if the object's value is a NULL pointer, all evaluate similarly. The STR_IS_NULL macro is defined in the include file `ctalkdefs.h`, so to use it, add the following line at the start of a class library.

```
#include <ctalk/ctalkdefs.h>
```

Ctalk also assigns an object the value of '`(null)`' when an object is created or when Ctalk encounters a C NULL in an expression (which is a macro that expands to '`((void *)0)`').

Instance Methods

`! (void)` Overloads the '`!`' prefix operator.

`!= (Object` *arg*`)`
Returns a `Boolean` value of False if the receiver and the argument are the same object, True otherwise. Also returns False if the receiver and argument's values both evaluate to False.

`& (void)` When used to overload C's "address of" ('`&`') operator, is synonymous with the `addressOf` method, below.

`-> (String` *member_name*`)`

Given the name of an a C `OBJECT *` member name, like `__o_name`, `__o_classname`, `__o_class` and so on, return a `String`, `Integer`, or `Symbol` with the value of that member. See ⟨undefined⟩ [OBJECT typedef], page ⟨undefined⟩.

The class of the returned object depends on which OBJECT * is selected. Some of the results are returned as references contained in Symbol objects, which avoids unexpectedly changing the target object.

OBJECT Member	Expression	Class of Result Object
__o_name	obj -> __o_name	String
__o_classname	obj -> __o_classname	String
__o_superclassname	obj -> __o_superclassname	String
__o_value	obj -> __o_value	String
__o_class	obj -> __o_class	<obj's> class
__o_superclass	obj -> __o_superclass	<obj's> superclass
instancevars	obj -> instancevars	Symbol
classvars	obj -> classvars	<obj's> class
__o_p_obj	obj -> __o_p_obj	<obj's> parent's class
scope	obj -> scope	Integer
attrs	obj -> attrs	Integer
nrefs	obj -> nrefs	Integer

The method sets the OBJECT_IS_DEREF_RESULT attribute on objects that the method creates.

The result of retrieving an object's instance or class variables depends on the object. If the object is not a class object, then the `classvars` member will be NULL. The class of an instance variable is the parent object, so the Ctalk library returns the address of the first instance variable. If you want to check each variable, it is much easier to use the methods `mapInstanceVariables` or `mapClassVariables`, below.

Examples

```
/* Set a Symbol object and an Object instance to
   another object's class, and display the classes'
   name. */
int main () {

  String new myObject;
  Symbol new optr;
  Object new classObj;

  myObject = "String"; /*Needed by classObject, below.*/

  printf ("%p\n", myObject -> __o_class);
  printf ("%s\n", myObject -> __o_class -> __o_name);

  *optr = myObject -> __o_class;
  printf ("%p\n", (*optr) -> __o_class);
```

```
  printf ("%s\n", (*optr) -> __o_class -> __o_name);

  classObj = (*optr) classObject;
  printf ("%p\n", classObj);
  printf ("%s\n", classObj -> __o_name);

  classObj = myObject classObject;
  printf ("%p\n", classObj);
  printf ("%s\n", classObj -> __o_name);

}

/* To print an object's value. */
int main () {

  String new s;

  s = "StringValue";

  printf ("%s\n", s -> __o_value);

}

/* To save an object's scope to an Integer object, and
   print them. */
int main () {

  String new s;
  Integer new scopeInt;


  scopeInt = s -> scope;

  printf ("%d == %d\n", s -> scope, scopeInt);

}


/* To save an object's attributes to an Integer and
   print them. */

int main () {

  Integer new attrsInt;
  List new l;
  String new member1;
  String new member2;
```

```
    l = member1, member2;

    /* Check for an OBJECT_IS_MEMBER_OF_PARENT_COLLECTION attribute
       (currently its definition is (1 << 7), or 128), which is set
       in each of the list's Key objects. (The '->' operator in the
       expressions has a higher precedence than '*', so parentheses
       are necessary.) */
    attrsInt = (*l) -> attrs;

    printf ("%d == %d\n", (*l) -> attrs, attrsInt);
}
```

Note: The use of `__o_classname` and `__o_superclassname` as separate object member is being superceded by the `CLASSNAME` and `SUPERCLASSNAME` definitions. The `->` method recognizes both the old and new names, but if `->` is used with an `OBJECT *` as a C operator, then the program must use the `CLASSNAME` or `SUPERCLASSNAME` macros. See ⟨undefined⟩ [CLASSNAMEMacro], page ⟨undefined⟩.

`= (Object` *`new_object`*`)`
: A basic assignment method. Assigns *new_object* so that the receiver label can refer to it.

`== (Object` *`arg`*`)`
: Returns a `Boolean` value of True if the receiver and the argument are the same object, False otherwise.

  As a special case, the method returns True if the receiver and argument's values evaluate to a C NULL. Refer to the description of null objects above. See ⟨undefined⟩ [NullObjects], page ⟨undefined⟩.

`addInstanceVariable (char *`*`name`*`, OBJECT *`*`value`*`)`
: Add instance variable *name* with *value* to the receiver.

`addressOf (void)`
: Return a `Symbol` that contains the address of the receiver. This method is functionally equivalent to the C language '`&`' operator.

`asFloat (void)`
: Return a `Float` object with the value of the receiver.

`asString (void)`
: Return a `String` object of the receiver.

`asSymbol (void)`
: If the value of the receiver is an object reference, return a Symbol object with that reference. Otherwise, return a Symbol object with a reference to the receiver.

`backgroundMethodObjectMessage (Method` *`method_object`*`)`

Send the message defined in *method_object* to the receiver as a new process, which runs concurrently until *method_object* returns.

The argument, *method_object*, is a previously defined method which takes no arguments. The return value, if any is not saved when the background process exits.

Here is a brief example that opens an `ObjectInspector` on a process.

```
Object instanceMethod bgMethod (void) {
  self inspect "bgMethod> ";
  printf ("bgMethod printed this.\n");
}

int main () {
  Integer new i;
  Method new bgMethodObject;
  int status;

  bgMethodObject definedInstanceMethod "Object", "bgMethod";
  i backgroundMethodObjectMessage bgMethodObject;

  wait (&status);  /* Returns when the background */
                   /* process exits.              */

}
```

`basicNew (char *`*`name`*`)`
`basicNew (char *`*`name`*`, char *`*`value_expr`*`)`
`basicNew (char *`*`name`*`, char *`*`classname`*`, char *`*`superclassname`*`, char *`*`value_expr`*`)`

If given with one argument, the receiver must be a class object, and the method creates a new object with the name given as the argument and a value of '`(null)`'.

With two arguments, create a new object with the name and value given in the arguments, and with the class and superclass of the receiver, which needs to be a class object in this case also.

If the method is used with four arguments, create a new object with the name, class, superclass and value given by the arguments - in this case, `basicNew` doesn't use the class and superclass of the receiver.

In each case, the newly created object contains all of the instance variables defined by the class object and has a scope of LOCAL_VAR|METHOD_USER_OBJECT and a reference count of 1.

Note: The addition of the receiver's instance variables means you need to be careful when the receiver is not a class object. The `basicNew` method does not

check for cascading or circular instance variables. So if you're creating and then pushing a lot of items, make sure the method's receiver is a class object, and assigning the new object using a `Symbol` is more robust that assigning other types of objects.

That is, a set of statements like

```
*tokSym = String basicNew "token", valuebuf;
myList push *tokSym;
```

is much more reliable than, say

```
myTok = myString basicNew "token", valuebuf;
myList push myTok;
```

`become (OBJECT *original_object)`
Make the receiver a duplicate of the argument object.

`callStackTrace (void)`
Print a call stack trace.

`class (char *classname)`
Declare a new class `classname`. The class declaration consists of a superclass name, the `class` keyword, the name of the new class, and an optional documentation string. For example;

```
FileStream new WriteFileStream;

... or, with a docstring between the class name and semicolon ...

FileStream new WriteFileStream
"Defines the methods and instance variables that write data to files. Also
defines the class variables stdoutStream and stderrStream, which are the
object representation of the standard output and standard error streams.";
```

`className (void)`
Return a `String` object containing the class name of the receiver.

`classObject (void)`
Return the class object named by the receiver's value. Normally the receiver should be a `String` object - the method uses the C value of the receiver to retrieve the class object by its name. If the receiver is a class object, the method returns the receiver itself.

`copy (Object source_object)`
Copy *source_object* to the receiver. This method replaces the receiver, which received the message, with a completely new copy of the *source_object* argument. The original receiver object no longer exists.

`decReferenceCount (void)`
: Decrease the receiver object's reference count by one.

`delete (void)`
: Delete the receiver object and its instance variables.

  For some of the details of how Ctalk deletes objects, refer to the `__ctalkDeleteObject ()` API function. See ⟨undefined⟩ [__ctalkDeleteObject], page ⟨undefined⟩.

`disableExceptionTrace (void)`
: Disable method walkback displays when handling exceptions.

`docDir (void)`
: Returns a `String` that contains the path where the Ctalk-specific documentation is installed on the system (i.e., documentation other than man pages and Texinfo manuals).

`dump (void)`
: Create an `ObjectInspector` object and print the object's contents on the terminal. This is a convenience method for the ObjectInspector : formatObject method.

`enableExceptionTrace (void)`
: Enable method walkback displays when handling exceptions.

`getReferenceCount (void)`
: Return an Integer with the receiver's reference count.

`hasPointerContext (void)`
: Returns a `Boolean` value of True if the receiver appears in a pointer context; i.e., on the left-hand side of an assignment operator in an expression with some dereferencing method (usually a "`*`"), as in this example.

```
Symbol new myNewObjectPtr;
Symbol new myOldObjectPtr;


*myNewObjectPtr = myOldObjectPtr;
```

`incReferenceCount (void)`
: Increase the receiver object's reference count by one.

`inspect (void)`
`inspect (String` *`promptStr`*`)`
: Open an inspector on the receiver object. At the prompt, typing '`?`' or '`help`' displays a list of the inspector's commands. If given a String object as its argument, displays the string as the inspector's command prompt. This method is a shortcut for `inspect` in `ObjectInspector` class. See ⟨undefined⟩ [ObjectInspector_inspect], page ⟨undefined⟩.

`is (char *`*`className`*`)`

Return `TRUE` if the receiver is a member of the class given in the argument, `FALSE` otherwise. *Note:* Don't use an expression like, '`obj is Class`'. The argument must be a valid, defined class, which simplifies the method considerably.

To determine if an object is a class object, use `isClassObject`, which is mentioned below. This example illustrates the difference.

```
if (myObject is String) {   /* OK - String is a defined class. */
 ...
}

if (myObject is Class) {    /* Not OK - 'Class' is not a defined class. */
 ...
}

if (myObject isClassObject) {  /* OK - Just check whether the object is
 ...                              its own class object. */
}
```

`isClassMethod (char *`*`classMethodName`*`)`

Return `TRUE` if the receiver's class defines an instance method named *classMethodName*, `FALSE` otherwise. If the receiver is an instance or class variable, look for the method in the class of the receiver's value.

`isClassObject (void)`

Returns a Boolean value of True if the receiver is a class object, False otherwise.

`isInstanceMethod (char *`*`instanceMethodName`*`)`

Return `TRUE` if the receiver's class defines an instance method named *instanceMethodName*, `FALSE` otherwise. If the receiver is an instance or class variable, look for the method in the class of the receiver's value.

`isInstanceVariable (char *`*`instanceVariableName`*`)`

Return `TRUE` if the receiver has an instance variable named *instanceVariableName*, `FALSE` otherwise.

`instanceMethod (char *`*`alias`*`, char *`*`name`*`, (`*`args`*`)`

Declare a method of the receiver's class with the name *name*, with the arguments *args*, and, optionally, with the alias *alias*. See ⟨undefined⟩ [Methods], page ⟨undefined⟩.

`isNull (void)`

Return `TRUE` if the receiver is a null result object, `FALSE` otherwise.

`isNullValue (void)`

Return `TRUE` if the receiver's value is `NULL`, '`(null)`', or '`0`', `FALSE` otherwise. Note that this method is here for compatibility with older programs. New programs should not need it.

`isSubClassOf (String classname)`
: Returns True if the reciever's is a member of *classname* or one of its subclasses, False otherwise.

`libraryPath (void)`
: Return the file path name of the receiver's class library.

`mapClassVariables (OBJECT *(*fn)())`
: Apply the argument, *fn* to each of the class variables in the receiver's class object. The argument may be either a method or a code block. Refer to the *Ctalk Tutorial* for examples of inline method messages using methods like `map` and `mapInstanceVariables`.

`mapInstanceVariables (OBJECT *(*fn)())`
: Apply the argument, *fn*, to each of an object's instance variables. The argument may be either a method or a code block. Refer to the *Ctalk Tutorial* for examples of inline method messages using methods like `map` and `mapInstanceVariables`.

`methodObjectMessage (Method method_object)`
`methodObjectMessage (Method method_object, Object arg1, Object arg2)`
: Perform a method call with a `Method` class object. The *method_object* argument and its own arguments must have been previously defined. See ⟨undefined⟩ [Method], page ⟨undefined⟩.

`methodPoolMax (void)`
`methodPoolMax (Integer new_max_size)`
: Set (with one argument) and retrieve (with no arguments) the maximum number of objects in each method's object pool. When a method's pool size reaches this number of objects, the pool deletes the the oldest object in the pool to make room for the new object.

  Generally, the default pool size is suitable for the language tools and demonstration programs that come packaged with Ctalk. Some of the test programs in the `test/expect` subdirectory that run through many iterations (i.e., thousands of iterations) require a larger pool size. This is especially true if a program uses many C variables when iterating through its operations, and whether the C variables are simply scalar or constant values (e.g., ints, doubles, and literal strings), and whether the variables are pointers to objects in memory.

  The default maximum pool size is set when the Ctalk library is built. This size is reported in the `configure` status report during the build process.

`methodObjectMessage (Method methodObject)`
`methodObjectMessage (Method methodObject, Object arg1, Object arg2)`
: With three arguments, the method uses the second and third arguments as the method instance's arguments, without previous calls to the `withArg` method (defined in `Method` class).

  Method instances with two arguments are commonly used in graphical event dispatchers. They're found mostly in `X11PaneDispatcher` class, and the three-parameter form of `methodObjectMessage` allows the method instance calls to be more efficient.

The following examples of method calls shows how to call method instances with and without using the `withArg` method first.

This example is from `X11PaneDispatcher` class.

```
  /* This example pushes the arguments separately. */
__subPane handleKbdInput withArg __subPane;
__subPane handleKbdInput withArg __event;
__subPane methodObjectMessage __subPane handleKbdInput;

  /* The three-argument form of methodObjectMessage allows
     passing two arguments to a method instance with only
     a single expression. */
__subPane methodObjectMessage __subPane handleKbdInput,
  __subPane, __event;
```

This example is the `demos/ipc/sockproc.c` demonstration program, which shows how to define and start a method instance as a separate process.

```
#define SOCK_BASENAME "testsocket.$$"

String new gSocketPath; /* The socket path is common to both
   processes. */

Object instanceMethod startTimeServer (void) {
  UNIXNetworkStreamReader new reader;
  String new utcTimeString;
  CalendarTime new timeNow;
  SystemErrnoException new ex;

  reader openOn gSocketPath;
  if (ex pending) {
    ex handle;
    return ERROR;
  }

  while (1) {
    utcTimeString = reader sockRead;
    if (ex pending) {
      ex handle;
      return ERROR;
    }
    if (utcTimeString length > 0) {
      timeNow = utcTimeString asInteger;
      timeNow localTime;
      printf ("%s\n", timeNow cTimeString);
    }
```

```
  }
}

int main (int argc, char **argv) {

  CTime new thisTime;
  CTime new prevTime;
  UNIXNetworkStream new strObject;
  UNIXNetworkStreamWriter new client;
  Method new timeServerInit;
  Integer new childPID;
  SystemErrnoException new ex;
  Object new procRcvr;

  gSocketPath = strObject makeSocketPath SOCK_BASENAME;

  timeServerInit definedInstanceMethod "Object",
    "startTimeServer";
  childPID =
    procRcvr backgroundMethodObjectMessage timeServerInit;

  sleep (1); /* Wait a sec while the reader process creates the
socket. This interval might be system dependent. */

  client openOn gSocketPath;
  if (ex pending) {
    ex handle;
    return ERROR;
  }

  prevTime utcTime;

  while (1) {

    thisTime utcTime;

    if (thisTime != prevTime) {

      client sockWrite (thisTime asString);

      if (ex pending) {
ex handle;
return ERROR;
      }

      prevTime = thisTime;
```

```
    }
    usleep (1000);
  }
  exit (0);
}
```

`methods (Boolean` *`findClassMethods`*`)`
: Return a string containing the methods defined by the receiver's class, one method per line. If *findClassMethods* is True, return a list of class methods in the class. If the argument is False, return a list of the class's instance methods.

`name (void)`
: Return the name of the receiver as a `String` object.

`new (`*`newObject1`*`, `*`newObject2`*`, `*`newObject3...`*`)`
: Create an object or object's of the receiver's class with the names *newObject1*, etc. For example:

```
String new myString1, myString2, myString3;
```

`printSelfBasic (void)`
: Print the contents of the receiver to the program's standard error output.

`setReferenceCount (Integer` *`refCnt`*`)`
: Set the receiver's reference count to the value given as the argument. Note, however, that the reference count given as the argument does not include any changes in the reference count as this method is executed. If you're unsure about how to adjust the reference count it's safer to use `incReferenceCount` and `decReferenceCount` instead, or use the *__objRefCntSet ()* library function directly. See ⟨undefined⟩ [objRefCntSet], page ⟨undefined⟩.

`sizeof (`*`Object`*`)`
: Overloads the C `sizeof` operator when the argument is an object. If the argument is a C variable or a type expression, then Ctalk uses the compiler's `sizeof` operator.

`superclassName (void)`
: Return a `String` that contains the name of the object's superclass, or '`(null)`' if the receiver is an instance of `Object` class, which has no superclass.

`traceEnabled (void)`
: Return `TRUE` if exception walkback displays are enabled, `FALSE` otherwise.

`value (void)`
: Returns the value of the receiver object. When you use an object without a method, ctalk uses the `value` message to return the object's value. For example,

```
printf ("%s", stringObject);
```

is the same as,

```
printf ("%s", stringObject value);
```

3.3 `Bitmap` class

3.4 `DisplayFont` class

3.5 `X11Font` class

A `X11Font` object provides information about a X Window System font, like its resource ID, width, height, and X Logical Font Descriptor (XLFD) string. Normally, a pane or X window does not use font glyphs directly but uses Pane methods to display text using the font resource. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

For variable width fonts, the `maxWidth` instance variable provides the width of the widest character in the set.

For more information about X's font naming scheme, the *xfontsel(1)* manual page is a good place to begin.

Instance Variables

`ascent`

`descent` `Integer` objects that contain the number of pixels the character set extends above and below the baseline. These dimensions are typically used to determine interline spacing.

`fontDesc` A `String` containing the X Logical Font Descriptor of the receiver's font. After a call to `getFontInfo` (below), contains the XLFD of the actual font.

`fontId` An `Integer` that contains the X Window System's resource ID of the font.

`height` An `Integer` that contains the sum of the maximum descent and ascent; that is, the height of the font's tallest character.

`maxLBearing`
An `Integer` that contains the maximum distance in pixels from a character's drawing origin to its glyph.

`maxRBearing`
An `Integer` that contains the maximum distance in pixels between the right side of a font's character glyphs to the right-hand edge of the character space.

`maxWidth` An `Integer` that contains the width of the font's widest character glyph. Note that these metrics' uses differ for different fonts, and can vary a lot, especially in foreign language character sets. Generally, though, for Western, monospaced fonts, the width in pixels of a character plus its horizontal spacing is:

```
myFont maxLBearing + myFont maxWidth;
```

Instance Methods

`getFontInfo (String` *`fontDesc`*`)`
Fills in the font metrics (height, maxWidth, ascent, descent, X font ID) for the font named by *fontDesc*. Also fills in the receiver's `fontDesc` instance variable with the string given as the argument.

`textWidth (String text)`
: Returns an `Integer` with the width of *text* in pixels when rendered in the reciever's font. The program should call the `getFontInfo` method before calling this method.

3.6 `X11Cursor` class

The `X11Cursor` class represents cursor objects and provides wrapper methods that create X11 cursor resources. A `X11Cursor` object's value is the ID of the newly created cursor resource.

To display the cursor in a `X11Pane` object's window, see the `useCursor` method in `X11Pane` class. This example shows the sequence of expressions. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

```
X11Pane new xPane;         /* Create our objects. */
X11Cursor new waitCursor;

...

waitCursor watch;          /* Get the resource ID of a wait cursor
                              (normally, a watch or a spinner). */
...

xPane useCursor waitCursor;  /* Display the wait cursor on the xPane's
                                window. */
```

The `defaultCursor` method (class `X11Pane`) restores the window's cursor to its parent window's cursor, which is the default cursor for new windows. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

```
xPane defaultCursor;
```

Instance Methods

`arrow (void)`
: Sets the cursor to the upward-left pointing arrow that is the default for most new windows.

`grabMove (void)`
: Creates a grab cursor.

`scrollArrow (void)`
: Creates a scrollbar vertical double arrow cursor.

`watch (void)`
: Creates a cursor that displays a small watch.

`xterm (void)`
: Creates a xterm cursor. That is, the vertical bar used for editing text.

3.7 X11FreeTypeFont class

This class contains the methods that handle FreeType fonts using the Xft and Fontconfig libraries. In classes and methods that use fonts, FreeType fonts take priority if they are available and the program has initialized the library with the `initFontLib` method, which is described in further detail below. Otherwise, applications use the X Window System's builtin bitmap fonts.

Internally Ctalk uses the Fontconfig library to locate the system's `fonts.conf` file. The `fonts.conf` file's default location is configured when the library is built. Generally, the `font.conf` file's path is `/etc/fonts/fonts.conf` or `/usr/local/etc/fonts/fonts.conf`.

However, an alternate `fonts.conf` pathname may be set in the environment with the '`FONTCONFIG_FILE`' environment variable. If its value begins with a '`~`', the path is relative to the user's '`HOME`' directory. If the name doesn't start with a '`/`', then the pathname is relative to the fontconfig libraries' default installation directory. This directory can be changed by setting the '`FONTCONFIG_PATH`' environment variable to an alternate directory.

Classes that use X Window System graphics (that is, `X11Pane` and its subclasses) can use either the X libraries' built-in bitmap fonts (via the pane's `fontVar` instance variable), or the scalable Freetype fonts (via the pane's `ftFontVar` instance variable).

The default is to use X's built in bitmap fonts. These fonts are initialized when the program connects to the display server.

However, to enable drawing with scalable fonts, programs need to call the `initFontLib` method during program initialization, with a line similar to this one.

```
myMainWindow ftFontVar initFontLib;
```

Fontconfig Font Selection

Fontconfig font descriptions have their own syntax. A Fontconfig pattern has the following elements.

```
<fontfamily>[-<fontsize>][:[[<name>=]<value>] ...]
```

Here are a few examples.

```
DejaVu Sans Mono
Nimbus Sans L-12
Ubuntu Mono-10.0:italic
FreeMono-10:slant=italic:weight=bold
Ubuntu Mono-10.0:italic
Roboto-12:style=bold
```

These font descriptors work with the `selectFontFromFontConfig` method. The receiver, of course, must be a `X11FreetypeFont` object, or the pane object's `ftFontVar` instance variable. Here is an example

```
myWindow ftFontVar selectFontFromFontConfig "DejaVu Sans Mono-12"
```

The Fontconfig user guide describes the library's font selection syntax. The documentation is available from many software package archives and also on the web at http://freedesktop.org/fontconfig.

Basic `fonts.conf` Files

Ctalk also uses a subset of the `fonts.conf` tag syntax and the older style of finding a system's or users' `fonts.conf` file:

1. The file path named by the '`$XFT_CONFIG`' environment variable.
2. '`$HOME/.fonts.conf`'
3. '`$XDG_CONFIG_HOME/fontconfig/fonts.conf`'
4. `/etc/fonts/fonts.conf`
5. The file `/usr/X11R6/lib/X11/XftConfig`

If you want to write your own font config file, the Ctalk libraries can use a subset of the `fonts.conf` markup.

In this case, the entries simply cause the libraries to add the fonts in the directories to the internal font database.

```
<fontconfig>
<dir>/usr/share/fonts/truetype/dejavu</dir>
<dir>/usr/share/fonts/truetype/freefont</dir>
<dir>/usr/share/fonts/type1/gsfonts</dir>
</fontconfig>
```

There is a sample '`XftConfig`' file in the Ctalk source distribution, `doc/XftConfig.sample`. The *fonts.conf (5)* manual page and the Xft user and developer documentation provide even more information.

Further Information About Fontconfig and libXft

For further information about the the Fontconfig libraries, refer to: https://freedesktop.org/fontconfig. Information about Xft is available at http://freedesktop.org/wiki/Software/Xft.

Instance Variables

`family` A `String` that contains the family of the currently selected font. See ⟨undefined⟩ [String], page ⟨undefined⟩.

`fgRed`

`fgGreen`

`fgBlue`

`fgAlpha` The parameters of the font's RGBA color. The variables are `Integers` and can have a range of 0 - 0xffff. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`rotation`

`scaleX`

`scaleY` `Float` instance variables that control the scaling and rotation of the rendered fonts.

`slant`

`weight`

`dpi` `Integers` that contain the slant, weight, and dpi resolution of the receiver font. For a description of the Xft library's constants, refer to the `selectFont` method entry, below. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`pointSize`
A `Float` that contains the font's point size. See ⟨undefined⟩ [Float], page ⟨undefined⟩.

`ascent` An `Integer` that contains the font's height in pixels above the baseline.

`descent` An `Integer` that contains the font's height in pixels below the baseline.

`height` An `Integer` that contains the dimensions in pixels of the font's total height, both above and below the baseline, and any extra vertical spacing.

`maxAdvance`
An `Integer` that contains the maximum width in pixels of a font's characters horizontal dimensions.

Instance Methods

`alpha (Integer` *value*`)`
Set the current Xft font's foreground alpha channel. The argument *value* must be an unsigned short int; i.e., in the range 0 - 65535 (0xffff hex).

`blue (Integer` *value*`)`
Set the current Xft font's foreground blue channel. The argument *value* must be an unsigned short int; i.e., in the range 0 - 65535 (0xffff hex).

`currentDescStr (void)`
Returns a `String` containing a font descriptor for the currently selected font. The descriptor contains the attributes that the Ctalk libraries use to select fonts: family, point size, slant, and weight.

For the complete descriptor that the Freetype library derives from the selected font's pattern, refer to the `X11FreeTypeFont : selectedFont` method.

`errors (void)`
This is a shortcut for

```
myFont notifyLevel XFT_NOTIFY_ERRORS
```

Refer the the `X11FreeTypeFont : notifyLevel` method for more information.

`green (Integer value)`
: Set the current Xft font's foreground green channel. The argument *value* must be an unsigned short int; i.e., in the range 0 - 65535 (0xffff hex).

`initFontLib (void)`
: Initializes the font library and sets the reciever's instance variables to the default selected font. Apps need to use this method before most of the other other methods in this class.

  If the system is not configured to use outline fonts then the library prints a message and exits the program with a non-zero value. See ⟨undefined⟩ [ctalkXftInitLib], page ⟨undefined⟩. If the library is already initialized, then the method returns `0`.

`isMonospace (void)`
: Returns a read-only `Boolean` value of true if the application's selected font is monospace, false otherwise.

`libIsInitialized (void)`
: Returns a non-zero `Integer` value if the Xft libraries are initialized and a font is selected.

`listFonts (String pattern)`
: List the font descriptors in the FreeType library that contain *pattern*. If *pattern* is an empty string '`""`' or a '`*`', match every font descriptor.

`loadNotify (void)`
: This is a shortcut for

```
myFont notifyLevel XFT_NOTIFY_LOAD
```

  This causes methods like `selectFont` to display the family and size given to them, and a brief font descriptor that contains the font parameters that Ctalk uses to select fonts: family, point size, slant, and weight.

  Refer to the `X11FreeTypeFont : notifyLevel` method for a description of information and error message levels. To return a string containing the font descriptor that the Freetype libraries derive from the selected font's pattern, refer to the `X11FreeTypeFont : selectedFont` method.

`namedX11Color (String colorName)`
: Sets the selected font's red, green, and blue values from the X11 color name given as the argument.

  Note: For a list of the colors that the X Window System recognizes, consult the file `rgb.txt`, which is often stored with the server's configuration files when the GUI is installed.

`notifyLevel (Integer level)`
: Set the Xft libraries' notification level to *level*. Ctalk defines the following constants in `/ctalk/ctalkdefs.h`.

```
XFT_NOTIFY_NONE
XFT_NOTIFY_ERRORS
XFT_NOTIFY_LOAD
XFT_NOTIFY_VERBOSE
```

These definitions produce the following information and warning messages.

`XFT_NOTIFY_NONE`
: The font libraries produce no output when the program is run.

`XFT_NOTIFY_ERRORS`
: The font libraries display a warning if the library's selected font does not match the user or appication defined font specification.

`XFT_NOTIFY_LOAD`
: The font library displays a message containing information about the requested font, and a brief font descriptor that contains the attributes that the Ctalk libraries use to select fonts: family, point size, slant, and weight.

`XFT_NOTIFY_VERBOSE`
: The font library displays a message that contains information about the requested font, and the complete font descriptor that the font libraries derive from the selected font's patterm.

Note: programs should call this method before launching any processes. Generally this is before the `X11Pane : openInputStream` method. Otherwise, the program only displays font loads for the process it was called from.

`qualifyName (String` *`xftpattern`*`)`
: Returns a fully qualified `String` containing a fully qualified Xft font descriptor that matches *xftpattern*.

`quiet (void)`
: Sets the font libraries' reporting level to XFT_NOTIFY_NONE. For a description of the libraries' notification levels, refer to the `notifyLevel` method.

`red (Integer` *`value`*`)`
: Set the current Xft font's foreground red channel. The argument *value* must be an unsigned short int; i.e., in the range 0 - 65535 (0xffff hex).

`saveSelectedFont (void)`
: This method does the work of setting the receiver's instance variables to the values of the selected font.

`selectedFont (void)`
: Returns the `String` with the descriptor of the selected font. Because a complete font descriptor contains a lot of information, this method can generate a lot of output.

`RGBAColor (void)`
`RGBAColor (int` *`red`*`, int` *`green`*`, int` *`blue`*`, int` *`alpha`*`)`
: If given with no arguments, sets the current font's color to the values of the receiver's '`ftRed, fgGreen, fgBlue,`' and '`fgAlpha`' values. If the method is

used with arguments, set's the current font's color to their values. The values are `Integers` and can have a range of 0-0xffff.

`selectFont (void)`
`selectFont (String` *`family`*`, Integer` *`slant`*`, Integer` *`weight`*`, Integer` *`dpi`*`, Float` *`pointSize`*`)`

If used without arguments, selects the font named by the receiver's instance variables. If the statement provides arguments, only the *family* argument needs to describe an actual font family; that is, it has no default value. Defaults for the other arguments are described below. In this case, the method sets the receiver's instance variables to the selected font.

For the `slant` and `weight` arguments, the Xft library defines the constants and their meanings as the following.

```
     SLANT
0    Roman
100  Italic
110  Oblique

     WEIGHT
0    Light
100  Medium
180  Demibold
200  Bold
210  Black
```

The *dpi* parameter should be set to the resolution of the display.

An easier way to describe a font may be to use the `selectFontFromFontConfig` method, which takes a FontConfig format font specification as its argument. See ⟨undefined⟩ [selectFontFromFontConfig], page ⟨undefined⟩.

Refer to the `X11FreeTypeFont : notifyLevel` method for a description of information and error message levels. To return a string containing the font descriptor that the Freetype libraries derive from the selected font's pattern, refer to the `X11FreeTypeFont : selectedFont` method.

`selectFontFromXLFD (String` *`xlfd`*`)`

Similar to `selectFont`, above, except that its argument is a string in XLFD format. That is, the method translates the XLFD argument into the Xft library's font metrics. This makes possible font descriptions like the following.

```
-*-DejaVu Sans-medium-r-*-*-12-72-72-*-*-*-*-*
```

As with all of the `selectFont*` methods, refer to the `X11FreeTypeFont : notifyLevel` method for a description of information and error message levels. To return a string containing the font descriptor that the Freetype libraries derive from the selected font's pattern, refer to the `X11FreeTypeFont : selectedFont` method.

`selectFontFromFontConfig (String` *`font_config_string`*`)`

Selects a font using the FontConfig specification given as its argument.

Refer to the beginning of the `X11FreeTypeFont` section for a description of Fontconfig descriptors.

Again, as with all of the `selectFont*` methods, refer to the `X11FreeTypeFont : notifyLevel` method for a description of information and error message levels. To return a string containing the font descriptor that the Freetype libraries derive from the selected font's pattern, refer to the `X11FreeTypeFont : selectedFont` method.

`textHeight (String` *text*`)`
: Return an `Integer` with the height in pixels of *text* in the currently selected FreeType font. If the Xft library is not available or not initialized, the method returns 0.

`textRBearing (String` *text*`)`
: Return an `Integer` with the right bearing (the distance between the rightmost segment of a character glyph and the right edge of its character box) in pixels of *text* in the currently selected FreeType font. If the Xft library is not available or not initialized, or the application has not selected a font, the method returns 0.

`textWidth (String` *text*`)`
: Return an `Integer` with the width in pixels of *text* in the currently selected FreeType font. If the Xft library is not available or not initialized, the method returns 0.

`verbose`
: Causes the Ctalk libraries to output verbose information about font selection and loading, like the complete font descriptor that the Freetype libraries derive from the currently selected font's pattern. Refer to the `X11FreeTypeFont : notifyLevel` method for more information about this class' notification levels..

`verbosity (void)`
: Returns an `Integer` with the Xft libraries' reporting level. The possible values, which are defined in `ctalk/ctalkders.h`, are:

```
XFT_NOTIFY_NONE
XFT_NOTIFY_ERRORS
XFT_NOTIFY_LOAD
XFT_NOTIFY_VERBOSE
```

3.8 `X11Bitmap` class

A `X11Bitmap` object is similar to a Bitmap object - it contains the address of a bitmap in memory. In addition, a `X11Bitmap` object has its own X11 graphics context for graphics operations.

Instance Variables

`backgroundColor`
: A `String` that contains the name of a X color. Setting this variable does not affect the color of a displayed bitmap - programs should call the `X11Bitmap : background` method, which also sets this variable.

`depth`
: An `Integer` that contains the depth of the bitmap.

`displayPtr`
: A `Symbol` that contains the address of the connection to the X server that was returned by a call to *XOpenDisplay*. This is generally the address of the main window pane's `displayPtr` instance variable, and in this class, it is expressly set in the `create` method.

`height`
: An `Integer` that contains the height of the bitmap.

`modal`
: A `Boolean` that helps determine how Ctalk draws graphics on the X11Bitmap. True for X11Bitmap objects that are displayed in popup windows, false otherwise.

`parentDrawable`
: An `Integer` that contains the window ID of the bitmap's parent drawable. Normally this is set by the `create` method, below.

`width`
: An `Integer` that contains the width of the bitmap.

`xGC`
: A `Symbol` object that contains the address of the X11 graphics context.

`xID`
: The X resource ID of the Pixmap associated with this object.

Instance Methods

`background (String` *`color`*`)`
: Set the background color of the receiver to *color*. *Note:* When changing foreground and background color, make sure that an application has is receiving and processing events, using the `openEventStream` method in class `X11Pane`.

  Clear the receiver to the background color.

`clearRectangle (Integer` *`x`*`, Integer` *`y`*`, Integer` *`width`*`, Integer` *`height`*`)`
: Clear the specified area to the background color.

  Copies the contents of *srcBitmap* to the receiver's drawing surface. The source dimensions are determined by *srcX*, *srcY*, *srcWidth*, and *srcHeight*. The method draws the source bitmap's contents with the source's upper left-hand corner at *destX*, *destY*.

`initialize (void *displayPtr, int` *`parentWindow`*`, int` *`width`*`, int` *`height`*`, int` *`depth`*`)`
: Create a X11Bitmap object and its graphics data.

`delete (void)`
: Delete a pixmap and its graphics data.

`drawCircle (Circle aCircle, Integer filled, String bgColor)`
`drawCircle (Circle aCircle, Pen aPen, Integer filled, String bgColor)`
Draw the circle defined by *aCircle* in the receiver's paneBuffer. If *filled* is true, draws a filled circle. If the *aPen* argument is given, draws the circle with the color and the line width defined by the `aPen`, and fills the interior of the circle with *bgColor*.

`drawFilledRectangle (Rectangle aRect, Pen aPen)`
Draw a filled rectangle with the dimensions given in *aRect*, using the color given by *aPen*. For other variations on this method refer to `drawRectangle`.

`drawLine (Integer xStart, Integer yStart, Integer xEnd, Integer yEnd, Pen aPen)`
`drawLine (Line aLine, Pen aPen)`
Draw a point on the receiver's drawing surface between the points *xStart*,*yStart* and *xEnd*,*yEnd* using the color, width, and transparency defined by the argument *aPen*. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

If a program uses the two-argument form of the method, *aLine* is a member of class *Line*. See ⟨undefined⟩ [Line], page ⟨undefined⟩.

`drawPoint (Integer x, Integer y, Pen aPen)`
Draw a point on the receiver's drawing surface at *x*,*y* using the color, width, and transparency defined by the argument *aPen*. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

`drawFilledRectangle (Rectangle aRect, Pen aPen)`
`drawFilledRectangle (Rectangle aRect, Pen aPen,`
`Integer` *fill*, `Integer` *corner_radius*)

`drawFilledRectangle (Integer xOrg, Integer yOrg, Integer xSize, Integer ySize,`
`Ineter fill,`
`Integer` *line_width*, `String` *color*, `Integer` *corner_radius*) Draw a rectangle on the receiver bitmap. The *fill* argument, if given, draws a filled rectangle if the argument is true. If *fill* is false, use the *pen_width* argument to determine the line width. If *radius* is non-zero, draw the rectangle with rounded corners with the radius in pixels given by the argument.

`entryIcon (int iconID)`
`entryIconEye (void)`
`entryIconEyeSlashed (void)`
Draws an eye or slashed eye icon on the receiver bitmap. The bitmap should already be created with a call to the `X11Bitmap : create` method, and should have the dimensions '`ENTRY_ICON_WIDTH_PX`' by '`ENTRY_ICON_HEIGHT_PX`'. These values are defined in `ctalkdefs.h`, which can be included in programs and classes with the line:

```
#include <ctalk/ctalkdefs.h>
```

In the case of the `entryIcon` method, the *iconID* argument should have one of the following values.

```
ENTRY_ICON_EYE_NONE
ENTRY_ICON_EYE_OK
ENTRY_ICON_EYE_SLASHED
```

These constants are also defined in `ctalkdefs.h`.

`faceRegular (void)`
`faceBold (void)`
`faceItalic (void)`
`faceBoldItalic (void)`
Selects the typeface of the currently selected font. The font should have been selected by a previous call to `font`, below, which loads the regular, bold, italic (or oblique), and bold italic flavors of the selected font if they are available.

`font (String` *`font_desc`*`)`
Set the font of the receiver.

`foreground (String` *`color`*`)`
Set the foreground color of the receiver. *Note:* When changing foreground and background color, make sure that an application has is receiving and processing events, using the `openEventStream` method in class `X11Pane`.

`icon (int` *`iconID`*`)`
`iconStop (void)`
`iconCaution (void)`
`iconInfo (void)`
These methods draw a 64x64 dialog icon on the receiver's drawing surface. In the case of the `icon` method, the *iconID* argument can be one of the following.

```
ICON_NONE
ICON_STOP
ICON_CAUTION
ICON_INFO
```

Before calling any of these methods, a program must have created the pane's drawing surface with the `create` method. It is also generally necessary to set the pane's background color to that of the surrounding window. Here is an example.

```
myIcon create aPane displayPtr, aPane paneBuffer xID,
  ICON_WIDTH_PX, ICON_HEIGHT_PX, pPane paneBuffer depth;
myIcon background "blue";
```

Because icons are generally rendered off-screen, the program should use the parent pane's display connection, `paneBuffer xID,` and depth. Then the program can use the `X11Bitmap : copy` method to draw the bitmap on the surround pane's buffer. Here's another example.

```
aPane paneBuffer copy myIcon, 0, 0, ICON_WIDTH_PX, ICON_HEIGHT_PX,
      iconLeftOrigin, iconTopOrigin;
```

The constants `ICON_WIDTH_PX` and `ICON_HEIGHT_PX` are defined in `ctalkdefs.h`, so programs and classes that use icons should add this line near the top of the source module.

```
#include <ctalk/ctalkdefs.h>
```

`putStr (Integer` *x*`, Integer` *y*`, String` *text*`)`
`putStr (Integer` *x*`, Integer` *y*`, String` *text*`, String` *font_desc*`)`
`putStr (Integer` *x*`, Integer` *y*`, String` *text*`, String` *font_desc*`, String` *colorName*`)`

Print *text* on the receiver's drawable X resource, at *x*,*y*.

If *font_desc*, (which is either a X Logical Font Descriptor if using X bitmap fonts or a Fontconfig descriptor if using Freetype fonts), is given, use that font to display the text.

For information about Fontconfig descriptors, refer to the `X11FreeTypeFont` section See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩. For information about X Logical Font Descriptors, refer to the `X11Font` section See ⟨undefined⟩ [X11Font], page ⟨undefined⟩.

If *colorName* is also given, render the text using that color.

`pixmapFromData (int` *x_org*`, int` *y_org*`, char *`*xpm_data[]*`)`

Draw the X pixmap defined by *xpm_data* with the upper left corner at *x_org*,*y_org* on the receiver's pane buffer.

The *xpm_data* argument is the name of the array declared at the start of a `xpm` file's data array.

In Ctalk's standard libraries, the image will appear on a `X11CanvasPane` object, and programs can use the `pixmapFromData` method in `X11CanvasPane` instead of calling this method directly. See ⟨undefined⟩ [X11CanvasPane_pixmapFromData], page ⟨undefined⟩.

`resize (Integer` *parentVisual*`, Integer` *new_width*`, Integer` *new_height*`)`

Resize the receiver Pixmap. This method is normally called by an event handler in a parent class (typically, that's a `X11CanvasPane`). If you need to resize a graphics pane's buffers, then the program should also call `subPaneNotify` (class `X11Pane`), which invokes the handler for a `RESIZENOTIFY` event.

Here is the `X11CanvasPane` classes' `RESIZENOTIFY` event handler.

```
X11CanvasPane instanceMethod subPaneResize (Object __subPane,
  InputEvent __event) {
  "Resize the receiver pane.  This is the resize event
  handler called by the parent window's pane
  dispatcher."

  if (__ctalkX11ResizeWindow (__subPane, __event xEventData3,
```

```
        __event xEventData4,
        __subPane depth) > 0) {
      /* refreshReframe uses viewWidth and viewHeight, refresh uses size x
         and size y */
      __subPane viewWidth = __event xEventData3;
      __subPane viewHeight = __event xEventData4;
      __subPane size x = __event xEventData3;
      __subPane size y = __event xEventData4;
      (X11Bitmap *)self paneBuffer resize self xWindowID,
        __event xEventData3, __event xEventData4;
      (X11Bitmap *)self paneBackingStore resize self xWindowID,
        __event xEventData3, __event xEventData4;
    }

    return NULL;
  }
```

For an example of how a program can handle `RESIZENOTIFY` events, refer to the examples in the `X11CanvasPane` section. See ⟨undefined⟩ [X11CanvasPane], page ⟨undefined⟩.

`xpmInfoFromData (char **xpm_data, Array dataReturn)`
: Fills in the `Array` *dataReturn* with the information from the X pixmap data referred to by *xpm_data*: [width, height, colors, characters_per_color];

`xpmCharsPerColorFromData (char **xpm_data)`
: Returns an `Integer` with the number characters per color in the X pixmap referred to by *xpm_data*.

`xpmColorsFromData (char **xpm_data)`
: Returns an `Integer` with the number of colors in the X pixmap referred to by *xpm_data*.

`xpmHeightFromData (char **xpm_data)`
: Returns an `Integer` with the height of the X pixmap referred to by *xpm_data*.

`xpmWidthFromData (char **xpm_data)`
: Returns an `Integer` with the width of the X pixmap referred to by *xpm_data*.

3.9 `Boolean` Class

`Boolean` class objects are used mainly as the return values of methods of various classes that test equality, like `<`, `>`, `==`, `!=`, `&&`, and `||`.

An expression can treat `Boolean` methods differently than methods that perform numerical calculations, which usually return objects of `Magnitude` class and its subclasses.

Methods can return one of the `Boolean` class variables, `boolTrue` or `boolFalse`. In addition, methods can also define their own `Boolean` objects to return. The API function *__ctalkRegisterBoolReturn ()* does this automatically when it encounters a return statement in a method that has declared a `Boolean` return value.

Class Variables

`boolFalse`
`boolTrue` Objects that represent the logical values true and false. The objects can be used as return values in methods that return boolean values and in API functions like *__ctalkRegisterBoolReturn ()*. See ⟨undefined⟩ [ctalkRegisterBoolReturn], page ⟨undefined⟩.

Instance Methods

`!= (Integer i)`
: Return a `Boolean` value of '`True`' if the values of the receiver and the argument are different, '`False`' otherwise.

`&& (Boolean b)`
: Returns a `Boolean` value of '`True`' if both the receiver and the method's operand evaluate to true.

`= (Integer i)`
: Sets the value of the receiver to True or False. The method uses the Ctalk library to perform the numeric conversion of the argument to a C `int`, then casts the result to True or False.

`== (Integer i)`
: Return a `Boolean` value of '`True`' if the values of the receiver and the argument are the same, '`False`' otherwise.

`|| (Boolean b)`
: Returns a `Boolean` value of '`True`' if either the receiver or the method's operand evaluate to true.

3.10 `Collection` class

The `Collection` class is the superclass of all objects that contain groups of objects.

Internally, a `Collection` is made up of a series of `Key` objects. See ⟨undefined⟩ [Key], page ⟨undefined⟩. The only function of `Key` has is to maintain a reference to one of the collection's items. The `Key` objects have no other references, while the contents that they refer to may be used elsewhere.

But programs can also refer to `Key` objects individually. Most of the math operators that are overloaded to work with `Collections` actually work on `Key` objects.

If you write a method that adds objects to collections, it's important to add the attribute OBJECT_IS_MEMBER_OF_PARENT_COLLECTION to each `Key` object, which tells Ctalk that the `Key` object can be referred to individually, and not just by its parent object. See ⟨undefined⟩ [Attributes], page ⟨undefined⟩.

Instance Methods

`* (void)` When used as a unary operator to overload C's dereference ('`*`') prefix operator, returns the first element of the array or `Collection`. This method is actually a shortcut for the method `head`, below.

`+ (Integer n)`

Returns the *nth* member of the receiver `Collection`. This method is actually a shortcut for the following expressions.

```
Collection new myCollection;

.....  /* Add members to the collection. */

if (*(myCollection + 1)) {
...
}

Collection instanceMethod + nth (Integer new n) {
  Key new k;

  k = *self;
  k += n;
  return k;
}
```

`at (char *keyString)`

Retrieve the element of the receiver Collection at *keyString*.

`atPut (char * keyString, OBJECT *elemObject)`

Add *elemObject* to the receiver with the key *keyString*.

`delete (void)`

Remove all of the items in the receiver collection, leaving an empty collection. If any of the items are temporary (for example, a C variable alias), delete the item also.

`getValue (void)`

Return the value of the receiver, a `Key` object.

`integerAt (String keyName)`

Returns the value for *keyString* as an `Integer`. The method does not do any checking to make sure that the value is a valid `Integer` object; the program should ensure that the collection element is either an `Integer` object or is validly translatable to an `Integer`.

`isEmtpy (void)`

Return `TRUE` if the receiver collection is emtpy, `FALSE` otherwise.

`head (void)`

Return the first `Key` in the receiver collection, or NULL if the collection is empty.

`keyExists (String keyName)`

Return an `Integer` value of True if the key given as the argument is present in the receiver collection, False otherwise.

`map (OBJECT *(*method)())`
: Execute *method*, an instance method of class `AssociativeArray,` for each key/value pair of the receiver array. The receiver of *method* is a `Key` object with the parent class of `AssociativeArray`. The return value of `mapKeys` is NULL.

`removeAt (String key_name)`
: Remove the object stored at *key_name* from the receiver collection and return it. Returns NULL if the entry isn't present in the receiver collection. In that case, the method does not create a new entry, so programs should then call the `atPut` method.

`replaceAt (String key_name, Object new_value)`
: Replace the value at *key_name* in the receiver collection. Returns the old value object, or NULL if the key doesn't exist. In that case, the method does not create a new entry, so in that case it's necessary to add the key/value entry to the receiver collection using `atPut`.

`size (void)`
: Return an `Integer` with the number of items in the receiver collection.

`tail (void)`
: Return the last `Key` object that was added to the receiver collection.

3.11 `Array` Class

The `Array` class contains objects ordered by index from 0... n.

Instances of `Array` objects store unique copies of objects, so they can be added and deleted without worrying too much if an `Array` member is referred to by another object (unless the program later refers to the `Array` member; i.e., after it's been stored with a method like `atPut`).

So most statements on `Array` members are safe, but you should be careful when using multiple `Array` operations in the same expression. For example, the following expression is not safe.

```
myArray atPut 0, ((myArray at 0) - 3);
```

That's because the `atPut` method replaces the element at 0 while the original element is still being used as an argument. It's safer to use another object to work on an `Array` element.

```
tmpInt = ((myArray at 0) - 3);
myArray atPut 0, tmpInt;
```

Instance Variables

`value` The value is the name of the array.

`0... n` These instance variables refer to the elements of the array.

Instance Methods

`= (Array a)`
: Set the receiver array's elements equal to the argument array's elements.

`asString (void)`
: Returns the contents of the receiver `Array` as a `String` object. If any member of the receiver is a `Character` object, unquotes the `Character` object's value and concatenates the character to the result string. Otherwise the method concatenates each of the receiver `Array`'s values into the result string.

`at (int n)`
: Retrieve the *n*th element of the receiver array.

`atPut (int index, OBJECT *item)`
: Add *item* to the receiver at *index*.

`map (method)`
: Executes *method* with each element of the receiver.

  The argument, *method*, is a method that should belong to the same class as the receiver. When *method* is executed, its receiver is each successive array element, and *method* can refer to it with `self`.

`size (void)`
: Return the number of elements in the receiver.

3.12 `List` Class

Objects of class `List` allow sets of objects to be sequentially added, removed, and used as method receivers.

Instance Methods

`+= (Object obj1, ...)`
: Add the items given in the argument to the receiver List. For example:

```
int main () {
  List new l;

  l = "first", "second", "third", "fourth";
  l += "fifth", "sixth", "seventh", "eighth";

  l map {
    printf ("%s\n", self);
v  }
}
```

`= (Object obj1, ...)`
: Push the arguments on to the receiver List. The arguments are a comma separated list of objects. For example:

```
int main () {
  List new l;
```

```
  l = "first", "second", "third", "fourth";

  l map {
    printf ("%s\n", self);
  }
}
```

The = method initializes a list with only the objects that are given as arguments. The memthod deletes any previous list contents.

`append (Object` *`obj1`*`, ...)`
: Add the objects given in the argument to the receiver List. This is a synonym for the += method, above.

`init (Object` *`obj1`*`, ...)`
: This is a synonym for the = method, above.

`map (OBJECT *(*`*`method`*`)())`
`map (OBJECT *(*`*`method`*`)(), Object` *`argument`*`)`
`map (OBJECT *(*`*`method`*`)(), Object` *`argument1`*`, Object` *`argument2`*`)`
: Execute *method*, an instance method of class `List`, for each member of the receiver List. Within *method*, `self` refers to each successive list element. Here is an example.

```
List instanceMethod printElement (void) {
  printf ("%s\n", self);  /* Here, for each call to the printElement
                             method, "self" is each of myList's
                             successive members, which are String
                             objects. */
}

int main () {

  List new myList;

  myList = "item1", "item2", "item3";  /* Initialize the List with
                                          three String objects. */
  myList map printElement;

}
```

The `map` method can also use a code block as its argument. The example above, written with a code block, would look like this.

```
int main () {

  List new myList;
```

```
  myList = "item1", "item2", "item3";  /* Initialize the List with
                                          three String objects. */
  myList map {
    printf ("%s\n", self);
  }

}
```

If given with two arguments, *method's* parameter list must have one parameter. The parameter's class is significant within *method*; that is, `map` can use any class of object for *argument*. Here is the example above with one argument for the printElement method;

```
List instanceMethod printElement (String leftMargin) {
  printf ("%s%s\n", leftMargin, self);

}

int main () {

  List new myList;
  String new leftMargin;

  myList = "item1", "item2", "item3";  /* Initialize the List with
                                          three String objects. */
  leftMargin = "- ";

  myList map printElement, leftMargin;

}
```

Calling `map` with three arguments works similarly. The `map` methods in `List` class all return NULL.

`mapRev (OBJECT *(*`*method*`)())`

Like `map`, except that it executes *method* for the last member that was added to the receiver List, then the previous member, and so on until the `mapRev` executes *method* for the first member of the list before returning.

`new (`*list1*`,` *list2*`,` *list3*`, ...;)`

Create the `List` objects *list1*, etc. For example:

```
List new list1, list2, list3;
```

`pop (void)`

Remove the object from the end of the receiver's list and return the object.

`popItemRef (void)`
: Here for backward compatibility; it is now the same as `pop`.

`push (OBJECT *(*object)(int))`
: Add *object* to the end of the receiver's list contents.

`pushItemRef (OBJECT *(*object)(int))`
: Also here for backward compatibility, this method is now the same as `push`.

`shift (OBJECT *(*object)(int))`
: Add *object* as the first element of the receiver's list contents.

`sortAscending (void)`
`sortDescending (void)`
`sortAscendingByName (void)`
`sortDescendingByName (void)`
: Sorts the receiver list based on either the members' values or names, in ascending or descending order. The sort algorithm is very simple minded, but due to the mechanics of determining earlier/later `List` members, the methods are as fast for small and medium `Lists` as more complex sort algorithms.

  If possible, you should try to add members in order rather than try to re-arrange a `List` later. For this, refer to the methods in `SortedList` class See ⟨undefined⟩ [SortedList], page ⟨undefined⟩.

`unshift (void)`
: Remove the first object from the receiver's list and return the object.

`value (void)`
: Class `List` objects have no `value` instance variable. Instead, return the `List` object's contents, or `NULL` if the list is empty.

3.13 `AssociativeArray` Class

Objects of class `AssociativeArray` contain members that are stored and retrieved using `Key` objects. See ⟨undefined⟩ [Key], page ⟨undefined⟩.

`AssociativeArray` objects use the `atPut` and `at` methods in `Collection` class to store and retrieve objects. See ⟨undefined⟩ [Collection], page ⟨undefined⟩.

The `keyAt` method returns the key/value pair of the array element named as its argument.

The `map` method maps over each object stored in an `AssociativeArray`, and provides that element as the receiver to the method or code block given as the argument.

If you want to work with the object stored in the array and the key it is stored with, the `mapKeys` method iterates over each key/value pair of the `AssociativeArray`

The following example shows how to retreive the keys and values stored in an `AssociativeArray`. Each value that is stored in the array is a `String` object, so the program does not need to check for different classes of objects that are stored in the array.

```
AssociativeArray instanceMethod printValue (void) {

  printf ("%s\t", self name);
```

```
  printf ("%s\n", self value);
  return NULL;
}

AssociativeArray instanceMethod printKeyValue (void) {

  String new valueObject;
  printf ("%s =>\t", self name);
  valueObject = self getValue;
  printf ("%s\n", valueObject);
  return NULL;
}

int main () {
  AssociativeArray new assocArray;
  String new s1;
  String new s2;
  String new s3;

  WriteFileStream classInit;

  s1 = "string1 value";
  s2 = "string2 value";
  s3 = "string3 value";

  assocArray atPut "string1", s1;
  assocArray atPut "string2", s2;
  assocArray atPut "string3", s3;

  stdoutStream printOn ("%s\n%s\n%s\n\n", (assocArray at "string1"),
(assocArray at "string2"),
(assocArray at "string3"));
  stdoutStream printOn "%s\n%s\n%s\n\n", (assocArray at "string1"),
    (assocArray at "string2"),
    (assocArray at "string3");
  stdoutStream printOn "%s\n%s\n%s\n\n", assocArray at "string1",
    assocArray at "string2",
    assocArray at "string3";

  assocArray map printValue;

  assocArray mapKeys printKeyValue;

  return 0;
}
```

Instance Methods

`at (char *key)`
: Retrieve the element of the receiver array stored at *key*.

`init (...)`
`= (...)`
: Append the arguments to the receiver `AssociativeArray`. The argument list may be any number of *key,value* pairs. Here is an example.

```
myAssocArray init "key1", "first", "key2", "second", "key3", "third",
  "key4", "fourth";
myAssocArray append "key5", "fifth", "key6", "sixth", "key7", "seventh",
  "key8", "eigth";

... or ...

myAssocArray = "key1", "first", "key2", "second", "key3", "third",
  "key4", "fourth";
myAssocArray += "key5", "fifth", "key6", "sixth", "key7", "seventh",
  "key8", "eigth";
```

`atPut (char *key, OBJECT *item)`
: Add *item* to the receiver at *key*.

`init (...)`
`= (...)`
: Initializes the receiver `AssociativeArray` to the values given as the arguments, which may be composed of any number of *key,value* pairs. Here is an example.

```
myAssocArray init "key1", "value1", key2", "value2", "key3", "value3";

... or ...

myAssocArray = "key1", "value1", key2", "value2", "key3", "value3";
```

`keyAt (String keyName)`
: Returns the `Key` object named by *keyName*.

`map (OBJECT *(*method)())`
: Execute *method*, an instance method of class `AssociativeArray`, for each member of the receiver array. Each time *method* is executed, `self` refers to the object stored in the associativeArray. This method returns NULL.

`setValue (void)`
: A wrapper method for `getValue` in class `Key`. If received by an instance of `Collection` or its subclasses instead of a `Key` object, this method generates an exception.

3.14 `SortedList` Class

Instance Methods

`+= (Object` *obj1*`, ...)`
: Add the items given in the argument to the receiver list. For example:

```
int main () {
  SortedList new l;

  l = "first", "second", "third", "fourth";
  l += "fifth", "sixth", "seventh", "eighth";

  l map {
    printf ("%s\n", self);
  }
}
```

`= (Object` *obj1*`, ...)`
: Push the arguments on to the receiver list. The arguments are a comma separated list of objects. For example:

```
int main () {
  SortedList new l;

  l = "first", "second", "third", "fourth";

  l map {
    printf ("%s\n", self);
  }
}
```

The `=` method initializes a list with only the objects that are given as arguments. The memthod deletes any previous list contents.

`append (Object` *obj1*`, ...)`
: Add the objects given in the argument to the receiver list. This is a synonym for the `+=` method, above.

`init (Object` *obj1*`, ...)`
: This is a synonym for the `=` method, above.

`pushAscending (Object` *new_item*`)`
`pushDescending (Object` *new_item*`)`
: Adds *new_item* to the receiver list at the point where its value maintains the receiver list items' ascending or descending sort order.

3.15 `Stream` Class

`Stream` class is the superclass of classes that provide sequential read and write access to objects and I/O devices.

The subclasses of `Stream` class are listed here. This manual describes each class in its own section.

```
Stream
 FileStream
  DirectoryStream
  ReadFileStream
  WriteFileStream
 TerminalStream
  ANSITerminalStream
  Win32TerminalStream
  X11TerminalStream
 NetworkStream
  UNIXNetworkStream
   UNIXNetworkStreamReader
   UNIXNetworkStreamWriter
  TCPIPV6NetworkStream
   TCPIPV6NetworkStreamReader
   TCPIPV6NetworkStreamWriter
  TCPIPNetworkStream
   TCPIPNetworkStreamReader
   TCPIPNetworkStreamWriter
```

3.16 `FileStream` Class

`FileStream` and its subclasses contain methods and variables that read and write to files and other I/O devices.

Instance Variables

`streamMode (Class Integer)`
: File permissions mode.

`streamDev (Class Integer)`
: Device ID of a file.

`streamRdev (Class Integer)`
: Device ID if a special file.

`streamSize (Class LongInteger)`
: File size in bytes.

`streamAtime (Class LongInteger)`
: Time of last file access.

`streamMtime (Class LongInteger)`
: Time of last file modification.

`streamCtime (Class LongInteger)`
: Time of last file status change.

`streamPos (Class LongInteger)`
: Offset of read or write operation in the file, if a regular file.

`streamErrno`
: System error number of the last file operation, EOF at the end of the file, or 0 on success.

`streamPath`
: A `String` object, containing the path name of a regular file.

Class Variables

None.

Instance Methods

`= (`*arg*`)` Assign the value of the file stream in the argument to the receiver. This method does not duplicate the file handle.

`closeStream (void)`
: Close the receiver's input file stream. Note that `closeStream` does *not* delete the receiver object. You must call `delete` (Class Object) to delete the object explicitly. Deleting global and local objects is still not complete - you must check the receiver and argument stacks, and the local variables of the function or method, to make sure that the object is removed from its dictionary.

`deleteFile (String` *file_path*`)`
: Deletes the file given as the argument. If the deletion is not successful, the method raises a `SystemErrnoException` which an application can check for.

`exists (char *`*__path_name*`)`
: Return `TRUE` if the file exists, `FALSE` otherwise.

`isDir (void)`
: Return an `Integer` object that evaluates to `TRUE` if the receiver is a directory, `FALSE` otherwise.

`renameFile (String` *oldname*`, String` *newname*`)`
: Renames the file named by *oldname* to *newname*. Raises a `SystemErrnoException` if renaming the file causes an error.

`seekTo (Integer` *file_pos*`)`
: Reposition the file's reading and writing position indicator at *file_pos*. If the request is unsuccessful, the method raises a `SystemErrnoException` that a program can check for.

`seekToBeginning (void)`
: Reposition the file's reading and writing position indicator at the beginning of a file. If the request is unsuccessful, the method raises a `SystemErrnoException` that a program can check for.

`seekToEnd (void)`
: Reposition the file's reading and writing position indicator at the end of a file. If unsuccessful, the method raises a `SystemErrnoException` that programs can check for.

`size (void)`
: Returns the size of the receiver's file as a `LongInteger`.

`statFile (String file_path)`
: A wrapper method for the C `stat(2)` function. This method fills in the receiver stream's `streamMode`, `streamDev`, `StreamRdev`, `streamSize`, `streamAtime`, `streamMtime`, and `streamCtime` instance variables. The stream does not need to be opened on the file given by *file_path*.

  The method returns an `Integer` with a value of '`0`' on success, or if an error occurs, returns '`-1`' and raises a `SystemErrnoException`.

`statStream (void)`
: Another wrapper method for the C `stat(2)` function. The method fills in the `streamMode`, `streamDev`, `StreamRdev`, `streamSize`, `streamAtime`, `streamMtime`, and `streamCtime` instance variables of the receiver, an open file stream.

  The method returns an `Integer` with a value of '`0`' on success, or if an error occurs, returns '`-1`' and raises a `SystemErrnoException`.

`streamEof (void)`
: Returns a non-zero `Integer` object if at the end of the receiver's file stream, a zero-value `Integer` otherwise.

Class Methods

None.

3.17 `DirectoryStream` Class

The `DirectoryStream` class contains methods for creating and deleting directories, and for reading files in directories.

On machines which have library support for file globbing, `DirectoryStream` provides methods that read directories and files that match patterns which contain metacharacters like '`*`', '`?`', and '`[`'. This is in addition to whatever file pattern expansion the shell performs on file patterns provided on the command line. See below for more details.

Metacharacter Expansion.

If a command provides a file specification that contains a metacharacter, often the shell expands the pattern into a list of files that the program receives in the function `main's argv` array.

If the command line provides a quoted metacharacter as one of the program's arguments, however, the program can use the `DirectoryStream` methods `globPattern` and `globCwd` to expand the pattern into a list of filenames.

```
$ myfileprog some_arg *    # The shell expands '*' into a list of files.

$ myfileprog some_arg '*'  # The shell adds a literal '*' to the app's
                           # arguments.
```

Not all shells provide metacharacter expansion before the program executes `main`, however, and so the app should check for metacharacters as arguments also. The following example shows one way to do this.

```
List new fileNames;

String instanceMethod globFile (void) {

  DirectoryStream new d;
  List new globFiles;

  if (d hasMeta self) {  /* The argv entry contains a wildcard character. */
    d globPattern self, globFiles;
    globFiles map {
      fileNames push self;
    }
  } else {             /* The argv entry is an unambiguous file name. */
    fileNames push self;
  }

}

int main (int argc, char *argv[]) {

  int i;

  for (i = 1; i < argc; ++i) {
    argv[i] globFile;
  }

  fileNames map {
    printf ("%s\n", self);
  }
}
```

Directory Modes

The default mode for new directories is `0755` (‘`drwxr-xr-x`’) modified by the user’s `UMASK`. Programs can change the default directory permissions by redefining the macro `CTALK_DIRECTORY_MODE`.

Error Handling

The methods `chDir`, `mkDir`, and `rmDir` raise a `SystemErrnoException` on error and return an `Integer` with the value `-1`.

Instance Methods

`chDir (char *`*`dir`*`)`

Change to the directory *dir*.

`directoryList (char *dirname, List dirlist)`
: List directory *dirname*, and store it in the `List` *dirlist*. The order of the elements in the list depends on the operating system. To expressly create a sorted directory listing, see the `sortedDirectoryList` method.

`getCwd (void)`
: Return the current directory as a `String` object.

`globPattern (String file_name_pattern, List matching_file_names)`
: Adds the file names that match *file_name_pattern* to *matching_file_names*. This method uses the machine's *glob* library call, if it is available, to do the pattern matching. If the system doesn't support filename pattern matching in its C library, the method prints a warning message and returns. For more information about how the machine expands file globbing patterns into file names, refer to the *glob(3)* and *glob(7)* manual pages.

`hasMeta (String file_name_pattern)`
: Returns a `Boolean` value of True if *file_name_pattern* contains one of the opening metacharacters '`*`', '`?`', or '`[`', False otherwise.

`mkDir (char *directoryname)`
: Create the directory *directoryname*.

`rmDir (char *directoryname)`
: Delete the directory *directoryname*.

`sortedDirectoryList (char *dirname, SortedList dirlist)`
: List directory *dirname*, and store it in the `SortedList` *dirlist*. The members of the directory listing are stored in ascending order.

3.18 `ReadFileStream` Class

The `ReadFileStream` class contains methods and objects for reading files.

Instance Variables

None

Class Variables

`stdinStream`
: The `stdinStream` object contains the value of the application's standard input channel.

Instance Methods

`new (stream1, stream2, stream3,...)`
: Creates one or more new `ReadFileStream` objects with the names given in the argument list; e.g.,

```
ReadFileStream new stream1, stream2, stream3;
```

`openOn (String path)`
: Open file *path* with mode '`r`' and set the receiver's value to the file input stream.

`readAll (void)`
: Returns a String with the contents of the receiver's file stream.

`readChar (void)`
: Returns a Character from the stream defined by the receiver, or '`EOF`' at the end of the input.

`readFormat (char *fmt, ...)`
: Read formatted input from the receiver and store in the objects given as arguments.

`readLine (void)`
: Returns a String of characters up to and including the next newline of the stream defined by the receiver, or a String containing '`EOF`' at the end of the input.

`readRec (Integer record_length)`
: Return a `String` that contains *record_length* characters from the receiver's input stream.

`readVec (LongInteger data_length)`
: Reads *data_length* bytes from the receiver stream and returns a new `Vector` object that points to the data. This is useful for reading binary data which may have embedded NULLs and other non-human readable characters. See ⟨undefined⟩ [Vector], page ⟨undefined⟩.

`value`
: Returns the receiver's `value` instance variable.

Class Methods

`classInit`
: Called automatically by `new` when the first `ReadFileStream` object is created. Initializes `stdinStream`.

3.19 `WriteFileStream` Class

Class `WriteFileStream` provides objects and methods for writing to files, including the application's `stdout` and `stderr` output channels.

Because the way UNIX and Linux file stream modes work, a WriteFileStream opens file non-destructively; that is, for reading and writing, which means that programs can actually perform both reads and writes from a WriteFileStream.

All of the methods for reading files are defined in ReadFileStream class, however. See ⟨undefined⟩ [ReadFileStream], page ⟨undefined⟩. It's safer to handle existing files when they are opened for reading; i.e., as a `ReadFileStream` object.

That means `openOn` won't truncate an existing file. To start with a new file, use `deleteFile` (class `FileStream`). See ⟨undefined⟩ [FileStream], page ⟨undefined⟩.

In that case, and also any time that `openOn` doesn't find a file with that path name, it creates the file first.

```
writeFile deleteFile "my/file/path";
writeFile openOn "my/file/path";
```

To append to an existing file, use `seekToEnd` (also in `FileStream` class). See ⟨undefined⟩ [FileStream], page ⟨undefined⟩.

```
writeFile openOn "my/file/path";
writeFile seekToEnd;
writeFile writeStream appendLine;
writeFile closeStream;
```

Instance Variables

None

Class Variables

`stdoutStream`
: The `stdoutStream` object contains the application's standard output channel.

`stderrStream`
: The `stderrStream` object contains the application's standard error channel.

Instance Methods

`new (`*`stream1`*`, `*`stream2`*`, `*`stream3`*`, ...;)`
: Creates one or more new `WriteFileStream` objects with the names given in the argument list; e.g.,

```
WriteFileStream new stream1, stream2, stream3;
```

`openOn (String `*`path_name`*`)`
: Open file *path* and set the receiver's value to the file output stream. The file is created if it does not exist. Raises a `systemErrnoException` if an error occurs.

`printOn (char *`*`fmt`*`, ...)`
: Format and print the method's arguments to the receiver, which must be a `WriteFileStream` object that has been opened with `openOn`.

`value` Returns the receiver's `value` instance variable.

`writeChar `*`char`*
: Writes *char* to the receiver's output file stream.

`writeFormat (char *`*`fmt`*`, ...)`
: Writes its arguments to the receiver using format *fmt*.

`writeStream `*`string`*
: Write *string* to the receiver's output file stream.

`writeVec (Vector vector_object)`
: Write the contents of *vector_object* to the receiver stream. The length of the data written is determined by *vector_object's* `length` instance variable. This allows the method to write binary data that may contain NULLs and other non-human readable characters. See ⟨undefined⟩ [Vector], page ⟨undefined⟩.

3.20 `TerminalStream` Class

`TerminalStream` is the superclass of terminal classes that provide terminal and console input and output capabilities.

This class implements the input queue for its subclasses. The queue is a `List` object whose members are `InputEvent` objects. See ⟨undefined⟩ [InputEvent], page ⟨undefined⟩.

Instance Variables

`inputQueue`
: A `List` contains that contains `InputEvent` objects.

`inputQueueMax`
: The maximum number of `InputEvent` objects in the `inputQueue`

`nInputEvents;`
: The actual number of `InputEvent` objects in the `inputQueue`

Instance Methods

`inputPending (void)`
: Returns `TRUE` if a terminal stream has pending input objects.

`isATty (void)`
: Return `TRUE` if the receiver's stream is a tty device, `FALSE` otherwise.

`nextInputEvent (void)`
: Return the next `InputEvent` object from the stream's input queue.

`queueInput (void)`
: Turn on queueing of input events.

`queueInputEvent (void)`
: Queue this input event.

3.21 `ANSITerminalStream` Class

`ANSITerminalStream` objects and their methods provides basic I/O capabilities for consoles that use standard input and output input, as well as serial terminals.

`ANSITerminalStream` also provides basic input translation and output formatting for ANSI, VT100, XTerm, and similar terminal types.

`ANSITerminalStream` does not provide curses-like screen buffering methods. The class's output methods output characters correctly whether the terminal is in canonical or raw mode.

If a program sets the terminal to raw mode (either with the `rawMode` or `getCh` methods), it should also output characters with `printOn`, and restore the terminal before exiting, as this example illustrates.

```
int main () {
  ANSITerminalStream new term;
  Character new c;

  term rawMode;
  term clear;
  term gotoXY 1, 1;

  while ((c = term getCh) != EOF)
    term printOn "%c", c;

  term restoreTerm;
}
```

If a program needs to handle cursor and other non-alphanumeric keys, then it must open the `TerminalStream` input queue (with the `openInputQueue` method) and read input events. See ⟨undefined⟩ [TerminalStream], page ⟨undefined⟩.

At present, Ctalk recognizes the following input event classes.

```
KBDCHAR         # ASCII Characters
KBDCUR          # Cursor Keys.
```

When reading an `InputEvent` (with `nextInputEvent` in `TerminalStream` class), the event class is stored in the `eventClass` instance variable of the `InputEvent` object.

Here is an example of a program that opens a terminal input queue and reads characters by retrieving input event objects from the queue.

```
int main () {
  ANSITerminalStream new term;
  Character new c;
  InputEvent new iEvent;

  term rawMode;
  term clear;
  term cursorPos 1, 1;
  term openInputQueue;

  while ((c = term getCh) != EOF) {
    iEvent become (term nextInputEvent);
    if (iEvent eventClass == KBDCHAR) {
      term printOn "<KEY>%c", iEvent eventData;
    }
    if (iEvent eventClass == KBDCUR) {
      term printOn "<CURSOR>%c", iEvent eventData;
    }
  }
}
```

Because ANSITerminalStream objects wait until the user has typed a key, programs can read input events synchronously, but programs that use other input stream classes may read the input queue asynchronously.

Newly created ANSITerminalStream objects use standard input and standard output as their file streams. Programs that read and write to serial terminals can open the terminal's serial line with openOn, and set the communication parameters with setTty, as in this example.

```
int main () {
  ANSITerminalStream new term;
  SystemErrnoException new s;

  term openOn "/dev/ttyS1";
  if (s pending)
    s handle;

  term setTty 9600, 8, N, 1;
  term printOn "Hello, world!\r\nHello, world!\r\n";
  term closeStream;
}
```

However, programs also need to take note of the differences between xterms, console displays, and serial terminals. For example, some serial terminals won't send an 'ESC' (0x1b hex) character until another character is typed at the terminal to flush the terminal's output. Also, programs should not depend on a particular terminal's newline translation protocol, and some terminals echo tab characters without special configuration. Some terminal parameters are tunable in the ANSITerminalStrea class, but if a program needs special character handling, you may need to write a subclass of ANSITerminalStream for a particular terminal.

Instance Variables

inputHandle
: A ReadFileStream object that contains the terminal's input stream. The handle is initialized to standard input when the ANSITerminalStream object is created. Also refer to the information for outputHandle, below.

outputHandle
: A WriteFileStream object that contains the terminal's output stream. The handle is initialized to standard output when the ANSITerminalStream object is created. Programs can use the methods openOn and setTty (below), to open and configure handles for serial terminals.

  In practice, objects that are instances of ANSITerminalPane and its subclasses use the inputHandle stream for both input and output with serial terminals, but programs can configure ouputHandle independenly if necessary.

queueInput
: True if queueing input events.

rawModeFlag
: True if the terminal stream parameters are set to raw mode.

`termioCIFlag`
`termioCOFlag`
`termioCLFlag`
`termioCCFlag`
: Terminal settings. These variables are set with the current terminal parameters when creating an `ANSITerminalStream` object with `new`, and restored by the `restoreTerm` method.

`ttyDevice`
: A `String` containing the path terminal's device file, if different than `stdinStream` and `stdoutStream`. This variable is currently unused.

Instance Methods

`clear (void)`
: Clear the terminal stream.

`closeStream (void)`
: Close a terminal stream that was previously opened with `openOn`, below.

`cursorPos (int` *`row`*`, int` *`column`*`)`
: Set the cursor position to *row*, *column*. The upper left-hand corner of screen is `1,1`.

`getCh (void)`
: Get a character from the terminal without echoing it. This method handles *`C-c`*, *`C-d`*, and *`C-z`* control characters by exiting. This method calls `rawMode`, so the application must call `restoreTerm` before exiting.

`gotoXY (int` *`row`*`, int` *`column`*`)`
: Set the cursor position to *row*, *column*. The upper left-hand corner of screen is `1,1`. This method is a synonym for `cursorPos`, above.

`new (`*`stream1`*`,` *`stream2`*`, ...` *`streamN`*`; )`
: Create one or more new `ANSITerminalStream` objects and initialize their input and output handles to `stdinStream`. See ⟨undefined⟩ [ReadFileStream], page ⟨undefined⟩, and `stdoutStream`. See ⟨undefined⟩ [WriteFileStream], page ⟨undefined⟩. For example,

```
ANSITerminalStream new stream1, stream2;
```

`openInputQueue (void)`
: Begin queueing input events. See ⟨undefined⟩ [InputEvent], page ⟨undefined⟩.

`openOn (char *`*`tty_device_name`*`)`
: Open a tty device for the receiver.

`printOn (char *`*`fmt`*`, ...)`
: Print the formatted output to the receiver's output stream.

`rawMode (void)`
: Set the terminal input and output streams to raw mode.

`readChar (void)`
: Read a character from the receiver's input stream.

`readLine (void)`
: Read a line of text from the receiver's input stream and return a `String` object with the text.

`restoreTerm (void)`
: Restore the terminal parameters to the values when the `ANSITerminalStream` object was created. Applications must call this method after using the `getCh` and `rawMode` methods, or the terminal may be unusable after the the application exits.

`setGraphics (char` *`attribute`*`)`
: Set the graphics attribute for the following characters. The *attribute* argument can be one of the following characters.

```
0       Attributes Off
1       Bold
4       Underscore
5       Blink
7       Reverse
```

`setTty (int` *`speed`*`, int` *`data_bits`*`, char` *`parity`*`, int` *`stop_bits`*`)`
: Set the communication parameters of a terminal device opened with `openOn`, above.

3.22 `Win32TerminalStream` Class

The `Win32TerminalStream` class provides keyboard input capabilities for Win32 `cmd.exe` consoles. It also provides basic output formatting capabilities using DJGPP's `conio.h` functions. This class does not provide curses-style output buffering.

This class is not strictly a stream, and is likely to remain limited to displays in the `cmd.exe` window until Ctalk can implement a genuine *fork(3)* system call for non-POSIX systems.

`Win32TerminalStream` simulates raw terminal I/O using BDOS and C library function calls. If the program needs to handle extended keys like cursor and keypad keys, it should open the `TerminalStream` input queue (with the `openInputQueue` method) and read input events. See ⟨undefined⟩ [TerminalStream], page ⟨undefined⟩.

At present, Ctalk recognizes the following input event classes. In `Win32TerminalStream` class, the `getCh` method generates a `KBDCUR` input events for all extended keys.

```
KBDCHAR         # ASCII Characters
KBDCUR          # Extended keys (first character read returns 0).
```

For examples of how to read characters from the keyboard, see the program listings in the `ANSITerminalStream` class section of the manual. See ⟨undefined⟩ [ANSITerminalStream], page ⟨undefined⟩.

Instance Variables

`queueInput`
: True if queueing input events. The current state of the keyboard's modifier keys. Set by the `biosKey` and `getShiftState` methods, below.

Instance Methods

Waits for a key and returns the scan code of a BIOS Int 16h, function 0 call. Also sets the shift state in the receiver's `shiftState` instance variable. See the `getShiftState` method, below.

`cGetStr (void)`
: Returns a `String` object that contains input typed by the user. The input is echoed to the console.

`clear (void)`
: Clears the terminal window and moves the cursor to the upper left-hand corner.

`cPutStr (char *str`
: Prints *str* to the console at the current cursor position.

`cusorPos (int x, inty)`
: Positions the cursor at character position *x*, *y*. The upper left-hand corner of the screen is row 1, column 1. This is a synonym for `gotoXY,` below.

`getCh (void)`
: Essentially a wrapper for the BDOS character input without echo function. This method handles *C-c*, *C-d*, and *C-z* by exiting the application.

`getShiftState (void)`
: Get the state of the keyboard's modifier keys with by calling BIOS Int 0x16, function 12h. Stores the result in the receiver's `shiftState` instance variable.

`gotoXY (int x, inty)`
: Positions the cursor at character position *x*, *y*. The upper left-hand corner of the screen is row 1, column 1.

`openInputQueue (void)`
: Begin queueing input events. See ⟨undefined⟩ [InputEvent], page ⟨undefined⟩.

`printOn (char *fmt, ...)`
: Print the formatted output to the receiver's output stream.

`screenColor (char *fgcolor, char*bgcolor)`
: Sets the window's foreground and background colors for following writes using `cPutStr,` above. The arguments are class `String,` and may have the following values.

```
black
blue
green
cyan
red
magenta
brown
lightgray
darkgray
lightblue
lightgreen
```

```
lightcyan
lightred
yellow
white
```

3.23 X11TerminalStream Class

`X11TerminalStream` objects and methods handle input events from X Window System displays. Ctalk's X11 support focuses on the windows themselves, so this class should almost always be used with a `X11Pane` object. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

Here is an example program that uses `InputEvent` objects created by the window's input stream, `xPane`, to configure the dimensions of the pane's window. The `X11TerminalStream` object is contained in the `X11Pane`'s `inputStream` instance variable.

```
int main () {

  X11Pane new xPane;
  InputEvent new e;

  xPane initialize 25, 30, 100, 100;
  xPane map;
  xPane raiseWindow;
  xPane openEventStream;

  WriteFileStream classInit;

  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;
      switch (e eventClass value)
{
case CONFIGURENOTIFY:
  stdoutStream printOn "ConfigureNotify\t%d\t%d\t%d\t%d\n",
    e xEventData1,
    e xEventData2,
    e xEventData3,
    e xEventData4;
  stdoutStream printOn "Window\t\t%d\t%d\t%d\t%d\n",
    xPane origin x,
    xPane origin y,
    xPane size x,
    xPane size y;
  break;
case WINDELETE:
  xPane deleteAndClose;
  exit (0);
```

```
      break;
    default:
      break;
    }
        }
      }
    }
```

When compiling this program, you might also need to add the directory that contains the X include files to the include search path. For example:

```
$ ctcc -I /usr/include/X11 windemo.c -o windemo       # Recent Linux
```

Ctalk's libraries are linked with the system's X libraries during installation if they are present, but applications may also need to link with X libraries directly in some cases.

Event Classes

A `X11TerminalStream` object recognizes the following input event classes, and returns the following information in the `xEventData` instance variables, which are defined in `InputEvent` class. See ⟨undefined⟩ [InputEvent], page ⟨undefined⟩.

Event Class	xEventData1	xEventData2	xEventData3	xEventData4	xEventData5
BUTTONPRESS	x	y	state	button	-
BUTTONRELEASE	x	y	state	button	-
KEYPRESS	x	y	state	keycode	x_keycode[1]
KEYRELEASE	x	y	state	keycode	x_keycode[1]
CONFIGURENOTIFY	x	y	height	width	border
MOVENOTIFY	x	y	height	width	border
RESIZENOTIFY	x	y	height	width	border
MOTIONNOTIFY	x	y	state	is_hint	-
MAPNOTIFY	event	window	-	-	-
EXPOSE	x	y	width	height	count
SELECTIONREQUEST	-	-	-	-	-
SELECTIONCLEAR	-	-	-	-	-
WINDELETE	-	-	-	-	-

[1] The values of X keycodes are defined in X11/keysymdefs.h.

The `x` and `y` coordinates of the key and button events are relative to the window's origin. The `x` and `y` coordinates of a `CONFIGURENOTIFY` event are relative to the origin of the root window, normally the upper left-hand corner of the display.

The `MOVENOTIFY` and `RESIZENOTIFY` events are both derived from `CONFIGURENOTIFY` events, depending on whether the receiver's window was moved or resized.

Instance Variables

`inputPID` The process ID of the `X11TerminalStream` event handler.

`clientFD` The file descriptor that the `X11TerminalStream` event handler uses when communicating with the main window's process.

`eventMask`
An `Integer` that contains the OR'd values of the window system events that the program wants to be notified of. The window system events are listed in the table above. If `eventMask's` value is zero, then the event handler assumes

that the program wants to receiver all window system events. Any non-zero value, however, causes the program to receive only those events; for example:

```
myMainWindow inputStream eventMask = WINDELETE|KEYPRESS;
```

In this case the program only receives notice of key presses (`KEYPRESS` events) and requests from the GUI desktop to close the window (`WINDELETE` events) and exit the program.

Instance Methods

`openInputClient (void)`
: Open the X input handler. This method is normally used by `X11Pane` class's `openEventStream` method. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

`parentPane (void)`
: Return the receiver's `X11Pane` parent object if present.

`queueInput (void)`
: Queue window system events as `InputEvent` objects in the receiver's `inputQueue` instance variable. The `inputQueue` instance variable is inherited from `TerminalStream` class. See ⟨undefined⟩ [TerminalStream], page ⟨undefined⟩.

3.24 `NetworkStream` Class

`NetworkStream` is the superclass of objects that use socket connections to send and receive data. The subclasses of `NetworkStream` provide support for connections between processes within a program, between programs on the same machine or a local network, or between Internet sites.

Each of `NetworkStream's` subclasses, listed here, has its own section.

```
TCPIPNetworkStream
  TCPIPNetworkStreamReader
  TCPIPNetworkStreamWriter
TCPIPV6NetworkStream
  TCPIPV6NetworkStreamReader
  TCPIPV6NetworkStreamWriter
UNIXNetworkStream
  UNIXNetworkStreamReader
  UNIXNetworkStreamWriter
```

3.25 `TCPIPNetworkStream` Class

`TCPIPNetworkStream` class contains methods for sockets that are used for both writing and reading.

Methods that are specific to programs that read from sockets are defined in `TCPIPNetworkStreamReader` class, See ⟨undefined⟩ [TCPIPNetworkStreamReader],

page ⟨undefined⟩, and methods that work with programs that write to network connections are defined in `TCPIPNetworkStreamWriter` class See ⟨undefined⟩ [TCPIPNetworkStreamWriter], page ⟨undefined⟩.

Here is a program that writes data to a network connection.

```
static char *msg = "First connect.\n";

int main (int argc, char **argv) {
  SystemErrnoException new ex;
  TCPIPNetworkStreamWriter new client;

  if (argc != 2) {
    printf ("Usage: ip6writer <server_hostname>\n");
  }
  client openOn argv[1];
  client writeText msg;
  if (ex pending) {
    ex handle;
  }

  client closeSock;
}
```

And here is the corresponding program that reads the data from the network connection and displays it.

```
int main () {
  TCPIPNetworkStreamReader new server;
  Exception new ex;
  String new output;
  int newsocket;

  server openOn;

  newsocket = server acceptSock;

  if (newsocket > 0) {
    output = server readText newsocket;
    if (ex pending) {
      ex handle;
    }
    printf ("%s\n", output);
    server closeSock newsocket;
  } else if (ex pending) {
    ex handle;
  } else {
```

```
        printf ("Connection timed out.\n");
      }

      server closeSock;
    }
```

Instance Variables

`sock` An `Integer` that contains the file handle number of the socket created when a network connection is opened.

Instance Methods

`addrInfo (String` *`hostName`*`, String` *`canonicalNameOut`*`, List` *`addrsOut`*

Performs a lookup of *hostName's* IP addresses. If the lookup is successful, returns the host's canonical name in *canonNameOut* and each of the host's addresses as a string in *addrsOut*. The method's return value is an `Integer` with the number of addresses found.

If the lookup causes an error, the method raises an `Exception` (*not* a `SystemErrnoException`) and returns 0.

`closeSock (void)`
`closeSock (Integer` *`sock_fh`*`)`

With no arguments, closes the receiver's socket. With one argument, closes the socket given as the argument. If closing a socket causes an error, the method raises a `SystemErrnoException`.

`createSocketBasic (void)`

Creates a socket with the domain AF_INET and the protocol SOCK_STREAM, and sets the receiver's sock instance variable to the new socket's file handle number, an Integer, and then returns the socket's file handle number. If not successful, the method raises a SystemErrnoException and returns 0.

`hostToAddress (String` *`hostname`*`)`

Given the name of a network host as the argument, the method returns a `String` containing the host's dotted quad Internet address.

If the hostname lookup doesn't return any results, the method raises an `Exception` (*not* a `SystemErrnoException`) and returns an empty `String`.

```
#include <stdio.h>   /* contains printf prototype */

int main () {
  TCPIPNetworkStream new net;
  String new address;

  address = net hostToAddress "MyHostName"; /* Substitute your host's name.

  printf ("%s\n", address);
```

```
}
```

`readText (void)`
`readText (Integer sock_fh)`

Reads a socket's input and returns the input as a `String` object. With no arguments, the method uses the receiver's socket file handle, which is normally assigned by `createSocketBasic`, above.

If a socket file handle is given as the argument, then the method performs the read on that handle. This is useful when performing reads after a call to `acceptSock` or a similar method. See ⟨undefined⟩ [TCPIPNetworkStreamReader], page ⟨undefined⟩.

If an error occurs while reading, the methods raise a `SystemErrnoException`.

`readVec (Integer sock_fh, Vector data_vec_out)`

Reads binary data from the socket given as the first argument and returns the data in the `Vector` object give as the second argument.

The method raise a `SystemErrnoException` if an error occurs while reading.

This example is a simple server that receives image data and writes it to a file.

```
int main () {
  TCPIPNetworkStreamReader new server;
  SystemErrnoException new ex;
  Vector new output;
  WriteFileStream new writeF;
  int newsocket;
  char *socket_out;

  server openOn;

  newsocket = server acceptSock;  /* INADDR_ANY */

  if (newsocket > 0) {
    server readVec newsocket, output;
    if (ex pending) {
      ex handle;
    }
    server closeSock newsocket;
  } else if (ex pending) {
    ex handle;
  } else {
    printf ("Connection timed out.\n");
  }

  server closeSock;

  writeF openOn "image-copy.jpeg";
```

```
  writeF writeVec output;
  writeF closeStream;
}
```

Here is the corresponding client program that transmits the image data.

```
int main () {
  SystemErrnoException new ex;
  TCPIPNetworkStreamWriter new client;
  ReadFileStream new readF;
  Vector new photo;
  LongInteger new imageSize;

  readF openOn "image.jpeg";

  readF statStream;
  imageSize = readF streamSize;

  photo = readF readVec imageSize;

  readF closeStream;

  client openOn "127.0.0.1"; /* Edit with the reciever's actual network
                                address. */

  client writeVec photo;
  if (ex pending) {
    ex handle;
  }

  client closeSock;
}
```

`readText (String` *text*`)`

Writes the *text* given as the argument to the reciever's socket. If the number of bytes actually written isn't equal to the length of *text*, then the method raises a `SystemErrnoException`.

`readText (Vector` *data*`)`

Writes the data contained in the argument to the receiver's socket. The method raises an Exception if the argument is not a `Vector`, or a `SystemErrnoException` if an error occurs while writing.

Examples of client and server programs which handle binary data are given in the entry for `readVec`, above.

3.26 `TCPIPNetworkStreamReader` Class

`TCPIPNetworkStreamReader` class defines methods and instance variables that are used specifically for reading data from network connections.

For example programs, refer to the `TCPIPNetworkStream` class See ⟨undefined⟩ [TCPIP-NetworkStream], page ⟨undefined⟩.

Instance Variables

`timeout` An `Integer` that contains the number of seconds to wait for incoming connections.

Instance Methods

`acceptSock (void)`

Waits for an incoming connection on the receiver's socket, and if a connection is pending, returns the number of the socket that communications will take place on.

If the connection times out, then the method returns 0. The length of time the method should wait for incoming connections is given by the receiver's `timeout` instance variable.

If an error occurs, the method returns 0 and also raises a `SystemErrnoException`.

Note: The method can handle peer connections if the operating system supports it - binding listening sockets to specific addresses is not uniformly supported among operating systems. In these cases, the method can also wait for connections using the constant INADDR_ANY.

This is done by using the `openOn` method (below) with no network address.

```
mySock openOn "127.0.0.1";    /* bind the socket to the
                                 local network connection. */

mySock openOn;                /* bind a socket to listen for
                                 connections from any network
                                 address. */
```

In the second case, the `acceptSock` method can also function as the core of a more conventional network server.

`openOn (void)`
`openOn (String` *`address`*`)`
`openOn (String` *`address`*`, Integer` *`port`*`)`

Creates the receiver's socket and binds it to receive messages from the network *address* given as the argument. If no network address is given, the receiver's socket is bound to the constant INADDR_ANY.

If no *port* argument is given, the socket is bound to DEFAULT_TCPIP_PORT, which is defined in `classes/ctalkdefs.h`, and which you can set depending on the the systems' needs.

These methods raise an `Exception` if any of the networking functions return an error.

3.27 `TCPIPNetworkStreamWriter` Class

The `TCPIPNetworkStreamWriter` class defines supporting methods for programs that write to network connections.

For example programs, refer to the `TCPIPNetworkStream` class; See ⟨undefined⟩ [TCPIP-NetworkStream], page ⟨undefined⟩.

Instance Methods

`openOn (String` *`address`*`)`
`openOn (String` *`address`*`, Integer` *`port`*`)`

Creates a socket and connects it to the network *address* given as the argument.

If a *port* argument is given, the socket connects over that port; otherwise, the socket connects via DEFAULT_TCPIP_PORT, which is defined in `classes/ctalkdefs.h`, which you can adjust to suit the network's needs.

These methods raise a `SystemErrnoException` if any of the networking functions return an error.

3.28 `TCPIPV6NetworkStream` Class

The `TCPIPV6NetworkStream` class manages TCPIP version 6 stream objects. The class is a superclass of the `TCPIPV6NetworkStreamReader` and `TCPIPV6NetworkStreamWriter` classes, and the methods and instance variables that are defined by this class are common to both client and server programs.

Here is an example of simple IPv6 reader program.

```
int main () {
  SystemErrnoException new ex;
  TCPIPV6NetworkStreamReader new server, connection;
  String new data;

  server openOn;
  if (ex pending) {
    ex handle;
  }

  connection = server acceptSock;
  if (ex pending) {
    ex handle;
  }
  data = server readText connection;
  printf ("%s\n", data);

  server closeSock;
```

```
}
```

And here is a simple program that writes a message to a TCPIP v6 connection.

```
static char *msg = "First connect.\n";

#include <ctalk/ctalkdefs.h>

int main (int argc, char **argv) {
  SystemErrnoException new ex;
  TCPIPV6NetworkStreamWriter new client;

  if (argc != 2) {
    printf ("Usage: ip6writer <server_hostname>\n");
  }

  client openOn argv[1];
  if (ex pending) {
    ex handle;
  }

  client writeText msg;

  client closeSock;
}
```

Instance Variables

`sock` An `Integer` that contains the system-assigned file handle number of `TCPIPV6NetworkStreamReader` and `TCPIPV6NetworkStreamWriter` objects.

Instance Methods

`addrInfo (String` *`hostname`*`, String` *`canonnameout`*`, List` *`addrsOut`*`)`

Performs a lookup of *hostname's* IPv6 addresses. If the lookup is successful, returns the host's canonical name in *canonnameout* and each of the host's addresses as a string in *addrsOut*. The method's return value is an `Integer` with the number of addresses found.

If the lookup causes an error, the method raises an `Exception` (*not* a `SystemErrnoException`) and returns 0.

`closeSock (void)`
`closeSock (Integer` *`sockNum`*`)`

With no arguments, closes the receiver's socket. If a socket file handle number is given as the argument, closes that socket. If closing the socket causes an error, the method raises a `SystemErrnoException`.

`createSocketBasic`
: Creates a new IPv6 socket and sets the receiver's `sock` instance variable, an `Integer`, to the system-assigned file number, and returns the `sock` instance variable. If an error occurs while creating a socket, the method raises a `SystemErrnoException` and returns 0.

`readText (void)`
`readText (Integer` *`sock_fn`*`)`
: Read text from a network connection. With no arguments, the value of the receiver's `sock` instance variable provides the socket file handle. If one argument is given, the argument, an `Integer`, provides the file handle number.

  These methods return a `String` with the contents of the received message. If an error occurs while reading, the methods raise a `SystemErrnoException` and return `NULL`.

`readVec (Integer` *`sock_fn`*`, Vector` *`resultVec`*`)`
: Reads binary data from the socket handle *sock_fn* into *resultVec*, and returns *resultVec*. The method raises a `SystemErrnoException` if an error occurs.

`writeText (String` *`textArg`*`)`
: Writes the `String` given as the argument to the receiver's socket. If the number of bytes written does not match the length of *textArg*, the method raises a `SystemErrnoException`.

`writeVec (Vector` *`dataVec`*`)`
: Writes the contents of *DATAVEC* to the receiver's socket. If the number of bytes written does not match the length of the data, the method raises a `SystemErrnoException`.

3.29 `TCPIPV6NetworkStreamReader` Class

Instance Variables

`timeout` An `Integer` that contains the number of seconds that the `acceptSock` method (below) should wait for a connection. The default is 10 seconds.

Instance Methods

`acceptSock (void)`
: Listens for incoming connections on the socket that is bound to a network address and port by a previous call to `openOn`.

  If the program receives an incoming connection, `acceptSock` creates a new socket to read data from the connection and returns the socket number, an `Integer`, to the calling program.

  If the method times out, it returns 0 to the calling program. The `timeout` instance variable determines the number of seconds the method should wait for incoming connections. The default is 10 seconds.

  If an error occurs, the method returns -1 and raises a `SystemErrnoException`.

`openOn (void)`
`openOn (Integer` *`port`*`)`
`openOn (String` *`hostName`*`, Integer` *`port`*`)`

Binds a `TCPIPV6NetworkStreamReader` object to the hostname and port given as the arguments, if any. The return value is the socket handle number of the receiver, an `Integer`.

If no hostname or port is given, the method binds the socket to the system constant `in6addr_any`, and the port defined by DEFAULT_TCPIP_PORT, defined by the Ctalk library in the ctalkdefs.h header file.

If an error occurs, these methods raise an `Exception` and return 0.

3.30 `TCPIPV6NetworkStreamWriter` Class

Instance methods

`openOn (String` *`hostName`*`)`
`openOn (String` *`hostName`*`, Integer` *`port`*`)`

Open a socket connection to *hostName* for writing, optionally using *port* as the network port for the connection. If successful, the method returns the filehandle number of the socket object that made the connection as an `Integer`

If no *port* is given, the method uses the definition of `DEFAULT_TCPIP_PORT`, which is defined in `ctalkdefs.h`, as the port number. To include these definitions, you can add the following line to a program.

```
#include <ctalk/ctalkdefs.h>
```

The argument *hostName* must be the name of an IPv6 capable host, or the method raises an `Exception` and returns 0. The method also raises an `Exception` and returns 0 if an error occurs while trying to connect the socket.

3.31 `UNIXNetworkStream` Class

`UNIXNetworkStream` and its subclasses define instance and class variables and methods to communicate over UNIX domain sockets between processes and programs operating on the same machine.

As a type of interprocess communication, UNIX domain sockets are more flexible than named pipes - programs create and manage one or more sets of reader and writer objects independently of each other.

The section *UNIXNetworkStreamReader* describes the methods and instance data for creating and managing reader objects See ⟨undefined⟩ [UNIXNetworkStreamReader], page ⟨undefined⟩, and the section *UNIXNetworkStreamWriter* describes the details of writer objects See ⟨undefined⟩ [UNIXNetworkStreamWriter], page ⟨undefined⟩.

Class Variables

`socketPrefix`
: A `String` that defaults to the machine-specific directory name where the system stores its temporary files (e.g., `/tmp`, `/var/tmp`, etc.).

Instance Variables

`sock`
: An `Integer` that contains the file handle number the receiver's socket.

`socketBaseName`
: A `String` that contains the base name of the socket's path name. The string may contain escape characters. Refer to the `makeSocketPath` method, below.

`socketPath`
: A `String` that contains the fully qualified path name of a program's UNIX domain socket.

Instance Methods

`closeSocket (void)`
: Shuts down the receiver's socket (defined in the `sock` instance variable) and deletes the socket's file entry (which is defined in the `socketPath` instance variable), if present.

`makeSocketPath (String` *basename*`)`
: Constructs a fully qualified socket path from *basename* and the prefix given by `socketPrefix` (described above). If *basename* contains the characters '`$$`', the method replaces the character with the program's process ID.

  After constructing the fully qualified name, the method fills in the receiver's `socketBaseName` and `socketPath` instance variables, and returns the value of the `socketPath` instance variable.

  The small program here constructs and prints the file pathname of a program's UNIX socket, which includes the process ID of the program.

```
int main () {
  UNIXNetworkStream new un;
  String new sockName;

  sockName = "myprogsocket.$$";

  un makeSocketPath sockName;
  printf ("%s\n", un socketPath);
}
```

`removeSocket (void)`
: Deletes the socket's file entry by its name, which is defined by `makeSocketPath`, above.

3.32 `UNIXNetworkStreamReader` Class

`UNIXNetworkStreamReader` objects receive data using UNIX domain sockets on a single machine. The class defines methods to create a connection and check the connection for incoming data, and read any data for the application.

Here are slightly abreviated versions of the `sockread.c` and `sockwrite.c` example programs. This is the source code of `sockread.c`.

```
int main () {
  UNIXNetworkStreamReader new reader;
  SystemErrnoException new ex;
  FileStream new f;
  String new sockName;
  String new data;

  sockName = "testsocket";

  reader makeSocketPath sockName;
  printf ("reader socket:  %s\n", reader socketPath);

  /* Delete a socket from a previous connection if necessary. */
  if (f exists reader socketPath) {
    f deleteFile reader socketPath;
  }

  reader open;
  if (ex pending) {
    ex handle;
    unlink (reader socketPath);
    return -1;
  }

  while (1) {
    data = reader sockRead;

    if (data length > 0) {
      printf ("%s\n", data);
    }
    if (ex pending) {
      ex handle;
      break;
    }
    usleep (1000);
  }

  return 0;
}
```

And here is `sockwrite.c`. For details about its operation, refer to the `UNIXNetworkStreamWriter` section. See ⟨undefined⟩ [UNIXNetworkStreamWriter], page ⟨undefined⟩.

```
int main (int argc, char **argv) {
  UNIXNetworkStreamWriter new writer;
  SystemErrnoException new ex;
  String new sockName;
  String new data;

  sockName = "testsocket";

  writer makeSocketPath sockName;
  printf ("writer socket:  %s\n", writer socketPath);

  writer open;
  if (ex pending) {
    ex handle;
  }

  writer sockWrite argv[1];

  if (ex pending) {
    ex handle;
  }

  exit (0);
}
```

The programs communicate if started from different shells used by different virtual terminals or different windows on a graphical desktop, or if the reader program is started in the background and the writer started in the foreground.

Instance Variables

`charsRead`

An `Integer` that contains the number of characters read by the last reception of data from a connection, or zero if there is no data waiting. Refer to the method `sockRead`, below.

Instance Methods

`open (void)`

Creates a socket to read data from the socket name by the receiver's `socketPath` instance variable, which is defined in `UNIXNetworkStream` class. See ⟨undefined⟩ [UNIXNetworkStream], page ⟨undefined⟩.

The method returns the file number of the socket, or -1 if there is an error, and raises a `SystemErrnoException` if an error occurred in the network library routines.

`openOn (String` *socketpath*`)`
: Creates a new reader socket and binds it to *socketpath*. Sets the receiver's `socketPath` instance variable to the argument. Returns an `Integer` with the file number of the newly created socket.

`sockRead (void)`
: Returns a `String` containing data received from the socket previously created by the `open` method, above, and sets the `charsRead` instance variable to the number of characters read.

  If no data is waiting, the method returns an empty string and sets the receiver's `charsRead` instance variable to 0.

  If an error occurs during one of the system calls, the method raises a `SystemErrnoException`.

3.33 `UNIXNetworkStreamWriter` Class

`UNIXNetworkStreamWriter` objects send data over machine- local UNIX domain sockets. The class contains methods to open sockets and send data over the network connection.

For an example of how to use `UNIXNetworkStreamWriter` objects, refer to the programs in the `UNIXNetworkStreamReader` section. See ⟨undefined⟩ [UNIXNetworkStreamReader], page ⟨undefined⟩.

Instance Variables

`charsWritten`
: An `Integer` that contains the number of bytes written by the previous call to the `sockWrite` method, which is described below.

Instance Methods

`open (void)`
: Creates a machine-local UNIX socket connection with the socket name given by the `socketPath` instance variable, which is defined in `UNIXNetworkStream` class. See ⟨undefined⟩ [UNIXNetworkStream], page ⟨undefined⟩.

  If successful, the method returns the file handle number of the new socket. If an error occurs, the method returns -1 and raises a `SystemErrnoException` if the error occured in a library call.

`openOn (String` *socketpath*`)`
: Creates a new writer socket and binds it to *socketpath*. Sets the receiver's `socketPath` instance variable to the argument. Returns an `Integer` with the file number of the newly created socket.

`sockWrite (String` *data*`)`
: Writes *data* to the socket created by a previous call to the `open` method, above. Sets the receiver's `charsWritten` instance variable to the number of characters

written. If successful, returns 0, or if an error occured, returns -1 and raises a `SystemErrnoException`.

3.34 `TreeNode` Class

`TreeNode` objects and methods maintain links to other `TreeNode` objects (*siblings*), and to sets of objects (*children*). The methods can add sibling and child `TreeNodes`, traverse the tree, and set the content of each `TreeNode`.

The class provides basic methods for adding sibling and child nodes, adding content to each node (which is a `String` object), and to traverse the tree. The methods that visit each node in a tree– `print, format,` and `search`–are fairly generic. They perform a depth-first traversal which should work equally well with balanced and non-balanced trees.

Instance Variables

`children` A `List` of `TreeNode` objects, which are accessible only through the parent node. See ⟨undefined⟩ [List], page ⟨undefined⟩.

`content` A `String` that contains the node's displayable text.

`levelMargin`
A `String,` normally consisting of all spaces, that is the additional left margin for each level of the tree. See ⟨undefined⟩ [String], page ⟨undefined⟩.

`levelMarginLength`
An `Integer` that is the length of the `levelMargin` string.

`siblings` Another `List` of `TreeNode` objects that occur at the same level. See ⟨undefined⟩ [List], page ⟨undefined⟩.

Instance Methods

`addChild (TreeNode` *`child`*`)`
Add *child* to the end of the receiver's `children` list.

`format (void)`
Print the receiver tree to a `String` object. This method uses two other methods, `__formatChildren` and `__formatSiblings` to traverse each `TreeNode` object in the receiver tree. Returns a `String` object.

`makeSibling (TreeNode` *`sib`*`)`
Add *sib* to the end of the receiver's `sibling` list.

`map (OBJECT *(`*`method`*`)())`
Execute *method* over each member of the receiver tree. As with `map` methods in other classes, *method* must also belong to `TreeNode` class and takes no arguments.

`print (void)`
Print the receiver tree to the terminal. This method uses two other methods, `__printChildren` and `__printSiblings` to traverse each `TreeNode` object in the receiver tree.

`search (String searchString)`
: Return the first node in the receiver tree whose content matches *searchString*. Returns only the first node that contains *searchString*. Does not look for multiple matches.

`setContent (String str)`
: Sets the receiver `TreeNode's content` instance variable to *str*, a `String` object. See ⟨undefined⟩ [String], page ⟨undefined⟩.

3.35 `Event` Class

`Event` and its subclasses represent system and language events like applications, errors, input, and signals, and these classes provide the methods that receive and handle the events.

3.36 `Application` Class

Objects of `Application` class represent programs. This class provides basic methods for initializing and exiting programs, handling window size and placement for graphical programs, and starting subprograms, as well as other tasks.

Instance Variables

`cmdLineArgc`
: An `Integer` that contains the value of the `argc` argument to *main ()*. The value is normally set by the `parseArgs` method, below.

`cmdLineArgs`
: An `Array` of `Strings` that contains each element of the `argv` parameter to `main ()`. This is normally filled in by the `parseArgs` method, below.

`exitHandler`
: A `SignalHandler` object that contains a user-installable signal handler. See ⟨undefined⟩ [SignalHandler], page ⟨undefined⟩.

`geomFlags`
`winXOrg`

`winYOrg`

`winWidth`

`winHeight`
: Variables that specify the appication window's dimensions, if any. These can be set by the `parseX11Geometry` method, below.

Instance Methods

`__handleAppExit (__c_arg__ int signo)`
: A standard, user installable SIGINT (*`Ctrl-C`*) signal handler. If the application has enabled exception traces, print a stack trace and then call the default SIGINT handler. The *signo* argument contains the number of the signal that calls the method.

The application must first call the `installExitHandler` method (below) to define this method as a signal handler - then this method is called when the application receives a SIGINT, which is normally generated when the user presses Control-C.

This method does not return.

`__handleSigAbrt (__c_arg__ int` *signo*`)`
: A standard, user installable SIGABRT handler. The application must first call `installAbortHandler`, below. If the application has enabled exception traces, print a stack trace before exiting. This method does not return.

`classSearchPath (void)`
: Returns a `String` that contains Ctalk's class library search path, with each directory separated by a colon (':'). The default is usually `/usr/local/include/ctalk`.

  If there are any directories given as arguments to the '`-I`' command line option, and any directories named in the '`CLASSLIBDIRS`' environment variable, Ctalk includes those directories as well, and searches them first.

`execC (String` *commandLine*`)`
`execC (String` *commandLine*`, String` *commandOutput*`)`
: Execute the command and arguments given by *commandLine* and wait for the program to finish. The method displays the standard output of the subprocess.

  If a second argument is given, the program's standard output is saved in the *commandOutput* object, which should normally be a `String` object. Here is a simple example program.

```
int main () {
  Application new myApp;
  String new str, output;

  str = "/bin/ls -l";
  myApp execC str, output;
  printf ("%s\n", output);
}
```

  For any number of arguments, if the command redirects the standard outut, then the output is sent to the file that is the operand of a '`>`' or '`>>`' redirection operator.

  If *commandLine* is the name of a shell script, the shell script is executed by a sub-shell using the *system(3)* library call.

`getPID (void)`
: Returns an `Integer` with the program's process ID.

`installAbortHandlerBasic (void)`
: Installs a C handler for `SIGABRT` signals. The C handlers are more reliable, though less flexible, than handlers that use `SignalHandler` class.

This method catches a SIGABRT, and prints a walkback trace if tracing is enabled, before the application exits.

`installExitHandler (void)`
: Install a `SIGINT` (*`C-c`*) handler in a Ctalk program that performs cleanup before exiting the application.

`installExitHandlerBasic (void)`
: Installs a C handler for `SIGINT` signals (*`C-c`* or *`C-break`* for DJGPP) that is slightly more robust and reliable, though less flexible, than the `SignalHandler` classes.

  This method causes the application to exit when receiving a signal, and prints a walkback trace if tracing is enabled.

`installPrefix (void)`
: Returns a `String` with the name of the top-level directory where Ctalk's various component subdirectories are located. For example, in relative terms, this is where Ctalk's various components get installed.

```
Executables:          prefixdir/bin
Libraries:            prefixdir/lib
Class Libraries:      prefixdir/include/ctalk
Texinfo Manuals:      prefixdirshare/info
Manual Pages:         prefixdir/share/man
Searchable Docs:      prefixdir/share/ctalk
```

`membervars (void)`
: Returns a `String` with the member variable declarations of the class named by the receiver. If the receiver is a `String` object, `membervars` returns the variables for the class given by the `String's` value. If the receiver is a class object, `membervars` returns the member variables for that class. For nany other type of object, `membervars` returns the variables declared in the object's class. See ⟨undefined⟩ [ClassVariableKeyword], page ⟨undefined⟩.

  `Membervars` returns the complete documentation of the instance or class variable, if the declaration also contains a documentation string. See ⟨undefined⟩ [VariableDocStrings], page ⟨undefined⟩.

`methodDocString (String` *`method-source`*`)`
: Returns the documentation string from the source of the method given as the argument. See ⟨undefined⟩ [MethodDocStrings], page ⟨undefined⟩.

`methodPrototypes (String` *`input`*`)`
: If *input* is a method's source or a class library, returns a `String` containing the prototypes of the methods; that is, the declaration and the argument list.

`methodSource (String` *`className`*`, String` *`methodName`*`)`
: Returns a `String` object with the source code of the method(s) that match the declarations, The `methodSource` method does not distinguish methods by the number of arguments, so it returns the code of any method in a class that matches the *methodName* arguement.

```
className instanceMethod methodName
className classMethod methodName
```

The method generates a `SystemErrnoException` if it can't find the class file.

The `methodSource` method is built to be as fast as possible and has a rather simple minded view of what constitutes a method declaration.

The method only recognizes declarations that appear on a single line, in order to keep the regular expressions that do the matching as simple as possible, and it only matches spaces between tokens, and not tabs, at least at the moment. It can also be fooled by things that even *look* like a method declaration within the method body. For example, a phrase like,

```
className instanceMethod <some-method-name>
```

in the method's documentation, causes `methodSource` to signal the start of the next method.

`parseArgs (Integer argc, Array argv)`
Takes the *argv* and *argc* parameters to `main ()` and sets each element of the `cmdLineArgs` instance variable (above) to a `String` that contains each element of the system's `argv` array.

`parseX11Geometry (String geometryString)`
Parses a X Window System geometry string. If the string specifies any of the x, y, width, or height values for a window, the method sets the `winXOrg`, `winYOrg`, `winWidth`, or `winHeight` instance variables. If the geometry string omits any of the values, the method sets the corresponding instance variable to zero. The method sets the `geomFlags` instance variable to the geometry flags provided by the window system.

This method only parses the geometry string. It does not make any adjustments for the display dimensions, or the window dimensions or placement.

For information about the format of a X geometry string, refer to the *XParseGeometry(3)* manual page.

`spawnC (String command, Integer restrict)`
Starts the program given by the argument *command*, and resumes execution of the main program. The method returns an `Integer` with the process ID of the child process.

The child process becomes a daemon process, which means it has no interaction with the parent process. If you want the parent process to handle the child processes' output, refer to the `execC` method, above. Otherwise, communication between the parent and child process should be handled by UNIX's interprocess communication facilities.

If the *restrict* argument is non-zero, the method also changes the child processes' working directory to '`/`' and its umask to '`0`'.

The `spawnC` method does not use a shell when executing the child process, which means that the method doesn't handle shell facilities like IO redirection or file globbing. It's also necessary to provide the full path name of the program to be launched in the background.

The process that handles the session management when the daemon process is launched remains executing until the parent process exits. This means there can be *three* entries in the system's process table, but it helps minimize creating zombie processes in case any part of the program quits unexpectedly.

`uSleep (long long int usecs)`
: Sleep for *usecs* microseconds.

`useXRender (Boolean b)`
: If *b* is true, draw graphics using the X Render extension if it is available. If *b* is false, use Xlib for graphics drawing. The default is to draw using the X Render extension if it is available.

`usingXRender (void)`
: Returns a `Boolean` value of True if the program is using the X Render extension for drawing, False otherwise.

3.37 `ClassLibraryTree` Class

A `ClassLibraryTree` application is a utility program that formats a tree of the available Ctalk class hierarchy.

The class lists the libraries in Ctalk's default installation directory, and in any directories named by the '`CLASSLIBDIRS`' environment variable.

The class contains one method, `init`, which collects the class and superclass information into a set `TreeNode` objects. The prototype of init is:

```
TreeNode instanceMethod init (TreeNode classTree, Boolean verbose);
```

After the method finishes the `classTree` object contains the classes and subclasses. You can then print or format the with the `TreeNode` methods `format` and `print`.

The `verbose` argument, if True, tells the method to print dots to indicate its progress.

Here is an example program that displays the class hierarchy on a terminal.

```
Boolean new verbose;

int main (int argc, char **argv) {

  TreeNode new tree;
  ClassLibraryTree new classTree;
  Integer new nParams;
  Integer new i;
  String new param;
```

```
  verbose = True;

  classTree parseArgs argc, argv;
  nParams = classTree cmdLineArgs size;

  for (i = 1; i < nParams; i++) {

    param = classTree cmdLineArgs at i;

    if (param == "-q") {
      verbose = False;
      continue;
    }

    if (param == "-h" || param == "--help") {
      printf ("Usage: classes [-q] [-h]\n");
      exit (0);
    }

  }

  classTree init tree, verbose;
  tree print;

  exit (0);
}
```

Instance Methods

`init instance method (TreeNode` *`tree`*`, Boolean` *`printDots`*`)`

Creates a a tree of the class library with *tree* as the root node of the tree. If *printDots* is true, prints the method's progress on the terminal.

3.38 `GLUTApplication` Class

The `GLUTApplication` class provides a class library interface to the GLUT application programming interface, which is a platform-independent window API for OpenGL programs.

The GLUT API uses C functions to handle window system events. This is the simplest approach to adding event handlers. Ctalk does not, itself, use any GLUT or OpenGL functions, so it is normally safe to use Ctalk methods within a window callback, as in this example.

```
#include <ctalk/ctalkGLUTdefs.h>

GLUTApplication new teapotApp;

float angle = 0.0f;
```

```
void mydisplay (void) {       /* This callback updates the display. */
  glEnable (GL_NORMALIZE);
  glEnable(GL_DEPTH_TEST);
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
  glLineWidth (1.0f);

  glLoadIdentity ();
  glColor4f (0.0f, 0.0f, 1.0f, 1.0f);

  glRotatef (angle, 0.0f, 1.0f, 0.0f);
  glRotatef (10.0f, 1.0f, 0.0f, 0.0f);

  teapotApp teapot 3.0, 0;    /* Method call within the callback. */

  glutSwapBuffers ();
}

void animation (void) {  /* Callback to update the animation state. */
  angle += 0.2f;
  if (angle >= 360.0f)
    angle = 0.0f;
  glutPostRedisplay();
}

int main (int argc, char **argv) {

  teapotApp initGLUT(argc, argv);
  teapotApp initWindow (640, 480);
  teapotApp createMainWindow ("teapotApp -- GLUTApplication Class");
  teapotApp defineDisplayFn mydisplay;
  teapotApp defineIdleFn animation;
  teapotApp installCallbackFns;
  teapotApp run;

}
```

If you save the example in a file called teapotApp.ca, you can build this program with the following command on Linux/UNIX systems.

```
$ ctcc -x teapotApp.ca -o teapotApp
```

To build the example on Apple OS X machines, you need to link it with the GLUT application framework. Refer to the discussion below for more information.

You can also use the OpenGL API within methods. This lets programs use C functions, methods, and the GLUT and OpenGL APIs together with few restrictions. The `GLUTApplication` class does not, at least at this time, support all of the functions that the GLUT API provides, but the API is flexible enough that you can extend it if necessary

Apple OS X Machines

OS X machines include GLUT as an application framework, which Ctalk does not support directly. Ctalk provides the platform independent `ctalkGLUTdefs.h` include file, which you can use to add the GLUT definitions to programs, by adding a statement like this one to the program.

```
#include <ctalk/ctalkGLUTdefs.h>
```

Then you need to link the program with the framework to produce an executable. Normally you would use a series of commands like these.

```
$ ctalk -I /usr/X11R6/include teapotApp.ca -o teapotApp.i
$ gcc -framework GLUT teapotApp.i -o teapotApp -lctalk -lreadline \
      -L/usr/X11R6/lib -lGL -lGLU
```

This example isn't meant to be definitive. You might need to experiment to find the right build configuration for a particular OS X machine.

There is more platform specific information in the example programs and `README` file in the `demos/glut` subdirectory.

The `GLUTApplication` class doesn't provide a guide to the very involved subject of programming with the GLUT and OpenGL APIs. There are many references and tutorials available on the Internet and in bookstores that teach OpenGL programming.

Instance Methods

`createMainWindow (String` *`title`*`)`
: Create the main window. The argument, a `String` contains the window's title.

`cone (Float` *`base`*`, Float` *`height`*`, Integer` *`slices`*`, Integer` *`stacks`*`, Integer` *`fill`*`)`
: Draw a cone with a base of size *base*, height *height*, with *slices* longitudinal slices and *stacks* lateral slices. If *fill* is True, draw a filled cone; otherwise, draw a wireframe cone.

`cube (Float` *`size`*`, Integer` *`fill`*`)`
: Draw a cube with sides of *size* length. If *fill* is True, draw a filled cube; otherwise draw a wirefame cube.

`defineAnimationFn (Symbol fn)`
`defineButtonBoxFn (Symbol fn)`
`defineDialsFn (Symbol fn)`
`defineDisplayFn (Symbol fn)`
`defineEntryFn (Symbol fn)`
`defineIdleFn (Symbol fn)`
`defineKeyboardFn (Symbol fn)`
`defineMenuStateFn (Symbol fn)`
`defineMenuStatusFn (Symbol fn)`
`defineMotionFn (Symbol fn)`
`defineMouseFn (Symbol fn)`
`defineOverlayDisplayFn (Symbol fn)`
`definePassiveMotionFn (Symbol fn)`
`defineSpaceballMotionFn (Symbol fn)`
`defineSpaceballRotateFn (Symbol fn)`
`defineSpecialFn (Symbol fn)`
`defineTabletButtonFn (Symbol fn)`
`defineTabletMotionFn (Symbol fn)`
`defineVisibilityFn (Symbol fn)`
`defineTimerFn (Integer msec, Symbol fn, Integer argValue)`

Define callback functions for window system events. The argument is the name of the C function that handles the event. Refer to the method's documentation and the GLUT API's documentation for information about each callback function's parameters.

Note: `defineAnimationFn` and `defineTimerFn` both use the `glutTimerFunc ()` callback, so the actual callback is the last function defined (before calling `installCallBackFns`).

The difference is that `defineAnimationFn` executes its callback 24 times per second, while `defineTimerFn` uses an application defined interval, and takes three arguments: the timer interval in milliseconds, the name of the callback function, and an integer that is passed to the callback function as an argument.

`defineReshapeFn (Symbol fn)`

Defines the function used to reshape the window's 3D viewing space. The default is an orthographic projection, 5 grid units wide on each axis. If you need to duplicate the default viewing area in a program, here is the code that defines it. The variables '`width`' and '`height`' are provided by the GLUT application as arguments to the reshape function.

```
void defualt_reshape_fn (int width, int height) {
  float ar = (float)width / (float)height;
  glViewport(0, 0, w, h);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho (-5.0 * ar, 5.0 * ar, -5.0, 5.0, 5.0, -5.0);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
```

```
}
```

`dodecahedron (Integer fill)`
: Draw a dodecahedron. If `fill` is True, draw a solid dodecahedron; otherwise draw a wireframe dodecahedron.

`fullScreen (void)`
: Resize the main window so that it occupies the entire screen. A call to `reshape` (the method, not the callback) or `position` returns the window to its normal state.

`icosahedron (Integer fill)`
: Draw a icosahedron. If `fill` is True, draw a solid icosahedron; otherwise draw a wireframe icosahedron.

`initGlut (Integer argc, Array argv)`
: Initialize the GLUT window system. If there are any GLUT specific command line arguments, this method parses them and uses then uses the options to configure the window system. This method also calls `Application` method `parseArgs`, so the command line arguments are available in the `cmdLineArg` instance variable (an `Array`). See ⟨undefined⟩ [Application], page ⟨undefined⟩.

`initWindow (Integer width, Integer height)`
`initWindow (Integer xOrg, Integer yOrg, Integer width, Integer height)`
: Initialize the main window's size or, with four arguments, its size and position.

`installCallBackFns (void)`
: Install the callback functions defined by previous calls to `define*Fn` methods.

`octahedron (Integer fill)`
: Draw a octahedron. If `fill` is True, draw a solid octahedron; otherwise draw a wireframe octahedron.

`tetrahedron (Integer fill)`
: Draw a tetrahedron. If `fill` is True, draw a solid tetrahedron; otherwise draw a wireframe tetrahedron.

`run (void)`
: Enter the GLUT API's main event loop. When this method returns, the program typically exits. *Note:* GLUT does not provide any events to terminate program. To exit a program normally, use C's *exit (3)* function and *on_exit(3)* (or *onexit(3)*) to handle any program specific exit processing.

`sphere (Float radius, Integer slices, Integer stacks, Integer fill)`
: Draw a sphere with *radius* with *slices* longitudinal sections and *stacks* lateral sections. If *fill* is True, draw a filled sphere; otherwise draw a wireframe sphere.

`teapot (Integer fill)`
: Draw the classic teapot demonstration. If *fill* is True, draw a solid teapot; otherwise draw a wireframe teapot.

`torus (Float inner_radius, Float outer_radius, Integer size, Integer rings, Integer fill)`

Draw a torus with the inner and outer radii given by the arguments, with a section *size* and rendered in *rings* sections. If *fill* is True, draw a solid torus; otherwise draw a wireframe torus.

`windowID (String window_title)`

Return an `Integer` with the X window ID of the window with the title *window_title*. *Note:* Some OS's, like OSX/Darwin, don't use Xlib to draw windows; in that case, this method won't be able to provide a lower level window ID.

`xpmToTexture (char **xpm_data, Integer width_out, Integer height_out, Symbol texture_data_out)`

`xpmToTexture (char **xpm_data, Integer alpha, Integer width_out, Integer height_out, Symbol texture_data_out)`

Translates a XPM pixmap into an OpenGL texture. The argument *xpm_data* is the pixmap's `char *pixmap_name[]` declaration. If no *alpha* argument is given, then '`1.0`' is used to create an opaque texture. Alpha values can range from 0.0 (completely transparent) - 1.0 (completely opaque).

The method sets the arguments *width_out*, *height_out*, and *texel_data_out* with the height, width and data of the texture.

The resulting texture has the format GL_RGBA and the data type GL_UNSIGNED_INT_8_8_8_8, so you can create a 2D texture from a pixmap with statements like these.

```
Integer new xpmWidth;
Integer new xpmHeight;
Symbol new texData;

/*
 *  Note that the xpm_data argument should not normally need a
 *  translation from C.
 */
myGLUTApp xpmToTexture xpm_data, xpmWidth, xpmHeight, texData;
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, xpmWidth, xpmHeight, 0,
              GL_RGBA, GL_UNSIGNED_INT_8_8_8_8, texData);
```

The `xpmToTexture` method does not do any setup of the OpenGL texture environment. For basic textures, OpenGL works better with textures that have a geometry that is an even multiple of 2; e.g., 128x128 or 256x256 pixels.

Individual applications can add parameters for interpolation, blending, mipmap creation, and material rendering based on the program's requirements, though.

The Ctalk library only stores the data for one texture at a time, so if a program uses multiple textures, it should save the texture data to a separate `Symbol`, in orderq to avoid regenerating the texture each time it's used.

For an example of how to draw with textures, refer to the `texture.ca` and `texblend.ca` programs in the Ctalk distribution's `demos/glut` subdirectory.

3.39 `ObjectInspector` Class

The `ObjectInspector` class provides basic methods for examining the contents of objects.

The central method of this class is `formatObject,` which returns the contents of an object and its instance variables as a `String` object. However, applications can also use the `formatInstanceVariable` and `formatClassVariable` methods.

These methods use the `mapInstanceVariables` and `mapClassVariables` methods, defined in `Object` class. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

Other classes provide convenience methods that call `formatObject;` for example, `dump` in `Object` class.

Instance Variables

`promptString`
: A `String` object that contains the text of the inspector's command prompt. See ⟨undefined⟩ [String], page ⟨undefined⟩.

`verbose`
: A `Boolean` value that causes the inspector to print a verbose listing of objects when set to True. See ⟨undefined⟩ [Boolean], page ⟨undefined⟩.

Instance Methods

`formatClassVariable (Symbol` *`objectRef`*`)`
: Return a `String` containing the contents of an object and its instance variables. The argument is a reference to the object to be formatted. Both `formatClassVariable` and `formatInstanceVariable` are used by `formatObject,` below.

`formatInstanceVariable (Symbol` *`objectRef`*`)`
: Return a `String` containing the contents of an object and its instance variables. The argument is a reference to the object to be formatted.

`formatObject (Symbol` *`objectRef`*`)`
: Return a `String` containing the contents of an object and its instance variables. The argument is a reference to the object to be formatted.

`inspect (Symbol` *`objectRef`*`)`
`inspect (Symbol` *`objectRef`*`, String` *`promptStr`*`)`
: Suspends execution of the program and enters the Object inspector. This allows you to examine the object pointed to by *objRef*, continue execution, or exit the program. Typing '`?`' or '`help`' at the prompt prints a list of the inspector's commands. With a `String` object as the second argument, the inspector displays the string as its command prompt.

  The manual page, *inspect(3ctalk)*, contains a description the inspector's commands and a brief tutorial.

3.40 `LibrarySearch` Class

The methods in `LibrarySearch` class perform searches on the Ctalk reference documentation. The documentation is derived from this reference and is formatted to be easily searchable. Ctalk installs the documentation files during the installation process.

These methods are also used in the `searchlib` program to search the documentation.

Note that these methods raise an exception if they can't open the documentation file, in which case they return an empty string and leave it up to the calling function or method to handle the exception.

The `searchlib` program is described in the *searchlib(1)* man page and the `ctalktools` Texinfo documentation.

Instance Methods

`cAPIFunctionSearch (String` *`docPath`*`, String` *`search_pattern`*`)`
: Returns a `String` that contains the results of a search for the C API function or functions that match *search_pattern*.

`methodSearch (String` *`docPath`*`, String` *`search_pattern`*`)`
: Search the class library's method prototypes for *search_pattern*. Returns a `String` object with the results. Note that this function is designed to be rather inclusive; it searches the *entire* method prototype for a pattern, which may include the receiver class's name or any parameter name(s). So including the receiver class name is a search is perfectly valid - the syntax of a method prototype in the documentation is:

    `<receiver_class>::<method_selector> (param_list)`

3.41 `Exception` Class

`Exception` and its subclasses handle system and language errors. These classes provide default and user-defined handlers to respond to the Exceptions;

If a program calls `enableExceptionTrace` (class `Object`), then the `handle` method, below, and other exception handlers print a walkback of the program's method stack. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

Here is an example of how to handle an exception when trying to open a file.

```
Exception new e;
...
e enableExceptionTrace;
...
inputStream openOn fileArg;
if (e pending) {
  e handle;
  exit (1);
}
```

The `pending` method returns `TRUE` or `FALSE` depending on whether an exception is waiting to be processed. The `handle` method processes the exception.

The class's default method handlers format and print the text provided when a program calls `raiseException`. Here is the code from `openOn` (class `ReadFileStream`) that creates the exception.

```
SystemErrnoException new e;
...
if ((f = fopen (__streamPath, "r")) != NULL) {
  __ctalkObjValPtr (selfval, f);
  self streamPath = __streamPath;
  self statStream;
} else {
  e raiseCriticalException __streamPath;
  strcpy (selfval -> __o_value, "");
}
```

The object `e` is a `SystemErrnoException`. The `SystemErrnoException` class translates the operating system's errors into Ctalk exceptions. In the example above, the type of the exception is provided by the operating system.

The `Exception` class also provides the `installHandler` method to allow programs to use their own handlers. Here is an example.

```
Exception instanceMethod myHandler (void) {
  printf ("My exception handler: %s.\n",
  __ctalkGetRunTimeException ());
  return NULL;
}

int main () {
  Exception new e;
  e installHandler "myHandler";
  e raiseException USER_EXCEPTION_X, "my program's exception";
  e handle;
}
```

Some notes about exceptions:

1. The receiver class of an exception handler is the the same as the exception object; e.g., `myHandler` in the example above is the same as the class of `e`; that is, `Exception`. In addition, exception handlers take no arguments and rely on information at the time the exception is raised.

2. Programs must take care that they handle any exceptions that they raise as soon as possible. If not handled by the application, the run-time library API might then handle the exception, which could lead to confusing results.

3. Ctalk uses *critical exceptions* internally. At the application level, however, all exceptions are treated with the same priority. The application needs to take the appropriate action, depending on the type of exception.

4. Exception classes should not, as much as possible, depend on other classes being evaluated. This is because even basic classes use exceptions, and if the exceptions in turn use the basic classes, it can lead to circular reference. In fact, `Symbol` is one of the few classes that `Exception` can use without generating circular method and class references. That is why the `handlerMethod` instance variable (see below) is implmented as a `Symbol`.

5. When writing exception handlers, you should use as much as possible the following library API functions: __ctalkGetRunTimeException, __ctalkHandleRunTimeException, __ctalkPeekRunTimeException, __ctalkPeekExceptionTrace, and __ctalkPendingException. They are located in `lib/except.c` and described in a later section. See ⟨undefined⟩ [Ctalk library], page ⟨undefined⟩.

6. The only way to distinguish between different exceptions within a handler is to compare the text returned by a function like __ctalkGetRunTimeException. The exception information is generated at the time the exception is raised, which is the information that the program should be interested in. Handling exceptions as soon as possible also helps avoid confusion if one exception then causes other exceptions. Also, different operating systems map errors codes differently, so Ctalk uses the operating systems' interpretations of the errors.

7. Exception handlers in calling methods take priority over handlers in the methods they call. You need to be careful that exception handlers within the scope of a caller do not supercede each other. This is especially true if an application tries to use a global Exception object. You should try to keep Exception objects as local as possible.

Exception Codes

Exceptions have a corresponding integer code. The following macros correspond to the actual exceptions.

```
SUCCESS_X
CPLUSPLUS_HEADER_X
MISMATCHED_PAREN_X
FALSE_ASSERTION_X
FILE_IS_DIRECTORY_X
FILE_ALREADY_OPEN_X
UNDEFINED_PARAM_CLASS_X
PARSE_ERROR_X
INVALID_OPERAND_X
PTR_CONVERSION_X
UNDEFINED_CLASS_X
UNDEFINED_METHOD_X
METHOD_USED_BEFORE_DEFINE_X
SELF_WITHOUT_RECEIVER_X
UNDEFINED_LABEL_X
UNDEFINED_TYPE_X
UNDEFINED_RECEIVER_X
UNKNOWN_FILE_MODE_X
INVALID_VARIABLE_DECLARATION_X
WRONG_NUMBER_OF_ARGUMENTS_X
SIGNAL_EVENT_X
INVALID_RECEIVER_X
NOT_A_TTY_X
USER_EXCEPTION_X
```

Instance Variables

`handlerMethod`
: A `Symbol` object that contains the address of a user defined method to handle exceptions.

Instance Methods

`deleteLastException (void)`
: Delete the last generated exception from Ctalk's internal exception list.

`handle (void)`
: Execute the exception handler for the next pending method. If a program calls `enableExceptionTrace` (class `Object`), then `handle` also displays a walkback of the exception's copy of the program call stack. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

`exceptionHandler (void)`
: Handle an exception either with Ctalk's default exception handler or a user defined handler. If the program has called the `traceEnabled` method (class `Object`), print a stack trace also.

`installHandler (char *handler_method_name)`
: Install a user-defined method to handle exceptions. The method's receiver class must be of the same class as the exception; i.e., either `Exception` or `SystemErrnoException`.

`peek (void)`
: Returns a `String` containing the text of the first pending exception, if any.

`pending (void)`
: Return `TRUE` if any exceptions are pending, `FALSE` otherwise.

`raiseCriticalException (EXCEPTION ex, char *text)`
: Raise a critical exception. A critical exception is similar to a normal exception, below, except that it is not caught internally. The application must catch the exception with `pending` and `handle`.

  You should use one of the `EXCEPTION` macros defined above as the argument *ex*.

  Print a trace of the methods in the current exception's copy of the program's call stack.

`raiseException (EXCEPTION ex, char *text)`
: Raise an exception. You should use one of the `EXCEPTION` macros defined above as the argument *ex*.

3.42 `SystemErrnoException` class

The `SystemErrnoException` class translates operating system errors into Ctalk exceptions. The exceptions, which are specific to each operating system, correspond to the interpretation of the system's `errno` global variable and allows Ctalk to work with system errors even if the system implements its error codes via a built-in variable, a macro, or function.

There is an example of a system error handler in the `Exception` class section. See ⟨undefined⟩ [Exception], page ⟨undefined⟩.

Class Variables

`sysErrno` The `sysErrno` object contains the value of the C library's `errno` variable or macro.

Instance Methods

`raiseException (char *data)`
: Sets the value of the `sysErrno` class variable and raises an exception if the C library's `errno` variable or macro returns a nonzero value.

3.43 `InputEvent` Class

Instance Variables

`eventClass`
: An `Integer` that defines the class of the input event. Ctalk defines the following input event classes.

```
KBDCHAR         # ASCII Characters
KBDCUR          # Cursor Keys.
```

`eventData`
: An `Integer` that contains the data specific to an input event.

  For example, when used with subclasses of `ANSITerminalPane, eventData` holds the ASCII value of a keypress.

  When used with subclasses of X11Pane (actually, any subclass or program that uses X11TerminalStream : queueInput), `eventData` generally holds the ID of the window that received the event.

  The GUI event types and the data that a `X11TerminalStream` object returns are described in the `X11TerminalStream` class. See ⟨undefined⟩ [XEventClasses], page ⟨undefined⟩.

`xEventData1`
`xEventData2`
`xEventData3`
`xEventData4`
`xEventData5`
: Data provided by the `X11TerminalStream` input handler. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

3.44 `SignalEvent` Class

The `SignalEvent` class provides methods and variable definitions for Ctalk programs to handle signal events.

Signal handler (⟨undefined⟩ [SignalHandler], page ⟨undefined⟩) methods act like C functions when a program installs them to handle signals. The signal handler methods cannot, in most

cases, create objects. Ctalk provides the C function `__ctalkNewSignalEventInternal` to create and queue `SignalEvent` objects from the handler.

Here is an example of a signal handler method that creates `SignalEvent` objects.

```
#include <time.h>

SignalHandler instanceMethod handleSignal (__c_arg__ int signo) {
  time_t t;
  char buf[MAXLABEL];
  noMethodInit;
  t = time (NULL);
  /* Format the system time, then create a new SignalEvent object,
     and add the time data to the new object's text instance variable. */
  __ctalkDecimalIntegerToASCII (t, buf);
  __ctalkNewSignalEventInternal (signo, getpid (), buf);
  return NULL;
}
```

For more information, ⟨undefined⟩ [Method functions], page ⟨undefined⟩, and ⟨undefined⟩ [__ctalkNewSignalEventInternal], page ⟨undefined⟩.

Class Variables

`pendingEvents`
: A `List` of pending signal events.

Instance Variables

`processID`
: An `Integer` that contains the process ID of the program that received the signal.

`sigNo`
: An `Integer` that contains the signal number of the handler.

`data`
: A `String` object that contains data from the signal handler.

Instance Methods

`getPID (void)`
: Set the `SignalEvent` object's `pid` instance variable to the program's process ID.

`new (`*`event1`*`, `*`event2`*`, ... `*`event3`*`;)`
: Create one or more new `SignalEvent` objects with the names given in the argument lists.

```
SignalEvent new event1;
SignalEvent new event1, event2;
```

`nextEvent (void)`
: Return the next `signalEvent` object from the class' `pendingEvents` queue.

`pending (void)`
: Return `TRUE` if the class' `pendingEvents` queue contains `SignalEvent` event objects, `FALSE` otherwise.

`queueEvent (void)`
: Add the receiver to the class' `pendingEvents` queue.

3.45 `SignalHandler` Class

Class `SignalHandler` provides the methods that install handlers for signals from the operating system.

Applications can also define signal handlers with this class. Signal handler methods need to use the calling conventions of C functions. See ⟨undefined⟩ [Method functions], page ⟨undefined⟩.

For POSIX signals, you can use a method to set the signal number.

Signal	Method	Signal
SIGHUP	setSigHup	Termination of terminal process.
SIGINT	setSigInt	Interrupt from keyboard.
SIGQUIT	setSigQuit	Quit from keyboard.
SIGILL	setSigIll	Illegal instruction.
SIGABRT	setSigAbrt	Abort from C library abort(3).
SIGFPE	setSigFpe	Floating point exception.
SIGKILL	-	Kill process - non-catchable.
SIGSEGV	setSigSegv	Invalid memory address.
SIGPIPE	setSigPipe	Broken pipe or write to pipe with no reader.
SIGALRM	setSigAlrm	Timer signal from C library alarm(2).
SIGTERM	setSigTerm	Process termination.
SIGUSR1	setSigUsr1	User defined.
SIGUSR2	setSigUsr2	User defined.
SIGCHLD	setSigChld	Child process stopped or terminated.
SIGCONT	setSigCont	Continue stopped process.
SIGSTOP	-	Stop process - non-catchable.
SIGTSTP	setSigTstp	Stop from tty.
SIGTTIN	setSigTtin	Terminal input for background process.
SIGTTOU	setSigTtou	Terminal output from background process.

You can also set system-specific signals by number with `setSigNo`.

`SignalHandler` methods can also send `SignalEvent` objects to Ctalk programs. Refer to ⟨undefined⟩ [Method functions], page ⟨undefined⟩, ⟨undefined⟩ [SignalEvent], page ⟨undefined⟩, and ⟨undefined⟩ [__ctalkNewSignalEventInternal], page ⟨undefined⟩.

Internally, methods in `SignalHandler` class use the library functions `__ctalkIgnoreSignal ()`, `__ctalkDefaultSignalHandler ()`, `__ctalkInstallHandler ()`, and `__ctalkSystemSignalNumber`. Signal handlers need to be reset after each usage. Refer to the `timeclient.c` example program.

Instance Variables

`attributes`
: The value of `attributes` can be one of the following.

  `SIG_DEFAULT`
  : The application uses the operating system's default signal handler for the signal. The operating system's documentation describes how it handles signals.

  `SIG_IGNORE`
  : The application ignores the signal.

  `SIG_METHOD`
  : The application provides a method to handle the signal.

`handler`
: The value is an instance method of class `SignalHandler` that is provided by the application.

Instance Methods

`defaultHandler (void)`
: Install the operating system's default handler for the receiver's signal number `sigNo.` The operating system's documentation describes how it handles signals.

`ignoreSignal (void)`
: Set the receiver signal handler to ignore a signal.

`installHandler (OBJECT *(*method)(int))`
: Install *method* as the receiver's signal handler. The method must be callable as a C function. See ⟨undefined⟩ [Method functions], page ⟨undefined⟩.

`new (char *name)`
: Create new `SignalHandler` objects with the name(s) given in the argument list.

`raiseSignal (void)`
: Send the signal of the receiver's `sigNo` variable to the current program.

`setSigAbrt (void)`
: Set the signal of the receiver's handler to `SIGABRT.`

`setSigAlrm (void)`
: Set the signal of the receiver's handler to `SIGALRM.`

`setSigChld (void)`
: Set the signal of the receiver's handler to `SIGCHLD.`

`setSigCont (void)`
: Set the signal of the receiver's handler to `SIGCONT.`

`setSigFpe (void)`
: Set the signal of the receiver's handler to `SIGFPE.`

`setSigHup (void)`
: Set the signal of the receiver's handler to `SIGHUP.`

`setSigIll (void)`
: Set the signal of the receiver's handler to `SIGILL`.

`setSigInt (void)`
: Set the signal of the receiver's handler to `SIGINT`.

`setSigNo (int` *`signum`*`)`
: Set the signal number of the receiver to *signum*.

`setSigPipe (void)`
: Set the signal of the receiver's handler to `SIGPIPE`.

`setSigQuit (void)`
: Set the signal of the receiver's handler to `SIGQUIT`.

`setSigSegv (void)`
: Set the signal of the receiver's handler to `SIGSEGV`.

`setSigTerm (void)`
: Set the signal of the receiver's handler to `SIGTERM`.

`setSigTstp (void)`
: Set the signal of the receiver's handler to `SIGTSTP`.

`setSigTtin (void)`
: Set the signal of the receiver's handler to `SIGTTIN`.

`setSigTtou (void)`
: Set the signal of the receiver's handler to `SIGTTOU`.

`setSigUsr1 (void)`
: Set the signal of the receiver's handler to `SIGUSR1`.

`setSigUsr2 (void)`
: Set the signal of the receiver's handler to `SIGUSR2`.

`signalProcessID (int` *`processid`*`)`
: Send the signal `sigNo` of the receiver to process *processid*.

`sigName (Integer` *`signal_number`*`)`
: Return a `String` with the name of the signal whose number is given as the argument.

`sigNum (String` *`signal_name`*`)`
: Return an `Integer` with the value of the signal whose name is given as the argument.

`waitStatus (Integer` *`child_pid`*`, Integer`*`child_return_val`*`, Integer` *`child_signal`*`, Integer` *`errno`*`)`
: Checks for a change in the status of the child process given by *child_pid*.

  The return value is an `Integer` with the value 0, which indicates that the child process has not changed status, an `Integer` equal to *child_pid*, or -1.

  If the return value is equal to *child_pid*, then the processes' return code is returned in *child_return_val* if the process exited normally. If the child process was terminated by an uncaught signal, the signal's number is returned in *child_signal*.

If the return value is -1, the system *errno* is returned in *errno*, which indicates an error when the parent process called `waitStatus`.

Here is an example.

```
SignalHandler new s;
Integer new r, childProcessID, child_retval, child_sig,
        child_errno;

 ... do stuff ...

 r = s waitStatus childProcessID,
child_retval, child_sig, child_errno;

 if (r == childProcessID) {
    if (child_sig) {
       printf ("Child received signal %s - exiting.\n",
  s sigName child_sig);
  exit (1);
    }
 }

 ... do more stuff ...

 exit (0);
```

3.46 `Expr` Class

Ctalk uses objects of class `Expr` and its subclasses internally to represent C expressions, functions, and variables.

3.47 `CFunction` Class

Ctalk uses the `CFunction` class internally to represent C function calls. Ctalk installs the `CFunction` methods library templates in `CLASSLIBDIR/libc` and `CLASSLIBDIR/libctalk`.

Note that this class does not, at this time, provide methods for all C99 library function calls. If you want to add C library templates, look at the templates in `classes/libc` and at the API documentation. See ⟨undefined⟩ [Templates], page ⟨undefined⟩. Refer to the `README` file in the Ctalk distribution for instructions on submitting source code contributions.

Class Methods

`cAbs (int i)`
: Return the result object of an *abs(3)* library call.

`cAcos (double d)`
: Return the result object of an *acos(3)* library call.

`cAcosh (double d)`
: Return the result object of an *acosh(3)* library call.

`cAscTime (struct tm *tm)`
: Return the result object of an *asctime(3)* library call. The value of *tm* is the result of a *gmtime(3)* or *localtime(3)* call.

`cAsin (double d)`
: Return the result object of an *asin(3)* library call.

`cAsinh (double d)`
: Return the result object of an *asinh(3)* library call.

`cAtof (char *s)`
: Return the result object of an *atof(3)* library call.

`cAtoi (char *s)`
: Return the result object of an *atoi(3)* library call.

`cAtol (char *s)`
: Return the result object of an *atol(3)* library call.

`cAtoll (char *s)`
: Return the result object of an *atoll(3)* library call.

`cCbrt (double d)`
: Return the result object of a *cbrt(3)* library call.

`cCeil (double d)`
: Return the result object of a *ceil(3)* library call.

`cChdir (char *s)`
: Change the working directory to *s*. Returns `0` on success, `-1` on error, and sets the system error number.

`cClearErr (OBJECT * FILECLASSOBJECT)`
: Perform a *clearerr(3)* library call on the file stream of the argument.

`cClock (void)`
: Return the result object of a *clock(3)* library call.

`cCopySign (double d, double s)`
: Return the result object of a *copysign(3)* library call.

`cCos (double d)`
: Return the result object of a *cos(3)* library call.

`cCosh (double d)`
: Return the result object of a *cosh(3)* library call.

`cCTime (int *t)`
: Return the result object of a *ctime(3)* library call.

  Note: Because *ctime(3)* uses a pointer to `int` as its argument, arguments to `cCTime` should be of class `Symbol`. See ⟨undefined⟩ [Objects in Function Arguments], page ⟨undefined⟩.

`cDiffTime (int time1, int time2)`
: Return the result object of a *difftime(3)* library call.

`cErf (double d)`
: Return the result object of an *erf(3)* library call.

`cErfc (double d)`
: Return the result object of an *erfc(3)* library call.

`cExp (double d)`
: Return the result object of an *exp(3)* library call.

`cExpm1 (double d)`
: Return the result object of an *expm1(3)* library call.

`cFabs (double d)`
: Return the result object of a *fabs(3)* library call.

  Note: The functions *fabsf(3)* and *fabsl(3)* functions are not supported on Solaris systems that don't have `math_c99.h`. Use *fabs(3)* or `cFabs` instead.

`cGetchar (void)`
: Return an object of class `Integer` from the standard input.

`cGetCwd (void)`
: Return the current directory as a `String` object.

  Return a `String` object with the result of a *getenv(3)* C library function call.

`cGetPID (void)`
: Return an `Integer` object with the result of a *getpid(3)* C library function call.

`cLrint (double)`
`cLrintf (float)`
`cLrintl (long double)`
`cLlrint (double)`
`cLlrint (float)`
`cLlrint (long double)`
: Return an `Integer` or `LongInteger` object that is the result of a *lrint(3)*, *lrintf(3)*, *lrintl(3)*, *llrint(3)*, *llrintf(3)*, or *llrintf(3)* library call.

`cStrcat (char *s1, char *s2)`
: Return the result of a *strcat(3)* library function call.

`cStrcasecmp (char *s1, char *s2)`
: Return the result of a *strcasecmp(3)* library call.

`cStrcmp (char *s1, char *s2)`
: Return the result of a *strcmp(3)* library call.

`cStrlen (char *s)`
: Return the result of a *strlen(3)* library call.

`cStrncat (char *s1, char *s2, int n)`
: Return the result of a *strncat(3)* library call.

`cStrncasecmp (char *s1, char *s2, int n)`
: Return the result of a *strncasecmp(3)* library call.

`cStrncmp (char *s1, char *s2, int n)`
: Return the result of a *strncmp(3)* library call.

`cStrncpy (char *s1, char *s2, int n)`
: Return the result of a *strncpy(3)* library call.

`cRand (void)`
: Return the result of a *rand(3)* library call.

This class is not complete. The use of C library functions is described later in this manual. See ⟨undefined⟩ [C library functions], page ⟨undefined⟩.

3.48 `Magnitude` Class

`Magnitude` is the superclass of all object classes that contain quantities.

Instance Methods

`! (void)`
: When used as a prefix operator, overloads C's '`!`' operator in expressions that contain objects.

`* (void)`
: When used as a prefix operator, overloads C's '`*`' dereference operator and returns the first element of an `int`, `long int`, `long long int`, or `double` array as an `Integer`, `LongInteger`, or `Float` object.

`- (void)`
: Overloads the unary '`-`' prefix operator. Returns the negated value of an `Integer`, `LongInteger`, or `Float`, receivers.

`asCharacter (void)`
: Return an object with the receiver's value as a `Character` object.

`asFloat (void)`
: Return the value of an `Integer` or `LongInteger` receiver as a `Float` object. If the receiver is a `Float` object, returns the receiver. For all other classes, prints a warning and returns `0.0f`.

`asInteger (void)`
: Return an object with the receiver's value as an `Integer` object.

`asLongInteger (void)`
: Return an object with the receiver's value as a `LongInteger` object.

3.49 `Character` Class

The value of `Character` class objects is (on most hardware platforms) an 8-bit integer corresponding to the ISO-8859-1 character set.

Character Constants

Ctalk recognizes the following escape sequences as character constants.

```
\a      Alert
\b      Bell
\e      Escape
```

```
\f      Form Feed
\n      New line
\r      Carraige return
\t      Horizontal tab
\v      Vertical tab
\0      NUL
\"      Literal double quote.
\'      Literal single quote.
```

The '`\e`' escape sequence is an extension to the C language standard.

Ctalk itself doesn't interpret the character constants, though - it simply recognizes that the character or escaped character can be part of a literal string or character. However, if a method encounters a constant, it should interpret the character constant based on the method's function.

Very often a character constant's value is enclosed in single quotes if they are part of the token's name. Again it's up to the method to interpret the character's value. The Ctalk library includes the macros `TRIM_CHAR()` and `TRIM_CHAR_BUF()` that can remove the quotes from the character sequences if necessary.

Note: Ctalk does not support negative `Characters` (i.e., `ch < 0`) universally, even though C `char` types are "signed" by default. This is due to the uneven support in the Ctalk and C libraries for negative characters. So if you need negative values, it's generally safer to use `Integers`.

Additionally, if an expression doesn't need a separate buffer for function arguments, it can also use the `CHAR_CONSTANT_VALUE` macro, which returns a string with the first character pointing to the Character constant's actual value. The argument to the macro is a C `char *`. Here is an example of how to store the value of a Character object in a C `char`.

```
char c;
OBJECT *self_value;

self_value = self value;

sscanf (CHAR_CONSTANT_VALUE(self_value -> __o_value), "%c", &c);
```

Instance Variables

`value` The value of an 8-bit character in the ISO-8859-1 character set.

Instance Methods

`! (void)` Return a character value of true if the receiver evaluates to zero, false otherwise.

`!= (char` *`character`*`)`
: Returns true if the receiver is not equal to *character*.

`& (char` *`character`*`)`
: Returns a bitwise AND of the receiver and the argument.

`&& (char character)`
: Returns `TRUE` if both operands are `TRUE, FALSE` otherwise.

`* (char c)`
: Multiply the receiver and the operand. The result is a `Character` object.

`*= (char c)`
: Multiply the receiver by the operand, and return the receiver.

`+ (char c)`
: Add the receiver and the operand. The result is a `Character` object.

`++ (void)` The prefix and postfix increment operators for `Character` objects.

`+= (char c)`
: Add the operand to the receiver and the operand and return the receiver.

`- (char c)`
: Subtract the receiver and the operand. The result is a `Character` object.

`-- (void)` The prefix and postfix decrement operators for `Character` objects.

`-= (char c)`
: Subtract the operand from the receiver and return the receiver.

`/ (char c)`
: Divide the receiver and the operand. The result is a `Character` object.

`/= (char c)`
: Divide the receiver by the operand and return the receiver.

`< (char character)`
: Returns `TRUE` if the receiver is less than the operand, `FALSE` otherwise.

`<< (int i)`
: Shift the receiver left by the number of bits in the operand, which must be an `Integer`.

`<= (char character)`
: Returns `TRUE` if the receiver is less than or equal to the operand, FALSE otherwise.

`= (char character)`
: Set the value of the receiver object to *character*.

`> (char character)`
: Returns `TRUE` if the receiver is greater than the operand, `FALSE` otherwise.

`>> (int i)`
: Shift the receiver right by the number of bits in the operand, which must be an `Integer`.

`>= (char character)`
: Returns `TRUE` if the receiver is greater than or equal to the operand, FALSE otherwise.

`== (char character)`
: Returns true if the receiver is equal to *character*.

`^ (char *character*)`
: Returns a bitwise XOR of the receiver and the argument.

  Returns a `Character` object that is the bitwise complement of the receiver. This method simply calls `bitComp`, below.

`bitComp (void)`
: Perform a bitwise complement of the receiver.

`invert (void)`
: Returns `TRUE` if the receiver evaluates to `FALSE, FALSE` if the receiver evaluates to `TRUE.`

`isASCII (void)`
: Returns `TRUE` if the receiver is a 7-bit ASCII character '`0-127`', FALSE otherwise.

`isAlNum (void)`
: Returns `TRUE` if the receiver is an alphanumeric character '`0-9`', '`A-Z`', '`a-z`', `FALSE` otherwise.

`isAlpha (void)`
: Returns `TRUE` if the receiver is an alphabetic character '`A-Z`', '`a-z`', `FALSE` otherwise.

`isBlank (void)`
: Returns `TRUE` if the receiver is a space ' ' or horizontal tab '`\t`' character, `FALSE` otherwise.

`isCntrl (void)`
: Returns `TRUE` if the receiver is a control character, `FALSE` otherwise.

`isDigit (void)`
: Returns `TRUE` if the receiver is a character '`0-9`', `FALSE` otherwise.

`isGraph(void)`
: Returns `TRUE` if the receiver is any character except a space, `FALSE` otherwise.

`isLower (void)`
: Returns `TRUE` if the receiver is a lower case character, `FALSE` otherwise.

`isPrint (void)`
: Returns `TRUE` if the receiver is a printable character, `FALSE` otherwise.

`isPunct (void)`
: Returns `TRUE` if the receiver is a printable non-alphanumeric character, `FALSE` otherwise.

`isSpace (void)`
: Returns `TRUE` if the receiver is a space, horizontal tab (`\t`), newline (`\n`), vertical tab (`\v`), form feed (`\f`), or carriage return (`\r`) character, `FALSE` otherwise.

`isUpper (void)`
: Returns `TRUE` if the receiver is an upper case letter, `FALSE` otherwise.

`isXDigit (void)`
: Returns `TRUE` if the receiver is a character '`0-9`', '`a-f`', or '`A-F`', `FALSE` otherwise.

`toLower (void)`
: If the receiver is an upper case letter, returns the lower case version.

`toUpper (void)`
: If the receiver is a lower case letter, returns the upper case version.

`| (char` *`character`*`)`
: Returns a bitwise OR of the receiver and the argument.

`|| (char` *`character`*`)`
: Returns `TRUE` if either operand is `TRUE, FALSE` otherwise.

3.50 `String` Class

Objects of `String` class contain arrays of ASCII characters. The value of a `String` object is similar to the C language organization of strings as a NUL terminated array of `char` values.

In most cases, `String` objects can be used like a collection of `Character` objects. The overloaded operators `++`, `--`, `+`, and `-`, all work similarly to the operators in `List` or `AssociativeArray` objects.

Some of `String` classes' methods add semantics to operators, like the `+=` method, which behaves differently depending on whether its argument is another `String` object, or an `Integer` object.

```
myString = "Hello, ";    /* The resulting value is, */
myString += "world!";    /* "Hello, world!"         */

myString = "Hello, ";    /* The resulting value is, */
myString += 3;           /* "lo, "                  */
```

The main exception to this is the `map` method, which doesn't allow incrementing `self` within an argument block. This is because `String` objects don't use `Key` objects internally to order a `String` object's individual `Character` objects. If it's necessary to treat a `String` object as a collection, the `asList` method will organize the receiver `String` into a `List` of `Character` objects.

Conversely, `Array` and `List` classes contain the `asString` method, which translates an `Array` or `List` into a String object.

In addition, methods like `matchRegex`, `=~`, and `!~` can accept as arguments strings that contain regular expression metacharacters and use them to perform regular expression matches on the receiver `String`. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

Instance Variables

`value` The value is a pointer to the character string.

Instance Methods

`* (void)` When used as a prefix operator, overloads C's '`*`' dereference operator and returns the first element of the receiver, a `Character` object.

`= (char *s)`
: Set the value of the receiver object to *s*.

`== (char *s)`
: Return `TRUE` if *s* and the receiver are identical, `FALSE` otherwise.

`=~ (char *pattern)`
: Returns a `Boolean` value of `true` if the receiver contains the regular expression *pattern*, false otherwise. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`!~ (char *pattern)`
: Returns a `Boolean` value of `false` if the receiver does not contain the argument, *pattern*, which may contain regular expression metacharacters. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`!= (char *s)`
: Return `FALSE` if *s* and the receiver are not identical, `TRUE` otherwise.

`!= (char *s)`
: Return `FALSE` if *s* and the receiver are not identical, `TRUE` otherwise.

`+ (String s)`
`+ (Integer i)`
: If the argument is a `String`, concatenate the receiver and *s* and return the new `String`. If the argument is an `Integer`, return a reference to the receiver plus *i*.

`++ (void)` Increment the value of the receiver as a `char *`. This method uses *__ctalkIncStringRef ()* to handle the pointer math.

In other words, this method effectively sets the receiver `String's` value from, for example, '`Hello, world!`' to '`ello, world!`'. If the receiver is incremented to the end of its contents, then its value is `NULL`.

`+= (String s)`
`+= (Integer i)`
: If the argument is an `Integer`, increment the reference to the receiver by that amount. If the argument is a `String` or any other class, concatenate the argument to the receiver and return the receiver, formatting it as a string first if necessary.

`- (Integer i)`
: Return a reference to the receiver `String` minus *i*. If the reference is before the start of the string, return NULL. That means the method is only effective after a call to `++` or a similar method.

```
String new str;

str = "Hello, world!";

str += 1;
```

```
printf ("%s\n", str);    /* Prints, "ello, world!" */

--str;

printf ("%s\n", str);    /* Prints, "Hello, world!" */
```

`-- (void)` Decrement the value of the receiver as a `char *`. The effect is the converse of `++`, above. The method doesn't decrement the reference so that it points before the beginning of the `String` object's contents. That means, like `-` above, the method only returns a pointer to somewhere in the receiver's value after a previous call to `++` or a similar method. For example,

```
String new str;

str = "Hello, world!";

++str;

printf ("%s\n", str);    /* Prints, "ello, world!" */

--str;

printf ("%s\n", str);    /* Prints, "Hello, world!" */
```

`-= (Integer i)`
: If the argument is an `Integer`, decrement the reference to the receiver's value by the amount given as the argument, an `Integer`. Like the other methods that decrement the reference to the receiver's value, the program must first have incremented it further than the start of the string.

`asExpanded (void)`
: Return the expanded directory path for a directory glob pattern contained in the receiver.

`asInteger (void)`
: Return an `Integer` object with the value of the receiver.

`asList (List` *`newList`*`)`
: Store each character of the receiver `String` as `Character` object members of *newList*.

`at (int` *`index`*`)`
: Return the character at *index*. The first character of the string is at index 0. If *index* is greater than the length of the string, return '`NULL`'.

`atPut (int n, char c)`

Replace the *n*'th character of the receiver with *c*. Has no effect and returns `NULL` if n is greater than the length of the receiver.

The `atPut` method interprets the following character sequences (with their ASCII values)

```
Sequence   ASCII Value
\0         0
\a         7
\b         7
\n         10
\e         27
\f         10
\r         13
\t         9
\v         11
```

The '`\e`' escape sequence is an extension to the C language standard.

The method returns the receiver (with the new value) if successful.

You should note that the method does not do any conversion of the argument; that is, if *c* isn't a `Character` object, then the results are probably not going to be what you want. For example, if you try to store an `Integer` in a `String`, like this:

```
myInt = 1;

myString atPut 0, myInt + '0';
```

The results aren't going to be what you want; adding ASCII '`'0'`' *doesn't* convert `myInt` to a `Character` object. You still need to use the `asCharacter` method from `Magnitude` class to create a `Character` object, as in this example.

```
myInt = 1;

myString atPut 0, (myInt + '0') asCharacter;
```

The parentheses in the second argument are necessary; otherwise, `asCharacter` would use '`'0'`' as its receiver because `asCharacter`, which is a method message, has a higher precedence than '`+`'. Instead, `asCharacter's` receiver should be the value of '`myInt + '0'`', so we enclose the first part expression in parentheses so it gets evaluated first.

`callStackTrace (void)`

Print a call stack trace.

`charPos (char c)`

Return an `Integer` with the position of *c* in the receiver. Returns an `Integer` between 0 (the first character) and the receiver's length, minus one (the last character). If the receiver does not contain *c*, returns -1.

`charPosR (char c)`

Return an `Integer` with the position of the last occurence of *c* in the receiver. Returns an `Integer` between 0 (the first character) and the receiver's length, minus one (the last character). If the receiver does not contain *c*, returns -1.

`chomp (void)`

Removes a trailing newline character ('`\n`') if the receiver contains one. Named after Perl's very useful string trimming function.

`consoleReadLine (String promptStr)`

Print the *promptStr* on the terminal and wait for the user to enter a line of text. If Ctalk is built with the GNU readline libraries, adds readline's standard line editing and command history facilities. In that case, Ctalk also defines the `HAVE_GNU_READLINE` preprocessor definition to '`1`'. You can build Ctalk with or without readline; see the options to `./configure` for further information.

Here is a sample program that shows how to use `consoleReadLine`.

```
int main (int argc, char **argv)    String new s;
  String new promptStr;

  if (argc > 1)
    promptStr = argv[1];
  else
    promptStr = "Prompt ";

  printf ("Readline test.  Type ^C or, \"quit,\" to exit.\n");
#if HAVE_GNU_READLINE
  printf ("Ctalk built with GNU Readline Support.\n");
#else
  printf ("Ctalk built without GNU Readline Support.\n");
#endif
  while (1)      s consoleReadLine promptStr;
    printf ("You typed (or recalled), \"%s.\"\n", s);
    /*
     *  Matches both, "quit," and, "quit\n."
     */
    if (s match "quit")
      break;
  }
}
```

`contains (String pattern)`
`contains (String pattern, Integer starting_offset)`

Returns a `Boolean` value of True if the receiver string contains an exact match of the text in *pattern*, False otherwise.

With a second argument *n*, an `Integer`, the method begins its search from the *n*'th character in the receiver string.

`envVarExists (char *envVarName)`
: Test for the presence of an environment variable. Return `TRUE` if the variable exists, `FALSE` otherwise.

`getEnv (char *envVarName)`
: Return the value of environment variable *envVarName* as the value of the receiver, or `(null)`. Note that this method generates an internal exception of the environment variable does not exist. To test for the presence of an environment variable without generation an exception, see `envVarExists`, above.

`getRS (void)`
: Returns a `Character` with the current record separator.

  The record separator determines whether the regular expression metacharacters '`^`' and '`$`' recognize line endings. The default value of the record separator is a newline '`\n`' character, which means that a '`^`' character will match an expression at the start of a string, or starting at the beginning of a text line. Likewise, a '`$`' metacharacter matches both the end of a line and the end of the string.

  To match only at the beginning and end of the string, set the record separator to a NUL character ('`\0`'). See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`isXLFD (void)`
: Returns a Boolean value of True if the receiver is a XLFD font descriptor, False otherwise. For more information about font selection, refer to the `X11Font` class See ⟨undefined⟩ [X11Font], page ⟨undefined⟩, and the `X11FreeTypeFont` class See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`length (void)`
: Return an object of class `Integer` with the length of the receiver in characters.

`map (OBJECT *(*method)())`
: Execute *method*, an instance method of class `String`, for each character of the receiver object. For example,

```
String instanceMethod printSpaceChar (void) {
  printf (" %c", self);  /* Here, for each call to the printSpaceChar
                            method, "self" is each of myString's
                            successive characters. */
}

int main () {

  String new myString;

  myString = "Hello, world!";

  myString map printSpaceChar;
```

```
  printf ("\n");
}
```

The argument to `map` can also be a code block:

```
int main () {

  String new myString;

  myString = "Hello, world!";

  myString map {
    printf (" %c", self);
  }

  printf ("\n");
}
```

`match (char *pattern)`
: Returns `TRUE` if *pattern* matches the receiver `String` regardless of case, false otherwise. Both `match` and `matchCase`, below, are being superceded by `matchRegex` and `quickSearch`, also below.

`matchAt (Integer idx)`
: Returns the text of the *idx'th* parenthesized match resulting from a previous call to `matchRegex`, `=~`, or `!~`. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`matchCase (char *pattern)`
: Returns `TRUE` if *pattern* matches the receiver case- sensitively, false otherwise. Like `match`, above, `matchCase` is being superceded by `matchRegex` and `quickSearch`, below.

`matchIndexAt (Integer idx)`
: Returns the character position in the receiver `String` of the *idx'th* parenthesized match resulting from a previous call to `matchRegex`, `=~`, or `!~`. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`matchLength (void)`
: Returns the length of a regular expression match from the previous call to the `matchRegex` method, below.

`matchRegex (String pattern, Array offsets)`
: Searches the receiver, a `String` object, for all occurrences of *pattern*. The `matchRegex` method places the positions of the matches in the *offsets* array, and returns an `Integer` that contains the number of matches. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

The `quickSearch` method, below, matches exact text only, but it uses a much faster search algorithm.

`nMatches (void)`
: Returns an `Integer` with the number matches from the last call to the `matchRegex` method.

`printMatchToks (Integer yesNo)`
: If the argument is non-zero, print the tokens of regular expression patterns and the matching text after each regular expression match. This can be useful when debugging regular expressions. See ⟨undefined⟩ [DebugPattern], page ⟨undefined⟩.

`printOn (char *fmt, ...)`
: Format and print the method's arguments to the receiver.

`quickSearch (String pattern, Array offsets)`
: Searches the receiver, a `String` object, for all occurrences of *pattern*. The `quickSearch` method places the positions of the matches in the *offsets* array, and returns an `Integer` that contains the number of matches.

  Unlike `matchRegex`, above, `quickSearch` matches exact text only, but it uses a much faster search algorithm.

`readFormat (char *fmt, ...)`
: Scan the receiver into the arguments, using *fmt*.

`search (String pattern, Array offsets)`
: This method is a synonym for `matchRegex`, above, and is here for backward compatibility.

`setRS (char record_separator_char)`
: Sets the current application's record separator character, which determines how regular expression metacharacters match line endings, among other uses. See ⟨undefined⟩ [RecordSeparator], page ⟨undefined⟩. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`split (char delimiter, char ** resultArray)`
: Split the receiver at each occurrence of *delimiter*, and save the result in *resultArray*. The *delimiter* argument can be either a `Character` object or a *String* object. If *delimiter* is a `String`, it uses Ctalk's pattern matching library to match the delimiter string. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

  However, the pattern matching library only records the length of the last match, so if you use a pattern like '`" *"`' then the results may be inaccurate if all of the delimiters are not the same length.

`subString (int index, int length)`
: Return the substring of the receiver of *length* characters beginning at *index*. String indexes start at 0. If *index* + *length* is greater than the length of the receiver, return the substring from *index* to the end of the receiver.

`sysErrnoStr (void)`
: Sets the receiver's value to the text message of the last system error (the value of *errno(3)*).

`tokenize (List tokens)`
: Splits the receiver `String` at each whitespace character or characters (spaces, horizontal and vertical tabs, or newlines) and pushes each non-whitespace set of characters (words, numbers, and miscellaneous punctuation) onto the `List` given as the argument. The method uses *ispunct(3)* to separate punctuation, except for '_' characters, which are used in labels.

  Note that this method can generate lists with hundreds or even thousands of tokens, so you need to take care with large (or even medium sized) input `Strings` as receivers.

`tokenizeLine (List tokens)`
: Similar to tokenize, above. This method also treats newline characters as tokens, which makes it easier to parse input that relies on newlines (for example, C++ style comments, preprocessor directives, and some types of text files).

`vPrintOn (String calling_methods_fmt_arg)`
: This function formats the variable arguments of its calling method on the receiver `String` object.

  The argument is the format argument of the calling method. When `vPrintOn` is called, it uses the argument as the start of the caller's variable argument list.

  Here is an example of `vPrintOn's` use.

```
Object instanceMethod myPrint (String fmt, ...) {
  String new s;
  s vPrintOn fmt;
  return s;
}

int main () {
  Object new obj;
  Integer new i;
  String new str;

  i = 5;

  str = obj myPrint "Hello, world no. %d", i;

  printf ("%s\n", str);
}
```

`writeFormat (char *fmt,...)`
: Write the formatted arguments using *fmt* to the receiver. Note that Ctalk stores scalar types as formatted strings. See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

3.50.1 String Searching and Pattern Matching

`String` class defines a number of methods for searching and matching `String` objects. The `matchRegex` method recognizes some basic metacharacters to provide regular expression search capabilities. The `quickSearch` method searches `String` objects for exact text patterns, but it uses a much faster search algorithm.

The operators, `=~` and `!~` return true or false depending on whether the receiver contains the pattern given as the argument. If the argument contains metacharacters, then Ctalk conducts a regular expression search; otherwise, it tries to match (or not match, in the case of `!~`) the receiver and the pattern exactly.

If you want more thorough information about the search, the `matchRegex` and `quickSearch` methods allow an additional argument after the text pattern: an `Array` object that the methods use to return the character positions of the matches within the receiver. After the method is finished searching, the second argument contains the position of the first character wherever the text pattern matched text in the receiver. The last offset is '`-1`', indicating that there are no further matches. The methods also return an `Integer` object that contains the number of matches.

Here is an example from `LibrarySearch` class that contains the additional '`offsets`' argument.

```
if ((inputLine match KEYPAT) &&
       (inputLine matchRegex (pattern, offsets) != 0)) {

...

}
```

Searches can provide even more information than this, however. Pattern strings may contain *backreferences*, which save the text and position of any of the receiver string's matched text that the program needs. The sections just below describe backreferences in detail.

All of these methods (except `quickSearch`) recognize a few regular expression metacharacters. They are:

'`.`' Matches any single character.

'`^`' Matches text at the beginning of the receiver `String's` text.

'`$`' Matches text at the end of the receiver `String's` text, or the end of a line (that is, the character before a '`\n`' or '`\r`' newline character).

'`*`' Matches zero or more occurrences of the character or expression it follows.

'`+`' Matches one or more occurences of the character or expression it follows.

'`?`' Matches zero or one occurrence of the character or expression it follows.

'`\`' Escapes the next character so it is interpreted literally; e.g., the sequence '`\*`' is interpreted as a literal asterisk. Because Ctalk's lexical analysis also performs the same task, so if you want a backslash to appear in a pattern, you need to type, '`\\`', for example,

```
myPat = "\\*";   /* The '\\' tells Ctalk's lexer that we really
                    want a '\' to appear in the pattern string,
                    so it will still be there when we use myPat
                    as a regular expression. */
```

However, Ctalk also recognizes patterns, which only need to be evaluated by the regular expression parser. Patterns do not get checked immediately for things like for balanced quotes and ASCII escape sequences; instead, they get evaluated by the regular expression parser when the program actually tries to perform some pattern matching. Otherwise, patterns are identical to `Strings`. Expressed as a pattern, `myPat` in the example above would look like this.

```
myPat = /\*/;
```

Pattern strings are described in their own section, below. See ⟨undefined⟩ [Pattern Strings], page ⟨undefined⟩.

‘`(`’

‘`)`’ Begin and end a match reference (i.e., a *backreference*). Matched text between ‘`(`’ and ‘`)`’ is saved, along with its position in the receiver `String`, and can be retrieved with subsequent calls to the `matchAt` and `matchIndexAt` methods. The match information is saved until the program performs another pattern match.

‘`\W`’

‘`\d`’

‘`\p`’

‘`\w`’

‘`\l`’ In patterns, these escape sequences match characters of different types. The escape sequences have the following meanings.

```
Character Class      Matches
---------------      ------
\W                   'Word' Characters (A-Z, a-z)
\d                   Decimal Digits (0-9)
\w                   White Space (space, \t, \n, \f, \v)
\p                   Punctuation (Any other character.)
\l                   'Label' Characters (A-Z, a-z, 0-9, and _)
\x                   Hexadecimal Digits (0-9, a-f, A-F, x, and X)
```

The following program contains a pattern that looks for alphabetic characters, punctuation, and whitespace.

```
int main (int argc, char **argv) {
  String new str;
```

```
  str = "Hello, world!";

  if (str =~ /e(\W*\p\w*\W)/) {
    printf ("match - %s\n", str matchAt 0);
  }
}
```

When run, the expression,

```
str =~ /e(\W*\p\w*\W)/
```

Produces the following output.

```
match - llo, w
```

'|' Matches either of the expressions on each side of the '|'. The expressions may be either a character expression, or a set of characters enclosed in parentheses. Here are some examples of alternate patterns.

```
a|b
a*|b*
a+|b+
\W+|\d+
(ab)|(cd)
```

When matching alternate expressions, using '*' in the expressions can produce unexpected results because a '*' can provide a zero-length match, and the '|' metacharacter is most useful when there is some text to be matched.

If one or both expressions are enclosed in parentheses, then the expression that matches is treated as a backreference, and the program can retrieve the match information with the `matchAt` and `matchIndexAt` methods.

The following example shows how to use some of the matching featues in an actual program. This program saves the first non-label character (either a space or parenthesis) of a function declaration, and its position, so we can retrieve the function name and display it separately.

```
int main (argc, argv) {
  String new text, pattern, fn_name;
  List new fn_list;

  fn_list = "strlen ()", "strcat(char *)", "strncpy (char *)",
    "stat (char *, struct stat *)";

  /* Match the first non-label character: either a space or a
     parenthesis.  The double backslashes cause the content of
     'pattern' (after the normal lexical analysis for the string) to
```

```
      be,

        "( *)\("

      So the regular expression parser can check for a backslashed
      opening parenthesis (i.e., a literal '(', not another
      backreference delimiter).
  */

  pattern = "( *)\\(";

  fn_list map {
    if (self =~ pattern) {
      printf ("Matched text: \"%s\" at index: %d\n",
      self matchAt 0, self matchIndexAt 0);
      fn_name = self subString 0, self matchIndexAt 0;
      printf ("Function name: %s\n", fn_name);
    }
  }

  return 0;
}
```

When run, the program should produce results like this.

```
Matched text: " " at index: 6
Function name: strlen
Matched text: "" at index: 6
Function name: strcat
Matched text: " " at index: 7
Function name: strncpy
Matched text: " " at index: 4
Function name: stat
```

Note that the first backreference is numbered '`0`', in the expression '`self matchAt 0`'. If there were another set of (unescaped) parentheses in `pattern`, then its text would be refered to as '`self matchAt 1`'.

You should also note that the second function match saved an empty string. That's because the text that the backreferenced pattern referred to resulted in a zero-length match. That's because '`*`' metacharacters can refer to *zero* or more occurrences of the character that precedes it.

The program could also use the `charPos` method to look for the ' ' and/or '`(`' characters, but using a regular expression gives us information about which non-label character appears first more efficiently.

Here's another example. The pattern contains only one set of parentheses, but Ctalk saves a match reference every time the pattern matches characters in the target string.

```
int main () {
  String new string, pattern;
  Array new offsets;
  Integer new nMatches, i;

  pattern = "(l*o)";
  string = "Hello, world! Hello, world, Hello, world!";

  nMatches = string matchRegex pattern, offsets;

  printf ("nMatches: %d\n", nMatches);
  offsets map {
    printf ("%d\n", self);
  }
  for (i = 0; i < nMatches; ++i) {
    printf ("%s\n", string matchAt i);
  }
}
```

When run, the program produces output like this.

```
nMatches: 6
2
8
16
22
30
36
-1
llo
o
llo
o
llo
o
```

The character classes match anywhere they find text in a target string, including control characters like '`\n`' and '`\f`', regardless of the record separator character. For a brief example, refer to the section, *The Record Separator Character*, below.

This example matches one of two patterns joined by a '`|`' metacharacter.

```
int main () {
  String new s, pat;
  Array new matches;
  Integer new n_matches, n_th_match;
```

```
  pat = "-(mo)|(ho)use";

  s = "-mouse-house-";

  n_matches = s matchRegex pat, matches;

  for (n_th_match = 0; n_th_match < n_matches; ++n_th_match) {
    printf ("Match %d. Matched %s at character index %ld.\n",
    n_th_match, s matchAt n_th_match, s matchIndexAt n_th_match);
  }

  matches delete;

}
```

When run, the program should produce output like this.

```
Match 0. Matched mo at character index 0.
Match 1. Matched ho at character index 6.
```

You should note that if a pattern in a backreference results in a zero length match, then that backreference contains a zero length string. While not incorrect, it can produce confusing results when examining matched text. The following program shows one way to indicate a zero-length backreference. It prints the string '`(null)`' whenever a backreference contains a zero-length string.

```
int main () {
  String new s;
  String new pat;
  Integer new n_matches;
  Array new offsets;
  Integer new i;

  s = "1.mobile 2mobile mobile";
  pat = "(\\d\\p)?m";

  n_matches = s matchRegex pat, offsets;

  for (i = 0; i < n_matches; ++i) {
    printf ("%Ld\n", offsets at i);
  }

  for (i = 0; i < n_matches; ++i) {
    if ((s matchAt i) length == 0) {
      printf ("%d: %s\n", s matchIndexAt i, "(null)");
```

```
    } else {
      printf ("%d: %s\n", s matchIndexAt i, s matchAt i);
    }
  }
}
```

When run, the program should produce output that looks like this.

```
0
10
17
0: 1.
17: (null)
22: (null)
```

3.50.1.1 Pattern Strings

When writing a regular expression, it's necessary to take into account all of the processing that String objects encounter when they are evaluated, before they reach the Ctalk library's regular expression parser. To help facilitate lexical analysis and parsing, Ctalk also provides *pattern strings*, which allow Ctalk to defer the evaluation of a pattern until the regular expression parser actually performs the text matching.

Ctalk also provides operators that provide shorthand methods to match patterns with text, the `=~` and `!~` operators.

Pattern constants at this time may only follow the `=~` and `!~` operators, but you can use the `matchAt` and `matchIndexAt`, and `nMatches` methods to retrieve the match information. You must, as with `Strings` that are used as patterns, enclose the pattern in '`(`' and '`)`' metacharacters in order to create a backreference.

Here is a simple string matching program that matches text against a pattern constant.

```
int main () {

  String new s;
  Integer new n_offsets;
  Integer new i;

  s = "Hello?";

  if (s =~ /(o\?)/) {
    printf ("match\n");
    i = 0;
    n_offsets = s nMatches;
    while (i < n_offsets) {
      printf ("%d: %s\n", s matchIndexAt i, s matchAt i);
      ++i;
```

```
    }
  }
}
```

The most obvious example of how a pattern provides an advantage for text matching is when writing backslash escapes. To make a backslash appear in a pattern string, you need to write at least two backslashes in order for a backslash to appear when it's needed to escape the following character. If you want to match an escaped backslash, then you need to write at least *four* backslashes.

```
String         Pattern
"\\*"          /\*/        # Matches a literal '*'.
"\\\\*"        /\\*/       # Matches the expression '\*'.
```

To create a pattern, you delimit the characters of the pattern with slashes (`//`) instead of double quotes. Other delimiters can signify patterns also if the pattern starts with a `m` character, followed by the delimiter character, which must be non-alphanumeric.

```
String         Pattern     Alternate Pattern
"\\*"          /\*/        m|\*|
"\\\\*"        /\\*/       m|\\*|
```

There is no single rule that governs how often `String` objects are evaluated when a program runs. So writing patterns helps take some of the work out of testing an application's pattern matching routines.

3.50.1.2 Debugging Pattern Matches

Ctalk allows you to view the parsed pattern tokens, and the text that each token matches. Token printing is enabled using the `printMatchToks` method, like this.

```
myString printMatchToks TRUE;
```

When token printing is enabled, then Ctalk's pattern matching routines print the tokens of the pattern and the text that each token matches after every pattern match attempt.

If we have a program like the following:

```
int main () {

  String new s;

  s printMatchToks TRUE;

  s = "192.168.0.1";

  if (s =~ /\d+\.(\d+)\.\d+\.\d+/) {
```

```
      printf ("match!\n");
    }

}
```

Then, when this program is run with token printing enabled, the output should look similar to this.

```
joeuser@myhost:~$ ./mypatprogram
PATTERN: /\d+\.(\d+)\.\d+\.\d+/         TEXT: "192.168.0.1"
TOK: d+          (character class)               MATCH: "192"
TOK: .           (literal character)             MATCH: "."
TOK: (           (backreference start)           MATCH: ""
TOK: d+          (character class)               MATCH: "168"
TOK: )           (backreference end)             MATCH: ""
TOK: .           (literal character)             MATCH: "."
TOK: d+          (character class)               MATCH: "0"
TOK: .           (literal character)             MATCH: "."
TOK: d+          (character class)               MATCH: "1"
match!
joeuser@myhost:~$
```

The processed token text is followed by any attributes that the regular expression parser finds (for example, then a pattern like '`\d+`' becomes the token '`d+`' with the attribute of a character class identifier, or the '`(`' and '`)`' characters' backreference attributes). Then, finally, the library prints the text that matches each token.

Successful matches have text matched by each token in the pattern (except for zero-length metacharacters like '`(`', '`)`', '`^`', or '`$`').

Unsuccessful matches, however, may display text that matches where you don't expect it. That's because the regular expression parser scans along the entire length of the text, trying to match the first pattern token, then the second pattern token, and so on.

Although this doesn't always pinpoint the exact place that a match first failed, it can provide a roadmap to help build a complex pattern from simpler, perhaps single-metachar patterns, which shows what the regular expression parser is doing internally.

3.50.1.3 The Record Separator Character

Ctalk uses a record separator character to determine how the metacharacters '`^`' and '`$`' match line endings, among other uses.

The default record separator character is a newline ('`\n`'). In this case a '`^`' metacharacter in an expression matches the beginning of a string as well as the character(s) immediately following a newline. Similarly, a '`$`' metacharacter anchors a match to the characters at the end of a line and at the end of a string.

Setting the record separator character to NUL ('`\0`') causes '`^`' and '`$`' to match only the beginning and the end of a string.

Here is an example that prints the string indexes of matches with the default newline record separator and with a NUL record separator character.

When the record separator is '`'\n'`', the '`$`' metacharacter in our pattern matches the text immediately before a '`\n`' character, as well as the text at the end of the string.

```
int main () {

  String new s;
  Integer new n_indexes;
  Array new match_indexes;
  String new pattern;

  printf ("\tMatch Indexes\n");

  /* Begin with the default record separator ('\n'). */

  s = "Hello, world!\nHello, wo\nHello, wo";
  pattern = "wo$";
  n_indexes = s matchRegex pattern, match_indexes;

  printf ("With newline record separator:\n");
  match_indexes map {
    printf ("%d\n", self);
  }

  s setRS '\0';   /* Set the record separator to NUL ('\0'). */

  match_indexes delete; /* Remember to start with an empty Array again. */

  n_indexes = s matchRegex pattern, match_indexes;

  printf ("With NUL record separator:\n");
  match_indexes map {
    printf ("%d\n", self);
  }
}
```

When run, the program should produce output like this.

```
        Match Indexes
With newline record separator:
21
31
-1
With NUL record separator:
31
-1
```

Likewise, a '^' metacharacter matches text immediately after the '\n' record separator, or at the beginning of a string.

It's also possible, though, to match newlines (and other ASCII escape characters) in patterns, either with a character class match, or by adding the escape sequence to the pattern. To do that, the program should use a double backslash with the ASCII escape sequence, as with the newline escape sequence in this example.

```
int main () {
  String new s;

  s = "Hello,\nworld!";

  if (s =~ /(\W\p\\n)/)
    printf ("%s\n", s matchAt 0);

}
```

3.51 `Float` Class

Objects of `Float` class represent double precision, floating point numbers.

Instance Variables

`value` The value is the formatted representation of a double precision floating point number.

Instance Methods

`&& (double` *d*`)`
: Return an `Integer` that evaluates to `TRUE` if both operands are `TRUE, FALSE` otherwise.

`= (double` *d*`)`
: Set the value of the receiver object to *d*.

`+ (double` *d*`)`
: Add *d* to the receiver.

`+= (double` *d*`)`
: Add *d* to the receiver's value. Set the receiver to the new value, and return the receiver.

`- (void)`

`- (double` *d*`)`
: Subtract *d* from the receiver. When used as a prefix operator, negate the receiver.

`-= (double` *d*`)`
: Subtract *d* from the receiver's value. Set the receiver to the new value, and return the receiver.

`* (double` *d*`)`
: Multiply the receiver by *d*.

`*= (double d)`
: Multiply *d* by the receiver's value. Set the receiver to the new value, and return the receiver.

`* (double d)`
: Divide the receiver by *d*.

`/= (double d)`
: Divide *d* by the receiver's value. Set the receiver to the new value, and return the receiver.

`< (double d)`
: Return an `Integer` that evaluates to `TRUE` if the receiver is less than the argument, `FALSE` otherwise.

`<= (double d)`
: Return an `Integer` that evaluates to `TRUE` if the receiver is less than or equal to the argument, `FALSE` otherwise.

`> (double d)`
: Return an `Integer` that evaluates to `TRUE` if the receiver is greater than the argument, `FALSE` otherwise.

`>= (double d)`
: Return an `Integer` that evaluates to `TRUE` if the receiver is greater than or equal to the argument, `FALSE` otherwise.

`asInteger (void)`
: Return the integer portion of the receiver.

`|| (double d)`
: Return an `Integer` that evaluates to `TRUE` if either operand is `TRUE, FALSE` otherwise.

3.52 `Integer` Class

Objects of `Integer` class represent signed and unsigned integers of the C types `int` and `long int`.

Instance Variables

`value` The value is the formatted representation of the receiver.

Instance Methods

`!= (int i)`
: Return `TRUE` if the receiver and the argument are not equal, `FALSE` otherwise.

`& (int i)` As a binary operator, perform a bitwise and of the receiver and the argument. The `Object` class's `&` method overloads C's unary "address of" prefix operator. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

`% (int i)` Return an `Integer` that is the modulus of the receiver and the argument.

`%= (int i)`
: Perform a modulus of the receiver and its argument, and store the result in the receiver. Returns the receiver.

`&& (int i)`
: Return `TRUE` if the receiver and the argument evaluate to `TRUE`.

`&= (Integer i)`
: Perform a bitwise and of the receiver and the argument, and assign the result to the receiver.

`+ (int i)` Add *i* and the receiver, as in this example.

`++ (void)` Postfix and prefix increment operators for `Integer` objects.

`+= (int arg)`
: Add the value of *arg* to the receiver.

`- (int i)` Subtract *i* from the receiver.

`- (void)` When used as a unary minus prefix operator, negate the expression.

`-- (void)` Postfix and prefix decrement operators for `Integer` objects.

`-= (Integer arg)`
: Subtract the value of *arg* from the receiver.

`* (int i)` Multiply the receiver by *i*.

`*= (int arg)`
: Multiply the receiver by *arg*.

`/ (int i)` Divide the receiver by *i*.

`/= (int arg)`
: Divide the receiver by *arg*.

`< (int i)` Return `TRUE` if the receiver is less than the argument, `FALSE` otherwise.

`<< (int i)`
: Perform an arithmetic left shift on the receiver by the number of bits in the argument.

`<= (int i)`
: Return `TRUE` if the receiver is less than or equal to the argument, `FALSE` otherwise.

`= (int i)` Set the value of the receiver object to *i*. Also checks for `Symbol` pointer contexts and other aliases.

```
intObject = 2;

resultInt = intObject + intObject;
```

`== (int i)`
: Return `TRUE` if the receiver and the argument are equal, `FALSE` otherwise.

`> (int i)` Return `TRUE` if the receiver is greater than the argument, `FALSE` otherwise.

`>= (int i)`
: Return `TRUE` if the receiver is greater than or equal to the argument, `FALSE` otherwise.

`>> (int i)`
: Perform an arithmetic right shift on the receiver by the number of bits in the argument.

`^ (int i)` Return the result of a bitwise xor of the receiver and its argument.

`^= (int i)`
: Preform a bitwise xor of the receiver with its argument, and assign the value to the receiver. Returns the receiver.

`bitComp (int i)`
: Return the bitwise complement of the receiver.

`invert (void)`
: Return `TRUE` if the receiver evaluates to `FALSE`, `FALSE` if the receiver evaluates to `TRUE`.

`| (int i)` Perform a bitwise or of the receiver and the argument.

`|= (Integer i)`
: Perform a bitwise or of the receiver and the argument, and assign the result to the receiver.

`|| (int i)`
: Return `TRUE` if either the receiver or the argument, or both, evaluate to `TRUE`.

`^ (int i)` Perform a bitwise exclusive or of the receiver and the argument.

Returns a `String` formatted as a decimal or hexadecimal integer. The `asString` method is a synonym for `asDecimalString`.

`~ (void)` When used to overload C's '~' operator, is synonymous with the `bitComp` method, above.

3.53 `CTime` Class

Objects of `CTime` class represent the system's UTC clock which measures time in seconds since 1900. This class also implements the methods that convert UTC time into calendar time.

The return value of the methods `gmTime` and `localTime` is an `Array` that contains the following values.

```
returnArray at 0    Seconds (0... 59)
returnArray at 1    Minutes (0... 59)
returnArray at 2    Hours (0... 23)
returnArray at 3    Day of the Month (1... 31)
returnArray at 4    Month (0... 11)
returnArray at 5    Year (Number of years since 1900.)
returnArray at 6    Day of the Week (0 = Sunday ... 6 = Saturday)
returnArray at 7    Day of the Year (1... 365)
returnArray at 8    > 0 = Daylight Savings Time; 0 = Standard Time;
                    < 0 = Not Available
```

Instance Methods

`cTime (void)`
: Returns a formatted `String` with the date and time of the receiver.

  The return `String` of a `cTime` call is formatted as in this example.

```
"Sun Jan 6 13:04:00 2008\n"
```

`gmTime (void)`
: Returns an `Array` with the values described above for the current Greenwich Mean Time.

`haveDST (void)`
: Return an `Integer` that evaluates to True if the system provides daylight savings time information, False otherwise.

`isAM (void)`
: Returns an `Integer` that evaluates to True or False depending on whether the local time is a.m. or p.m.

`localTime (void)`
: Returns an `Array` with the values described above for the current local time.

`timeZoneName (void)`
: Return a `String` with the name of the time zone provided by the system.

`timeZoneOffset (void)`
: Return an `Integer` with the time zone offset in seconds from GMT. Not all systems provide this information.

`utcTime (void)`
: Return an `Integer` object containing the current UTC time.

3.54 `CalendarTime` class

`CalendarTime` class provides instance variables for elements of a broken-down clock and calendar UTC time: seconds, minutes, hours, day of the month, and so on. The methods in this class might be more convenient in many cases than their functional equivalents in `CTime` class.

All of the methods in this class expect that the receiver has made a previous call to `getUTCTime` (which is defined in `CTime` class). The `getUTCTime` method fills in the receiver's value with the UTC seconds since the epoch. See ⟨undefined⟩ [CTime], page ⟨undefined⟩.

The `localTime` and `gmTime` methods contain calls to the system's *localtime(3)* and *gmtime(3)* library functions (or localtime_r and gmtime_r). These manual pages provide more information about how the system translates seconds since 1900 into a local time zone's calendar time, or UTC calendar time.

This brief example prints an ISO-format date and time string.

```
int main () {
  CalendarTime new ct;
```

```
    ct utcTime;
    ct localTime;
    printf ("%s\n", ct isoTimeString);
  }
```

Instance Variables

`seconds`

`minutes`

`hours`

`dom`

`month`

`year`

`dow`

`doy` `Integers` that contain the clock time and calendar date represented by the receiver's UTC time. The values are translated using the local time zone information if necessary.

Instance Variable	Range
seconds	0 - 59
minutes	0 - 59
hours	0 - 23
dom (day of the month)	1 - 31
month	0 - 11 (0 = January)
year	Years since 1900.
dow (day of the week)	0 - 6 (0 = Sunday)
doy (day of the year)	0 - 365
isdst	> 0 : true 0 : false < 0 : not available

`isdst` An `Integer` value that indicates whether Daylight Savings Time is in effect on systems that support it. If the value is positive, Daylight Savings Time is in effect, zero indicates that DST is not in effect, and a value less than zero indicates that the information is not available.

`timeZone` An `Integer` that contains the seconds west of GMT of the local time zone.

`tzStd`

`tzDst`

`gmtOff` `String` objects that contain the abbreviation of the standard local time zone, the local daylight savings time zone, and the hours from GMT, usually expressed

as a four digit number; e.g., '`-0700`' for the MST time zone. The value of `gmtOff` is the result of dividing the value of the `timeZone` instance variable by -3600, them multiplying the result by 100. The `haveDst` instance variable is `true` if the timezone supports daylight savings time, but not all systems support this. To find out whether daylight savings time is in effect, the `isdst` instance variable above, can provide that information if the machine supports it.

Calling the `localTime` and `zoneInfo` methods fill in the time zone information.

Instance Methods

`cTimeString (void)`

Returns a formatted string with the date and time given by the receiver.

The returned String is formatted similarly to the output of the *ctime(3)* C function, except that the string does not include a trailing newline.

Programs should call the `utcTime` method to get the current UTC time, and then either the `localTime` or `gmTime` method to convert the UTC time into calendar day and date information, before calling this method.

`dayName (void)`

Returns a `String` with the three-letter abbreviation of the current day: '`Sun`', '`Mon`', '`Tue`', '`Wed`', '`Thu`', '`Fri`', and '`Sat`'.

`gmTime (void)`

Fills in the receiver's instance variables with the elements of the UTC calendar time.

Programs should call the `utcTime` method to get the current UTC time before calling this method.

`isoTimeString (void)`

Returns a formatted string with the date and time given by the receiver. The returned String has the format of an ISO date and time string.

Programs should call the `utcTime` method to get the current UTC time, and then either the `localTime` or `gmTime` methods to convert the UTC time into calendar day and date information, before calling this method.

`localTime (void)`

Fills in the receiver's instance variables with the elements of the local calendar time, as determined by the system's time zone setting.

Programs should call the `utcTime` method to get the current UTC time before calling this method.

`monName (void)`

Returns a `String` with the three-letter abbreviation of the current time's month: '`Jan`', '`Feb`', '`Mar`', '`Apr`', '`May`', '`Jun`', '`Jul`', '`Aug`', '`Sep`', '`Oct`', '`Nov`', and '`Dec`'.

`zoneInfo (void)`

Fills in the receiver's `timeZone`, `tzStd`, `tzDst`, `gmtOff`, and `haveDst` with information about the machine's current time zone.

3.55 `LongInteger` class

Objects of `LongInteger` class represent signed and unsigned integers of the C type `long long int`.

Instance Variables

`value` The value is the formatted representation of a signed `long long int`.

Instance Methods

`!= (long long int l)`
: Return `TRUE` if the receiver is not equal to *l*.

`= (long long int l)`
: Set the value of the receiver to *l*.

`== (long long int l)`
: Return `TRUE` if the receiver is equal to *l*.

`+ (long long int l)`
: Add *l* to the the receiver.

`++ (void)` Implements both the C prefix and postfix increment operators for `LongInteger` objects.

`- (long long int l)`
: Subtract *l* from the receiver.

`- (void)` When used as a prefix operator, negate the argument.

`-- (void)` The C decrement operator, both prefix and postfix, for `LongInteger` objects.

`% (long long int l)`
: Return a `LongInteger` that is the modulus of the receiver and the argument.

`%= (int i)`
: Perform a modulus of the receiver and its argument, and store the result in the receiver. Returns the receiver.

`& (long long int l)`
: Perform a bitwise AND of the receiver and the operand. The `Object` class's `&` operator overloads C's unary "address of" prefix operator. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

`&& (long long int l)`
: Return `TRUE` if both the receiver and the operand evaluate to `TRUE`.

`& ( int i)`
: Perform a bitwise AND of the receiver and the operand, and store the result in the receiver. Returns the receiver.

`* (long long int l)`
: Multiply the receiver by *l*.

`/ (long long int l)`
: Divide the receiver by *l*.

`< (long long int l)`
: Return `TRUE` if the receiver is less than the operand, `FALSE` otherwise.

`<= (long long int l)`
: Return `TRUE` if the receiver is less than or equal to the operand, `FALSE` otherwise.

`<< (int i)`
: Shift the receiver left by the number of bits in the operand, an `Integer.`

`> (long long int l)`
: Return `TRUE` if the receiver is greater than the operand, `FALSE` otherwise.

`>= (long long int l)`
: Return `TRUE` if the receiver is greater than or equal to the operand, `FALSE` otherwise.

`>> (int l)`
: Shift the receiver right by the number of bits in the operand, an `Integer.`

`^ (long long int l)`
: Perform a bitwise XOR of the receiver and the operand.

`^= (int i)`
: Perform a bitwise XOR of the receiver and the operand, and store the result in the receiver. Returns the receiver.

`bitComp (void)`
: Return a bitwise complement of the receiver.

`invert (void)`
: Return `TRUE` if the receiver evaluates to `FALSE, FALSE` if the receiver evaluates to `TRUE.`

`| (long long int l)`
: Perform a bitwise OR of the receiver and the operand.

`|= (int i)`
: Perform a bitwise OR of the receiver and the operand, and store the result in the receiver. Returns the receiver.

`| (long long int l)`
: Return `TRUE` if either the receiver or the operand evaluate to `TRUE.`

`~ (void)` When overloading C's unary '`~`' operator, is synonymous with the `bitComp` method, above.

3.56 `Pen` class

`Pen` class objects define drawing parameters for graphics primitives when drawing points or lines. Currently the only parameter that `Pen` implements with the `width` instance variable, which controls the width of graphics shapes like points and lines.

The `Point` class section contains an example of how to use `Pen` objects when drawing graphics. See ⟨undefined⟩ [Point], page ⟨undefined⟩.

Instance Variables

`alpha` Defines an object's opacity when using drawing libraries that support alpha channel blending (presently that is only the `Line` class drawing library, which uses the X Render extension). The value is an Integer in the range, 0..65535 (0..0xffff hexadecimal). For graphics library functions that do not support alpha blending, this value is not used.

`colorName`
A `String` containing the name of the `Pen` object's color. The X Window System's libraries use color names defined in the server's `rgb.txt` file, which is normally located a directory that contains the server's configuration files (e.g., `<`*prefix*`>/etc/X11` or similar).

`width` The width in pixels of a line when drawing non-filled shapes. The default is 1.

3.57 `Point` class

Objects of `Point` class describe a location with X- and Y-axis coordinates. These objects are commonly used to describe locations on a display.

The `draw` and `drawWithPen` methods allow you to draw `Point` objects on a `X11CanvasPane` object at the coordinates given by the point's `x` and `y` instance variables. There is an example program at the end of this section. See ⟨undefined⟩ [point-example], page ⟨undefined⟩.

If you use the `draw` method and don't provide a `Pen` object, Ctalk draws the point with the default diameter of one pixel, and the default color is black.

Instance Variables

`x` The x coordinate of the receiver object.

`y` The y coordinate of the receiver object.

Instance Methods

`draw (X11Pane` *pane_object*`)`
Draw the receiver on *pane_object* at the coordinates given by the receiver's `x` and `y` instance variables.

This method draws to the `X11Pane` objects buffer. To draw offscreen to a separate `X11Bitmap`, object refer to the method `drawPoint` in class `X11Bitmap`. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`drawWithPein (X11Pane` *pane_object*`, Pen` *pen_object*`)`
Draw the receiver on *pane_object* at the coordinates given by the receiver's `x` and `y` instance variables, with the diameter and color supplied by the *pen_object* argument. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

This method also draws to the `X11Pane` object's buffer. To draw offscreen to a separate `X11Bitmap` object, refer to the method `drawPoint` in class `X11Bitmap`. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

```
int main () {
```

```
X11Pane new xPane;
InputEvent new e;
X11PaneDispatcher new xTopLevelPane;
X11CanvasPane new xCanvasPane;
Application new paneApp;

paneApp enableExceptionTrace;
paneApp installExitHandlerBasic;

xPane initialize 10, 10, 250, 250;
xPane inputStream eventMask =        /* Tell the main window's event */
  WINDELETE|EXPOSE;                  /* object, a X11InputStream,    */
                                     /* which types of events we     */
                                     /* plan to use.                 */

xTopLevelPane attachTo xPane;        /* The attachTo methods also */
xCanvasPane attachTo xTopLevelPane;  /* set the dimensions of the */
                                     /* subpanes before they are  */
                                     /* mapped and raised along   */
                                     /* with the top-level pane.  */

xPane map;
xPane raiseWindow;
xPane openEventStream;               /* Before we can do any      */
                                     /* drawing on the window, we */
                                     /* need to start sending and */
                                     /* receiving events from the */
                                     /* X server.  That is what   */
                                     /* openEventStream does.     */

xPane background "yellow";           /* Setting the background of */
xPane clearWindow;                   /* an X11Pane object sets the*/
                                     /* background of the actual  */
                                     /* window.                   */

xCanvasPane background "yellow";     /* Setting the background of */
                                     /* a buffered pane like a    */
                                     /* X11CanvasPane sets the    */
                                     /* background color of its   */
                                     /* buffer.                   */

xCanvasPane clearRectangle 0, 0, 250, 250; /* In both cases, we   */
                                           /* need to update the  */
                                           /* pane before the new */
                                           /* color is visible,   */
                                           /* with either,        */
                                           /* "clearWindow," or,  */
```

```
                                                /* "clearRectangle."   */
    xCanvasPane pen width = 100;
    xCanvasPane pen colorName = "red";
    xCanvasPane drawPoint 40, 40;
    xCanvasPane pen colorName = "green";
    xCanvasPane drawPoint 120, 40;
    xCanvasPane pen colorName = "blue";
    xCanvasPane drawPoint 80, 90;

    while (TRUE) {
      xPane inputStream queueInput;
      if (xPane inputStream eventPending) {
        e become xPane inputStream inputQueue unshift;
        xPane subPaneNotify e;          /* We need to notify subPanes */
                                        /* e.g., xCanvasPane of the   */
                                        /* input events from the GUI. */
        switch (e eventClass value)
          {
          case WINDELETE:
            xPane deleteAndClose;
            exit (0);
            break;
          case EXPOSE:
          case RESIZENOTIFY:
    xCanvasPane pen width = 100;
    xCanvasPane pen colorName = "red";
    xCanvasPane drawPoint 40, 40;
    xCanvasPane pen colorName = "green";
    xCanvasPane drawPoint 120, 40;
    xCanvasPane pen colorName = "blue";
    xCanvasPane drawPoint 80, 90;
            break;
          default:
            break;
          }
      }
    }
  }
```

3.58 `Line` class

A `Line` object contains the coordinates for a line graphics shape. `Line` objects use `Pen` objects to specify the width and color of the line. If an application doesn't specify a `Pen` object, the default width is one pixel and the default color is black. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

Here is an example of how to draw a line on a `X11Pane` window.

```
int main () {
  X11Pane new xPane;
  InputEvent new e;
  Pen new bluePen;
  Line new basicLine;

  xPane initialize 10, 10, 100, 100;
  xPane map;
  xPane raiseWindow;
  xPane openEventStream;
  bluePen width = 10;
  bluePen colorName = "blue";
  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;
      switch (e eventClass value)
{
case WINDELETE:
  xPane deleteAndClose;
  exit (0);
  break;
case EXPOSE:
case RESIZENOTIFY:
  basicLine start x = 90;
  basicLine start y = 10;
  basicLine end x = 10;
  basicLine end y = 90;
    basicLine drawWithPen xPane, bluePen;
  break;
default:
  break;
}
    }
  }
}
```

Instance Variables

`start` A `Point` object that contains the receiver's starting `x` and `y` coordinates.

`end` A `Point` object that contains the receiver's ending `x` and `y` coordinates.

Instance Methods

`draw (X11Pane` *`paneObject`*`)`

Draw a line on the *paneObject*'s visible area at the receiver's `start` and `end` coordinates, using a default pen width of one pixel and default color of black.

This method is mainly used for drawing on a buffer already attached to a `X11Pane` object. To draw offscreen to a separate `X11Bitmap` object, use `drawLine` in class `X11Bitmap`. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`drawWithPen (X11Pane` *`paneObject`*`, Pen` *`penObject`*`)`

Draw a line on the *paneObject*'s visible area at the receiver's `start` and `end` coordinates. The *penObject* argument contains the width and color of the line. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

This method is also used mainly for drawing on a buffer already attached to a `X11Pane` object. To draw offscreen to a separate `X11Bitmap` object, use `drawLine` in class `X11Bitmap`. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

3.59 `Rectangle` class

Programs can use `Rectangle` objects to describe and draw four-sided square and rectangular shapes.

Although the `Rectangle` objects are meant to be displayed on X11Pane displays that require only the origin and extent of a rectangle to display it, the class defines each side's line separately, in case a program needs to work with all of the object's dimensions.

The methods `drawWithPen` and `fillWithPen` draw either the sides of a rectangle or a filled rectangle, using the line width and color defined by a `Pen` object. The methods `draw` and `fill` use a default line width of one pixel and the color black.

Here is a simple example program.

```
int main () {
  X11Pane new xPane;
  InputEvent new e;
  Pen new bluePen;
  Rectangle new rectangle;

  xPane initialize 10, 10, 100, 100;
  xPane map;
  xPane raiseWindow;
  xPane openEventStream;

  /*
   *  The rectangle's sides are four Line objects:
   *  top, bottom, left, and right.  There is also
   *  a "dimensions" method that fills in all of
   *  the sides' dimensions.
   */
  rectangle top start x = 10;
  rectangle top start y = 10;
  rectangle right end x = 80;
  rectangle right end y = 80;

  bluePen width = 3;
  bluePen colorName = "blue";
```

```
        while (TRUE) {
          xPane inputStream queueInput;
          if (xPane inputStream eventPending) {
            e become xPane inputStream inputQueue unshift;
            switch (e eventClass value)
              {
              case WINDELETE:
                xPane deleteAndClose;
                exit (0);
                break;
              case EXPOSE:
              case RESIZENOTIFY:
        /*
         *  To draw only the outline of the rectangle,
         *  use the "draw" method instead.
         */
        rectangle fillWithPen xPane, bluePen;
                break;
              default:
                break;
              }
          }
        }
      }
```

Instance Variables

`top`

`right`

`bottom`

`left` `Line` objects that contain the dimensions of each of the rectangle's sides. Each `Line` instance variable in turn contains `Point` objects that contain the endpoints of each side.

Note: Unlike `Line` objects, which specify endpoints as absolute window coordinates, `Rectangle` objects specify the endpoints of each side as relative to the origin of the `Rectangle` object; that is, as height and width dimensions. See ⟨undefined⟩ [Line], page ⟨undefined⟩.

Instance Methods

`clear (X11Pane` *`paneObject`*`)`

Clear the rectangle defined by the receiver to the background color of *paneObject*. This method requires that the dimensions of the rectangle be completely defined. The easiest way to do this is with the `dimensions` method, below.

`draw (X11Pane paneObject)`
: Draw an outline of the receiver's rectangle dimensions on *paneObject*'s display area, using a line width of one pixel and the color black.

`drawWithPen (X11Pane paneObject, Pen penObject)`
: pDraw an outline of the receiver's rectangle dimensions on *paneObject*'s display area, using the line width and color given by *penObject*.

`dimensions (Integer xOrigin, Integer yOrigin, Integer xSize, Integer ySize)`
: A convenience method that fills in the dimensions of each of the receiver's sides from the arguments, which specify the origin, height, and width of the receiver.

`fill (X11Pane paneObject)`
: Draw a solid rectangle on *paneObject*'s display area, using a default line width of one pixel and the default color, black.

`fillWithPen (X11Pane paneObject, Pen penObject)`
: Draw a solid rectangle on *paneObject*'s display area, using the color given by *penObject*.

3.60 `Circle` class

Objects of `Circle` class contain instance variables that define the center and radius of a circle. The class also defines methods for drawing circles on X displays.

The methods defined in this class can render circles on any GUI drawable surface. To work with the pane buffering mechanism, however, `X11CanvasPane` class also defines drawing methods. See ⟨undefined⟩ [X11CanvasPane], page ⟨undefined⟩.

Circle drawing methods also require an interior color argument. This is the name of the color within the circle. To give the appearance of drawing just the circle's rim, a program can set the interior color to the window's background color. If the fill argument is TRUE, then the circle is filled with the rim color. If the fill argument is FALSE, the width and color of the rim are determined by the pen object used to draw the circle, and the color inside the circle is determined by the interior color argument.

```
int main () {
  X11Pane new xPane;
  InputEvent new e;
  X11PaneDispatcher new xTopLevelPane;
  X11CanvasPane new xCanvasPane;
  Application new paneApp;
  Circle new inner;
  Circle new outer;
  Circle new middle;
  Pen new innerPen;
  Pen new middlePen;
  Pen new outerPen;
  String new bgColor;

  paneApp enableExceptionTrace;
```

```
paneApp installExitHandlerBasic;

bgColor = "white";

xPane initialize 10, 10, 300, 300;
xTopLevelPane attachTo xPane;
xCanvasPane attachTo xTopLevelPane;
xPane map;
xPane raiseWindow;
xPane openEventStream;
xPane clearWindow;
xCanvasPane background bgColor;

inner center x = 150;
inner center y = 150;
inner radius = 30;

innerPen colorName = "navy";
innerPen width = 10;

middle center x = 150;
middle center y = 150;
middle radius = 40;

middlePen colorName = "blue";
middlePen width = 10;

outer center x = 150;
outer center y = 150;
outer radius = 50;

outerPen colorName = "sky blue";
outerPen width = 10;

xCanvasPane pen width = 1;
xCanvasPane pen colorName = "black";

xCanvasPane drawCircle outer, outerPen, FALSE, bgColor;
xCanvasPane drawCircle middle, middlePen, FALSE, bgColor;
xCanvasPane drawCircle inner, innerPen, FALSE, bgColor;
xCanvasPane drawLine 50, 150, 250, 150;
xCanvasPane drawLine 150, 50, 150, 250;
xCanvasPane refresh;

while (TRUE) {
  xPane inputStream queueInput;
  if (xPane inputStream eventPending) {
```

```
            e become xPane inputStream inputQueue unshift;
            xPane subPaneNotify e;
            switch (e eventClass value)
              {
              case WINDELETE:
                xPane deleteAndClose;
                exit (0);
                break;
              case EXPOSE:
              case RESIZENOTIFY:
        xCanvasPane drawCircle outer, outerPen, TRUE, bgColor;
        xCanvasPane drawCircle middle, middlePen, TRUE, bgColor;
        xCanvasPane drawCircle inner, innerPen, TRUE, bgColor;
        xCanvasPane drawLine 50, 150, 250, 150;
        xCanvasPane drawLine 150, 50, 150, 250;
        xCanvasPane refresh;
                break;
              default:
                break;
              }
          }
        }
      }
```

Instance Variables

`center` A `Point` object that defines the `x,y` coordinates of the circle's center. If drawn on a X display the units represent pixels on the display. See ⟨undefined⟩ [Point], page ⟨undefined⟩.

`radius` An `Integer` that defines the `Circle` object's radius.

Instance Methods

`draw (Pane` *`paneObject`*`, Integer` *`filled`*`, String` *`bgColor`*`)`
`draw (Pane` *`paneObject`*`, Pen` *`penObject`*`, Integer` *`filled`*`, String` *`bgColor`*`)`
Draw the receiver circle on the display. If a `Pen` argument is given, use the `Pen` object's color and line width to draw the circle. If a `Pen` argument is not given use a default `Pen`, and fill the interior of the circle with *bgColor* if *filled* is false.

3.61 `Method` Class

The `Method` class provides a virtual mechanism for objects to maintain references to methods and also to call the methods.

A `Method` instance variables can contain all of the data of an actual method. Much of the method's instance data is used only by the Ctalk front end, and the instance data may or may not be needed at run time when a `Method` object is created or used. This class defines the instance variables anyway in case a program needs them.

Note that defining a `Method` object does not include the method receiver's class library. The program must already have included a method's receiver class, either by constructing an object of that class, or by including the class with the `require` keyword.

Note also that, in this version, `Method` object calls are only tested with a receiver, so you may want to use a virtual method as part of an expression, you might want to write a wrapper method for it, as in this example. The `methodObjectMessage` method is defined in class `Object`. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

```
String instanceMethod selfConcat (String __arg) {
  self = self + __arg;
  return NULL;
}

int main () {
  Method new m;
  String new s;
  Exception new e;

  s = "Hello,";

  m definedInstanceMethod "String", "selfConcat";
  m withArg " world!";
  s methodObjectMessage m; /* methodObjectMessage is defined */
                           /* in Object class.               */
  if (e pending) {
    e handle;
  } else {
    printf ("%s\n", s);
  }
}
```

Instance Variables

`methodName`
: A `String` object that contains the method's name.

`methodSelector`
: A `String` object that contains the name of the method's selector.

`returnClass`
: A `String` object that contains the name of the method's return class.

`rcvrClassObject`
: A `Symbol` that refers to the method's receiver class object.

`methodFn` A `Symbol` that contains the address of the method's function call.

`methodSource`
: A `String` that contains the method's source code.

`methodParameters`
: A `List` that contains the method's paramter definitions.

`nParams` An `Integer` that contains the number of parameters the method requires.

`varargs` An `Integer` that is either `TRUE` or `FALSE` depending on whether the method takes a variable number of arguments.

`nArgs` An `Integer` that contains the number of arguments the method is called with.

`errorLine`
An `Integer` that contains the first line of the method in a class library or program input.

`errorColumn`
An `Integer` that contains the first column of the method in a class library or program input.

`argFrameTop`
An `Integer` that contains the stack index of the method's first argument.

`rcvrFrameTop`
An `Integer` that contains the stack index of the method's receiver.

`imported` An `Integer` that is either `TRUE` or `FALSE` depending on whether the method is imported from a class library.

`queued` An `Integer` that is either `TRUE` or `FALSE` depending on whether a method is queued for output.

`methodArgs`
A `List` of references to the method's arguments.

`localObjects`
A `List` of references to the method's local objects.

`userObjects`
A `List` of references to objects created during the method's execution.

`localCVARs`
A `List` of references to the method's local C variables.

`isInitialized`
A `Boolean` that is True only if the `Method` object has been initialized to refer to an actual method. This variable should only be set by the methods `definedInstanceMethod` and `definedClassMethod` (in `Object` class).

Instance Methods

`definedClassMethod (String` *`classname`*`, String` *`name`*`)`
Initialize the receiver with the class method named by the arguments.

`definedInstanceMethod (String` *`classname`*`, String` *`name`*`)`
Initialize the receiver with the instance method named by the arguments.

`withArg (Object` *`method_argument`*`)`
Define an argument for the receiver method.

This method normally precedes a `methodObjectMessage` call. For examples of its use, refer to the `methodObjectMessage` section. See ⟨undefined⟩ [methodObjectMessage], page ⟨undefined⟩.

`setCallbackName (String` *`name`*`)`

Sets the receiver object's name to the name of the callback, which is generally set when the program is compiled. This does not change the name of the actual method, only the name by which the Method object that represents it is referred to.

Here is a slightly hypothetical example of the steps that use `setCallbackName` to set up a callback method.

```
/*
 * 1. In MyClass, the callback here is defined as an instance
 * variable.
 */
MyClass instanceVariable myCallback Method NULL;

...

/*
 * 2. Also in MyClass, define a method to configure the callback.
 */
MyClass instanceMethod onEvent (String methodClassName,
                                String methodName) {
  Method new callbackMethod;

  callbackMethod definedInstanceMethod methodClassName, methodName;

  /* This sets the class and name of callbackMethod to the name
      of the callback instance variable defined above, at the
      start of the class. */
  callbackMethod setCallbackName "myCallback";

  self addInstanceVariable "myCallback", callbackMethod;
}

/*
 * 3. In the program's source file, define the callback method
 * itself.
 */
MyClass instanceMethod callbackMethod (void) {
  ... do something ...
}

...
```

```
/*
 * 4. Also in the program's source file, set up the callback
 * method during program initialization.
 */
int main () {
  MyClass new myProgram;

  ...

  myProgram onEvent "MyClass", "callbackMethod";

  ...
}
```

For a working example, refer to the class `GLXCanvasPane`, which uses this process to assign callbacks.

3.62 `Pane` Class

`Pane` is the superclass of all classes that handle graphical displays.

Subclasses need to provide their own constructor and destructor methods for extra object construction and cleanup. `Pane` subclasses need to provide at least a `paneBuffer` (below) allocated during object creation and freed during object deletion, unless the subclass is very simple.

Instance Variables

`children` A `List` of child panes.

`cursor` A `Point` object that contains the X and Y coordinates within the pane where text writes will occur.

`mapped` An `Integer`. If `TRUE`, display the pane when it, or its parent pane, receives a `refresh` message.

`origin` A `Point` object that contains the X and Y coordinates of the upper left-hand corner of the pane within the parent pane, or the terminal window or display if the pane is the parent pane.

`paneBackingStore`
A `Symbol` object that contains the memory address of the pane's backing store buffer. Subclasses should provide constructors that allocate this memory, using the *__ctalkCreateWinBuffer* library function. The methods `map` or `unmap` handle the task of saving and restoring window contents, by calling functions like *__ctalkANSITerminalPaneMapWindow* and *__ctalkANSITerminalPaneUnMapWindow*. Classes that re-implement `map` and `unmap` need to handle these tasks also.

`paneBuffer`
: A `Symbol` object that contains the memory address of the pane's display buffer. Subclasses should provide constructors that allocate this memory, using the *__ctalkCreateWinBuffer* library function.

`size`
: A `Point` object that contains the width and height of the pane.

Instance Methods

`attach (Pane childPane)`
: Attach *childPane* to the receiver. If mapped, then the child pane is displayed the next time the parent pane and its children are updated. If *childPane* is an `ANSITerminalPane` object, the child pane inherits the parent's input and output stream handles.

`deletePaneBuffer`
: Deletes the `paneBuffer` backing store memory that constructors should allocate when creating subclass objects. Subclasses should also use this message in destructor or cleanup methods.

3.63 `ANSITerminalPane` Class

The `ANSITerminalPane` class allows applications to write output to multiple panes or windows on ANSI terminals, consoles and `xterm`s.

Printing output to a pane does not immediately update the display; instead, the pane's contents are updated, and the pane is displayed with the `refresh` message.

The `ANSITerminalPane` class supports the display multiple panes. Each pane is a child of the initial, or parent, pane. Child panes attached to the parent pane (with the `Pane : attach` method) inherit the input and output streams of the parent pane.

The `map` configures a child pane for display on top of its parent pane, and the child pane is displayed on top of the parent pane at the next `refresh` message.

Coordinates and dimensions are numbered from 1,1, which is the upper left-hand corner of a terminal or a content region.

`ANSITerminalPanes` support bold, blinking, reverse, and underline text attributes for its contents. Some attributes, like blinking and underlined text, or foreground and background colors, are not universally supported.

The program listing shows how to display parent and child `ANSITerminalPane` objects.

```
int main () {
  ANSITerminalPane new parentPane;
  ANSITerminalPane new childPane;

  /*
   *  Use openOn with the terminal device
   *  to open a remote serial terminal.
   */
/*   parentPane paneStream openOn "/dev/ttya"; */
  parentPane initialize 1, 1, 79, 24;
```

```
    parentPane withShadow;
    parentPane withBorder;
    parentPane refresh;

    parentPane gotoXY 29, 10;
    parentPane printOn "Parent Pane";
    parentPane gotoXY 25, 11;
    parentPane printOn "Please press [Enter].";
    parentPane refresh;

    getchar ();  /* Actual apps should use getCh, etc. */

    childPane initialize 10, 10, 40, 10;
    childPane withShadow;
    childPane withBorder;
    childPane boldOnOff;
    parentPane map childPane;
    parentPane attach childPane;

    childPane gotoXY 13, 2;
    childPane printOn "Child Pane";
    childPane gotoXY 10, 3;
    childPane printOn "Please press [Enter].";

    parentPane refresh;

    getchar ();

    parentPane unmap childPane;

    parentPane refresh;

    parentPane paneStream closeStream;

    childPane delete;
    parentPane delete;
}
```

Instance Variables

`paneStream`

An `ANSITerminalStream` object containing the input and output channels and parameters for the pane and its child panes. When created with `new`, initializes input and output to the terminal's standard input and standard output file handles. Communication settings, including TTY settings, can be set with methods from the `ANSITerminalStream` class. See ⟨undefined⟩ [ANSITerminalStream], page ⟨undefined⟩.

`shadow` If `TRUE`, draw a shadow beneath the pane. The shadow is drawn over the background window and is not part of the pane's content region.

`border` If `TRUE`, draw a border around the edges of the pane. The border is within the pane's content region, and it can objscure text beneath it.

`parentOrigin`
A `Point` object containing the X and Y coordinates of the parent pane's origin.

`parentClip`
A `Point` object containing the X and Y dimensions of the parent pane's content area.

Instance Methods

`blinkOnOff (void)`
Toggle the pane's blinking graphics attribute; if enabled, display blinking text; if disabled, display normal text. Blinking text is not supported on all terminals.

`boldOnOff (void)`
Toggle the pane's bold graphics attribute; if enabled, display bold text; if disabled, display normal text.

`childRefresh (void)`
Refresh the receiver, a child pane. This method is called by `refresh` for each of a parent pane's children.

`cursorPos (int x, int y)`
Position the software cursor at coordinates *x*, *y* within the pane's content area. This method is a synonym for `gotoXY`, below.

`delete` The class destructor. This method performs the extra cleanup that the `ANSITerminalPane` class requires.

`gotoXY (int x, int y)`
Position the software cursor at coordinates *x*, *y* within the pane's content area.

`initialize (int x_org, int y_org, int x_size, int y_size)`
Initialize the receiver pane's coordinate instance variables, and screen buffers.

`map (ANSITerminalPane __child)`
Enable the display of a child pane, and buffer any screen contents. The receiver should be a pane that completely encloses the child pane's content region and shadow if any. The pane is displayed when the parent pane receives a `refresh` message.

`new (char *__paneName)`
Create a new `ANSITerminalPane` object. If more than one label is given in the argument list, create new `ANSITerminalPane` objects with those names. This method also creates and initializes the `paneStream` (class `ANSITerminalStream`) instance variable. See ⟨undefined⟩ [ANSITerminalStream], page ⟨undefined⟩.

`printOn (char *__fmt, ...)`
: Print the arguments to the pane's content area at the software cursor position. The output is displayed after the receiver pane receives a `refresh` message.

`putCharXY (intx, inty, charc)`
: Put character *c* at coordinates *x*,*y* of the pane's content area. The character is displayed immediately using the pane's current graphics mode.

`refresh (void)`
: Display the receiver pane content and decorations like borders and shadows on the screen. This method also calls `childRefresh` for each child pane that is attached (with `attach`, class `Pane`) to the receiver pane. See ⟨undefined⟩ [Pane], page ⟨undefined⟩.

`resetGraphics (void)`
: Reset the pane's graphics attributes to normal text; i.e., bold, underline, reverse, and blink attributes are turned off.

`reverseOnOff (void)`
: Toggle the pane's reverse graphics attribute; if enabled, display text in inverse video; if disabled, display normal text.

`terminalHeight`
: Returns an `Integer` with the terminal's height in character rows.

  Note: If an operating system has a terminal interface that Ctalk doesn't know about, then the method returns 0.

`terminalWidth`
: Returns an `Integer` with the terminal's width in character columns.

  See the note about terminal compatibility in the `terminalHeight` entry, above.

`underlineOnOff (void)`
: Toggle the pane's underline graphics attribute; if enabled, display text underlined; if disabled, display normal text. Underlined text is not supported on all terminals.

`unmap (ANSITerminalPane __child)`
: Hide a child pane, and restore any screen contents that were obscured when the child pane was displayed. The child pane is withdrawn when the parent pane receives the next `refresh` message.

`withBorder (void)`
: Enable the display of a border around the edges of the pane's content area, using ANSI line drawing characters. The border is within the pane's content area and can obscure text at the edges of the pane.

`withShadow (void)`
: Enable the display of a shadow underneath the pane. The shadow is outside of the pane's content area, and should be within the clipping area of the parent pane.

3.64 `ANSIWidgetPane` Class

`ANSIWidgetPane` contains methods and instance variables that are useful when creating widgets, which are made up of one or more pane objects for display on ANSI terminals.

Widgets should be designed so they can be displayed either independently or above a main pane object. That means a widget class needs to provide its own methods for rendering the widget, handling input, returning the input to the application program, and cleaning up if necessary.

The `ANSIWidgetPane` class provides basic methods for these tasks, but subclasses probably will need to implement their own versions of these methods. For example, the `ANSITextEntryPane` class contains its own `handleInput` method.

Or, for example, a subclass can provide a `withdraw` method, as in this method from `ANSITextEntryPane`, which unmaps the widget pane if it was popped up over another pane. See ⟨undefined⟩ [ANSITextEntryPane], page ⟨undefined⟩.

```
ANSIWidgetPane instanceMethod withdraw (void) {
  if (self parentPane) {
    self unmap;
  } else {
    self paneStream clear;
  }
  return NULL;
}
```

A Note About Using Widgets with Serial Terminals.

The `ANSIWidgetPane` classes do not, at this time, provide any methods for setting terminal parameters for a widget and all subpanes. That means applications must see the terminal parameters of each subwidget, and there does not yet exist a general mechanism for handling the input and output of entire sets of widgets. However, classes can always implement convenience methods if necessary. For an example, see the `ANSIMessageBoxPane` section. See ⟨undefined⟩ [ANSIMessageBoxPane], page ⟨undefined⟩.

Instance Variables

`hasFocus` An `Integer` that is either `TRUE` or `FALSE` depending on whether the subwidget has the input focus. Methods should set and check the `isInputPane` instance variable, below, to make sure that a subwidget's class accepts user input.

`isInputPane`
An `Integer` that is either `TRUE` or `FALSE` depending on whether the pane can take the input focus.

`paneID` An `Integer` that contains an identifier of a widget pane or subpane.

`parentPane`
A `Key` object that contains the address of a parent pane, if the widget is displayed over a normal pane. If this variable is set, then the widget needs to map and unmap itself from the parent pane as with a normal child pane.

`titleString`
An optional `String` object that can be displayed as the widget's title.

`withdraw` Delete the receiver from the display, by clearing the display if the widget is displayed independently, or by unmapping it from a parent widget.

Instance Methods

`addBuffer (Integer` *`width`*`, Integer` *`height`*`,`
`Integer` *cellSize*) A convenience method that creates the receiver pane's buffers with the width, height and character cell size given as the arguments. The *cellSize* argument should almost always be '`1`'. If the pane has buffers created previously by another method, `addBuffer` deletes the old buffers first.

`handleInput`
Process `InputEvent` objects from the receiver's `paneStream` input handle. This `handleInput` definition provides only basic functionality. Subclasses should re-implement this method with the additional functions that the widget needs.

`map (void)`
Maps the receiver widget pane over another pane that was defined by a previous `parent` message (below).

`mapSubWidget (ANSIWidgetPane` *`subPane`*`)`
Maps the receiver widget pane over another pane that was defined by a previous `parent` message (below), and sets the *subPane*'s `mapped` instance variable to `TRUE`, and adds *subWidgetPane*, to the parent pane's `children` list. *subPane* may be any subclass of `ANSIWidgetPane`.

`new (`*`paneName1`*`, `*`paneName2`*`, ... `*`paneNameN`*`;)`
Create one or more new `ANSIWidgetPane` objects. The object is similar to an `ANSITerminalPane` object, but it contains the additional instance variables listed in the previous section. This method also relies on the `ANSITerminalPane` methods `withShadow` and `withBorder`, and the `ANSITerminalStream` method `openInputQueue`.

`parent (ANSITerminalPane` *`parentPaneObject`*`)`
Sets the receiver's `parentPane` instance variable to the main `ANSITerminalPane` (or subclass) object that the receiver is to be mapped over. This method also provides the subWidget with copies of the parent's input and output stream handles.

`title (String` *`titleString`*`)`
Set the receiver's title string to the argument.

`unmap (void)`
Unmaps the receiver from its parent pane. Used after previous `parent` and `map` messages (above).

3.65 `ANSIButtonPane` Class

`ANSIButtonPane` objects represent pushbutton widgets. Generally, they are sub-panes of other widget classes. For program examples, see See ⟨undefined⟩ [ANSIMessageBoxPane], page ⟨undefined⟩, and the methods in the `ANSIButtonPane` and `ANSIMessageBoxPane` class libraries.

Instance Variables

`buttonText`
: The text that will appear within the button.

`outputBuffer`
: The buffer that contains the ANSIButtonPane's return text. Normally, the ANSIButtonPane object is drawn by a parent widget, which uses its `handleInput` and `show` methods to retrieve the button's result.

  In cases where the ANSIButtonPane object is displayed on its own and uses the ANSIButtonPane `handleInput` and `show` methods listed in the next section, this variable contains an empty string if the user presses ***`Escape`***, or the button text if the user presses ***`Return`***, which is returned by the `show` method, below.

Instance Methods

`focusHighlightOnOff`
: Toggle the button's highlight.

`handleInput (void)`
: Wait for input from the receiver's `paneStream` object. Withdraws the receiver if the the user types an escape (`0x1b`) or carraige return (`0x0d`) character, or the `paneStream` object receives these characters from another input source.

  The `paneStream` instance variable (which is declared in `ANSITerminalPane` class), contains a reference to an `ANSITerminalStream` object. See ⟨undefined⟩ [ANSITerminalStream], page ⟨undefined⟩.

  This method sets the value of the `outputBuffer` instance variable, as described above.

`new (`*`button1_name, button2_name, ...;`*`)`
: Create one or more new `ANSIButtonPane` object, with border and shadow decorations, for each name given in the argument list. The buttons' exact sizes are determined by the `withText` method, below.

`show (Integer` *`x_origin`*`, Integer` *`y_origin`*`)`
: Display the button or map it to a parent widget. The *x_origin* and *y_origin* are relative to the upper left of the display if the button is displayed independently, or relative to the upper left-hand corner of a parent pane.

  This method returns the `outputBuffer` instance variable, which contains the result of input from the user or another source, as described above.

`withText (String` *`button_text`*`)`
: Set the text that will appear inside the button. The method adjusts the button dimensions to fit the text.

3.66 `ANSILabelPane` Class

The `ANSILabelPane` class draws text labels on a terminal, with or without border and shadow decorations, and with the graphics attributes provided by `ANSITerminalPane` class. See ⟨undefined⟩ [ANSITerminalPane], page ⟨undefined⟩.

The default is to draw a single-line or multi-line label in a pane with a border and drop shadow. When drawing multi-line labels, it is necessary to give the pane's dimensions to the `appendLine` method, below.

To draw a single-line label in reverse video, without any decorations and large enough to contain only the text, use a routine like the following.

```
ANSILabelPane new labelBox;

labelBox borderLess = TRUE;   /* The decorations are specified with */
labelBox border = FALSE;      /* instance variables.                */
labelBox shadow = FALSE;
labelBox reverseOnOff;        /* Defined in ANSITerminalPane class. */

labelBox appendLine "LabelBox Text", 20, 0;
```

Note that when creating pane buffers, Ctalk numbers the lines from zero, so it is safe to specify a pane's height as '`0`'.

When drawing a label alone, pressing Escape or Enter withdraws the label from the display.

Instance Variables

`borderLess`
: Specify that the label not contain extra space around the edges for a border.

`text`
: nA `List` containing the label text, one item per line of text.

`viewHeight`
: An `Integer` that contains the height of the label's viewable area in character rows.

`viewWidth`
: An `Integer` that contains the width of the label's viewable area in character columns.

Instance Methods

`appendLine (String` *`text`*`)`
`appendLine (String` *`text`*`, Integer` *`width`*`, Integer` *`height`*`)`
: Add a line of text to the label's contents. If given with a width and height, specify the size of the label. Otherwise, the label is drawn large enough to display the contents.

  If the label contains multiple lines of text, then the dimensions given (or calculated) for the last line determine the size of the label.

`cleanup (void)`
: Delete the buffers associated with the pane object.

`display (Integer` *`x`*`, Integer` *`y`*`)`
: Display the pane at *x*,*y* on the display or parent pane. Unlike `show`, below, does not wait for user input.

`handleInput (void)`
: Process input for the pane. This method is normally called by the `Show` method, below. When the pane is displayed by itself, pressing Escape or Enter returns from the method, and the `Show` method withdraws the pane from the display.

`new (label1, label2, label3,...;)`
: Create new `ANSILabelPane` objects for each member of the argument list. The arguments specify the names of the new object.

`refresh (void)`
: Draw the pane's contents on the terminal.

`show (Integer x, Integer y)`
: Display the pane at the position *x*, *y* on the terminal. This method also calls the `handleInput` method, above, and waits for the user's input before returning.

`sizePane (Integer width, Integer height)`
: Set the size of the pane object and its buffers. This method is normally called by the `appendLine` method, above.

3.67 `ANSIListBoxPane` Class

An `ANSIListBoxPane` object displays a list of items, and allows the user to select one of the items in the list by using the cursor motion keys.

Each item in the list is an `ANSILabelPane`. To modify the appearance of the items, refer to the `ANSILabelPane` class. See ⟨undefined⟩ [ANSILabelPane], page ⟨undefined⟩.

After exiting by pressing Escape or Enter, a program can retrieve the text of the selected item.

Here is a simple program that displays a list of items, then prints the selected item's text before exiting.

```
int main () {
  ANSIListBoxPane new listBox;

  listBox appendLine "Item 1";
  listBox appendLine "Item 2";
  listBox appendLine "Item 3";
  listBox appendLine "Item 4";
  listBox appendLine "Item 5";

  listBox show 2, 2;   // Waits for the user to press Escape or Enter
                       // before returning.

  printf ("\nYou selected %s.\n", listBox selectedText);

  listBox cleanup;
}
```

Instance Variables

`items` A `List` of `ANSILabelPane` items that contain the text of the list selections.

`oldSelectedContent`
An `ANSILabelPane` object that contains the content of the previous selection. Used for erasing the previous selection before drawing the new selection.

`prevSelectedLine`
An `Integer` that contains the index of the previously selected item. The `ANSIListBoxPane` indexes items starting with 1 for the first item.

`selectedContent`
An `ANSILabelPane` object that contains the contents of the currently selected item.

`selectedLine`
An `Integer` that contains the index, counting from 1, of the currently selected item.

`totalLines`
An `Integer` that contains the number of items to be displayed.

Instance Methods

`appendLine (String` *`text`*`)`
Creates a new `ANSILabelBox` object with the contents *text*, then adds the `ANSILabelBox` to the `items` list.

`cleanup (void)`
Deletes the buffers associated with the `ANSIListBoxPane` object and its items.

`handleInput (void)`
Waits for input from the user and processes it. Pressing a cursor key or an Emacs or vi next/previous line key shifts the selection. Pressing Escape or Enter causes the method to return.

`new (`*`listPane1`*`, `*`listPane2`*`, ...;)`
The `ANSIListBoxPane` constructor. The argument contains the names of one or more new `ANSIListBoxPane` objects.

`refresh (void)`
Draw the list pane and items on the terminal.

`refreshSelectionFirst (void)`
Highlight the initially selected item. Should only be called after a call to `refresh`.

`refreshSelection (void)`
Redraw the highlightd selected item, and un-highlight the previously selected item. Should only be called after a call to `selectNext` or `selectPrev`

`selectedText (void)`
Returns the text of the selected item as a `String` object.

`selectNext (void)`
`selectPrev (void)`
: Select the next or previous item of the list box's contents. Also saves the index and contents of the previously selected item.

3.68 `ANSIMessageBoxPane` Class

`ANSIMessageBox` objects present users with a pop-up dialog that contains a messages, and an `ANSIButtonPane` '`Ok`' button to close the widget.

Here is a simple example.

```
int main () {
  ANSIMessageBoxPane new messageBox;
  messageBox withText "Hello, world!";
  messageBox show 10, 10;
  messageBox cleanup;
}
```

Subclasses can always implement convenience methods to set input and output stream parameters if necessary.

You should also take care of the differences between xterms, consoles, and serial terminals. In particular, even though the `parent` method (class `ANSIWidgetPane`) See ⟨undefined⟩ [ANSITerminalStream], page ⟨undefined⟩.

Instance Variables

`okButton` An `ANSIButtonPane` object that contains the pane's '`Ok`' button.

`messageText`
: A `String` object that contains the text that appears in the message box.

Instance Methods

`cleanup (void)`
: Delete the receiver's extra data before deletion. The receiver objects themselves are deleted normally.

`new (String` *`message_box_name`*`)`
: Create one or more new `ANSIMessageBox` objects.

`withText (String` *`text`*`)`
: The argument is the text that will appear in the message box. The method adjusts the pane's dimensions to fit the text.

`show (int` *`x_origin`*`, int` *`y_origin`*`)`
: Display the receiver at *x_origin*, *y_origin*. If the receiver is displayed independently, the origin is relative to the upper left-hand corner of the display, or if the receiver is to be displayed over a parent pane, it is mapped to the parent pane with the origin relative the the parent pane's upper left-hand corner.

3.69 `ANSIProgressBarPane` Class

An `ANSIProgressBarPane` object displays a horizontal progress bar on a serial terminal or xterm. If you display a progress bar pane independently (using the `show` method), you can close the pane by pressing *`Enter`* or *`Esc`*.

The '`progress`' instance variable specifies how much of the progress bar is highlighted. It's better to use the `percent` method, though, which calculates the progress bar's hightlighted area and sets the percent legend.

As with all other ANSI pane widgets, you either display or omit the border and shadow decorations. Progress bars can also display a title string, which you can set using the `title` method, implemented in `ANSIWidgetPane` class. See ⟨undefined⟩ [ANSIWidgetPane], page ⟨undefined⟩.

Here is an example program that displays an `ANSIProgressBarPane` object.

```
int main () {
  ANSIProgressBarPane new progressBar;


  progressBar shadow = 1;
  progressBar border = 1;

  progressBar title "Progress";

  progressBar percent 65.0;

  progressBar show 2, 2;

  progressBar cleanup;
}
```

You can set the size of a progres bar using the `dimension` method. In that case, setting the progress bar using the `percent` method adjusts for the pane's width. The widget's display area is always drawn as one character row tall, however.

If you display the progress bar as a widget in an application, then you can update the progress bar on the screen using the `display` method instead of `show`, because `display` does not wait for user input before returning.

Instance Variables

`pctLabelMargin`
: An `Integer` that specifies where to draw the percent legend in the progress bar's content area.

`percentInt`
: An `Integer` that specifies percent of the progress bar's internal area that is highlighted. Setting this using the `percent` method, below, also adjusts the hightlight's dimensions for the progress bar's width. This variable is also used to display the text of the percent logo in the widget's content area.

`progress` An `Integer` that specifies the number of character columns to highlight in the progress bar's viewable area.

`viewHeight`
An `Integer` that specifies the height in character rows of the widget's viewable area. The progress bar highlight is always drawn as one character row in height.

`viewWidth`
An `Integer` that specifies the width in character rows of the widget's viewable area.

Instance Methods

`dimension (Integer width, Integer height)`
Set the width and height of the pane in character columns.

`show (Integer x, Integer y)`
Display the pane at row and column *x*,*y*, and return immediately.

`handleInput (void)`
Wait for the user's input. Pressing *`Esc`* or *`Enter`* closes the pane. and returns.

`new (String paneName)`
Create new `ANSIProgressBarPane` objects, one for each label given in the method's argument list.

`percent (Float percent)`
Set the percent of the progress bar's highlighted area. This method adjusts for the width of the progress bar and sets the text for the percent logo.

`refresh (void)`
Redraw the progress bar on the terminal.

`show (Integer x, Integer y)`
Display the pane at row and column *x*,*y*, and wait for the user's input.

3.70 `ANSIScrollingListBoxPane` Class

An `ANSIScrollingListBoxPane` object is similar to an `ANSIListBoxPane`; it displays a list of items in a text mode terminal or `xterm` and allows the user to use the terminal's *`Cursor-Up`*/*`Cursor-Down`* keys, Emacs compatible *`C-n`*/*`C-p`* keys, or vi compatible *`j`*/*`k`* keys to select an item in the list.

Pressing *`Enter`* or *`Escape`* closes the widget. An application can retrieve the selected item's text with the `selectedText` method, which is described below.

In addition, an `ANSIScrollingListBoxPane` object can scroll the list if the number of items is greater than the height of the widget's viewable area, in order to keep the selected item visible. The widget also displays a read-only scroll bar that indicates which portion of the list is visible.

```
int main () {
  ANSIScrollingListBoxPane new sListBox;
```

```
    sListBox enableExceptionTrace;

    sListBox withShadow;
    sListBox noBorder;    /* Not all terminals support line
                             drawing characters.           */

    sListBox appendLine "Item 1";
    sListBox appendLine "Item 2";
    sListBox appendLine "Item 3";
    sListBox appendLine "Item 4";
    sListBox appendLine "Item 5";
    sListBox appendLine "Item 6";
    sListBox appendLine "Item 7";
    sListBox appendLine "Item 8";
    sListBox appendLine "Item 9";
    sListBox appendLine "Item 10";
    sListBox appendLine "Item 11";
    sListBox appendLine "Item 12";
    sListBox appendLine "Item 13";
    sListBox appendLine "Item 14";
    sListBox appendLine "Item 15";
    sListBox appendLine "Item 16";
    sListBox appendLine "Item 17";
    sListBox appendLine "Item 18";
    sListBox appendLine "Item 19";
    sListBox appendLine "Item 20";
    sListBox appendLine "Item 21";
    sListBox appendLine "Item 22";
    sListBox appendLine "Item 23";
    sListBox appendLine "Item 24";
    sListBox appendLine "Item 25";
    sListBox appendLine "Item 26";
    sListBox appendLine "Item 27";
    sListBox appendLine "Item 28";
    sListBox appendLine "Item 29";
    sListBox appendLine "Item 30";
    sListBox appendLine "Item 31";
    sListBox appendLine "Item 32";

    sListBox show 5, 6;

    printf ("%s\n", sListBox selectedText);

    sListBox cleanup;
}
```

Instance Variables

`items` A `List` of the items that the receiver displays. See ⟨undefined⟩ [List], page ⟨undefined⟩.

`oldSelectedContent`
An `ANSILabelPane` object that contains previously selected item. See ⟨undefined⟩ [ANSILabelPane], page ⟨undefined⟩.

`prevSelectedLine`
An `Integer` that contains the index of the previously selected item. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`selectedContent`
An `ANSIScrollPane` object that draws the widget's scroll bar. See ⟨undefined⟩ [ANSIScrollPane], page ⟨undefined⟩.

`selectedContent`
An `ANSILabelPane` object that contains selected item. See ⟨undefined⟩ [ANSILabelPane], page ⟨undefined⟩.

`selectedLine`
An `Integer` that contains the index of the selected item. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`totalLines`
An `Integer` that contains total number of list items. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`viewStartLine`
An `Integer` that specifies which item begins the list's viewable portion. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`viewHeight`
An `Integer` that contains the height in text lines of the widget's visible area. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`viewWidth`
An `Integer` that contains the height in character columns of the widget's visible area. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

Instance Methods

`appendLine (String item_text)`
Adds *item_text* to the list's contents.

`cleanup (void)`
Deletes the display buffers associated with the list box and scroll bar.

`handleInput (void)`
Waits for the user's input. Changes the selected item when the user cursors through the list using the terminal's cursor keys, Emacs compatible *C-n*/*C-p*, or vi compatible *j*/*k*. Restores the terminal and returns when the user presses *Enter* or *Escape*.

`noBorder (void)`
`withBorder (void)`
: Set or unset the border for the main scroll pane and the scroll bar. These methods are equivalent to the following expressions.

```
/* To display borders. */
listPane border = 1;
listPane scrollBar border = 1;

/* To hide the borders. */
listPane border = 0;
listPane scrollBar border = 0;
```

Note that not all terninals support line drawing characters.

`noBorder (void)`
`withBorder (void)`
: Set or unset the shadow for the main scroll pane and the scroll bar. The methods are a shortcut for these statements.

```
/* To display shadows. */
listPane shadow = 1;
listPane scrollBar shadow = 1;

/* To hide the shadows. */
listPane shadow = 0;
listPane scrollBar shadow = 0;
```

`new (String` *`object_name`*`)`
: Constructs a new `ANSIScrollingListBoxPane` for each label given in the argument list. Sets the dimensions for the viewable areas, decorations, and creates the display buffers for the list box and the scroll bar.

`refresh (void)`
: Redraws the list box.

`refreshSelection (void)`
: Un-highlights the previously selected item and highlights the currently selected item.

`refreshSelectionFirst (void)`
: Highlights the currently selected item, which is normally the first item in the list when first drawing the widget–i.e., when there is no previously selected item.

`scrollThumbSize (void)`
: Sets the `scrollBar` instance variable's `thumbHeight` instance variable based on which portion of the list is visible. See ⟨undefined⟩ [ANSIScrollPane], page ⟨undefined⟩.

`scrollThumbStart (void)`
: Sets the `scrollBar` instance variable's `thumbStartLine` instance variable based on which portion of the list is visible. See ⟨undefined⟩ [ANSIScrollPane], page ⟨undefined⟩.

`selectedText (void)`
: Returns a `String` containing the text of the selected item. See ⟨undefined⟩ [String], page ⟨undefined⟩.

`selectNext (void)`
: Sets the next `selectedLine` and `selectedContent` instance variables to the next item in the list. If the selected item is already the last item in the list, the method does nothing.

`selectPrev (void)`
: Sets the next `selectedLine` and `selectedContent` instance variables to the previous item in the list. If the selected item is already the first item in the list, the method does nothing.

`show (Integer` *x*`, Integer` *y*`)`
: Displays the widget an character row and column *x*,*y* on the terminal display, then calls the `handleInput` method to process user input.

3.71 `ANSIScrollPane` Class

An `ANSIScrollPane` object draws a vertical scroll box on a text-mode terminal or X terminal. Applications can use the pane's instance variables to set the position and height of the scroll thumb, and the border and shadow decorations.

The class provides a `handleInput` method that allows users to move the scroll thumb independently if the pane is used by itself.

This app draws a scroll pane independently.

```
int main () {
  ANSIScrollPane new scrollBox;

  scrollBox shadow = 0;
  scrollBox border = 1;

  scrollBox thumbHeight = 3;    // Sets the position and height
  scrollBox thumbStartLine = 1; // of the scroll thumb.

  scrollBox show 2, 2;          // The Up and Down arrow keys
                                // move the scroll thumb while
                                // the pane is displayed.

  scrollBox cleanup;
}
```

When used to indicate the position of other panes, applications should set and read the scroll thumb's using the `thumbHeight` and `thumbStartLine` instance variables. If the application wants to use the `ANSIScrollPane` object to set another pane's position, it needs to handle the Up and Down cursor motion keys in the app's `handleInput` method,

In that case, the app should use the classes' `display` method instead of `show` to display the `ANSIScrollPane` object, because `display` doesn't wait for user input on its own.

There's no terminal independent way, however, to indicate that the `ANSIScrollPane` object has the input focus. An app might do this by either setting or omitting the pane's shadow, but that can occupy an extra line of terminal space.

The class is simplified by keeping the width of the scroll thumb at one column. Apps can draw a wider scroll pane, but that does not affect the scroll thumb's width.

Instance Variables

`viewHeight`
: An `Integer` that specifies the height of the scroll channel in character rows.

`viewWidth`
: An `Integer` that specifies the width of the scroll channel in character columns.

`thumbHeight`
: An `Integer` that specifies the height of the scroll thumb in character rows.

`thumbStartLine`
: An `Integer` that specifies the row of the top of the scroll thumb. This should be less than the `viewHeight` instance variable for the thumb to be visible.

Instance Methods

`cleanup (void)`
: Delete the buffers associated with the pane object.

`dimension (Integer` *`width`*`, Integer` *`height`*`)`
: Set the width and height of the receiver pane. Also adjusts the width of the `viewWidth` and `viewHeight` instance variables.

`display (Integer` *`x`*`, Integer` *`y`*`)`
: Draw the pane on the terminal at row and column *x*,*y*, and return immediately.

`handleInput (void)`
: Waits for user input from the pane's `ANSITerminalStream` object. Pressing Enter or Escape withdraws the pane from the display and returns. Pressing the Up or Down arrow keys moves the scroll thumb.

`new (String` *`paneName`*`)`
: Create a new `ANSIScrollPane` object, with the name *paneName*. If more than one name is given in the argument list, create new `ANSIScrollPane` objects with the arguments' names.

`refresh (void)`
: Draw the pane on the terminal.

`show (Integer x, Integer y)`
: Draw the pane on the terminal at row and column *x*,*y*, and wait for the user's input.

3.72 `ANSITextBoxPane` Class

An `ANSITextBoxPane` object displays a text-mode dialog box with an application's text in the window, and a "Dismiss" button at the bottom of the display area. Like other widgets that are subclasses of `ANSIWidgetPane`, you can close the window by pressing *`Esc`* or *`Enter`*, and you can scroll through the text with the terminal's up and down arrow keys, the `vi` keys *`j`* and *`k`*, and the `emacs` keys *`C-n`* and *`C-p`*.

Here is a brief example of how to open and display text in an `ANSITextBoxPane` object.

```
int main () {
  ANSITextBoxPane new textBox;

  textBox resize 75, 30;

  textBox appendLine "Hello, world!";
  textBox appendLine "This is another line of text.";
  textBox appendLine "More text to follow.";

  textBox show 1, 1;

  textBox cleanup;
}
```

Instance Variables

`dismissButton`
: An `ANSIButtonPane` object that displays a "Dismiss" button at the bottom of the pane's display area.

`text`
: A `List` object that contains the text to be displayed in the pane, one line per list element.

`viewStartLine`
: An `Integer` object that indicates the topmost line of text to be displayed in the pane.

`viewHeight`
: An `Integer` that contains the height of the pane's text display area. The text display area is the width and height of the pane, not including a window border if any, and not including the bottom five lines of the pane, which is used to display the `dismissButton` widget.

`viewWidth`
: An `Integer` that contains the width of the viewable text area. As mentioned above, the `viewWidth` dimension is the width of the pane minus the window borders, if any.

`viewXOffSet`
: An `Integer` that contains the starting column of each line within the window.

Instance Methods

`appendLine (String text)`
: Append a line to the widget's `text` (class `List`) instance variable. The text will be visible after the next `refresh` message.

`cleanup (Integer lineNumber)`
: Delete the extra buffers that the receiver uses for screen data. The normal object cleanup routines delete the receiver pane itself.

`clearLine (Integer lineNumber)`
: Erase the line *lineNumber* in the pane's view area.

`handleInput (void)`
: Wait for the user's input from the keyboard and redisplay or withdraw the receiver widget depending on which key the user presses.

`new (String paneName)`
: Constructs a new `ANSITextBoxPane` object. The object's dimensions are 40 columns wide by 20 rows high, with a "Dismiss" button at the bottom of the window, and with a border and shadow.

  If more that one name is given in the argument list, construct new `ANSITextBoxPane` objects with the labels' names.

`refresh (void)`
: Redraws the receiver object and any text to be displayed in the pane's visible area.

`resize (Integer xSize, Integer ySize)`
: Resize the pane to the dimensions *xSize*, *ySize*.

`show (Integer xOrigin, Integer yOrigin)`
: Pop up the pane's window at the terminal coordinates *xOrigin*, *yOrigin*, and wait for the user's input.

3.73 `ANSITextEntryPane` Class

A `ANSITextEntryPane` object prompts the user for text input and returns the input to the application program. Like other subclasses of `ANSIWidgetPane`, this class uses the methods of that class or re-implements them as necessary. See ⟨undefined⟩ [ANSIWidgetPane], page ⟨undefined⟩.

The widget can be displayed independently; that is, it can be popped up on its own, as in this example.

```
int main () {
  ANSITextEntryPane new textEntry;
  String new inputText;

  textEntry withPrompt "Please enter some text: ";
```

```
  inputText = textEntry show 10, 10;
  printf ("\nYou typed: %s\n", inputText);
  textEntry cleanup;
}
```

To pop up an `ANSITextEntryPane` over another pane, the program must also configure and define the widget's parent pane.

```
int main () {
  ANSITerminalPane new mainPane;
  ANSITextEntryPane new textEntry;
  String new inputText;

  mainPane initialize 1, 1, 80, 24;
  mainPane refresh;

  mainPane gotoXY 29, 10;
  mainPane printOn "Parent Pane";
  mainPane gotoXY 25, 11;
  mainPane printOn "Please press [Enter].";
  mainPane refresh;

  getchar ();  /* Actual apps should use getCh, etc. */

  textEntry parent mainPane;
  textEntry withPrompt "Please enter some text: ";
  inputText = textEntry show 10, 10;
  mainPane refresh;

  printf ("\nYou typed: %s\n", inputText);

  getchar ();

  textEntry cleanup;
  mainPane cleanup;
}
```

Instance Variables

`promptText`
: The text of the entry pane's prompt. The default is an empty `String` object.

`inputBuffer`
: A `String` object that contains the user's input.

`inputLength`
: The width in characters of the text entry. The default is 20.

Instance Methods

`handleInput (void)`
: Process `InputEvent` objects from the receiver's `paneStream` input handle.

`inputWidth (Integer` *`width`*`)`
: Set the width, in characters, of the input entry box. The default is 20.

`new (String` *`paneName`*`)`
: Creates a new `ANSITextEntryPane` object. Also uses the `withShadow` and `withBorder` messages from `ANSITerminalPane` class, and the `openInputQueue` message from `ANSITerminalStream` class.

  If more that one argument is given in the argument list, create new `ANSITextEntryPane` objects with the arguments' names.

`show (int` *`x_origin`*`, int` *`y_origin`*`)`
: Display the receiver pane and return input from the user.

`withdraw`
: Remove the receiver widget from the display. If the widget is drawn over another pane object, unmap the receiver from the parent pane. If the receiver is displayed independently, clear the display before returning.

`withPrompt (String` *`promptText`*`)`
: Set the receiver's prompt to *promptText*.

3.74 `ANSIYesNoBoxPane` Class

An `ANSIYesNoBoxPane` object presents the user with a dialog that contains a text message and waits for the user's '`Yes`' or '`No`' response.

Here is an example of opening an `ANSIYesNoBoxPane` object using standard input and output (e.g., when displaying the pane on a xterm).

```
int main () {
  ANSIYesNoBoxPane new messageBox;
  String new answer;

  messageBox withText "Do you want to quit?";
  answer = messageBox show 10, 10;
  messageBox cleanup;
  printf ("You answered, \"%s\"\n", answer);
}
```

Here is an example of opening an `ANSIYesNoBoxPane` object in a serial terminal (for a Linux serial device). For other systems, change the '`/dev/ttyS1`' argument to the device node that connects to the serial terminal. It's necessary to adjust the arguments to `setTty` to match the terminal's settings.

```
int main () {
  ANSIYesNoBoxPane new messageBox;
  String new answer;

  messageBox paneStream openOn "/dev/ttyS1";  /* Linux serial device. */
```

```
    messageBox paneStream setTty 9600, 1, 'n', 8;

    messageBox noBorder;  /* Not all terminals support line drawing characters. */

    messageBox withText "Are you sure you want to quit?";
    answer = messageBox show 10, 10;
    messageBox cleanup;
    printf ("You answered, \"%s\"\n", answer);
}
```

As with any dialog widget, pressing *Esc* or *Return* closes the ANSIYesNoBoxPane object. The *Tab* key selects between the "Yes" and "No" buttons, as do the *Y* and *N* keys,

Instance Variables

button1 The ANSIButtonPane widget that controls the 'Yes' response.

button2 The ANSIButtonPane widget that controls the 'No' response. (ANSIYesNoBoxPane class)

messageText
A String object that contains the text displayed in the pane.

Instance Methods

cleanup (void)
Delete the widget's data before exiting.

getFocusWidgetText (void)
Return the text associated with the button that has the input focus.

handleInput (void)
Wait for the user's input and return the response from the widget.

new (*newPaneName*
Create a new ANSIYesNoBox object with the name given as an argument. If the argument list contains more than one name, create ANSIYesNoBoxPane objects for each argument.

```
ANSIYesNoBoxPane new yesnobox1, yesnobox2;
```

nextFocus (void)
Set the input focus to the next button widget.

noBorder (void)
withBorder (void)
Set or unset the border for the main pane and the button labels. These methods are equivalent to the following expressions.

```
/* To display borders. */
yesnoBox border = 1;
```

```
yesnoBox button1 border = 1;
yesnoBox button2 border = 1;

/* To hide the borders. */
yesnoBox border = 0;
yesnoBox button1 border = 0;
yesnoBox button2 border = 0;
```

Note that not all terninals support line drawing characters.

`noBorder (void)`
`withBorder (void)`
Set or unset the shadow for the main pane and the buttons. The methods are a shortcut for these statements.

```
/* To display shadows. */
yesnoBox shadow = 1;
yesnoBox button1 shadow = 1;
yesnoBox button2 shadow = 1;

/* To hide the shadows. */
yesnoBox shadow = 0;
yesnoBox button1 shadow = 0;
yesnoBox button2 shadow = 0;
```

`show (int x_origin, int y_origin)`
Display the `ANSIYesNoBoxPane` object at *x_origin*, *y_origin*. If displayed over another pane, the origin is relative to the parent pane's origin. If displayed independently, the origin is relative the the upper left-hand corner of the terminal.

`withText (char *text)`
Defines the text that is to appear within the pane. This method adjust's the pane's size to fit the text.

3.75 `X11Pane` Class

The `X11Pane` class provides the basic methods and instance variables for creating and displaying a window on a X display.

The `X11Pane` class does not, itself, provide methods for moving, resizing, or handling input or changing focus.

The `X11TerminalStream` class handles X input events. The `X11Pane` constructor `new` also creates a `X11TerminalStream` object in the `X11Pane's inputStream` instance variable. There is a short example program in the `X11TerminalStream` section. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

This class uses the default visual or a X window without buffering, so applications need to be careful to handle InputEvents correctly, or graphics drawing may result in flicker,

depending how the window manager handles X Window System events. See the *Ctalk Tutorial* for information about how to use `InputEvent` objects in applications.

However, `X11Pane` class provides the address of the window's graphics context in the `xGC` instance variable, so applications that use this class for graphical displays can create and use their own visuals if necessary.

Instance Variables

`backgroundColor`

A `String` object that contains the name of the window's background color. The value is the background color of the window, independent of the background color of any buffers used by subpanes. Normally this value is set by the `background` method, described below.

Note that if a subclass of `X11Pane` has set a '`backgroundColor`' resource, then that color overrides this variable's value when a program creates subpane windows. The window creation generally happens when attaching a pane object to its parent pane, with an `attachTo` method.

If unset, the default background color is black.

`borderWidth`

An `Integer` object that contains the window's border width in pixels. The default is 1 pixel.

`container`

A `Symbol` that refers to a pane's container (i.e., parent) Pane object. For top-level `X11Pane` objects this value should be `NULL`.

`depth` An `Integer` that contains the default depth of the display screen. Ctalk sets this value when creating the window of a `X11Pane` object or an instance of one of `X11Pane's` subclasses. Normally applications should not need to change this value.

`displayPtr`

A `Symbol` that holds the pointer to the display connection; i.e., the `Display *` returned by *XOpenDisplay(3)*. Most windows use the display connection opened when the main window is created. Dialogs, which create their own main windows, open their own connection to the display. Generally, programs should not need to change this.

When using subpanes, it's convenient to set the subpane's `displayPtr` variable to the value of the main window pane's value. This is generally done in the `attachTo` methods, with a line that looks something like this.

`haveXft` A `Boolean` that has the value of '`true`' if Ctalk is built with the Xft and Fontconfig libraries, and they have been initialized with a call to `X11FreeTypeFont : initFontLib`.

`modal` A `Boolean` that determines how Ctalk draws in the window. True for popup windows, false otherwise.

```
self displayPtr = self mainWindow displayPtr;
```

`fontVar` A `X11Font` object that contains information about the Window or Pane's current font.

`fontDesc` A `String` that contains a X Logical Font Descriptor for the font to be used by the window. If the value is '`(null)`', the window uses the system's fixed font.

`foregroundColor`
A `String` object that contains the name of the window's foreground color. If unset, the default foreground color is black.

`ftFontVar`
An `X11FreeTypeFont` object that contains the pane's current font, if the `X11TextEditorPane` object uses outline fonts.

If, on the other hand, the pane uses X11 bitmap fonts, the Pane uses the `fontVar` instance variable. Which requires no additional initialization.

To determine whether a machine has outline fonts available, a program could use a set of statements like this in its initialization.

```
Boolean new useXFonts;
X11Pane new myPane;

...

if (myPane ftFontVar version >= 10) {
  myPane ftFontVar initFontLib;
  useXFonts = false;
} else {
  useXFonts = true;
}
```

`inputStream`
A `X11TerminalStream` object that provides `InputEvent` objects to the application. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`resources`
This is an `AssociativeArray` that an object can use to store any data that determine its run-time appearance, like colors, dimensions, or options. Generally a `Pane` class sets default resources in its `new` method, which application programs can then update with its own resources after the `Pane` object has been constructed.

As a brief example, here's a sample initialization from the `X11MessageBoxPane` : `new` method.

```
paneName resources atPut "backgroundColor", "gray";
paneName resources atPut "pad", 10;
paneName resources atPut "messageText", "Your\nMessage\nHere";
```

```
paneName resources atPut "buttonText", "Ok";
```

Then, when an application program constructs the widgets when it is run, it can use a line like this.

```
l_label background self resources at "backgroundColor";
l_label multiLine self resources at "messageText";
```

Remember that the `Collection : atPut` method does not check for an existing key, so be sure to remove the old value first. This is done with the `Collection : removeAt` method, or simply with the `Collection : replaceAt` method.

```
if (l_label resources keyExists "backgroundColor")
  l_label resources removeAt "backgroundColor";
l_label resources atPut "backgroundColor", "blue";
```

or,

```
if (!l_label resources replaceAt "backgroundColor", "blue")
  l_label resources atPut "backgroundColor", "blue";
```

Refer to the `Collection` class section for more information about the methods it defines. See ⟨undefined⟩ [Collection], page ⟨undefined⟩.

`xGC` A `Symbol` that contains the address of the window's graphics context. The `X11Pane` class does not, at this time, provide methods or instance data for drawables, so the application needs to implement its own drawable for each window.

`xWindowID`
An `Integer` that contains the window's id that a program can use with X library functions.

Class Variables

`xColormap`
An `Integer` that contains the resource ID of the default color map.

Instance Methods

`attachTo (Object` *parentPane*`)`
Attach the receiver to *parentPane*. The value of *parentPane*'s *subWidget* instance variable is the receiver.

`background (String` *color_name*`)`
Sets the background color of the pane object's window. For buffered Panes, programs need to set the background color of that Pane independently. This method also has the effect of clearing the window.

If a program doesn't set a background color for the window, the default background color is white.

`clearRectangle (int x, int y, int width, int height)`
Clear the area of the receiver's window with the dimensions given as the arguments.

`clearWindow (void)`
Clear the receiver's window. This method clears only the main window and does not affect any image buffers associated with the pane object. To also clear the pane object's image buffers, use `clearRectangle`, above.

`defaultCursor (void)`
Restore the window's default cursor, which is normally the cursor of the parent window.

`deleteAndClose (void)`
Delete the receiver window and close the display.

`displayHeight (void)`
Returns an `Integer` with the display's height in pixels.

`displayWidth (void)`
Returns an `Integer` with the display's width in pixels.

`faceRegular (void)`
`faceBold (void)`
`faceItalic (void)`
`faceBoldItalic (void)`
These methods select which `X11FreeTypeFont` typeface a pane should use. These methods need the program to initialize the Xft libraries (with the `initFontLib` method in `X11FreeTypeFont` class), and the pane has established a connection to the X server (with the `openEventStream` method).

`font (String font_desc)`
Set the X font used for drawing on the window. This font can be set independently of the fonts used by subpanes. See ⟨undefined⟩ [X11TextPane], page ⟨undefined⟩, and See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

Because the Ctalk libraries use shared memory to manage font information, it is generally necessary to call this method after the `openEventStream` method, for the program to calculate the character spacing of multiple typefaces correctly.

`foreground (String colorName)`
Sets the window's foreground color (the default color for drawing on the window) to *colorName*. If a program doesn't set the window's foreground color, the default color is black.

`ftFont (String family, Integer slant,`
`Integer` *weight* `Integer` *dpi*, `Float` *pointSize*) Selects an outline font for use by the `X11Pane` object. The method selects the font, and fills in the `ftFontVar` instance variable with the font's information. Programs should first determine if FreeType fonts are available on the system, by using the `X11FreeTypeFont` method `version` first, as in this example.

```
Boolean new useXFonts;
X11Pane new myPane;

...

if (myPane ftFontVar version >= 10) {
  myPane ftFontVar initFontLib;
  useXFonts = false;
} else {
  useXFonts = true;
}

...

/* Selects the font DejaVu Sans Mono, regular slant, normal weight,
   72 dpi, 12 points. */
myPane ftFont "DejaVu Sans Mono", 0, 80, 72, 12.0;
```

The `X11FreeTypeFont` class also provides methods that use Xft and X11 font descriptors to select fonts. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`initialize (int` *`width`*`, int` *`height`*`)`
`initialize (int` *`x`*`, int` *`y`*`, int` *`width`*`, int` *`height`*`)`

Create the window and its graphics context with the width and height given as the arguments. This method also opens a connection to the X server if necessary. This method uses the window system to set the window's initial position.

Create the window and its graphics context with the dimensions given as the arguments. This method also opens the connection to the X server if necessary.

This method is here for older programs, or programs that set the window position directly. Otherwise, use the form of `initialize`, below, that takes the window's geometry flags as an argument, and only set the window's position if the user provides one on the command line and the program retrieves it with `parseX11Geometry`.

Even more simply, a program can set *x_org* and *y_org* to zero, and let the window system handle any positioning. Or use the form of `initialize` that uses only the *width* and *height* arguments.

`initialize (int` *`x`*`, int` *`y`*`, int` *`width`*`, int` *`height`*`, int` *`geom_flags`*`)`

Create the window and its graphics context with the dimensions given as the arguments. Like the other forms of `initialize`, this method also opens the connection to the X server if necessary.

The *geom_flags* argument provides placement hints for the window's initial position. It has the format provided by the `parseX11Geometry` method in `Application` class. See ⟨undefined⟩ [parseX11Geometry], page ⟨undefined⟩.

The *x* and *y* parameters can also be given directly if the program sets the window position itself. If these arguments are zero, then the window manager or the user supplied window geometry determine the window placement.

`initialize (int x, int y, int width, int height, int geom_flags, char *win_title)`
: This method is similar to the five-argument form of `initialize`, and additionally sets the window's title using the contents of the string *win_title*.

`isTopLevel (void)`
: Return `TRUE` if the receiver's container pane is `NULL`.

`mainWindow (void)`
: Returns the `X11Pane` object that manages a program's main window. The method retrieves the main window object by following the references of each subpane's `container` instance variable. If the top-level pane object is not a `X11Pane`, the method prints a warning.

`map (void)`
: Map the receiver window onto the X display.

`openEventStream (void)`
: Open the window's input stream, a `X11TerminalStream` object. The `X11TerminalStream` section of the manual describes X input event handling. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`putStrXY (Integer xOrg, Integer yOrg, String str)`
: Draw the string on the receiver's drawable surface at *xOrg*, *yOrg*, using the selected font.

`putStrXY (Integer xOrg, Integer yOrg, String str)`
: Draw the string on the receiver's drawable surface at *xOrg*, *yOrg*. Currently this method is the same as `putStrXY,` above.

`raiseWindow (void)`
: Display the receiver window above other windows on the X display. Note that the method's name was changed from `raise` to avoid warnings when including the C *raise(3)* function.

`setResources (String resourceName, String resourceClass)`
: Set the resource name and class of the main window. Normally the resource name is the name of the application, which may vary from the window title. The resource class should be used to identify the window for X resources.

  A program can call this method either before or after connecting to the server with the `openEventStream` method. Generally, if the resources affect the appearance of decorations provided by the system, like the window frame or icon, the window needs to be remapped for the changes to be visible, but this may vary with the application and the type of window manager.

`setWMTitle (char *title)`
: Set the window's title. Because setting a X window's title requires communication between the window and the display server, this method requires that

the window is first mapped and raised (with the `map` and `raiseWindow` methods, above), and has a connection to the display server (which is done with the `openEventStream` method, also above).

In other words, when setting a window's title for the first time, this method works best when used just before processing any other events.

`subPaneNotify (InputEvent event)`
: Called by applications that need to invoke sub-pane handlers in response to window events. For examples of its usage, refer to the section for `X11CanvasPane`, See ⟨undefined⟩ [X11CanvasPane], page ⟨undefined⟩, and other `X11Pane` subclasses.

`useCursor (Cursor cursor_object)`
: Display the X11 cursor defined by *cursor_object*, a `X11Cursor` object, in the receiver's window. To create cursors, See ⟨undefined⟩ [X11Cursor], page ⟨undefined⟩.

`useXRender (Boolean b)`
: If *b* is true, draw graphics using the X Render extension if it is available. If *b* is false, use Xlib for graphics drawing. The default is to draw using the X Render extension if it is available.

`usingXRender (void)`
: Returns a `Boolean` value of True if the program is using the X Render extension for drawing, False otherwise.

3.76 `GLXCanvasPane` Class

The `GLXCanvasPane` class displays 3 dimensional graphics drawn with OpenGL in a X window. The class requires that the display server supports GLX visuals, which is the case with most modern X server installations.

The class includes instance variables that select GLX visual properties, methods that create and display the window, and provides a simple API for the window's application program.

There is an example program that displays a `GLXCanvasPane` window at the end of this section. See ⟨undefined⟩ [GLXExampleProgram], page ⟨undefined⟩.

GLXCanvasPane Applications

The `GLXCanvasPane` class provides a simple application framework that is compatible with OpenGL's single threading model. The framework consists of a number of callback methods and a `run` method, that are configured when the program starts.

The methods that install callbacks are:

```
onButtonPress
onIdle
onKeyPress
onExpose
onPointerMotion
onResize
```

```
onTimerTick
onAnimation
```

There is a complete description of each of these methods in the section, *Instance Methods*.

Typically, a program installs its callback methods and initializes the OpenGL system, and then calls the method `run` to begin the program's event loop. The example program given at the end of this section follows this process. The program's initialization and startup, which is contained in *main ()*, is shown here.

```
int main () {
  GLXCanvasPane new pane;

  pane initialize (1, 150, 500, 500);
  pane title "GLXCanvasPane Demonstration";
  pane map;
  pane raiseWindow;

  pane onKeyPress "myKeyPressMethod";
  pane onExpose "myExposeMethod";
  pane onTimerTick "myTimerTickHandler";
  pane onResize "myResizeMethod";

  pane initGL;

  pane run;

}
```

Selecting GLX Visuals

The `GLXCanvasPane` class selects visuals based on the values of many of the instance variables. These instance variables correspond with the attributes recognized by the *glXChooseVisual(3)* library call.

The instance variables' default settings select a double buffered, TrueColor or DirectColor visual, with 24 color bits per pixel, and a stencil buffer with 8 bit planes, which is supported by many common GLX servers. The equivalent C code for these attributes, formatted as an argument for *glXChooseVisual(3)*, would be:

```
static GLint att[] = {GLX_RGBA, GLX_DEPTH_SIZE, 24, GLX_STENCIL_SIZE, 8,
                      GLX_DOUBLEBUFFER, None};
```

These attributes also correspond to the available attributes output by a program like *glxinfo(1)*. Refer to the *glXChooseVisual(3)* and *glxinfo(1)* manual pages for more information

Drawing with X Fonts

The `GLXCanvasPane` class defines three methods, `useXFont`, `drawText`, `drawTextW`, and `freeXFont` that facilitate drawing text with X fonts.

Typically, a program calls `useXFont` with the name of the X font when it initializes OpenGL (that is, after first creating the window and mapping the GLX context to the display), then using `drawText` to draw the text in the program's drawing routine. Finally, the program calls `freeXFont` to release the font data before exiting, or when changing fonts if the program uses multiple fonts for drawing.

```
myPane useXFont "fixed";    /* Call during OpenGL initialization. */
                            /* The argument, "fixed," is the name */
                            /* of the font to be used.            */

...

                            /* Called from within the program's   */
                            /* drawing routine.                   */
myPane drawText "Text to appear in the window";


...

myPane freeXFont;           /* Called during program cleanup or   */
                            /* before calling useXFont again to   */
                            /* draw text in a different font.     */
```

The Ctalk distribution contains an example program in the `demos/glx` subdirectory, `xfont.ca` that demonstrates this drawing method.

Drawing with FreeType Fonts

The methods to draw text using the freetype libraries are similar to those that render X bitmap fonts. The Freetype libaries support text rendering using Freetype, Truetype, and Postscript Type1 fonts.

The main differences are that, because of the way the fonts are rendered on the screen, their measurements are given in the coordinates of the current viewing and transformation matrices.

In addition, when loading a font using `useFTFont`, the method uses the path name of the font file, not an identifier. This is the only interface that the Freetype libraries use. To use the system's font caching, refer to See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

There is a demo program that renders Freetype fonts in the `demos/glx` subdirectory of the Ctalk source package, `ftfont.ca`.

Display Synchronization

On OpenGL releases that support synchronization, `GLXCanvasPane` applications can synchronize buffer swapping with the video display's refresh rate. `GLXCanvasPane` class

provides the methods `syncSwap,refreshRate,` and `frameRate` which allow programs to adjust buffer swapping to match the video refresh rate. The demonstration program, `demos/glx/glxchaser.ca` provides an example of how to use these methods.

To find out which GLX extensions the display server supports, the `extensions` method, below, returns the extensions as a `String` object.

Currently, video syncronization support is limited to MESA releases that provide the `GLX_MESA_swap_control` and `GLX_OML_sync_control` extensions.

Instance Variables

`animationHandler`
: Defines the callback method that is called 24 times a second to perform animation.

`buttonPressHandler`
: A `Method` object that defines the callback method that is executed when a mouse button is pressed.

`buttonState`
: An `Integer` that records whether a mouse button is currently pressed. The class defines macro constants to record the states, and you can include these definitions in your programs to interpret the value of `buttonState`.

```
#define buttonStateButton1 (1 << 0)
#define buttonStateButton2 (1 << 1)
#define buttonStateButton3 (1 << 2)
#define buttonStateButton4 (1 << 3)
#define buttonStateButton5 (1 << 4)
```

So to check if the mouse button 1 is pressed, the program could contain an expression like the following.

```
if (myPane buttonState & buttonStateButton1) {
   printf ("Button 1 pressed.\n");
}
```

`colormap` An `Integer` that contains the X resource ID of the default colormap of the current display and screen.

`displayPtr`
: A `Symbol` that contains the address of the X server's display handle as provided to the application. The `displayPtr` instance variable is filled in by the `initialize` methods.

`exposeHandler`
: Defines the method that is called each time the program's window receives an Expose event from the display. This handler is essential to displaying the window in coordination with other display events. If this variable is not initialized, then the `run` method calls the `swapBuffers` method.

`glxContextPtr`
: A `Symbol` that contains the address of the `GLXContext` that is current for the `GLXCanvasPane's` window. This variable is normally filled in by the `map` method.

`idleHandler`
: A `Method` that contains the callback that the program executes when not processing events from the display.

`keyPressHandler`
: A `Method` that handles the KeyPress events that the display sends to the program's window. This variable should be set during program initialization using the `onKeyPress` method before the program starts the `run` method.

`pointerMotionHandler`
: A `Method` that is called whenever the window receives a MotionNotify event.

`resizeHandler`
: A `Method` that is called whenever the window receives a ConfigureNotify event. The variable should be set using the `onResize` method before the program starts the `run` method.

`shiftState`
: An `Integer` that records whether the any of the Shift, Control, or Alt keys is currently pressed. The class defines macro constants to record the states, and you should also include the definitions in your program if it needs to monitor the state of the modifier keys.

```
#define shiftStateShift (1 << 0)
#define shiftStateCtrl  (1 << 1)
#define shiftStateAlt   (1 << 2)
```

So, for example, to test whether a Control key is pressed, you can use an expression like the following.

```
if (myPane shiftState & shiftStateCtrl)
   printf ("Control key pressed.\n");
```

`timerMSec`
: An `Integer` that defines the time in milliseconds between `onTimerTick` handler calls. The default is 1 millisecond.

`timerTickHandler`
: A `Method` that defines the callback that is executed when the classes' interval timer reaches 0.

`visualAuxBuffers`
: An `Integer` that, in combination with `visualSetAuxBuffers`, defines the minimum number of auxiliary buffers that the selected visual must have.

`visualBufferSize`
: An `Integer` that defines the desired color index buffer size. The instance variable `visualSetBufferSize` must also be set to true.

`visualDepthSize`
: An `Integer` that contains the size of the visual's depth buffer. The `visualSetDepthSize` instance variable must also be true for this value to take effect.

`visualDoubleBuffer`
: A `Boolean` that selects a double-buffered GLX visual if true, or a single-buffered visual if false.

`visualInfoPtr`
: A `Symbol` that contains the address of a X visual selected when the pane's window is created, which normally happens when a program calls one of the `initialize` methods described below.

`visualRedSize`
`visualGreenSize`
`visualBlueSize`
`visualAlphaSize`
: `Integer` values that, if greater than zero, try to select thea largest buffer for that color channel of at least the specified size. If one of the values is zero, then *glXChooseVisual(3)* tries to select the smallest available buffer for that color channel.

`visualRedAccumSize`
`visualGreenAccumSize`
`visualBlueAccumSize`
`visualAlphaAccumSize`
: `Integer` values that, if greater than zero, try to select thea largest accumulator buffer for that color channel of at least the specified size. If one of the values is zero, then *glXChooseVisual(3)* tries to select a visual with no accumulator buffer for that color channel.

`visualRGBA`
: A `Boolean` that selects a TrueColor or DirectColor visual if true, or a PseudoColor or StaticColor visual if false. Also, the `visualSetBufferSize` and `visualBufferSize` instance variables are ignored when this variable is true.

`visualStencilPlanes`
: An `Integer` the selects the number of stencil bitplanes if greater than zero. If zero, then a visual with no stencil buffer is selected if possible.

`visualStereo`
: A `Boolean` value that selects a stereo visual if true.

`xLineHeight`
: An `Integer` that contains the line height in pixels of a font that has been selected by the `useXFont` method. This variable is read only.

`xMaxCharWidth`
: An `Integer` that contains the maximum width in pixels of a character for a X font that has been selected by `useXFont`. This value is read only.

Instance Methods

`alpha (Float alpha)`
: Sets the alpha channel (opacity) when rendering outline fonts. Values should be between 0.0 (transparent) and 1.0 (opaque). The Ctalk library's default value is 1.0. Calling this method also sets the value of the receiver pane's `ftFontVar fgAlpha` instance variable.

`deleteAndClose (void)`
: Releases the receiver pane's GLX context and deletes the X11 window, and shuts down the application's X11 input client.

`displayHeigth (void)`
`displayWidth (void)`
: These methods return an `Integer` with the display height and width in pixels, respectively.

`drawFmtText (Float xOrg, Float yOrg, String fmt, ...)`
: Draws the text given by *fmt* and its arguments at the matrix position given by *xOrg*,*yOrg*.

`drawFmtTextFT (Float xOrg, Float yOrg, String fmt, ...)`
: Display the string given by *fmt* and its arguments at the matrix coordinates *xOrg*, *yOrg* in the currently selected Freetype font. This call, like all calls that render text, should be preceded by a call to `useFTFont`.

`drawFmtTextW (Integer xOrg, Integer yOrg, String fmt, ...)`
: Draws the formatted text of *fmt* and its arguments at the pixel position given by *xOrg*, *yOrg*. OpenGL uses the lower left-hand corner of the window as the origin for pixel coordinates.

`drawText (Float xOrg, Float yOrg, String text)`
`drawText (Float xOrg, Float yOrg, Float red, Float green, Float blue, String text)`
: Draws *text* at the matrix position given by *xOrg*,*yOrg*.

  The program must have registered a X font for drawing with a previous call to the `useXFont` method.

  If the *red*, *green*, and *blue* arguments are given, the method draws the text in that color. Otherwise, the method (via OpenGL) draws the text using the last OpenGL color setting.

`drawTextFT (Float xOrg, Float yOrg, String text)`
`drawTextFT (Float xOrg, Float yOrg, Float red, Float green, Float blue, Float alpha, String text)`
: Draws the string given by *text* at the matrix coordinates *xOrg*, *yOrg*.

  The *red, green, blue,* and *alpha* arguments, if used, should be between the values of 0.0 and 1.0, so they can be passed along to the OpenGL API directly,

and also to set the receiver's `ftFontVar` instance variable (a `X11FreeTypeFont` values for its instance variables: `fgRed`, `fgGreen`, `fgBlue`, and `fgAlpha` See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`drawTextW (Float` *`xOrg`*`, Float` *`yOrg`*`, String` *`text`*`)`
`drawTextW (Float` *`xOrg`*`, Float` *`yOrg`*`, Float` *`red`*`, Float` *`green`*`, Float` *`blue`*`, String` *`text`*`)`

Draws *text* using the window's *xOrg*,*yOrg* pixel as the origin.

If the *red*, *green*, and *blue* arguments are given, the method draws the text in that color. Otherwise, the method (via OpenGL) draws the text using the last OpenGL color setting.

This method allows text to be positioned relative to the window's pixels, which avoids the need for programs to translate a matrix position into a pixel position manually. This allows a program to position text more easily when it is necessary to measure spaces using the dimensions of the text and font that are being displayed.

The coordinates' origin (0,0) is at the lower left-hand corner of the window, and the pixel coordinates increase as the position moves toward the top and right of the window.

The GL function *glWindowPos2i*, which this method uses internally, is an extension in many GL implementations. Ctalk checks for the function when building the Ctalk libraries. If *glWindowPos2i* is not present in the machine's GL libraries, then programs that try to use these methods display an error message on the terminal and exit.

`extensions (void)`

Returns the supported GLX extensions from the display server as a `String` object.

`frameRate (void)`

Returns the rate that the program updates the display, in frames per second. The algorithm that calculates the frame rate measures frames over a five-second interval.

`freeFTFont (void)`

Releases the Freetype font in use.

`freeXFont (`*`void`*`)`

Frees the font data that was allocated by a previous call to `useXFont`. Programs should call this method when cleaning up before program exit, or when switching fonts by a subsequent call to `useXFont`.

`initialize (Integer` *`x`*`, Integer` *`y`*`, Integer` *`width`*`, Integer` *`height`*`, Integer` *`geomFlags`*`)`
`initialize (Integer` *`x`*`, Integer` *`y`*`, Integer` *`width`*`, Integer` *`height`*`)`
`initialize (Integer` *`width`*`, Integer` *`height`*

Creates the receiver pane's window and configures the window for display. The `initialize` method also fills in the receiver's `visualInfoPtr` instance variable with a pointer the X visual info structure specified by the receiver, which is provided by the receiver's instance variables.

With two arguments, the method initializes the receiver window with the width and height given as arguments.

With four arguments, the method initializes the receiver window with the window's x and y origin and the width and height given as arguments.

With five arguments, the *geom_flags* argument provides placement hints for the window's initial position. It has the format provided by the `parseX11Geometry` method in `Application` class. See ⟨undefined⟩ [parseX11Geometry], page ⟨undefined⟩.

When used, the *x* and *y* parameters can be given directly if the program sets the window position itself. If these arguments are zero, then the window manager or the user supplied window geometry determines the window placement.

`hasExtension (String` *`extensionName`*`)`
: Returns a `Boolean` value of true if the system's OpenGL library supports the GLX extension *extensionName*, false otherwise.

`map (void)`
: Maps the `GLXCanvasPane's` window to the display, and internally creates a `GLXContext` for the window, and makes the `GLXContext` current.

  This method fills in the receiver's `glxContextPtr` instance method.

`namedColorFT (String` *`colorName`*`, Float` *`redOut`*`, Float` *`greenOut`*`, Float` *`blueOut`*`)`
: Return the GLX compatible color values for *colorName*; i.e., the red, green, and blue values are `Floats` between 0.0 and 1.0. The *colorName* argument can be any of the colors supported by the X11 display server. Refer to *showrgb(1)* for a list of colors.

`onAnimation (String` *`animationHandlerName`*`)`
: Installs the callback method that the program calls 24 times a second. The method needs to have the prototype:

```
GLXCanvasPane instanceMethod <methodName> (void);
```

`onButtonPress (String` *`buttonPressHandlerName`*`)`
: Installs the callback method that handles ButtonPress events from the display. The callback method needs to have this prototype.

```
GLXCanvasPane instanceMethod <methodName> (Integer winX, Inte-
ger winY,
                                           Integer screenX, In-
teger screenY,
                                           Integer buttonState,
                                           Integer eventTime);
```

  The parameters *winX* and *winY* give the position of the pointer relative to the window's origin. The parameters *screenX* and *screenY* give the pointer's position relative to the upper left-hand corner of the root window.

Note this does not generally mean that the program can receive events when a button is pressed outside of the program's window. This depends on how the desktop GUI interprets button presses; with many desktop programs, the program doesn't receive events when a button is clicked outside of the program's window.

The *buttonState* parameter's value records which buttons are pressed at the time of the event. Note that many systems interpret a multiple button click (a "chord") as a unique button. For example, pressing the left and right buttons of a two-button mouse at the same time results in a buttonState that indicates button 2 is pressed, not that button 1 and button 3 are pressed simultaneously.

The *time* parameter is the time that the event occurred, so programs can interpret a series of ButtonPress events as multiple mouse clicks if necessary.

To install a buttonPress callback method, the program needs to include an expression like this one in its initialization code.

```
myGLXPane onButtonPress "myButtonPressHandler";
```

`onExpose (String` *`exposeHandlerName`*`)`

Installs the callback method to handle Expose events received from the display. The callback method should have the following prototype.

```
GLXCanvasPane instanceMethod <methodName> (Integer nEvents);
```

To install the callback method, the program's initialization should contain an expression like this one.

```
myPane onExpose "myExposeHandler";
```

The parameter *nEvents* contains the number of Expose events that the window is waiting to receive. This allows programs to execute the handler's statements once per group of Expose events; that is, when *nEvents* reaches 0.

This handler is important because it updates the window in coordination with other display events. If a callback method is not installed, then the `run` method calls the `swapBuffers` method.

`onIdle (String` *`callbackMethodName`*`)`

Installs a callback method that the program executes when it is not processing events from the display.

The callback method has the prototype:

```
GLXCanvasPane instanceMethod <idleHandler> (void);
```

To install the handler, the program's initialization needs to contain an expression like this.

```
myPane onIdle "myIdleCallback";
```

`onKeyPress (String` *`callbackMethodName`*`)`

Configures the receiver's `keyPressHandler` instance variable to refer to the application's actual KeyPress handler method, which is called when the program's window receives a KeyPress event from the display.

The actual callback method has the prototype:

```
GLXCanvasPane instanceMethod <methodName> (Integer xKeySym,
                                           Integer keyCode,
                                           Integer shiftState);
```

This example shows a simple KeyPress handler that closes the window and exits the program when the *`Escape`* key is pressed.

```
/* This definition comes from the machine's X11/keysymdef.h file. */
#define XK_Escape 0xff1b

GLXCanvasPane instanceMethod myKeyPressMethod (Integer xKeySym,
       Integer keyCode,
       Integer shiftState) {
  if (xKeySym == XK_Escape) {
    self deleteAndClose;
    exit (0);
  }
}
```

The first parameter is the X Window System symbol for the key, which is specific to the machine's keyboard configuration. The complete set of X key symbols is located in the machine's `X11/keysymdef.h` file.

The second parameter is the ASCII value of alphanumeric keys and punctuation keys. In the case of alphabetic characters, the value is the same whether the keypress is shifted or unshifted. That means that pressing *`A`* and `a` both result in the `keyCode` argument having the value 97.

The third parameter, shiftState, indicates whether a modifier key is currently being pressed. The parameter is the receiver's `shiftState` instance variable. The variable's description describes how to interpret its value.

Then, during the program's initialization the program's code should include an expression like the following.

```
myProgram onKeyPress "myKeyPressMethod";
```

There is a more detailed description of how to configure callback methods in section that discusses `Method` class. See ⟨undefined⟩ [CallbackSetup], page ⟨undefined⟩.

`onPointerMotion (String` *`callbackMethodName`*`)`
: Installs the callback method that handles pointer motion events from the display. The callback method must have the prototype:

```
GLXCanvasPane instanceMethod <methodName> (Integer winX,
                                           Integer winY,
                                           Integer screenX,
                                           Integer screenY);
```

The program's initialization should contain an expression like this one:

```
myPane onPointerMotion "myPointerMotionMethod";
```

`onResize (String` *`callbackMethodName`*`)`
: Installs the callback method that handles resize notifications from the display. The callback method needs to have the prototype:

```
GLXCanvasPane instanceMethod <methodName> (Integer width,
                                           Integer height);
```

The program's initialization code should contain an expression like this one.

```
myPane onResize "myResizeMethod";
```

`onTimerTick (String` *`callbackMethodName`*`)`
: Installs the callback method to be executed when the classes' interval timer reaches zero. The callback method needs to have the following prototype.

```
GLXCanvasPane instanceMethod <methodName> (void);
```

The interval in milliseconds between the callback method's execution is set in the `timerMSec` instance variable,

`pixelHeightFT (Integer` *`pxHeight`*`)`
: Set the pixel height of the selected font to the argument. The default height for rendering fonts with the Freetype libraries is 18 pixels.

`refreshRate (void)`
: Returns a `Float` with the display's refresh rate. If the machine's OpenGL does not support reporting the refresh rate, returns -1.

`run (void)`

Runs the event loop that receives X events from the display server, and sends them to the callback methods that are configured for the application.

Without any callback methods defined, the `run` method handles only '`Expose`' events (by calling `swapBuffers`), and '`ClientMessage`' events, which check for the `WM_DELETE_WINDOW` Atom and if present, delete the pane's window and GLX context, and exit the program.

`swapBuffers (void)`

Swaps the pane window's offscreen rendering buffer with the window's visible buffer.

`syncSwap (Integer` *`interval`*`)`

If *interval* > 0, sets the swap interval to 1/*interval*, which enables swap synchronization with the display's vertical refresh rate if the machine's OpenGL installation supports the `GLX_MESA_swap_control` extension.

An *interval* value of 0 disables swap synchronization.

Returns 0 on sucess, or -1 if the extension is not supported.

`textWidth (String` *`text`*`)`

Returns an `Integer` with the width of *text* in pixels in the currently selected X font. If no font is selected, the method returns '`-1`'.

`textWidthFT (String` *`text`*`)`

Returns a `Float` with the width of *text* in matrix coordinates for the currently selected Freetype font.

`title (String` *`title_string`*`)`

Set's the window's title. This method should be called as soon as possible after the program calls the `initialize` method.

`useFTFont (String` *`fontFileName`*`)`

Load a TrueType, FreeType or Type 1 font. Also initializes the font and GLEW libraries if needed.

The method uses the font's file name as its argument. To use a system's font aliasing and lookup, refer to See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`useXFont (String` *`fontName`*`)`

Register a X font for use with drawing text in the receiver pane. The argument, *fontname*, is th X Logical Font Descriptor of a X font that is available on the system - refer to *xlsfonts(1)* or *xfontsel(1)* for more information.

This method should be called during OpenGL initialization (that is, after the `GLXCanvasPane` object has been created and the GLX context established).

`xpmToTexture (Symbol` *`xpmdata`*`, Integer` *`widthout`*`, Integer` *`heightout`*`, Symbol` *`texeldataout`*`)`
`xpmToTexture (Symbol` *`xpmdata`*`, Integer` *`alpha`* `Integer` *`widthout`*`, Integer` *`heightout`*`, Symbol` *`texeldataout`*`)`

Translates a XPM pixmap into an OpenGL texture. The argument *xpm_data* is the pixmap's `char *pixmap_name[]` declaration. If no *alpha* argument is given, then '`1.0`' is used to create an opaque texture.

Alpha values can range from 0 (completely transparent) - 0xffff (completely opaque), although in practice, the alpha channel's effect might not be apparent, because OpenGL has its own set of functions that perform texture blending.

The method sets the arguments *width_out*, *height_out*, and *texel_data_out* with the height, width and data of the texture.

Mesa OpenGL textures, used with Linux systems, internally have the format GL_RGBA and the data type GL_UNSIGNED_INT_8_8_8_8, so you can create a 2D texture from a pixmap with statements like these.

```
Integer new xpmWidth;
Integer new xpmHeight;
Symbol new texData;

/*
 *  Note that the xpm_data argument should not normally need a
 *  translation from C.
 */
myGLUTApp xpmToTexture xpm_data, xpmWidth, xpmHeight, texData;
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, xpmWidth, xpmHeight, 0,
              GL_RGBA, GL_UNSIGNED_INT_8_8_8_8, texData);
```

Apple OpenGL implementations use a different internal format, so a program would create the equivalent texture like this.

```
Integer new xpmWidth;
Integer new xpmHeight;
Symbol new texData;

myGLUTApp xpmToTexture xpm_data, xpmWidth, xpmHeight, texData;
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, xpmWidth, xpmHeight, 0,
              GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV, texData);
```

The `xpmToTexture` method does not do any setup of the OpenGL texture environment. For basic textures, OpenGL works better with textures that have a geometry that is an even multiple of 2; e.g., 128x128 or 256x256 pixels.

Individual applications can add parameters for interpolation, blending, mipmap creation, and material rendering based on the program's requirements, though.

The Ctalk library only stores the data for one texture at a time, so if a program uses multiple textures, it should save the texture data to a separate `Symbol`, in orderq to avoid regenerating the texture each time it's used. Many OpenGL implementations also provide API functions for texture caching.

For an example of how to draw with textures, refer to the `glxtexture.ca` program in the Ctalk distribution's `demos/glx` subdirectory.

Sample GLXCanvasPane Application

```
#include <X11/Xlib.h>
#include <GL/glx.h>

#define DEFAULT_WIDTH 500
#define DEFAULT_HEIGHT 500

float face1[3][3] = {{0.0f, 2.0f, 0.0f},
     {-2.0f, -2.0f, 2.0f},
     {2.0f, -2.0f, 2.0f}};
float face2[3][3] = {{0.0f, 2.0f, 0.0f},
     {2.0f, -2.0f, 2.0f},
     {2.0f, -2.0f, -2.0f}};
float face3[3][3] = {{0.0f, 2.0f, 0.0f},
     {2.0f, -2.0f, -2.0f},
     {-2.0f, -2.0f, -2.0f}};
float face4[3][3] = {{0.0f, 2.0f, 0.0f},
     {-2.0f, -2.0f, -2.0f},
     {-2.0f, -2.0f, 2.0f}};

float base[4][3] = {{2.0f, -2.0f, 2.0f},
    {2.0f, -2.0f, -2.0f},
    {-2.0f, -2.0f, -2.0f},
    {-2.0f, -2.0f, 2.0f}};

float angle = 20.0;

GLXCanvasPane instanceMethod draw (void) {
  glEnable (GL_NORMALIZE);
  glEnable(GL_DEPTH_TEST);
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
  glLineWidth (1.0f);

  glLoadIdentity ();
  glColor4f (1.0f, 1.0f, 1.0f, 1.0f);
```

```
glRotatef (angle, 0.0f, 1.0f, 0.0f);
glRotatef (10.0f, 0.0f, 0.0f, 1.0f);

glBegin (GL_TRIANGLES);
glColor3f (1.0f, 0.0f, 0.0f);
glVertex3fv (face1[0]);
glColor3f (0.0f, 1.0f, 0.0f);
glVertex3fv (face1[1]);
glColor3f (0.0f, 0.0f, 1.0f);
glVertex3fv (face1[2]);

glColor3f (1.0f, 0.0f, 0.0f);
glVertex3fv (face2[0]);
glColor3f (0.0f, 1.0f, 0.0f);
glVertex3fv (face2[1]);
glColor3f (0.0f, 0.0f, 1.0f);
glVertex3fv (face2[2]);

glColor3f (1.0f, 0.0f, 0.0f);
glVertex3fv (face3[0]);
glColor3f (0.0f, 1.0f, 0.0f);
glVertex3fv (face3[1]);
glColor3f (0.0f, 0.0f, 1.0f);
glVertex3fv (face3[2]);

glColor3f (1.0f, 0.0f, 0.0f);
glVertex3fv (face4[0]);
glColor3f (0.0f, 1.0f, 0.0f);
glVertex3fv (face4[1]);
glColor3f (0.0f, 0.0f, 1.0f);
glVertex3fv (face4[2]);
glEnd ();

glBegin (GL_QUADS);

glColor3f (1.0f, 0.0f, 0.0f);
glVertex3fv (base[0]);
glColor3f (0.0f, 1.0f, 0.0f);
glVertex3fv (base[1]);
glColor3f (0.0f, 0.0f, 1.0f);
glVertex3fv (base[2]);
glColor3f (1.0f, 0.0f, 1.0f);
glVertex3fv (base[3]);

glEnd ();

glRotatef (20.0, 0.0f, 0.0f, 1.0f);
```

```
  glRotatef (angle, 0.0f, 1.0f, 0.0f);

  self swapBuffers;
}

GLXCanvasPane instanceMethod initGL (void) {
  glViewport (0, 0, DEFAULT_WIDTH, DEFAULT_HEIGHT);
  glClearColor(0.0, 0.0, 0.0, 1.0);
  glLineWidth (1.0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glEnable (GL_LINE_SMOOTH);
  glHint (GL_LINE_SMOOTH_HINT, GL_DONT_CARE);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  if (DEFAULT_WIDTH <= DEFAULT_HEIGHT) {
    glOrtho (-5.0, 5.0,
     -5.0 * ((float)DEFAULT_HEIGHT / (float)DEFAULT_WIDTH),
     5.0 * ((float)DEFAULT_HEIGHT / (float)DEFAULT_WIDTH),
     -5.0, 5.0);
  } else {
    glOrtho (-5.0, 5.0,
     -5.0 * ((float)DEFAULT_WIDTH / (float)DEFAULT_HEIGHT),
     5.0 * ((float)DEFAULT_WIDTH / (float)DEFAULT_HEIGHT),
     -5.0, 5.0);
  }
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
}

GLXCanvasPane instanceMethod myTimerTickHandler (void) {
  angle += 1.0;
  self draw;
}

/* This definition comes from the machine's X11/keysymdef.h file. */
#define XK_Escape 0xff1b

GLXCanvasPane instanceMethod myKeyPressMethod (Integer xKeySym,
       Integer keyCode,
       Integer shiftState) {
  if (xKeySym == XK_Escape) {
    self deleteAndClose;
    exit (0);
  }
}

GLXCanvasPane instanceMethod myExposeMethod (Integer nEvents) {
```

```
  if (nEvents == 0)
    self draw;
}

GLXCanvasPane instanceMethod myResizeMethod (Integer width,
     Integer height) {
  float ar;

  glViewport (0, 0, width, height);
  glMatrixMode (GL_PROJECTION);
  glLoadIdentity ();
  if (width <= height)
    ar = (float)height / (float)width;
  else
    ar = (float)width / (float)height;
  glOrtho (-5.0, 5.0, -5.0 * ar, 5.0 * ar, -5.0, 5.0);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity ();
}

int main () {
  GLXCanvasPane new pane;

  pane initialize (1, 150, 500, 500);
  pane title "GLXCanvasPane Demonstration";
  pane map;
  pane raiseWindow;

  pane onKeyPress "myKeyPressMethod";
  pane onExpose "myExposeMethod";
  pane onTimerTick "myTimerTickHandler";
  pane onResize "myResizeMethod";

  pane initGL;

  pane run;

}
```

3.77 `X11PaneDispatcher` Class

`X11PaneDispatcher` objects manage events from the system's display, control posititioning and rendering of subpanes within the main pane, and help communicate events between the main window's pane and the subwindows' panes.

The positioning of subwindows within the parent window is controlled by the arguments to the `attachTo` methods; in particular, these methods accept a geometry specification, as an argument, which has the following format.

```
width[%]xheight[%][+x[%]+y[%]]
```

If a dimension does not contain a percent (‘`%`’) sign, the dimensions are in pixels. With a percent sign, the dimensions are a fractional percentage of the parent pane's width or height.

Also, for some `Pane` classes, like dialog windows, if the geometry does not include the x and y dimensions, then the class positions the window centered over its parent window.

The `X11PaneDispatcher` method `new` creates a X window, although the window of the `X11PaneDispatcher` itself is not normally visible, the window is provided for subclass panes whose windows will appear within the main `X11Pane` window.

Subclasses can reimplement default methods like `new` and the event handler methods if necessary. It is also possible for subclasses to implement event handlers with other method messages than those given here. The `X11PaneDispatcher` object can use these subclasses' methods if they are available.

Subclasses, however, should use the `setMethodHandler` method, described below, to register these callback methods.

Handler Classes

The following event handlers and the default names of the callback method that handles the event, are available to `X11PaneDispatcher` objects and its subclasses. Handler methods need to take as arguments a reference to a subpane object, and the `InputEvent` object from the system. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

Handler Class	Event Class	Description	Default Handler Method
"resize"	RESIZENOTIFY	Called when the user resizes a window.	subPaneResize
"move"	MOVENOTIFY	Called when a user moves a window. Depending on the window manager, an application might also need to make sure the window is resized correctly.	subPaneMove
"expose"	EXPOSE	Called whenever the display server generates exposes a window due when raised, uncovered, resized, or other change.	subPaneExpose
"kbdinput"	KEYPRESS KEYRELEASE	Called when the user presses a key and the application's window has the focus.	subPaneKbdInput
"pointerinput"	BUTTONPRESS	Called when a mouse button	subPanePointerInput

```
               BUTTONRELEASE  is pressed or released.

"pointermotion" MOTIONNOTIFY  Called whe the pointer is       subPanePointerMotion
                              moved.

"selectionrequest"
             SELECTIONREQUEST Received when another program   subPaneSelectionRequest
                              requests the X selection.
                              This callback is here mainly
                              for completeness; presently, X
                              selection events are handled
                              internally, and only with the
                              library functions used by
                              X11TextEditorPane objects.

"selectionclear"
               SELECTIONCLEAR Received when another program
                              requests the X selection. Currently,
                              on library functions used by
                              X11TextEditorPane objects use this
                              internally but applications can use this
                              event to update their status if
                              necessary.
"wmfocuschange" WMFOCUSCHANGENOTIFY
                              Received from the window manager when the
                              pointer enters or leaves a window, or when
                              the window manager raises a window.  The
                              xEventData1 instance variable contains the
                              type of X event, either FocusIn or FocusOut.

"enternotify" ENTERWINDOWNOTIFY
                              Received when the pointer enters a
                              window from another window.

"leavenotify" LEAVEWINDOWNOTIFY
                              Received when the pointer leaves a
                              window.

"focusin"     FOCUSIN         Received when the application signals
"focusout"    FOCUSOUT        a change of widget focus; for example,
                              when the user presses the Tab
                              key.

"map"         MAPNOTIFY       Received when the pane's window is
                              mapped onto the display.

"destroy"     WINDELETE      Called when the user closes the
                              application window.
```

Handler Methods

A handler typically takes as its arguments a reference to a subpane, and the event. Although the subpane reference parameter can be declared as an `Object`, it is in actual use a `Symbol` object, with the reference defined by the `attachTo` method. The handler should be able to pass the event along to any subpanes of the receiver pane if necessary, by checking whether

subpanes implement their own handlers. Here, for example, is the `X11PaneDispatcher` class's `subPaneResize` method.

```
X11PaneDispatcher instanceMethod subPaneResize (Object __subPane,
      InputEvent __event) {
  "Dispatch an Resize event to the subpanes of the
   receiver pane."
  X11Pane new containerPane;
  self size x = __event xEventData3;
  self size y = __event xEventData4;
  containerPane = *self container;
  XResizeWindow (containerPane xDisplay, containerPane xWindowID,
 self size x, self size y);
  if (__subPane isInstanceMethod "subPaneResize") {
    __subPane methodObjectMessage __subPane handleResize, __subPane,
      __event;
  }
  return NULL;
}
```

The internals of the subpane API are likely to change and be expanded in future releases. Using the methods described here and in other sections should help insure that applications are compatible with future Ctalk releases.

Instance Variables

`canFocus` A `Boolean` that determines whether the widget is highlighed when the pointer passes over it, or the application sets the focus; for example, by pressing the `Tab` key.

If this variable is true, which is the default, then the class should also declare event handlers for focus in and focus out events. Otherwise, the program prints a warning each time the `shiftFocus` method, described below, calls a `NULL Method` object.

`handleDestroy`
: A `Method` that provides the default handler to delete a subpane window and its data.

`handleEnterNotify`
`handleLeaveNotify`
: Handlers for events that are generated when a pointer crosses from one window to another.

`handleFocusIn`
`handleFocusOut`
: Handles highlighting or un-highlighting a widget when receiving a FOCUSIN or FOCUSOUT event from the application; for example, when the user presses the `Tab` key.

`handleKbdInput`
: A `Method` that provides the default handler for subpanes to handle keyboard input.

`handleMap`
: Called when the program recieves a MAPNOTIFY event for the pane's window.

`handleMove`
: A `Method` that provides the default handler for moving subpane windows.

`handlePointerInput`
: A `Method` that provides the default handler for pointer input (mainly ButtonPress) events.

`handlePointerMotion`
: A `Method` that provides the default handler for pointer motion events.

`handleResize`
: A `Method` that provides the default handler for resizing subpane windows within the main window.

`handleSelectionClear`
: A `Method` that provides the default handler for SelectionClear events.

`handleSelectionRequest`
: A `Method` that provides the default handler for SelectionRequest events. Currently, this callback is here for completeness; X selection events are handled internally, and only in the library functions used by `X11TextEditorPane` objects.

`handleWMFocusChange`
: The method that handles the events received from the desktop's window manager when it changes the window focus in response to a pointer motion or click; or when the window manager raises a window.

`hasFocus` An `Integer` that is true if the current subpane has the input focus. For programs with only a single widget class, this variable is not used.

`highlight`
: A `Boolean` that indicates whether the widget is displayed highlighted.

`modalWin` When a dialog window is popped up over the main window, this holds the window ID of the popup. The `handleSubPaneEvent` method uses this to determine how the application's main window should respond to X events; generally it keeps the dialog window above the main window until the dialog window is withdrawn. When their is no window popped up over the main window, the value of `modalWin` is zero (‘`0`’).

`tabFocus` A `Boolean` that determines, if true (which is the default), whether the `handleSubPaneEvent` method intercepts the `Tab` key in order to shift the input focus when it is pressed. If `tabFocus` is true, the subpane's class must implement methods to handle the `focusin` and `focusout` events that shifting focus with the keyboard implements, or the program will display warning messages when it can't find the methods.

Instance Methods

`attachTo (Object` *`parentPane`*`)`
`attachTo (Object` *`parentPane`*`, String` *`geometry`*`)`

Attach the receiver to its parent pane, typically a `X11Pane` or `X11PaneDispatcher`. Also creates a X window, although `X11PaneDispatcher` windows themselves are not normally visible.

When the subwindow is attached to the parent window, the Ctalk library creates the pane object's window and graphics context, and clears the window to the background color of the pane object's `backgroundColor` instance variable; for example, with an expression like this:

```
myCanvasPane backgroundColor = "blue";
```

Otherwise, the method clears the subpane's window to black.

The *geometry* argument, if present, defines the size and placement of the subpane's window within the parent window. A geometry specification has the form:

```
width[%]xheight[%]+x[%]+y[%]
```

The dimensions are in pixels, unless a percent sign ('`%`') follows a dimension. In that case, the dimension is a fractional percentage of the parent pane's width or height. The `String` may contain a combination of absolute and relative dimensions.

`handleSubPaneEvent (InputEvent` *`event`*`)`

Handle an input event from the window system. Typically the parent pane's `inputStream` provides the event. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

This method also checks keypresses for the `Tab` key, and calls the `shiftFocus` method in order to shift focus between a window's subpanes when the user presses `Tab`.

`new (`*`dispatcherName`*`)`

Create a new `X11PaneDispatcher` object. Initializes the instance variables to the default subpane event handlers and the container mode to '`full`'. If the argument list contains more than one label, created new `X11PaneDispatcher` objects with the names of each label.

`setMethodHandler (String` *`handlerType`*`, Method` *`handlerMethod`*`)`

Set the pane's handler for *handlerType* to *handlerMethod*. Currently supported handler types are: '`resize`'.

`shiftFocus (void)`

When the user presses `Tab`, `handleSubPaneEvent` calls this method, which highlights a window's subpanes in succession, if the subpanes can take the input focus.

Refer to the `canFocus` instance variable, and the handlers for focus in and focus out events. These event handlers are called by the program, and are not the same as the `handleWMFocusChange` handler, which is called when the window focus changes on the desktop.

`clearFocus (void`
: Called before shifting the focus highlight to a new pane to insure that only one pane indicates that it should receive focus, including the synthetic focus that is assigned when shifting focus using the `Tab` key.

`subPaneDestroy (Object` *`subPaneRef`*`, InputEvent` *`destroyEvent`*`)`
: The default handler for `WINDELETE` events. Like the other method handlers, *subPaneRef* is typically a `Symbol` object. The `X11TerminalStream` section describes these events. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`subPaneGeometry`
: A `String` that stores a subpane's geometry specification, if any. For an explanation of geometry string's format, refer to the `attachTo` method, below.

`subPaneKbdInput (Object` *`subPaneRef`*`, InputEvent` *`kbdInputEvent`*`)`
: The default handler for `KEYPRESS` and `KEYRELEASE` events. Like the other method handlers, *subPaneRef* is typically a `Symbol` object. The `X11TerminalStream` section describes these events. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`subPaneMove (Object` *`subPaneRef`*`, InputEvent` *`moveNotifyEvent`*`)`
: The default event handler for `MOVENOTIFY` events from the system's GUI. Like the other method handlers, *subPaneRef* is typically a `Symbol` object. The `X11TerminalStream` section describes these events. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

  With some window managers, a `subPaneMove` method might also need to handle `RESIZENOTIFY` events.

`subPanePointerMotion (Object` *`subPaneRef`*`, InputEvent` *`event`*`)`
`subPanePointerInput (Object` *`subPaneRef`*`, InputEvent` *`event`*`)`
: The default handlers for `MOTIONNOTIFY`, and `BUTTONPRESS` and `BUTTONRELEASE` events. Like the other method handlers, *subPaneRef* is typically a `Symbol` object. The `X11TerminalStream` section describes these events. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`subPaneResize (Object` *`subPaneRef`*`, InputEvent` *`resizeNotifyEvent`*`)`
: The default resize handler for subpane windows. Typically *subPaneRef* is a `Symbol` object that contains a reference to the subpane object. The *resizeEvent* argument is typically a `RESIZENOTIFY` event from the system's GUI. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

3.78 `X11CanvasPane` Class

The `X11CanvasPane` class provides the instance variables and methods for basic X window graphics operations like drawing points, lines, and rectangles. The width and color of shapes is controlled by the `pen` instance variable.

Here is brief drawing program that uses a `X11CanvasPane` object. Clicking on the pane's window draws a dot at that point. If the program is given the argument '`-v`', the program displays the X events it receives from the display.

```
int main (int argv, char **argc) {
  X11Pane new xPane;
  X11PaneDispatcher new xTopLevelPane;
  X11CanvasPane new xCanvasPane;
  InputEvent new e;
  Integer new nEvents;
  Integer new verbose;
  Exception new ex;
  String new text;
  Application new paneApp;

  paneApp enableExceptionTrace;
  paneApp installExitHandlerBasic;

  xPane initialize 0, 0, 200, 100;
  xPane inputStream eventMask =
    WINDELETE|BUTTONPRESS|BUTTONRELEASE|MOVENOTIFY|EXPOSE;
  xTopLevelPane attachTo xPane;
  xCanvasPane attachTo xTopLevelPane;
  xPane map;
  xPane raiseWindow;

  xPane openEventStream;

  xCanvasPane clear;
  xCanvasPane background "blue";
  xCanvasPane pen width = 5;
  xCanvasPane pen colorName = "white";

  xCanvasPane refresh;

  verbose = FALSE;
  if (argc == 2) {
    if (!strcmp (argv[1], "-v")) {
      verbose = TRUE;
    }
  }

  WriteFileStream classInit;

  while (TRUE) {
    xPane inputStream queueInput;
```

```
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;
      xPane subPaneNotify e;  /* Call the classes' event handlers. */
      if (ex pending)
  ex handle;

      switch (e eventClass value)
{
  /*
   *  Handle both types of events in case the window
   *  manager doesn't distinguish between them.
   */
case MOVENOTIFY:
  if (verbose) {
    stdoutStream printOn "MOVENOTIFY\t%d\t%d\t%d\t%d\n",
      e xEventData1,
      e xEventData2,
      e xEventData3,
      e xEventData4;
    stdoutStream printOn "Window\t\t%d\t%d\t%d\t%d\n",
      xPane origin x,
      xPane origin y,
      xPane size x,
      xPane size y;
  }
  break;
case RESIZENOTIFY:
  if (verbose) {
    stdoutStream printOn "RESIZENOTIFY\t%d\t%d\t%d\t%d\n",
      e xEventData1,
      e xEventData2,
      e xEventData3,
      e xEventData4;
    stdoutStream printOn "Window\t\t%d\t%d\t%d\t%d\n",
      xPane origin x,
      xPane origin y,
      xPane size x,
      xPane size y;
  }
  break;
case EXPOSE:
  if (verbose) {
    stdoutStream printOn "Expose\t\t%d\t%d\t%d\t%d\t%d\n",
      e xEventData1,
      e xEventData2,
      e xEventData3,
      e xEventData4,
```

```
            e xEventData5;
        }
        break;
      case BUTTONPRESS:
        xCanvasPane drawPoint e xEventData1, e xEventData2;
        if (verbose) {
          stdoutStream printOn "ButtonPress\t\t%d\t%d\t%d\t%d\t%d\n",
            e xEventData1,
            e xEventData2,
            e xEventData3,
            e xEventData4,
            e xEventData5;
        }
        xCanvasPane refresh;
        break;
      case BUTTONRELEASE:
        if (verbose) {
          stdoutStream printOn "ButtonRelease\t\t%d\t%d\t%d\t%d\t%d\n",
            e xEventData1,
            e xEventData2,
            e xEventData3,
            e xEventData4,
            e xEventData5;
        }
        break;
      case WINDELETE:
          xPane deleteAndClose;
        exit (0);
        break;
      default:
        break;
      }
          }
        }
      }
```

Instance Variables

`dragStart`

A `Point` object that records the beginning of a canvas motion operation within a window or view port.

`moveCursor`

The `X11Cursor` displayed when moving the `X11CanvasPane` object within a window or view port.

`pen` A `Pen` object that contains the width in pixels and color of lines and points drawn on the pane's window.

`regions` An `AssociativeArray` that contains the rectangular regions defined by the `defineRegion` method, below.

`viewHeight`
An `Integer` that contains the height of the pane's window and buffers in pixels.

`viewWidth`
An `Integer` that contains the height of the pane's window and buffers in pixels.

`viewXOrg` The X coordinate of the upper right-hand corner of a canvas' visible rectangle within a window or view port.

`viewYOrg` The Y coordinate of the upper right-hand corner of a canvas' visible rectangle within a window or view port.

Instance Methods

`attachTo (Object` *`parent_pane`*`)`
`attachTo (Object` *`parent_pane`*`, String` *`geometry`*`)`
`attachTo (Object` *`parent_pane`*`, Integer` *`xOrg`*`, Integer` *`yOrg`*`)`
`attachTo (Object` *`parent_pane`*`, Integer` *`xOrg`*`, Integer` *`yOrg`*`, Integer` *`xSize`*`, Integer` *`ySize`*`)`

Attach a `X11CanvasPane` object to its parent pane, which is typically a `X11PaneDispatcher` object. With one argument, this method initializes the size of the pane's window and buffers to the parent pane's dimensions, and positions the pane at the upper left-hand origin of the main window.

If two arguments are present, the second is a `String` with the geometry specifcation for the subpane. Subpane geometry strings have the form:

```
[-]width[%]x[-]height[%][+[-]x[%]+[-]y[%]]
```

The dimensions are in pixels, unless a percent sign ('`%`') follows a dimension, or a minus sign ('`-`' precedes the dimension.

When a percent sign follows the dimension, the width of the pane in pixels is the fractional percentage of the parent pane's width or height.

If a minus sign precedes the dimension, the dimension is measured relative to the opposite edge of the parent window; i.e., a width of '`-15`' places the subpane's right edge 15 pixels in from the right edge of the parent window. A x position of '`-90`' places the subpane's *top* edge (not the subpane's bottom edge) 90 pixels from the bottom edge of the parent window.

The `String` may contain a combination of absolute and relative dimensions.

With only the width and height given, the method positions the pane at *xOrg*,*yOrg* within the parent pane, which usually is relative to the upper left hand origin of the window. This is generally used to center modal dialog panes over their parent window.

With all four dimensions given, the method positions the pane at *xOrg*,*yOrg* within the parent pane, with the width and height *xSize*,*ySize*.

`background (String color)`
Set the background of the pane to *color*. You need to update the pane using, for example, `clearRectangle`, for the new background to be visible. See the note for `X11Bitmap` class's `background` method. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`clear (void)`
Clear the pane to the background color.

`clearRectangle (Intger xOrg, Integer yOrg, Integer xSize, Integer ySize)`
Clear the pane's image to the window background in a rectangle bounded by the method's arguments, and update the top-level pane's window.

`copy (X11Bitmap src_bitmap, Integer src_x_ort, Integer src_y_org, Integer src_width, Integer src_height, Integer dest_x_org, Integer dest_y_org)`
Copies the contents of *src_bitmap* to the receiver's drawing surface. The source dimensions are determined by *src_x_org*, *src_y_org*, *src_width*, and *src_height*. The method draws the source bitmap's contents with the source's upper left-hand corner at *dest_x_org*, *dest_y_org*.

The `X11Bitmap's` parent drawable must be the receiver's drawable surface, and the color depths of the source and destination must match.

The process is similar to the `refresh` method, below, so programs do not need to call both `copy` and `refresh` for the same operation.

This slightly abbreviated example program is included in the Ctalk package at test/expect/examples-x11/canvas-copy.c as well as the XPM graphic, but almost any XPM should work as well.

```
#include "coffee-cup.xpm"

/*
   Set these to the width and height of your pixmap,
   and edit the pixmapFromData expression below to
   the xpm's declaration name.
*/
#define XPM_WIDTH 127
#define XPM_HEIGHT 141

X11CanvasPane instanceMethod drawXPMs (X11Bitmap xpmBitmap) {
  Integer new i;

  for (i = 0; i < 5; i++) {
    self copy xpmBitmap, 0, 0, XPM_WIDTH, XPM_HEIGHT, (i* 40), (i * 40);
  }

  self refresh;
}
```

```
int main () {
  X11Pane new xPane;
  InputEvent new e;
  X11PaneDispatcher new xTopLevelPane;
  X11CanvasPane new xCanvasPane;
  Application new paneApp;
  X11Bitmap new srcBitmap;

  paneApp enableExceptionTrace;
  paneApp installExitHandlerBasic;

  xPane initialize 10, 10, 300, 300;
  xTopLevelPane attachTo xPane;
  xCanvasPane attachTo xTopLevelPane;

  srcBitmap create xCanvasPane xWindowID, XPM_WIDTH, XPM_HEIGHT,
    xCanvasPane depth;

  xPane map;
  xPane raiseWindow;
  xPane openEventStream;

  xCanvasPane background "white";

  srcBitmap pixmapFromData (0, 0, coffee_cup);

  xCanvasPane drawXPMs srcBitmap;

  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;
      /* We don't have to use, "xPane subPaneNotify e" here, because
         the program doesn't need to handle any X events for the
         graphics classes. */
      switch (e eventClass value)
        {
        case WINDELETE:
          xPane deleteAndClose;
          exit (0);
          break;
        case EXPOSE:
        case RESIZENOTIFY:
  xCanvasPane drawXPMs srcBitmap;
          break;
        default:
```

```
                    break;
                }
            }
            usleep (100000);
        }
    }
```

Note: The `copy` method retrieves the `paneBuffer` instance variable. If you use an expresion like the following, then the program calls the `X11Bitmap : copy` method instead. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

```
myRcvrPane paneBuffer copy ...
```

`defineRegion (String` *`regionName`*`, Integer` *`xOrg`*`, Integer` *`yOrg`*`, Integer` *`xSize`*`, Integer` *`ySize`*`)`

Define a rectangular region with name *regionName* with the upper left-hand corner at *xOrg*, *yOrg* relative to the upper left-hand corner of the canvas. The region has the width *xSize* and height *ySize*. When handling input events, the window system clips the region to the canvas' viewable area.

`drawCircle (Circle` *`aCircle`*`, Integer` *`filled`*`, String` *`bgColor`*`)`
`drawCircle (Circle` *`aCircle`*`, Pen` *`aPen`*`, Integer` *`filled`*`, String` *`bgColor`*`)`

Draw the circle defined by *aCircle* in the receiver's paneBuffer. If *filled* is true, draws a filled circle. If the *aPen* argument is given, draws the circle with the color and the line width defined by the `Pen`, and fills the interior of the circle with *bgColor*.

For an example program, refer to the `Circle` section of this manual. See ⟨undefined⟩ [Circle], page ⟨undefined⟩.

`drawPoint (Intger` *`x`*`, Integer` *`y`*`)`

Draw a dot on the pane's window at the x and y coordinates given by the arguments.

`drawLine (Line` *`aLine`*`)`
`drawLine (Line` *`aLine`*`, Pen` *`aPen`*`)`
`drawLine (Intger` *`startX`*`, Integer` *`startY`*`, Integer` *`endX`*`, Integer` *`endY`*`)`
`drawLine (Intger` *`startX`*`, Integer` *`startY`*`, Integer` *`endX`*`, Integer` *`endY`*`, pen` *`aPen`*`)`

With one argument, a `Line` object, draws the line using the receiver's `Pen` instance variable. With two arguments, draws the `Line` object with the color and line width given by *aPen*.

If given the line's endpoints as arguments, the method draws a line on the pane's window from the point given by the *startX* and *startY* arguments to the point given by the *endX* and *endY* arguments, with the color and the line width given by the receiver's `Pen` object.

`drawLine (Intger` *`xOrg`*`, Integer` *`yOrg`*`, Integer` *`xSize`*`, Integer` *`ySize`*`)`

Draw a filled rectangle on the pane's window with the upper-left hand corner at the point given by the *xOrg* and *yOrg* arguments, with the width *xSize* and

the height *ySize*. If a `Pen` argument isn't given, uses the line width and color defined by the receiver's `pen` instance variable; otherwise uses the line width and color defined by the `Pen` argument.

`drawRoundedRectangle (Intger` *`xOrg`*`, Integer` *`yOrg`*`, Integer` *`xSize`*`, Integer` *`ySize`*`, Integer` *`radius`*`)`
: Similar to `drawFilledRectangle`, but this method takes an extra argument, the radius of the corner arcs that round the rendered rectangle's corners.

`drawRectangle (Intger` *`xOrg`*`, Integer` *`yOrg`*`, Integer` *`xSize`*`, Integer` *`ySize`*`)`
: Draw the borders of a rectangle on the pane's window with the upper-left hand corner at the point given by the *xOrg* and *yOrg* arguments, with the width *xSize* and the height *ySize*.

`drawRoundedRectangle (Intger` *`xOrg`*`, Integer` *`yOrg`*`, Integer` *`xSize`*`, Integer` *`ySize`*`, Integer` *`radius`*`)`
: This method is similar to `drawRectangle`, except that it takes an extra argument, *radius*, which specifies the radius of the arcs that form the rectangle's corners.

`directCopy (X11Bitmap` *`src_bitmap`*`, Integer` *`src_x_ort`*`, Integer` *`src_y_org`*`, Integer` *`src_width`*`, Integer` *`src_height`*`, Integer` *`dest_x_org`*`, Integer` *`dest_y_org`*`)`
: Similar to the `copy` method, above, except `directCopy` copies the `X11Bitmap` object given as its argument directly to the window. This might be quicker, and doesn't require that the program call `refresh` (below) to update the window contents, but this method may also cause flickering when the window is updated.

`foreground (String` *`color`*`)`
: Set the background of the pane to *color*. See the note for `X11Bitmap` class's `background` method. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`new (String` *`pane_name`*`)`
: Create a new `X11CanvasPane` instance and initialize its event handlers.

  If the argument list contains more than one label, create a new `X11CanvasPane` object with the names given by the arguments; for example,

```
X11CanvasPane new pane1, pane2, pane3;
```

`pixmapFromData (int` *`x_org`*`, int` *`y_org`*`, char *`*`xpm_data[]`*`)`
: Draw the X pixmap defined by *xpm_data* with the upper left corner at *x_org*,*y_org* on the receiver's pane.

  The *xpm_data* argument is the name of the array declared at the start of a `xpm` file's data array.

`refresh (void)`
: Redraw the pane on the main window.

`refreshReframe (void)`
: Redraw the pane on the main window. If the user has moved the pane by clicking and dragging on it, then reposition the pane within the window.

`putStrXY (Integer xOrg, Integer yOrg String text)`
`putStrXY (Integer xOrg, Integer yOrg String text, String font_desc)`
`putStrXY (Integer xOrg, Integer yOrg String text, String font_desc, String color_name))`

Write *text* on the receiver pane's drawing surface (usually a X11Bitmap) at position *xOrg*,*yOrg*. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

If the fourth argument is *font_desc*, the method draws the text using that font, which is either a X Logical Font Descriptor if using X bitmap fonts or a Fontconfig descriptor if using Freetype fonts.

For information about Fontconfig descriptors, refer to the `X11FreeTypeFont` section See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩. For information about X Logical Font Descriptors, refer to the `X11Font` section See ⟨undefined⟩ [X11Font], page ⟨undefined⟩.

If a *color_name* argument is also given, draws the text using that color.

`subPaneDestroy (Object subPaneRef, InputEvent event)`

Deletes the pane's window and its data when the user closes the pane's window.

`subPaneExpose (Object subPaneRef, InputEvent event)`

Redraws the pane's window whenever it is mapped or displayed after being covered by another window.

`subPanePointerInput (Object subPaneRef, InputEvent event)`

Default handler for mouse ButtonPress and ButtonRelease events. This method is a no-op here, but it can be re-implemented if necessary by subclasses. The application receives pointer events, like all other events, via the top-level window's `inputStream` (a `X11TerminalStream` object).

`subPaneResize (Object subPaneRef, InputEvent event)`

The handler for Resize events from the X display. Resizes the pane's X window and adjusts the pane's dimensions.

3.79 `X11ButtonPane` Class

The `X11ButtonPane` class defines instance variables and methods that draw buttons on X windows. The buttons' appearance is defined by the values in the resources and instance variables, and they may be used to define the format of customized buttons.

`X11ButtonPane` objects also contain a `X11LabelPane` object, which is used to render the visual elements, like the text, border, and highlighting, on the button's face.

In many cases the button's `X11LabelPane` subpane inherits the definitions of the `X11ButtonPane's` instance variables and resources, generally when a program constructs the pane during a call to the `attachTo` method.

Here are several example programs. The first draws beveled buttons, the second draws rounded, non-beveled buttons.

```
/* buttons.ca - X11ButtonPane Demonstration -*-c-*- */

#include <ctalk/ctalkdefs.h>
```

```
/* Uncomment this #define to use X bitmap fonts. */
/* #define XFONTS */

/* Also, uncomment this to draw a multiline label. */
/* #define MULTILINE */

/* Uncomment if you want the buttons to be highlighted with wider
   borders and a bold label (if the font supports it). */
/* #define BOLD_HILITE */

/* See the X11FreeTypeFont section of the the Ctalk reference. */
#define FTFONT_BOLD   200
#define FTFONT_MEDIUM 100

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
  X11ButtonPane new lbutton;
  X11ButtonPane new rbutton;
  X11LabelPane new label;
  InputEvent new e;

  mainWindow backgroundColor = "blue";

  label canFocus = false;
  label resources replaceAt "borderWidth", 0;

  label resources replaceAt "backgroundColor", "blue";
  label resources replaceAt "foregroundColor", "blue";
  label resources replaceAt "textColor", "white";
  lbutton resources replaceAt "backgroundColor", "blue";
  rbutton resources replaceAt "backgroundColor", "blue";
  lbutton resources replaceAt "ftFont", "sans serif-10";
  rbutton resources replaceAt "ftFont", "sans serif-10";

  mainWindow initialize 255, 200;
  mainWindow inputStream eventMask =
    EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR
  dispatcher attachTo mainWindow;
  lbutton attachTo dispatcher, "90x50+25+100";
  rbutton attachTo dispatcher, "90x50+135+100";
  label attachTo dispatcher, "147x80+44+15";

  mainWindow map;
  mainWindow raiseWindow;
```

```
  mainWindow openEventStream;

  mainWindow setWMTitle "X11ButtonPane Demo";

#ifndef XFONTS
  /*
   * This is the recommended way to set up fonts for each widget.
   */
  label ftFontVar initFontLib;
  label ftFont "DejaVu Sans", 0, FTFONT_BOLD, 0, 12.0;
  label ftFontVar saveSelectedFont;
  lbutton ftFont "DejaVu Sans", 0, FTFONT_MEDIUM, 0, 10.0;
  lbutton ftFontVar saveSelectedFont;
  rbutton ftFont "DejaVu Sans", 0, FTFONT_MEDIUM, 0, 10.0;
  rbutton ftFontVar saveSelectedFont;
#else
  lbutton label font "fixed";
#endif

  label multiLine "X11ButtonPane\nDemo";

#ifndef MULTILINE
  lbutton label text "Left";
  rbutton label text "Right";
#else
  lbutton label multiLine "Click\nHere";
  rbutton label multiLine "Click\nHere, Too";
#endif

#ifdef BOLD_HILITE
  lbutton label resources replaceAt "highlightForegroundColor",
    (lbutton resources at "foregroundColor");
  rbutton label resources replaceAt "highlightForegroundColor",
    (rbutton resources at "foregroundColor");
  lbutton label resources replaceAt "highlightBorderWidth", 2;
  rbutton label resources replaceAt "highlightBorderWidth", 2;
  lbutton label resources replaceAt "ftFont", "sans serif-10:bold";
  rbutton label resources replaceAt "ftFont", "sans serif-10:bold";
#endif

  lbutton draw;
  lbutton refresh;
  rbutton draw;
  rbutton refresh;
  label draw;
  label refresh;
```

```
  while (TRUE) {
    mainWindow inputStream queueInput;
    if (mainWindow inputStream eventPending) {
      e become mainWindow inputStream inputQueue unshift;

      mainWindow subPaneNotify e;

      switch (e eventClass value)
{
case EXPOSE:
  lbutton subPaneExpose (lbutton, e);
  rbutton subPaneExpose (rbutton, e);
  label subPaneExpose (label, e);
  break;
case WINDELETE:
    mainWindow deleteAndClose;
  exit (0);
  break;
default:
  if (lbutton haveClick) {
    printf ("left button!\n");
    lbutton clearClick;
  } else if (rbutton haveClick) {
    printf ("right button!\n");
    rbutton clearClick;
  }
  break;
}
    } else {
      usleep (1000);
    }
  }

}



/* roundbuttons.ca - X11ButtonPane Rounded Buttons -*-c-*- */

#include <ctalk/ctalkdefs.h>

/* To avoid overlapping arcs, corner_radius < (button_minor_dimen / 2) */
#define CORNER_RADIUS 13

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
```

```
X11ButtonPane new lbutton;
X11ButtonPane new rbutton;
X11LabelPane new label;
InputEvent new e;

mainWindow backgroundColor = "blue";
label resources replaceAt "backgroundColor", "blue";
rbutton resources replaceAt "backgroundColor", "blue";
lbutton resources replaceAt "backgroundColor", "blue";
rbutton resources replaceAt "foregroundColor", "blue";
lbutton resources replaceAt "foregroundColor", "blue";


mainWindow initialize 225, 150;
mainWindow inputStream eventMask =
  EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR
dispatcher attachTo mainWindow;
lbutton attachTo dispatcher, "80x30+22+100";
rbutton attachTo dispatcher, "80x30+124+100";
label attachTo dispatcher, "147x80+32+15";

mainWindow map;
mainWindow raiseWindow;

mainWindow openEventStream;

mainWindow setWMTitle "X11ButtonPane Demo";

label ftFontVar initFontLib;
label multiLine "X11ButtonPane\nRounded Button\nDemo";
label canFocus = false;
label borderWidth = 0;
label resources replaceAt "borderColor", "blue";
label resources replaceAt "ftFont", "sans-serif-14:bold";
label resources replaceAt "textColor", "white";
label resources replaceAt "backgroundColor", "blue";
label resources replaceAt "foregroundColor", "blue";

rbutton label ftFontVar selectFontFromFontConfig "sans serif-10";
rbutton label ftFontVar saveSelectedFont;
rbutton resources replaceAt "foregroundColor", "blue";
lbutton label ftFontVar selectFontFromFontConfig "sans serif-10";
lbutton label ftFontVar saveSelectedFont;
lbutton resources replaceAt "foregroundColor", "blue";
rbutton resources replaceAt "textColor", "white";
lbutton resources replaceAt "textColor", "white";
```

```
  lbutton label text "Left";
  rbutton label text "Right";

  lbutton bevelEdges = false;
  lbutton radius = CORNER_RADIUS;
  rbutton bevelEdges = false;
  rbutton radius = CORNER_RADIUS;

  lbutton draw;
  lbutton refresh;
  rbutton draw;
  rbutton refresh;
  label draw;
  label refresh;

  while (TRUE) {
    mainWindow inputStream queueInput;
    if (mainWindow inputStream eventPending) {
      e become mainWindow inputStream inputQueue unshift;

      mainWindow subPaneNotify e;

      switch (e eventClass value)
{
case EXPOSE:
  lbutton subPaneExpose (lbutton, e);
  rbutton subPaneExpose (rbutton, e);
  label subPaneExpose (label, e);
  break;
case WINDELETE:
    mainWindow deleteAndClose;
  exit (0);
  break;
default:
  if (lbutton haveClick) {
    printf ("left button!\n");
    lbutton clearClick;
  } else if (rbutton haveClick) {
    printf ("right button!\n");
    rbutton clearClick;
  }
  break;
}
    } else {
      usleep (1000);
    }
```

```
      }

    }
```

Retrieving Button States

The `X11ButtonPane` class provides several methods to retrieve the state of a button object.

`haveClick`
: Returns true or false depending on whether the button has been clicked recently. This means that programs do not need to constantly monitor a button's state. If `haveClick` returns true, then the click state can be cleared with the `clearClick` method (below).

`isClicked`
: This method returns true or false depending on whether the pointer's button 1 is currently pressed within the button's area.

`clearClick`
: Resets a previous clicked state as returned by the `haveClick` method, above.

`text`
: Returns a `String` with the button's text. If the text is displayed on several lines, the method concatenates the lines with a space character '' '' between them.

Resources

The resources that `X11ButtonPane : new` defines by default are stored in the `resources` instance variable, an `AssociativeArray` that is declared in `X11Pane` class. For a description, see the `resources` instance variable documentation. See ⟨undefined⟩ [PaneResources], page ⟨undefined⟩.

`backgroundColor`
: A `String` that contains the color used to draw the button's background. This includes the actual subwindow that receives the button's events from the display server. The resources' default value is '`gray`'.

  If you want the button's background to match its parent window, add a statement like the following *before* attaching the button to its parent window with the `attachTo` method (assuming in this example that the parent window's background is also '`blue`').

```
myButton resources replaceAt "backgroundColor", "blue";
```

  When the program creates the actual X subwindow (again, by calling the `attachTo` method), the Ctalk libraries then check for a resource or instance variable named either, '`background`' or '`backgroundColor`' and uses its value to set the X subwindow's background color.

`borderColor`
: A `String` that contains the name of the color used to draw the label's borders. The default value is '`black`'.

`borderWidth`
: An `Integer` that contains the width of the visible border in pixels. Its default value is '`1`'. To draw buttons without borders, programs can set this resource to '`0`'.

`foregroundColor`
: A `String` that defines the color used for the button's drawable face. Its default color is '`gray`'.

`ftFont`
: A `String` that contains the Fontconfig descriptor of the button's default font. Its default value is '`sans serif-12`'. The button's label inherits the descriptor as its default font when attaching the button (and its `X11LabelPane` label) to its parent window with the `attachTo` method.

  To change the label's font, include a statement like the one in the example, *after* the button has been fully initialized by the `attachTo` method.

```
myButton label resources replaceAt "ftFont", "URW Gothic L-10";
```

  The font, '`URW Gothic L,`' is a Type 1 font included in the Ghostscript `gsfonts` package.

`highlightForegroundColor`
: A `String` that contains the color used to fill the button's face when the pointer is over the button. Its default value is '`gray90`'.

  If you want buttons to display a thicker border and bold font for highlighting, you can include a set of statements like the following in the program, *after* the program attaches the button to its parent pane.

```
myButton label resources replaceAt "highlightForegroundColor",
  (myButton resources at "foregroundColor");
myButton label resources replaceAt "highlightBorderWidth", 2;
```

  To cause the button's text to be emphasized when the button is highlighted, add a line like this one.

```
myButton label resources replaceAt "highlightTextBold", true;
```

  This is only effective if the font library supports boldfacing (i.e., mainly Freetype and Type 1 fonts).

  In these cases, the label subpane doesn't automatically inherit these values, so it's necessary to use the label's declaration, `myButton label,` directly.

`highlightBorderColor`
: A `String` that defines the color to use when drawing a highlighted border. The default is '`black`'.

`highlightBorderWidth`
: An `Integer` that defines the border width in pixels when the button is highlighted. The default value is '`1`'. (That is, the button doesn't use a heavier border for emphasis by default.)

  The `label` subpane inherits this value when the `attachTo` method constructs the button pane.

`highlightHPenColor`
`highlightVPenColor`
`shadowPenColor`
: These are `String` objects that define the colors for the highlighted and shadowed button edges when drawing a beveled button.

`textColor`
: A `String` that contains the color used to draw the button's text. This value is inherited by the button's label widget. The `textColor` resources' default value is '`black`'.

Instance Variables

`bevelEdges`
: A `Boolean` that causes the widget to display beveled edges if true.

`bevelWidth`
: An `Integer` that defines the width of the button's bevelled edges in pixels.

`borderColor`
: A `String` that contains the name of the button's border color when drawing a non-beveled button.

`borderMargin`
: An `Integer` that defines the distance between the border and the pane's edge in pixels, when drawing a non-beveled button.

`borderWidth`
`borderHighlighWidth`
: `Integer` values that determine the width a non-beveled button when it is clicked on and in its non-highlighted state.

`clicked` A `Boolean` that is true when the pointer's Button 1 is pressed while over the widget, and false otherwise.

`highlightHPen`
`highlightVPen`
`shadowPen`
: `Pen` objects that defines the color of the edges' bevels when drawing a beveled widget.

`hover` A `Boolean` that is true if the pointer is over the button's window, false otherwise. Normally, this causes the button to draw its face using highlighted colors, fonts, and borders.

`label` A `X11LabelPane` object that contains a button's text and provides the methods to draw on the button's surface. See ⟨undefined⟩ [X11LabelPane], page ⟨undefined⟩.

`radius` If greater than zero ('0'), this `Integer` defines the radius in pixels of the curves displayed when the button is drawn with rounded corners.

Instance Methods

`attachTo (Object` *parentPane*`, String` *geometry*`)`

Attaches the receiver to the parentPane named by *parentPane*, with the placement and size given by `geometry`. The parent pane should generally be a `X11PaneDispatcher` which directs X events to the correct subpane.

The positioning of subwindows within the parent window is controlled by the arguments to the `attachTo` methods; in particular, these methods accept a geometry specification, as an argument, which has the following format.

```
width[%]xheight[%]+x[%]+y[%]
```

If a dimension does not contain a percent ('%') sign, the dimensions are in pixels. With a percent sign, the dimensions are a fractional percentage of the parent pane's width or height, or a horizontal or vertical distance that places the subwindow's upper left-hand corner that distance from the parent window's upper left-hand corner.

`clearClick (void)`

Resets a button's clicked state as returned by `haveClick` to false.

`draw (void)`

Draws the button and its label on the pane's buffer so the widget can be displayed with the refresh method.

If the program has saved a font specification to the widget's `ftFontVar` instance variable, then this method also selects the font before drawing the widget. If you want a button to display a different font than the surronding window, this is the way to declare and save a font specification.

```
/* The button inherits the ftFont method from X11Pane class. */

button ftFont "DejaVu Sans", FTFONT_ROMAN, FTFONT_MEDIUM, DEFAULT_DPI, 10.0;
button ftFontVar saveSelectedFont;
```

The `X11FreeTypeFont` section describes the parameters that the `X11FreeTypeFont` class uses when selecting fonts. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`haveClicked (void)`

Returns a `Boolean` true or false depending on whether the button has been clicked previously. In that case, the program should call the `clearClick` method to detect further button clicks.

`isClicked (void)`
: Returns a `Boolean` value of true or false depending on whether the pointer's button 1 is currently pressed within the button.

`new (String` *`newObjectName`*`)`
: The `X11ButtonPane` constructor. Creates a `X11ButtonPane` object with the classes' instance variables, and initialized the object's event handlers and `Pen` objects for drawing a beveled button.

`subPaneExpose (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneButtonPress (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneEnter (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneLeave (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneResize (Object` *`subPane`*`, InputEvent` *`event`*`)`
: The class's handlers for events generated by the X server.

`subPaneFocusIn (void)`
`subPaneFocusOut (void)`
: Handlers for focus changes generated by the application. These do not respond to X events and don't require any arguments. These methods are designed to be called by methods like `X11PaneDispatcher : shiftFocus` (e.g., in response to a `Tab` keypress).

`text (void)`
: Returns a `String` with the button's text. If the text is displayed on multiple lines, this method concatenates the lines with a space character (' ') between them.

3.80 `X11CheckBoxPane` Class

A single `X11CheckBoxPane` object draws a checkable box on the main window. The class has methods for initialization, drawing, changing the box's clicked/unclicked state, and the instance data for retrieving the state.

Here is an example program that displays a `X11CheckBoxPane` object.

```
/*
 *  X11CheckBoxPane demo.
 */

/*
 *  To display bevels, uncomment the following line.
 */
/* #define BEVEL */

/*
 * To draw a checkbox with rounded corners, uncomment the following
 * lines and set FRAME_RADIUS and FILL_RADIUS to non-zero values.  The
 * checkboxes use different radius' for the frame and the filled
 * interior of a checked checkbox because most radius are a
```

```
 * significant percent of the boxes' sides, which can make the frame
 * look like a rectangle with rounded corners, and also make the fill
 * rectangle look like an oval, if the shapes use the same radius.
 */
/* #define RADIUS */
/* #define FRAME_RADIUS 0 */
/* #define FILL_RADIUS 0 */

/*
 * To change the width, height, and internal margin of the
 * check box, edit these #defines.  The measurements are in
 * pixels.
 */
#define CHECKBOX_WIDTH   15
#define CHECKBOX_HEIGHT  15
#define CHECKBOX_MARGIN  1
#define CHECKBOX_BORDER  1

#include <ctalk/ctalkdefs.h>

int main (int argv, char **argc) @{
  X11Pane new xPane;
  X11PaneDispatcher new xTopLevelPane;
  X11CheckBoxPane new xCheckBoxPane;
  X11LabelPane new label;
  X11CanvasPane new canvas;
  InputEvent new e;
  Exception new ex;
  Pen new arrowPen;

  xPane ftFontVar initFontLib;

  xPane initialize 0, 0, 300, 300, "Checkbox Demo";
  xPane inputStream eventMask = BUTTONPRESS|BUTTONRELEASE|WINDELETE|EXPOSE;
  xTopLevelPane attachTo xPane;

  /* Note that the geometry string in the, "attachTo," call below
     contains only the X,Y origin of the check box.  In this case, the
     size of the check box is taken from the object's resources, which
     we can adjust any time before, "attachTo," sizes the actual
     drawing surfaces. */
  xCheckBoxPane resources replaceAt "width", CHECKBOX_WIDTH;
  xCheckBoxPane resources replaceAt "height", CHECKBOX_HEIGHT;
  xCheckBoxPane resources replaceAt "margin", CHECKBOX_MARGIN;
  xCheckBoxPane resources replaceAt "borderWidth", CHECKBOX_BORDER;
#ifdef RADIUS
  xCheckBoxPane resources replaceAt "frameRadius", FRAME_RADIUS;
```

```
  xCheckBoxPane resources replaceAt "fillRadius", FILL_RADIUS;
#endif

  label ftFontVar selectFontFromFontConfig "URW Gothic L-12";
  label ftFontVar saveSelectedFont;
  label justify = LABEL_LEFT;
  label text "Please Click";
  label resources replaceAt "borderWidth", 0;

  canvas attachTo xTopLevelPane, "280x280+10+10";
  xCheckBoxPane attachTo xTopLevelPane, "+20%+40%";
  label attachTo xTopLevelPane, "160x25+35%+40";

#ifdef BEVEL
  xCheckBoxPane resources replaceAt "bevel", true;
#endif

  xPane map;
  xPane raiseWindow;

  xPane openEventStream;

  xPane setWMTitle "Checkbox Demo";
  canvas background "white";
  label draw;
  xCheckBoxPane draw;
  arrowPen width = 1;
  arrowPen colorName = "black";

  while (TRUE) @{
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) @{
      e become xPane inputStream inputQueue unshift;
      xPane subPaneNotify e;  /* Call the classes' event handlers. */
      if (ex pending)
        ex handle;

      switch (e eventClass value)
        @{
        case BUTTONPRESS:
          if (e eventData == xCheckBoxPane xWindowID) @{
            if (xCheckBoxPane clicked) @{
              printf ("clicked\n");
            @} else @{
              printf ("unclicked\n");
            @}
          @}
```

```
          break;
        case EXPOSE:
          label draw;
          canvas paneBuffer drawRectangle 10, 10, 260, 260, false,
            1, "black", 0;
          canvas paneBuffer drawLine 110, 45, 70, 100, arrowPen;
          canvas paneBuffer drawLine 70, 100, 70, 90, arrowPen;
          canvas paneBuffer drawLine 70, 100, 81, 97, arrowPen;
          canvas refresh;
          break;
        case WINDELETE:
          xPane deleteAndClose;
          exit (0);
          break;
        default:
          break;
        @}
    @}
  @}
@}
```

Retrieving a `X11CheckBox` Object's State

The `X11CheckBoxPane` class declares the `checked` instance variable, which is either true or false depending on when and how many times a user has clicked on the checkbox. Successive clicks change the `checked` instance variable's state from false to true and back again. The `draw` method draws the checkbox filled or empty depending on the state of the `checked` instance variable.

Here is the portion of the program above that retrieves the checkbox's state. The check box object is named `xCheckBoxPane`. The condition, '`if (e eventData == xCheckBoxPane xWindowID)`' insures that the program uses only pointer clicks that fall within `xCheckBoxPane's` boundaries.

```
    if (e eventData == xCheckBoxPane xWindowID) {
      if (xCheckBoxPane clicked) {
        printf ("clicked\n");
      } else {
        printf ("unclicked\n");
      }
    }
```

Resources

`backgroundColor`
: A `String` that contains the color name of the checkbox's background. The default is '`white`'.

`bevel` A `Boolean` that determines whether to draw beveled edges on the checkbox's interior. The default is '`false`'.

`borderWidth`
An `Integer` that specifies the width in pixels of the checkbox's border. The default is 1px.

`clickColor`
A `String` that contains the color name of the checkbox's border and interior when checked. The default is '`darkslategray`'.

`fillRadius`
`frameRadius`
These are `Integer` objects which, if nonzero, cause the checkbox to be drawn with rounded corners. There is a separate dimension for the margin and the checkbox interior because the radius comprises a significant amount of a checkbox's size, so the margin might appear like a rectangle with rounded corners, while the interior might look like a circle or oval, if the shapes use the same corner radius.

`darkShadowColor`
`lightShadowColor`
The names of the colors of the light and dark shadows that the checkbox displays inside its border if the '`bevel`' resource is true.

`height`

`width` These are `Integer` values that contain the width and height of the checkbox, in pixels. Their default value, '`15px`', is set when the checkbox object is created, and may be changed before the checkbox is attached to its parent pane. If the checkbox's `attachTo` method is given a geometry that also contains a width and height, then this latest dimension setting determines the checkbox's size when displayed.

`margin` The distance in pixels between the inner edge of the border and the edge of the check in a filled checkbox. The default is 1px.

Instance Variables

`clicked` A `Boolean` that changes between true and false every time the check box is clicked. The variable's value also determines whether the checkbox's interior is filled.

Instance Methods

`attachTo (Object` *parentPane*`, String` *geometry*`)`
Attaches the receiver `X11CheckBoxPane` to its *parentPane* using the dimensions given by *geometry*. The method also creates the drawing surfaces with the correct dimensions.

The *geometry* argument may optionally omit the checkbox's width and height, which causes the method to use the width and height given by the checkbox object's resources. For example, if the program contains an expression like the following:

```
xCheckBoxPane attachTo xTopLevelPane, "+25%+25%";
```

Then the method will create the checkbox with the dimensions preset by the object's '`width`' and '`height`' resources. If the checkbox's dimensions are determined in this manner, a program my change the checkbox's size any time before the `attachTo` method is called. However, if the program contains a statement like the following,

```
xCheckBoxPane attachTo xTopLevelPane, "20x20+25%+25%";
```

Then the program uses the '`20x20`' size given as the argument, which is the most recent dimension that the program has defined..

`draw (void)`
: This method draws the checkbox. It uses the *clicked* instance variable state to determine whether to draw a filled checkbox, as well as the classes' resources that have been set in the object to determine check checkbox's style; i.e., the checkbox's size, whether its edges are beveled or rounded, and the checkbox's color.

`new (String` *paneName*`)`
: The `X11CheckBoxPane` constructor. Creates a new `X11CheckBox` with the name given by *paneName*, and makes it available to the method that contains the statement, or to the entire program if the new `X11CheckBoxPane` is declared in a global scope. The method also initializes the event handlers and resources that are defined in the class.

`onClick (Object` *subPane*`, InputEvent` *event*`)`
: This method receives a '`BUTTONPRESS`' event from the display system, and toggles the receiver checkbox's `clicked` instance variable between true and false. The method changes the `clicked` state only in response to '`BUTTONPRESS`' events; a corresponding '`BUTTONRELEASE`' event is ignored.

3.81 `X11LabelPane` Class

`X11LabelPane` objects display text in graphical windows. The class provides instance variables and methods that control the label's font, border, justification, whether the label is highlighted when the pointer is over it, and formatting of single and multiple line labels.

Formatting label text for multiple line labels, as well as controlling the text's justification, is described in the subsections below.

Here is a simple example program that displays a X11LabelPane widget.

```
#include <ctalk/ctalkdefs.h>

#define FTFONTS    /* Comment this line out to use X fonts. */
#define MULTILINE  /* Comment out to display a single line label. */
#define ROUNDED    /* Comment out to draw borders with straight edges. */
```

```
int main (int argv, char **argc) {
  X11Pane new xPane;
  X11PaneDispatcher new xTopLevelPane;
  X11LabelPane new xLabelPane;
  InputEvent new e;
  Exception new ex;

#ifdef FTFONTS
  xPane ftFontVar initFontLib;
#endif

  xLabelPane resources replaceAt "backgroundColor", "white";

  xPane initialize 0, 0, 300, 300;
  xPane inputStream eventMask =
    WINDELETE|EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY;
  xTopLevelPane attachTo xPane;
  xLabelPane attachTo xTopLevelPane, "150x150+25+50";
  xPane map;
  xPane raiseWindow;

  xPane openEventStream;

#ifndef FTFONTS
  xLabelPane font "-*-helvetica-medium-r-*-*-*-120-*-*-*-*-*-*";
#endif

#ifdef MULTILINE
  xLabelPane multiLine "Labels\ncan occupy\nmultiple\nlines.";
#else
  xLabelPane text "A Label";
#endif

#ifdef ROUNDED
  xLabelPane radius = 10;
#endif

  xLabelPane draw;

  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;
      xPane subPaneNotify e;  /* Call the classes' event handlers. */
      if (ex pending)
ex handle;
```

```
        switch (e eventClass value)
        {
        case WINDELETE:
           xPane deleteAndClose;
          exit (0);
          break;
        default:
          break;
        }
      }
    }
}
```

Multiple Line Labels

If the label's text contains embedded newline '`\n`' characters, and the text is defined using the `multiLine` method, below, then the text is display on multiple lines, with the breaks appearing where the newline characters appear in the text. The text is displayed centered vertically, and justified as described in the next section.

For single line labels, use the `text` method with a `String` argument to fit on one line.

Justification

The `X11LabelPane` widget contains a simple layout engine that can center label text, or justify it against the right or left margins.

The text's justification is determined by the setting of the `justification` instance variable, which can have one of three values, which are defined by the following macros:

```
LABEL_LEFT
LABEL_RIGHT
LABEL_CENTER
```

These macros are defined in the `ctalk/ctalkdefs.h` include file. To use them, add the line

```
#include <ctalk/ctalkdefs.h>
```

to the beginning of the source module.

The widget calculates the right and left margins as the following:

```
(aLabel margin) + (aLabel highlightBorderWidth) + (aLabel padding)
```

The highlighted border width is included regardless of whether the border actually is drawn or the label is hightlighted, so text alignment remains constant if a program draws and then obscures the border, or when the pointer passes over the label.

Resources

The resources that `X11LabelPane : new` defines by default are stored in the `resources` instance variable, an `AssociativeArray` that is declared in `X11Pane` class. For a description, see the `resources` instance variable documentation. See ⟨undefined⟩ [PaneResources], page ⟨undefined⟩.

`ftFont`
: A `String` that contains the text font's Fontconfig font descriptor. The default is '`sans serif-12`'.

  For more information about Fontconfig descriptors, refer to the `X11FreeTypeFont` class. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`backgroundColor`
: A `String` that contains the pane's background color. This is the color of the pane's X subwindow, so it is generally useful to set this to the color of the surrounding window, if you want the label to blend in with its surrounding drawing area.

`borderColor`
: A `String` that contains the name of the pane's border color. The default is `black`

  An `Integer` that contains the line width the pane's border. The default is '`1`'.

`foregroundColor`
: A `String` that contains the name of the color that fills the pane's visible area. The default is '`white`'.

`grayedColor`
: The color of the text and border when the pane is grayed. The default is `gray`.

`highlightBorderColor`
: The color of the border when the pane is highlighted. The default is '`black`'.

`highlightBorderWidth`
: The width of the border when the pane is highlighted the default is '`2`'.

`highlightForegroundColor`
: A `String` that contains the color name of the pane's visible area when it the pane is in a highlighted state, which is generally when the pointer is over the pane's window area. The default is '`orange`'.

`highlightTextBold`
: A `Boolean` that causes the label to display its text in a bold font when the label is highlighted, if the font library supports it. The default is '`false`'.

`highlightTextColor`
: A `String` that contains the text color when the pane is in a highlighted state, generally when the pointer is over the pane. The default is `black`.

`textColor`
: A `String` that contains the text color when the pane is not highlighted. The default is '`black`'.

`xFont` A `String` that contains the text fonts X Logical Font Descriptor. Refer to *xfontsel(1)* and *xlsfonts(1)* for information about XLFD's. The default is '`fixed`'.

Instance Variables

`border` A `Boolean` that controls whether the label displays a border. The default is '`true`'.

`borderColor`
A `String` that contains the name of the border's color if the label displays a border. The default is '`black`'.

`borderWidth`
An `Integer` that defines the width of the border in pixels when the widget is not highlighted. The default is '`2`'.

`grayed` A `Boolean` that determines whether to draw the widget in a grayed, or inactive, state. The state has no other effect for a `X11LabelPane` object. It can indicate the state of other controls in the window. The default is `false`.

`grayedColor`
A `String` that contains the name of the color to draw the widget when it is grayed. The default is is '`gray`'.

`haveMultiLine`
A `Boolean` that indicates whether the widget's text is defined using the `multiLine` method, and indicates that the widget should format the label's text to span multiple lines.

`highlightBackgroundColor`
A `String` that contains the color to draw a highlighted label's background.

`highlightBorderColor`
A `String` that contain's the name of a highlighed border's color. The default is '`black`'.

`highlightBorderWidth`
An `Integer` that determines the width in pixels of the label's border when the widget is highlighed. The default is two pixels.

`highlightTextColor`
A `String` that contains the name of the text color when the widget is drawn highlighted. The default is '`black`'.

`justify` An `Integer` that determines whether to draw the label's text centered horizontally, or justified on the right or left margins.

This variable recognizes three values, which are #defines in the `ctalk/ctalkdefs.h` include file. The definitions are:

```
LABEL_LEFT
LABEL_CENTER
LABEL_RIGHT
```

To use these macros, add the statement

```
#include <ctalk/ctalkdefs.h>
```

near the top of the source module.

`leading` An `Integer` that defines the space between lines of a multi-line label. This variable only affects text drawn using scalable fonts of the Freetype library. X11 bitmap fonts contain the leading in the character glyphs.

`margin` An `Integer` that defines the distance in pixels between the border and the edges of the `X11LabelPane` object.

`padding` An `Integer` that defines the minimum distance in pixels between the inner edge of the border and the pane object's text.

`radius` An `Integer` that describes the radius to round the corners of the border with. If the value is '`0`' (the default), then draw borders with straight edges.

`textColor`
A `String` that defines the name of the text color when the widget is not highlighted. The default is '`black`'.

`textLine` A `String` that contains the label's text if the text is to appear on a single line. The default is an empty string.

`textLines`
An `Array` that contains a multi-line widget's text, as set using the `multiLine` method.

Instance Methods

`draw (void)`
`drawButtonLayout (void)`
Draws the text of the label, and updates the visible label on the display. The instance variables in this class and the text font's class allow programs to adjust how the text is laid out.

The `drawButtonLayout` method is similar, but it uses a different algorithm that is more suited to laying out text on button labels.

If the widget has saved a font specification to its `ftFontVar` instance variable, then this method also selects the font before drawing the label. If you want a label to have a different font than the rest of the window, this is how to declare and save a font specification.

```
/* The label inherits the ftFont method from X11Pane class. */

label ftFont "DejaVu Sans", FTFONT_ROMAN, FTFONT_MEDIUM, DEFAULT_DPI, 10.0;
label ftFontVar saveSelectedFont;
```

The `X11FreeTypeFont` section describes the parameters that the `X11FreeTypeFont` class uses when selecting fonts. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`new` Creates a new `X11LabelPane` object, initializes the object's instance variables and event handlers, and sets the pane's default foreground and background colors.

`selectFont (void)`
A convenience method that is synonymous with, `self ftFontVar selectFont`.

`subPaneExpose (Object` *`subPane`*`, InputEvent` *`event`*`)`
The widget's Expose event handler. When the widget receives an Expose event, this method redraws the widget.

`subPaneEnter (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneLeave (Object` *`subPane`*`, InputEvent` *`event`*`)`
The widget's EnterNotify and LeaveNotify event handlers. These methods set the `subPane's highlight` instance variable to true or false (if the widget is configured to accept input focus), then calls the `draw` method to redraw the label.

`subPaneResize (Object` *`subPane`*`, InputEvent` *`event`*`)`
Handles resize events received from the display server.

`font (String` *`fontdesc`*`)`
Sets the X11 font to use for the label's text. When drawing using Freetype scalable fonts, the widget uses the `ftFont` method from `X11CanvasPane`.

`text (String` *`labelText`*`)`
Sets the text of a single-line label.

`text (String` *`labelText`*`)`
Sets the receiver's `textLine` instance variable for single-line labels.

`multiText (String` *`labelText`*`)`
Divides the text given as the argument into lines where newline characters '`\n`' appear in the text, and sets the elements of the receiver's `textLine` instance variable to one line for each array element. Also sets the `haveMultiLine` instance variable to `true`.

3.82 `X11ListPane` Class

The `X11ListPane` class displays a scrolling list of items. Clicking the left pointer buttons selects an item, which is available to the program for as long as the `X11ListPane` object is available.

The list may be larger than the pane can display at one time, and the list may be scrolled using the scrollbar on the left edge of the pane, or the `up` and `down` arrow keys, or the `C-p` and `C-n` keys.

Here is a program that displays a `X11ListPane` object, and prints the selected item on the terminal when the program exits.

Clicking on an item using the left mouse button (or the single button on pointing devices that have only one button) selects that item. Clicking on the item again de-selects the item, or if a different item is clicked, then that item is selected.

If the user presses the *Shift* key while clicking on an item, it is selected in addition to any items that have already been selected.

Pressing the *Ctrl* key while clicking selects every item in a range between the first selected item in the list and the last selected item.

```
/* listpane.ca, a X11ListPane demonstration. -*-c-*- */

#define N_ITEMS 30

#include <ctalk/ctalkdefs.h>

int main (int argv, char **argc) {
  X11Pane new xPane;
  X11PaneDispatcher new dispatcher;
  X11ListPane new listPane;
  X11LabelPane new label;
  X11ButtonPane new button;
  InputEvent new e;
  Integer new nEvents;
  Integer new verbose;
  Integer new i;
  X11Cursor new cursor;
  String new itemText, itemN;
  List new itemTextOut, itemNOut;

  xPane resources atPut "backgroundColor", "blue";
  label resources replaceAt "backgroundColor", "blue";
  label resources replaceAt "foregroundColor", "blue";
  label resources replaceAt "highlightForegroundColor", "blue";
  label resources replaceAt "textColor", "white";
  label resources replaceAt "highlightTextColor", "white";
  label border = false;

  xPane inputStream eventMask = EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY| \
    KEYPRESS|KEYRELEASE|WINDELETE|BUTTONPRESS|BUTTONRELEASE|MOTIONNOTIFY| \
     SELECTIONCLEAR|SELECTIONREQUEST|STRUCTURENOTIFY;
  xPane initialize 300, 400;
  dispatcher attachTo xPane;
  label attachTo dispatcher, "100x70+10+30%";
  button attachTo dispatcher, "60x45+50%+-75";
  listPane attachTo dispatcher, "-15x-90+100+24";

  xPane ftFontVar initFontLib;
```

```
  listPane selectFont;
  label selectFont;

  for (i = 1; i < N_ITEMS; ++i) {
    listPane add "item " + i asString;
  }

  xPane map;
  xPane raiseWindow;

  xPane openEventStream;

  xPane setWMTitle "X11TextListPane Demo";
  label multiLine "Select\nan\nItem";
  button label text "Done";

  listPane draw;
  listPane refresh;

  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;

      xPane subPaneNotify e;

      switch (e eventClass value)
        {
        case EXPOSE:
  listPane draw;
          listPane refresh;
          break;
        case WINDELETE:
          xPane deleteAndClose;
          exit (0);
          break;
        default:
          if (button haveClick) {
            button clearClick;
            xPane deleteAndClose;
    if (listPane nItemsSelected > 1) {
      i = 0;
      listPane selectedItems itemTextOut;
      listPane selectedItemsN itemNOut;
      itemNOut map {
itemN = self;
/* This is a convenient way to retrieve
```

```
   the i'th item in the itemTextOut list. */
itemText = *(itemTextOut + i);
printf ("%d: %s\n", itemN, itemText);
++i;
      }
    } else {
      printf ("%d: %s\n",
      listPane selectedItemN,
      listPane selectedItemText);
    }
            exit (0);
          }
        break;
        }
    }
  }
}
```

Retrieving Selected Items

The method `nItemsSelected` returns the number of list items that are currently selected. If only one item is selected, a program use the `selectedItemN` and `selectedItem` methods to retrieve the selected item's position in the list, and its text. This example shows how to print the list's selection when only one item is selected.

```
if (myListPane nItemsSelected == 1) {
   printf ("%d: %s.\n", myListPane selectedItemN,
     myListPane selectedItemText);
}
```

If there are multiple items selected (i.e., `nItemsSelected` returns an `Integer` greater than 1), a program can retrieve and display all of the results the `selectedItems` and `selectedItemsN` methods, and display all of the results using an expression like this (although this example doesn't print both sets of results at once).

```
List new itemTextOut, itemNOut;

...

if (myListPane nItemsSelected > 1) {
   myListPane selectedItems itemTextOut;
   myListPane selectedItemsN itemNOut;
   itemNOut map {
     printf ("%d\n", self)
   }
   itemTextOut map {
```

```
        printf ("%s\n", self)
      }
    }
```

Resources

`borderColor`
: A `String` that contains the named color of the pane's border. The default is '`gray`'.

`borderWidth`
: An `Integer` that defines the width in pixels of the pane's border The default is '`1`' pixel.

`font`
: A `String` that contains the Fontconfig descriptor of the font used to display the list items. The default is '`sans serif-12`'.

`keyScrollIncrement`
: The distance in pixels the list should scroll up or down when the user scrolls the list. The default is 4 pixels.

`leftMargin`
: An `Integer` that defines the width in pixels of between the left edge of pane's list area and the left margin of the item text. The default is 4 pixels.

`selectColor`
: A `String` that contains the background color of selected items. The default is '`lightblue`'.

`selectStyle`
: An `Integer` that defines whether a select bar extends to the right edge of an item's text, or to the right margin of the pane. The class recognizes the #defines '`LIST_SELECT_PANE`' and '`LIST_SELECT_ITEM`', which are defined in `ctalkdefs.h`. The default is '`LIST_SELECT_PANE`', so to change it to only highlight an item's text, a program should contain the line,

```
myListPane resources replaceAt "selectStyle", LIST_SELECT_ITEM;
```

`textColor`
: A `String` that contains the color of the list text. The default is '`black`'.

`thumbColor`
: A `String` that defines the color of the scroll bar's thumb. The default is '`darkgray`'.

`thumbMargin`
: An `Integer` that defines the distance in pixels between the thumb and its channel. The default is 2 pixels.

`vAlignHint`
: An `Integer` that helps display items vertically centered in their layout boxes. While X11ListPane objects generally can approximate how font renderers align

text within their layout boxes, fonts and even different library versions may vary this. Increasing this number shifts the item's text downward within the box when it is displayed. Smaller numbers (including negative numbers) shift the text upward. The default is 2px.

Instance Variables

`buttonState`
An `Integer` that records whether any of the pointer buttons are pressed. The possible values may be any combination of the following definitions, or none ('0').

```
BUTTON1MASK     (1 << 0)
BUTTON2MASK     (1 << 1)
BUTTON3MASK     (1 << 2)
```

`items` A `List` that contains the pane's list items. Each item uses an `ItemBox` object to store its information. The definition of the `ItemBox` class (which is also defined in the `X11ListPane` class library) is:

```
String class ItemBox;

ItemBox instanceVariable org Point 0;
ItemBox instanceVariable ext Point 0;
ItemBox instanceVariable size Point 0;
```

`iInit` An `Integer` that simply causes the `drawItemList` method to wait for class to initialize the fonts it needs.

`scrollWidth`
An `Integer` that defines the width in pixels of the scroll bar. It is initialized from the value of the `scrollWidth` resource when the `X11ListPane` object is attached to its parent window by the `attachTo` method. The default is 10 pixels.

`scrollWidth`
An `Integer` that records whether a `Shift` or `Ctrl` key is currently being pressed. The state is used when selecting multiple items, and may be either or both of the following, or none.

```
shiftStateShift     (1 << 0)
shiftStateCtrl      (1 << 1)
```

`thumbExt`

`thumbOrg` These are `Integers` that define the position and height of the scroll thumb in pixels. The value of `thumbExt` is calculated from the percentage of the list

contents that are visible in the window at any one time. The value of `thumbOrg` records the vertical position of the thumb where the user moved it using the pointer.

`viewStartY`
: An `Integer` that contains the distance from the top edge of the first list item (which is stored in the item's `org` instance variable) to the top of the portion of the list that is visible in the pane's window.

Instance Methods

`add (String itemText)`
: Addes a list item using *itemText* as its contents. New items are added sequentially to the end of the item list. The method also calculates the item's width and height in pixels, and the position of the top and bottom edges relative to the top edge of the zero'eth item in the list. Each item is stored in the pane's `items` list using an `ItemBox` object, which stores each item's dimensions as well as the item's contents. The `ItemBox` class is defined in the `X11ListPane` class library.

`attachTo (Object parentPane, String geomspect)`
: Attaches the `X11ListPane` object to *parentPane*, with the dimensions given by *geomspect*. Also initializes the object's drawing surfaces and the actual window within the parent window.

  The *geomspec* argument defines the width, height and position of the pane within the parent window. For a description of its contents, refer to the `X11PaneDispatcher` class See ⟨undefined⟩ [X11PaneDispatcher], page ⟨undefined⟩.

`calcThumb (void)`
: Calculates the height of the scroll thumb as a proportion of the percentage of the list contents that can be displayed on the pane at any one time, and stores the value in the pane's `thumbExt` instance variable (which is itself a `Point` object, so a program would retrieve the thumb's height like this).

```
thumbHeight = myListPane thumbExt y;
```

`draw (void)`
: This method draws the the pane's scroll frame and borders, and calls the `drawItemList` and `drawThumb` methods to display the list contents.

`drawItemList (void)`
: This method is called by the `draw` method to display the visible items on the pane's window.

`drawThumb (void)`
: Another method that is called by the `draw` method, this method draws the scroll thumb within the scroll bar.

`new (String` *paneName*`)`
: The `X11ListPane` constructor. This method creates a `X11ListPane` object with the name given by *paneName*, and initializes the object's resources and event handlers.

`nItemsSelected (void)`
: Returns an `Integer` with the number of items that are currently selected.

`onClick (Object` *subPane*`, InputEvent` *event*`)`
`onEnter (Object` *subPane*`, InputEvent` *event*`)`
`onExpose (Object` *subPane*`, InputEvent` *event*`)`
`onKey (Object` *subPane*`, InputEvent` *event*`)`
`onLeave (Object` *subPane*`, InputEvent` *event*`)`
`onMotion (Object` *subPane*`, InputEvent` *event*`)`
`onResize (Object` *subPane*`, InputEvent` *event*`)`
: These are the `X11ListPane` object's event handlers, which receive events from the display server. They are initialized by the `new` method.

`selectedItemN (void)`
: This method returns an `Integer` with the index of the selected item.

  The indexes are numbered from zero, which is the index of the first item.

  If no item is selected, this method returns '`-1`'.

`selectedItems (List` *itemsOut*`)`
: Pushes a `String` object with the text of each currently selected item onto the *itemsOut* list.

`selectedItemsN (List` *itemsOut*`)`
: Pushes an `Integer` with the numerical position of each selected item onto *itemsOut*. The indexes start with 0 reprsenting the first item, 1 representing the second item, and so on.

`selectedItemText (void)`
: Returns a `String` with the text of the selected item. If no item is selected, the method returns and empty `String`.

`selectFont (void)`
: Selects the font to display the list items, and saves the font parameters in the object's `ftFontVar` instance variable, which is inherited from `X11Pane` class.

3.83 `X11MessageBoxPane` Class

A `X11MessageBoxPane` object displays a popup window that contains a text message and a button that closes the window.

The button's text is displayed by a `X11LabelPane` object, and the button is a `X11ButtonObject`. Generally, `X11MessageBoxPane` objects override any input to the program that opened them, so they contain their own event loop, in the `show` method.

Here is an example of a program that creates its own `X11MessageBoxPane` object, and then opens it when the user clicks on the main window's button.

```
/* messagebox.ca - X11ButtonPane Demonstration -*-c-*- */

#include <ctalk/ctalkdefs.h>

/* See the X11FreeTypeFont section of the the Ctalk reference. */
#define FTFONT_BOLD   200
#define FTFONT_MEDIUM 100

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
  X11ButtonPane new button;
  X11LabelPane new label;
  X11MessageBoxPane new messageBox;
  InputEvent new e;

  label textColor = "white";
  label canFocus = false;
  label borderWidth = 0;

  label ftFontVar notifyLevel XFT_NOTIFY_ERRORS;

  mainWindow backgroundColor = "blue";
  label resources replaceAt "backgroundColor", "blue";
  button resources replaceAt "backgroundColor", "blue";
  messageBox resources replaceAt "backgroundColor", "blue";

  mainWindow initialize 255, 200;
  mainWindow inputStream eventMask =
    EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR

  dispatcher attachTo mainWindow;
  button attachTo dispatcher, "110x50+73+100";
  label attachTo dispatcher, "177x80+34+15";
  messageBox attachTo dispatcher, "300x200";

  mainWindow map;
  mainWindow raiseWindow;

  mainWindow openEventStream;

  mainWindow setWMTitle "X11MessageBoxPane Demo";

  label ftFontVar initFontLib;

  label multiLine "X11MessageBoxPane\nDemo";
```

```
  label resources replaceAt "textColor", "lightgray";
  label resources replaceAt "foregroundColor", "blue";
  label resources replaceAt "borderColor", "blue";

  button label multiLine "Open\nMessageBox";

  button label resources replaceAt "highlightForegroundColor", "gray90";

  /* The program uses the "replaceAt" method because the key/value
     entry for "backgroundColor" the X11MessageBoxPane : new method
     has alread created an entry for backgroundColor. */
  messageBox resources replaceAt "backgroundColor", "blue";
  messageBox resources replaceAt "foregroundColor", "blue";
  messageBox resources replaceAt "messageText",
            "Hello, messageBox!\nYour message text here.";

  button draw;
  button refresh;
  label draw;
  label refresh;

  while (TRUE) {
    mainWindow inputStream queueInput;
    if (mainWindow inputStream eventPending) {
      e become mainWindow inputStream inputQueue unshift;

      mainWindow subPaneNotify e;

      switch (e eventClass value)
{
case EXPOSE:
  button subPaneExpose (button, e);
  label subPaneExpose (label, e);
  break;
case BUTTONRELEASE:
  if (button haveClick) {
    button clearClick;
  }
  button highlight = false;
  button clicked = false;
  while (mainWindow inputStream eventPending)
    e become mainWindow inputStream inputQueue unshift;
  button draw;
  button refresh;
  messageBox show;
  break;
case WINDELETE:
```

```
        mainWindow deleteAndClose;
      exit (0);
      break;
    }
        } else {
          usleep (1000);
        }
      }
    }
```

Resources

`background`
: The window's background color, and the background color of the message label and the close button. The value is a `String` that contains a X11 named color. The default is '`gray`'. For a list of X11 color names, see *showrgb(1)*.

`buttonText`
: A `String` that contains the text that appears in the button. The default is '`Ok`'.

`foregroundColor`
: A `String` with the name of the color that the messagebox window is filled with. The default is '`gray`'.

`geometry`
: A `String` that contains the message window's size, and, optionally, its position. The value is set to the dimensions that are given as arguments to the `attachTo` method.

`iconID`
: An `Integer` that contains the identifier to the icon to be displayed. Ctalk defines the following constants for icons in `ctalkdefs.h` and in the graphics libraries.

```
ICON_NONE
ICON_CAUTION
ICON_INFO
ICON_QUESTION
ICON_STOP
```

  The default is '`ICON_NONE`', which causes no icon to be displayed. Any other constant causes an icon to be displayed to the left of the message text. The widget adjusts the size and position of the text automatically to provide space for the icon.

`messageColor`
: A `String` that contains the color of the message text. The default is '`black`'. The message box sets the label's `textColor` resource to this value.

`messageFont`
: A `String` that contains the Fontconfig font descriptor. The default is '`sans serif-12`'. The pane uses this value to set the label's '`fcFont`' resource.

`messageText`
: A `String` that contains the text of the message to be displayed. The default is '`Sample message text.`'.

`pad`
: An `Integer` that contains the space in pixels betwween the button and label widgets, and between each widget and the edge of the window. The default is 10 pixels.

`titleText`
: A `String` that contains the message window's title text. The default is '`Message`'.

Instance Variables

`button`
: A `X11ButtonPane` object that closes the window when clicked. The button's default text is, '`Ok`'. Programs can change the text with a statement like the following, which changes the button's text to, '`Close`'.

```
myMessageBox resources atPut "buttonText", "Close";
```

`initialized`
: A `Boolean` that is `true` if the pane has already been initialized. When a message box is closed, it is not deleted - it is simply unmapped from the display. This variable is set to true so that the message box is initialized only once.

`label`
: A `X11LabelPane` that displays the message box's text. As configured, the label is the same color as the main message box window, and uses no borders. The label's text is controlled by the resource, '`messageText`'. To set the message text, use a statement like the following.

```
myMessageBox resources atPut "messageText", "Today Is\nJanuary 3rd";
```

The class draws the label using multiline formatting, so if you want the text to appear on multiple lines, include the sequence '`\n`' in the text (that's a backslash and the character, '`n`', not a literal newline).

`keyState`
: An `Integer` which records whether the `Tab` or `Enter` keyboard shortcuts have been pressed. Pressing `Tab` highlights the pane's button, and pressing `Enter` withdraws the pane's window from the display, and the `show` method returns to the program that opened the message box.

`mainWindowPtr`
: A `Symbol` that contains a pointer to the main window.

Instance Methods

`attachTo (Object` *`parentPane`*`, String` *`geometry`*`)`
: Sets the message box's dimensions and creates the object's buffers and main window. Unlike other `X11Pane` subclasses, this method does not immediately attach the message box's object to its parent pane, so the message box isn't displayed until the program invokes the `show` or `showManaged` method.

`draw (void)`
: Draws the label and button subpanes in the message box's buffers and the buffers of the button and label subpanes. After calling this method, the `refresh` method makes the changes visible.

`fill (String` *`colorName`*`)`
: A convenience method that fills the pane window with the color given as the argument.

`initWidgets (void)`
: Initializes the message box's button and label subpanes when the pane's window is first opened. When constructed this method sets the `initialized` instance variable to '`true`', so when the message box appears again, this method is a no-op.

`new (String` *`newPaneName`*`)`
: The X11MessageBoxPane constructor.

`refresh (void)`
: After the method *draw* renders the main window and the label and the button subwindow's contents on the pane objects' buffers, this method updates the visible window with the contents of the buffers.

`show (void)`
: This method displays the message box, waits for events from the display system, and closes the message box when its button is clicked, then returns to the calling program.

`show (X11ButtonPane` *`buttonPane`*`)`
: This method is similar to the `show` method, but it takes as its argument the pane object from the main window that caused the message window to open.

  This method sets the main window's button to be unclicked and then redrawn.

  This method manages `X11ButtonPane` objects specifically, but it can be subclassed to manage any pane class object.

`subPaneButton (Object` *`subPane`*`, InputEvent` *`event`*`)`
: This is the event handler for button press and button release events from the display hardware. When a button is clicked on, the message box draws the button in its clicked state. When the button is released, the pane draws the button unclicked, then unmaps the message box window from the display, and returns to the main program.

`subPaneExpose (Object *subPane*, InputEvent *event*)`
: Handles drawing the message box when the program receives an Expose event from the display. This method calls the message box's `draw` and `refresh` methods.

`withdraw (void)`
: Unmaps the message box's window from the display after the pane object receives a button release event while processing the `show` method. When the `show` or `showManaged` method receives the button release event, this method is called to remove the pane's window from the display, and the `show` or `showManaged` method returns to the main program.

3.84 `X11ScrollBarPane` Class

A `X11ScrollBarPane` object draws a basic scroll bar and allows the user to move the scroll thumb by clicking and dragging.

Applications can set and read the scroll thumb's size and position via the object's instance variables. In particular, the `thumbHeight` instance variable sets the vertical size of the scroll thumb, and the `thumbDimensions` method sets the coordinates and height of the thumb when sliding it to a new position.

The scroll bar's `frame` and `thumb` instance variables are both `Rectangle` objects. Applications can use the `Rectangle` class's methods, like `dimension,` to set the scroll position, and can read the position using the `Rectangle` object's instance variables. See ⟨undefined⟩ [Rectangle], page ⟨undefined⟩.

Here is an example program that demonstrates the basic steps needed to draw a scroll bar.

```
int main (int argv, char **argc) {
  Integer new xWindowSize;
  Integer new yWindowSize;
  X11Pane new xPane;
  X11PaneDispatcher new xTopLevelPane;
  X11ScrollBarPane new xScrollBarPane;
  X11CanvasPane new xCanvasPane;
  InputEvent new e;
  Exception new ex;
  Application new scrollDemo;
  String new pctstr;
  Integer new strWidth;

  scrollDemo enableExceptionTrace;
  scrollDemo installExitHandlerBasic;
  scrollDemo installAbortHandlerBasic;

  xWindowSize = 400;
  yWindowSize = 400;

  xPane ftFontVar initFontLib;
```

```
  xPane ftFont "DejaVu Sans", 0, 0, 0, 48.0;

  xPane initialize xWindowSize, yWindowSize;
  xPane inputStream eventMask = WINDELETE|EXPOSE|BUTTONPRESS|BUTTONRELEASE|MOTIONNOTIF
  xTopLevelPane attachTo xPane;
  xScrollBarPane attachTo xTopLevelPane, "20x100%+0+0";
  xCanvasPane attachTo xTopLevelPane, "380x100%+20+0";

  xPane map;
  xPane raiseWindow;

  xPane openEventStream;

  xScrollBarPane background "lightblue";
  xScrollBarPane refresh;

  xCanvasPane background "navy";
  xCanvasPane foreground "navy";
  xPane ftFontVar namedX11Color "white";
  xCanvasPane clear;
  pctstr printOn "%0.2f%%", xScrollBarPane percent * 100.0;
  strWidth = xPane ftFontVar textWidth pctstr;
  xCanvasPane putStrXY (380 / 2) - (strWidth / 2), (400 / 2), pctstr;
  xCanvasPane refresh;

  while (TRUE) {

    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;

      if (ex pending)
   ex handle;

      xPane subPaneNotify e;

      switch (e eventClass value)
  {
case EXPOSE:
  xCanvasPane paneBuffer clear;
  pctstr printOn "%0.2f%%", xScrollBarPane percent * 100;
  strWidth = xPane ftFontVar textWidth pctstr;
  xCanvasPane putStrXY (380 / 2) - (strWidth / 2), (400 / 2), pctstr;
  xCanvasPane refresh;
  break;
  case WINDELETE:
     xPane deleteAndClose;
```

```
          exit (0);
          break;
         default:
        xCanvasPane paneBuffer clear;
        pctstr printOn "%0.2f%%", xScrollBarPane percent * 100;
        strWidth = xPane ftFontVar textWidth pctstr;
        xCanvasPane putStrXY (380 / 2) - (strWidth / 2), (400 / 2), pctstr;
        xCanvasPane refresh;
           break;
        }
          }
        }
      }
```

Instance Variables

`arrowCursor`
: A `X11Cursor` object that contains the resource ID of the widget's arrow cursor. See ⟨undefined⟩ [X11Cursor], page ⟨undefined⟩.

`bevel`
: A `Boolean` that determines whether the scroll thumb is drawn as a beveled widget.

`dragging`
: A `Boolean` object that is True while the right mouse button is clicked while over the scroll thumb. See ⟨undefined⟩ [Boolean], page ⟨undefined⟩.

`frame`
: A `Rectangle` object that contains coordinates of the scroll bar frame. See ⟨undefined⟩ [Rectangle], page ⟨undefined⟩.

`frameWidthVar`
: An `Integer` that contains the width of the scrollbar's visible frame in pixels.

`framePen`
: A `Pen` object that sets the line color and width of the scroll bar frame. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

`grabCursor`
: A `X11Cursor` object that contains the resource ID of the widget's grab cursor. See ⟨undefined⟩ [X11Cursor], page ⟨undefined⟩.

`lastY`
: An `Integer` object that contains the previous PointerMotion event's y coordinate while dragging. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`margin`
: An `Integer` that defines the distance in pixels between the pane's edge and the outer edge of the scrollbar frame.

`mouseOver`
: A `Boolean` that is true if the pointer is over the scrollbar frame, false otherwise.

`padding`
: An `Integer` that defines the distance in pixels between the inner edge of the scrollbar frame and the thumb.

`thumb`
: A `Rectangle` object that contains coordinates of the scroll thumb. See ⟨undefined⟩ [Rectangle], page ⟨undefined⟩.

`thumbBackgrondColor`
: A `String` that contains the name of the X11 color used to draw the thumb.

`thumbErasePen`
: A `Pen` object that sets the color and line width of the scroll thumb background when animating the thumb. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

`thumbHeight`
: An `Integer` that sets the height of the scroll thumb. See ⟨undefined⟩ [Integer], page ⟨undefined⟩.

`thumbPen` A `Pen` object that sets the color and line width of the scroll thumb. See ⟨undefined⟩ [Pen], page ⟨undefined⟩.

`thumbPix` A `X11Bitmap` that contains the thumb's beveled image, if the pane is drawn with `bevel` set to '`true`'.

Instance Methods

`attachTo (Object` *`parentPane`*`)`
: Attaches the `X11ScrollBarPane` object to its parent pane, which is usually a `X11PaneDispatcher` object. Also sizes the pane and its buffers to fit within the visible window, and sets the dimensions of the scrollbar's visible frame and thumb.

`background (String` *`colorName`*`)`
: Sets the pane's background color to *colorName*, and also sets the color to use when performing scrollbar animations.

`drawThumb (void)`
`eraseThumb (void)`
: These methods animate the thumb so that it tracks the pointer's position within the scrollbar.

`frameWidth (Integer` *`lineWidth`*`)`
: Calculates the scrollbar frame's dimensions within the pane's margins for the line width in pixels given as the argument.

`new (String` *`paneName`*`)`
: Creates a new `X11ScrollBarPane` object, initializes the pane's event handlers, and sets the default colors of the pane's elements.

`percent (void)`
: Returns a `Float` with the position of the thumb's top edge in the usable trough of the scrollbar, as a percentage between 0.0 and 1.0.

  The usable area of the trough is defined as the distance that the top edge of the scrollbar thumb can travel within the scrollbar's margins; i.e,

```
usableTrough = aScrollBar size y -
                    ((aScrollBar margin * 2) +
                     (aScrollBar frameWidthVar * 2) +
                     (aScrollBar padding * 2) +
```

```
                                (aScrollBar thumbHeight));
```

`pointIsInThumb (Integer` *x*`, Integer,` *y*`)`
: This method returns a `Boolean` value of true if the point *x*,*y* is within the scroll bar thumb.

`subPaneDestroy (Object` *subPane*`, InputEvent` *event*
`subPaneExpose (Object` *subPane*`, InputEvent` *event*
`subPanePointerInput (Object` *subPane*`, InputEvent` *event*
`subPanePointerMotion (Object` *subPane*`, InputEvent` *event*
`subPaneEnterNotify (Object` *subPane*`, InputEvent` *event*
`subPaneLeaveNotify (Object` *subPane*`, InputEvent` *event*
: The `X11ScrollBarPane` object's event handlers for DESTROY, EXPOSE, BUTTONPRESS/BUTTONRELEASE, MOTIONNOTIFY, ENTERWINDOWNOTIFY, and LEAVEWINDOWNOTIFY X Window System events.

`thumbDimensions (Integer` *y*`, Integer` *height*`)`
: Calculates the position and size thumb's rectangle within the pane using the vertical *y* position and the thumb's *height*.

`thumbPercent (Float` *pct*`)`
: Calculates the thumb's height as a percent of the trough's vertical distance. The argument, *pct*, is a `Float` between 0.0 and 1.0.

  Note that, in order to make these calculations and update the thumb height in the pane's viewing area, the scrollbar must already be attached to its parent pane, and the program needs to be receiving X events. So this method should only be used after a call to, for example, X11Pane : openEventStream.

3.85 `X11TextEntryPane` Class

A `X11TextEntryPane` object displays a basic, single line text entry box. Users can enter text when the pointer is over the entry box, and the application can retrieve the text that the user enters as the contents of the `X11TextEntryPane : entryText` method.

Editing is performed using the pointer's left button or cursor motion keys to position the cursor. Users may insert text at that point, or delete characters using the `Backspace` and `Delete` keys. The class also supports cursor motion using Emacs-compatible editing keys.

```
Action           Keys
------           ----
Character Left   Left Arrow,  Ctrl-B
Character Right  Right Arrow, Ctrl-F
Start of Text    Home,        Ctrl-A
End of Text      End,         Ctrl-E
Delete Right     Del,         Ctrl-D
Delete Left      Backspace
Delete Selection Backspace (If Selecting Text)
```

In addition, `X11TextEntryPane` objects support cutting and pasting of entry text using the X primary selection. The class uses the standard pointer buttons to select text and to paste text selected by other X client programs.

Action	Pointer Buttons
Set Insertion Point	Button 1 (Left Button) Press and Release
Paste Selected Text At Cursor	Button 2 (Center Button, or Both Left and Right Buttons) Press and Release
Select Text to Place on the X selection	Button 1 (Left Button) Press + Drag Pointer Across Text

Here is an example of an `X11TextEntryPane` object's use.

```
#include <ctalk/ctalkdefs.h>

int main (int argv, char **argc) {
  X11Pane new xPane;
  X11PaneDispatcher new dispatcher;
  X11TextEntryPane new entry;
  X11LabelPane new label;
  X11ButtonPane new button;
  InputEvent new e;
  Integer new nEvents;
  Integer new verbose;
  X11Cursor new cursor;
   xPane resources atPut "backgroundColor", "blue";
  label resources replaceAt "backgroundColor", "blue";
  label resources replaceAt "foregroundColor", "blue";
  label resources replaceAt "highlightForegroundColor", "blue";
  label resources replaceAt "textColor", "white";
  label resources replaceAt "highlightTextColor", "white";
  label border = false;

   xPane inputStream eventMask = EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY| \
    KEYPRESS|KEYRELEASE|WINDELETE|BUTTONPRESS|BUTTONRELEASE;
   xPane initialize 300, 200;
  dispatcher attachTo xPane;
  label attachTo dispatcher, "100x80+10+10";
  button attachTo dispatcher, "60x45+120+100";
  entry attachTo dispatcher, "140x32+120+32";
```

```
  xPane map;
  xPane raiseWindow;

  xPane openEventStream;

  xPane setWMTitle "X11TextEntryPane Demo";
  label multiLine "Enter your\ntext:";
  button label text "Done";

  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;

      xPane subPaneNotify e;

      switch (e eventClass value)
        {
        case EXPOSE:
          entry refresh;
          break;
        case WINDELETE:
          xPane deleteAndClose;
          exit (0);
          break;
        default:
          if (button haveClick) {
            button clearClick;
            xPane deleteAndClose;
            printf ("You entered: %s\n", entry entryText);
            exit (0);
          }
        break;
        }
    }
  }
}
```

Echoing Dots

Normally a `X11TextEntryPane` object displays characters as you type them. However, you can set the `dots` instance variable to '`true`' to cause the pane to display dots instead, like this.

```
entry dots = true;
```

Resources

For information about how to set and retrieve resources, refer to the `X11Pane` section See ⟨undefined⟩ [PaneResources], page ⟨undefined⟩.

`backgroundColor`
: The background color of the entry pane's window. The default is '`white`'.

`borderColor`
: The color of the entry pane's border. The default is '`gray`'.

`borderWidth`
: The width in pixels of the pane's border when the pointer is not above it. The default is 1 pixels.

`cursorAdvanceHint`
: The distance in pixels between the last character and the cursor when the point is positioned at the end of the buffer for appending text. The default is 2 pixels when built and run with MacOS, 4 pixels for Linux and other systems.

`font`
: A `String` that contains the display font's descriptor. In order to maintain alignment between the character display and insertion point, it is strongly recommended that the display font be monospaced.

  The default font for MacOS is '`Bitstream Vera Sans Mono-14:weight=medium;slant=roman`'. The default font for Linux and other systems is '`DejaVu Sans Mono-12:weight=medium;slant=roman`'.

`hoverBorderWidth`
: The width in pixels of the pane's border when the pointer is over it and the pane's window has the input focus. The default is 4 pixels.

`hPad`

`vPad`
: The distance in pixels between the edges of the text and the edge to the window, including the width of the border. The default is 4 pixels.

`spacingHint`
: The distance in pixels that the pane adjusts the displayed text's horizontal spacing. The default is -1 for MacOs and 0 for Linux and other systems.

`selectionColor`
: A `String` that contains the background color of selected text. The default is '`orange`'.

`textColor`
: The color that the pane uses to display text. The default is '`black`'.

Instance Variables

`baselineY`
: An `Integer` that contains the vertical location of the text's baseline in the entry window, in pixels.

`button`
: An `Integer` that records the state of the pointer buttons when the program receives a BUTTONPRESS or BUTTONRELEASE event.

`center` A `Boolean` that helps determine how a part of the text is displayed if the text is too large to fit within the pane.

`clipX` An `Integer` that contains the leftmost character that is displayed when the text is scrolled leftward.

`chars` A `List` that contains the entry pane's text. Each member of the list is a `CharCell`, a class used exclusively by `X11TextEntryPane` objects, which contains both the character and information about its dimensions and placement within the window.

`cursor` A `X11Cursor` object that contains a text editing cursor which is displayed when the pointer is over the pane and it has the keyboard input focus.

`cursorX` An `Integer` that contains the character index of the cursor and insertion point within the text.

`dots` A `Boolean`, which, if set by a program, causes the entry pane to echo dots instead of the typed characters. In order to display dots, the calling program should contain a statement like this.

```
entry dots = true;
```

`hover` A `Boolean` object that is true when the pointer is over the pane and the pane has the keyboard input focus.

`paneWidthChars`
The width of the pane as measured in the number of characters that can be displayed at one time using the current font. The pane uses the `paneWidth` variable's value to help determine which section of the text to display when the entire text is too wide to fit within the pane's window.

`point` An `Integer` that determines where each character that the user types is inserted into the buffer's text. If `point` is equal to the size of the `chars` list, then the pane appends the characters to the end of the text.

`selecting`
A `Boolean` that is true if the user is in the process of selecting text; that is, when the pointer's left button is depressed and the pointer is dragged across the text.

`sEnd`

`sStart` These are `Integer`, which, if non-zero contain the start and end indexes of selected text within the entry pane's buffer.

`shiftState`
An `Integer` that records whether a *`Shift`*, *`Control`*, or *`Caps Lock`* key is being pressed while typing.

`spacingHint`
An `Integer` that contains the value of the '`spacingHint`' resource.

Instance Methods

`attachTo (X11Pane parentPane, String geometry)`
: Attaches the entry pane to *parentPane* using the dimensions and placement given by *geometry*. For information about how to specify a pane's geometry, refer to the `X11PaneDispatcher` class See ⟨undefined⟩ [X11PaneDispatcher], page ⟨undefined⟩.

`calclulateSpaceAppend (void)`
: This method defers calculating the width of a space that the user appends to the text until a following character is added.

`charCellAt (Integer charIndex)`
: Returns the `CharCell` object for the `charIndex` position in the text. (I.e., the *nth* element of the `chars` instance variable's list).

`charIndex (Integer clickX, Integer clickY)`
: Returns an `Integer` with the index of the character displayed at *clickX*,*clickY*.

`clearSelection (void)`
: Clears the selection in response to events from the user or from another X client application requesting the selection ownership.

`deleteAt (Integer charIndex)`
: Deletes the character at *charIndex* and returns the element, which is a `CharCell` object. `CharCell` objects are a class that `X11TextEntryPane` class uses to store each character, as well as its dimensions and placement within the window.

`deleteForward (Integer charIndex)`
: This deletes the character at the insertion point and is called when the user presses the *Delete* or *Ctrl-D* key.

`draw (void)`
: Draws the pane's border, its text contents after determining how far left to scroll the text, and calls `drawCursor` to draw the insertion cursor. The drawing occurs on the pane's `paneBuffer` instance variable, which is a `X11Bitmap` object. To make the the pane's contents visible in the application's window, use the `refresh` method.

`drawCursor (void)`
: Draws the pane's editing cursor at the text insertion point.

`drawSelection (void)`
: If any text is selected, highlight the text using the color defined by the `selectionColor` resource.

`entryText (void)`
: Returns the text contents of the entry pane as a `String` object.

`inputWidth (Integer startIdx)`
: Returns an `Integer` that contains the width in pixels of the input text starting at character *startIdx*. This method determines how far leftward text should be scrolled to keep the insertion point and its surrounding text visible in the pane's window.

`insertAt (Integer` *charIndex*`)`
: Inserts a `CharCell` object at *charIndex* in the entry object's `chars` list.

`new (String` *paneName*`)`
: The `X11TextEntryPane` constructor. Initializes resources to their default values and the event handler instance variables to the classes' event handlers. These instance variables are declared in `X11PaneDispatcher` class See ⟨undefined⟩ [X11PaneDispatcher], page ⟨undefined⟩.

`reflow (void)`
: Recalculates the character placement in the entry object's `chars` list. This method should be used after any operation that inserts or deletes text in the middle of the *chars* list.

`refresh (void)`
: Displays the contents of the pane's drawing surfaces on the program's visible window.

`selectionToText (String` *textOut*`)`
: Sets its argument, a `String` object, to the currently selected text as a `string`.

`subPaneButtonPress (Object` *subPaneObject*`, InputEvent` *event*`)`
`subPaneEnter (Object` *subPaneObject*`, InputEvent` *event*`)`
`subPaneExpose (Object` *subPaneObject*`, InputEvent` *event*`)`
`subPaneLeave (Object` *subPaneObject*`, InputEvent` *event*`)`
`subPaneKbd (Object` *subPaneObject*`, InputEvent` *event*`)`
`subPanePointerMotion (Object` *subPaneObject*`, InputEvent` *event*`)`
`subPaneSelectionClear (Object` *subPaneObject*`, InputEvent` *event*`)`
: Handler methods for different types of X `InputEvent` objects. For more information about the handlers and event types, refer to the `X11PaneDispatcher` class. See ⟨undefined⟩ [X11PaneDispatcher], page ⟨undefined⟩.

3.86 `X11YesNoBoxPane` Class

A `X11YesNoBoxPane` object displays a window with a text message, optionally an icon, and two buttons, which are normally labelled '`Yes`' and '`No`'.

Clicking on one of the buttons closes the window and returns an identifier of the selected button as an `Integer`. The object also stores the text of the selected button's label.

Also, pressing the ***Tab*** key switches focus between buttons. Pressing ***Enter*** activates the button with the focus, also causing the window to close.

Here is an example program.

```
/* yesnobox.ca - X11YesNoBoxPane Demonstration -*-c-*- */

#include <ctalk/ctalkdefs.h>

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
```

```
X11ButtonPane new button;
X11LabelPane new label;
X11YesNoBoxPane new yesnoBox;
InputEvent new e;

label textColor = "white";
label canFocus = false;
label borderWidth = 0;

label ftFontVar notifyLevel XFT_NOTIFY_ERRORS;

mainWindow backgroundColor = "blue";
label resources replaceAt "backgroundColor", "blue";
button resources replaceAt "backgroundColor", "blue";
yesnoBox resources replaceAt "backgroundColor", "blue";

mainWindow initialize 255, 200;
mainWindow inputStream eventMask =
  EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR

dispatcher attachTo mainWindow;
button attachTo dispatcher, "110x70+73+100";
label attachTo dispatcher, "177x80+34+15";
yesnoBox attachTo dispatcher, "300x200";

mainWindow map;
mainWindow raiseWindow;

mainWindow openEventStream;

mainWindow setWMTitle "X11YesNoBoxPane Demo";

label ftFontVar initFontLib;

label multiLine "X11YesNoBoxPane\nDemo";
label resources replaceAt "textColor", "lightgray";
label resources replaceAt "foregroundColor", "blue";
label resources replaceAt "borderColor", "blue";

button label multiLine "Open\nYes/No\nDialog";

button label resources replaceAt "highlightForegroundColor", "gray80";

/* Icon IDs, like ICON_QUESTION, are defined in ctalkdefs.h. */
yesnoBox resources replaceAt "iconID", ICON_QUESTION;

/* The program uses the "replaceAt" method because the key/value
```

```
      entry for "backgroundColor" the X11MessageBoxPane : new method
      has alread created an entry for backgroundColor. */
  yesnoBox resources replaceAt "backgroundColor", "blue";
  yesnoBox resources replaceAt "foregroundColor", "blue";
  yesnoBox resources replaceAt "messageColor", "white";
  yesnoBox resources replaceAt "messageText",
            "Hello, yesnoBox!\nYour message text here.";

  button draw;
  button refresh;
  label draw;
  label refresh;

  while (TRUE) {
    mainWindow inputStream queueInput;
    if (mainWindow inputStream eventPending) {
      e become mainWindow inputStream inputQueue unshift;

      mainWindow subPaneNotify e;

      switch (e eventClass value)
{
case EXPOSE:
  button subPaneExpose (button, e);
  label subPaneExpose (label, e);
  break;
case BUTTONRELEASE:
  yesnoBox showManaged button;
  printf ("returnVal: %d: %s\n", yesnoBox returnVal,
  yesnoBox returnText);
  break;
case WINDELETE:
    mainWindow deleteAndClose;
  exit (0);
  break;
}
    } else {
      usleep (1000);
    }
  }

}
```

Return Values

Normally, after initializing the `X11YesNoBoxPane` object, a program calls either the `show` or `showManaged` method, which displays the object's window. After clicking on one of the buttons, or after either the calling program or the user otherwise closes the window, the method withdraws the window, and returns an `Integer` value that contains one of the following definitions.

```
YESNO_NONE
YESNO_LBUTTON
YESNO_RBUTTON
```

These constants are defined in `ctalkdefs.h`, so the calling program should contain the following statement.

```
#include <ctalk/ctalkdefs.h>
```

This value is also contained in the `X11YesNoBox` object's `returnVal` instance variable. In addition, the text of the selected button is contained in the object's `returnText` label.

In the example above, this is how the program displays the user's selection.

```
yesnoBox showManaged button;
printf ("returnVal: %d: %s\n", yesnoBox returnVal,
        yesnoBox returnText);
```

If the user or the calling program close the object's window by some other method, then the value of `returnText` is an empty string.

Resources

The resources that `X11YesNoBoxPane : new` defines by default are stored in the `resources` instance variable, an `AssociativeArray` that is declared in `X11Pane` class. For a description, see the `resources` instance variable documentation. See ⟨undefined⟩ [PaneResources], page ⟨undefined⟩.

`backgroundColor`
: A `String` that contains the color used to draw the window background. This includes the actual subwindow that receives the button's events from the display server. The resources' default value is '`gray`'.

`foregroundColor`
: A `String` with the name of the color that the messagebox window is filled with. The default is '`gray`'.

`geometry`
: A `String` that contains the message window's size, and, optionally, its position. The value is set to the dimensions that are given as arguments to the `attachTo` method.

`iconID` An `Integer` that contains the identifier to the icon to be displayed. Ctalk defines the following constants for icons in `ctalkdefs.h` and in the graphics libraries.

```
ICON_NONE
ICON_CAUTION
ICON_INFO
ICON_QUESTION
ICON_STOP
```

The default is '`ICON_QUESTION`', which causes a question mark icon to be displayed. The widget adjusts the size and position of the text automatically to provide space for the icon.

`leftButtonText`
`rightButtonText`
These are `Strings` that contain the text that appears in the left and right-hand buttons. The default is '`Yes`' and '`No`'.

`messageColor`
A `String` that contains the color of the message text. The default is '`black`'. The message box sets the label's `textColor` resource to this value.

`messageFont`
A `String` that contains the Fontconfig font descriptor. The default is '`sans serif-12`'. The pane uses this value to set the label's '`fcFont`' resource.

`messageText`
A `String` that contains the text of the message to be displayed. The default is '`Sample message text.`'.

`pad` An `Integer` that contains the space in pixels betwween the button and label widgets, and between each widget and the edge of the window. The default is 10 pixels.

`titleText`
A `String` that contains the message window's title text. The default is '`Message`'.

Instance Variables

`lbutton`

`rbutton` The `X11ButtonPane` objects that close the window when either is clicked. The buttons' default text are, '`Yes`' and '`No`'. Programs can change the text with a statement like the following.

```
myYesNoBox resources atPut "leftButtonText", "All Right";
myYesNoBox resources atPut "rightButtonText", "No Way!";
```

`initialized`
: A `Boolean` that is `true` if the pane has already been initialized. When a message box is closed, it is not deleted - it is simply unmapped from the display. This variable is set to true so that the message box is initialized only once.

`label`
: A `X11LabelPane` that displays the message box's text. As configured, the label is the same color as the main message box window, and uses no borders. The label's text is controlled by the resource, '`messageText`'. To set the message text, use a statement like the following.

```
myYesNoBox resources atPut "messageText", "Today Is\nJanuary 3rd.\nOkay?";
```

The class draws the label using multiline formatting, so if you want the text to appear on multiple lines, include the sequence '`\n`' in the text (that's a backslash and the character, '`n`', not a literal newline).

`mainWindowPtr`
: A `Symbol` that contains a pointer to the main window.

`returnText`
: A `String` that contains the text of the button that was clicked to withdraw the window. If the window was closed by some other method, then `returnText` contains an empty string.

`returnVal`
: When the `X11YesNoBoxPane` obect's window is closed, either by clicking on one of the buttons, or by otherwise closing the window, this `Integer` that contains one of the following values.

```
YESNO_NONE
YESNO_LBUTTON
YESNO_RBUTTON
```

The value of `returnVal` is also used as the `show` and `showManaged` methods' return value.

Instance Methods

`attachTo (Object` *`parentPane`*`, String` *`geometry`*`)`
: Sets the yes/no box's dimensions and creates the object's buffers and main window. Unlike other `X11Pane` subclasses, this method does not immediately attach the yes/no box's object to its parent pane, so the yes/no box isn't displayed until the program invokes the `show` or `showManaged` method.

`draw (void)`
: Draws the label and button subpanes in the yes/no box's buffers and the buffers of the button and label subpanes. After calling this method, the `refresh` method makes the changes visible.

`fill (String` *`colorName`*`)`
: A convenience method that fills the pane window with the color given as the argument.

`initWidgets (void)`
: Initializes the yes/no box's button and label subpanes when the pane's window is first opened. When constructed this method sets the `initialized` instance variable to '`true`', so when the yes/no box appears again, this method is a no-op.

`new (String` *`newPaneName`*`)`
: The X11YesNoBoxPane constructor.

`refresh (void)`
: After the method *draw* renders the main window and the label and the button subwindow's contents on the pane objects' buffers, this method updates the visible window with the contents of the buffers.

`show (void)`
: This method displays the yes/no box, waits for events from the display system, and closes the yes/no box when its button is clicked, then returns to the calling program.

`show (X11ButtonPane` *`buttonPane`*`)`
: This method is similar to the `show` method, but it takes as its argument the pane object from the main window that caused the yes/no window to open.

  This method sets the main window's button to be unclicked and then redrawn.

  This method manages `X11ButtonPane` objects specifically, but it can be subclassed to manage any pane class object.

`subPaneButton (Object` *`subPane`*`, InputEvent` *`event`*`)`
: This is the event handler for button press and button release events from the display hardware. When a button is clicked on, the yes/no box draws the button in its clicked state. When the button is released, the pane draws the button unclicked, then unmaps the yes/no box window from the display, and returns to the main program.

`subPaneExpose (Object` *`subPane`*`, InputEvent` *`event`*`)`
: Handles drawing the yes/no box when the program receives an Expose event from the display. This method calls the yes/no box's `draw` and `refresh` methods.

`withdraw (void)`
: Unmaps the yes/no box's window from the display after the pane object receives a button release event while processing the `show` method. When the `show` or `showManaged` method receives the button release event, this method is called to remove the pane's window from the display, and the `show` or `showManaged` method returns to the main program.

3.87 X11TextEntryBox Class

A X11TextEntryBox object pops up a dialog window that contains a X11TextEntryPane object, where the user can enter text. The contents of the text entry persist if the dialog window is re-opened, and the contents are available to the calling program, as is the text of the button and return code that indicates how the user closed the dialog box.

Here is an example program that demonstrates a X11TextEntryBox's use.

```
/* entrybox.ca, a X11TextEntryBox demonstration. -*-c-*- */

#include <ctalk/ctalkdefs.h>

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
  X11ButtonPane new button;
  X11LabelPane new label;
  X11TextEntryBox new textEntryBox;
  InputEvent new e;

  label textColor = "white";
  label canFocus = false;
  label borderWidth = 0;

  label ftFontVar notifyLevel XFT_NOTIFY_NONE;

  mainWindow backgroundColor = "blue";
  label resources replaceAt "backgroundColor", "blue";
  button resources replaceAt "backgroundColor", "blue";
  textEntryBox resources replaceAt "backgroundColor", "blue";

  mainWindow initialize 255, 200;
  mainWindow inputStream eventMask =
    EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR

  dispatcher attachTo mainWindow;
  button attachTo dispatcher, "110x70+73+100";
  label attachTo dispatcher, "177x80+34+15";
  textEntryBox attachTo dispatcher, "300x200";

  mainWindow map;
  mainWindow raiseWindow;

  mainWindow openEventStream;

  mainWindow setWMTitle "X11TextEntryBox Demo";
```

```
label ftFontVar initFontLib;

label multiLine "X11YTextEntryBox\nDemo";
label resources replaceAt "textColor", "lightgray";
label resources replaceAt "foregroundColor", "blue";
label resources replaceAt "borderColor", "blue";

button label multiLine "Open\nText Entry\nDialog";

button label resources replaceAt "highlightForegroundColor", "gray80";

/* The program uses the "replaceAt" method because the key/value
   entry for "backgroundColor" the X11MessageBoxPane : new method
   has alread created an entry for backgroundColor. */
textEntryBox resources replaceAt "backgroundColor", "blue";
textEntryBox resources replaceAt "foregroundColor", "blue";
textEntryBox resources replaceAt "messageColor", "white";
textEntryBox resources replaceAt "messageText",
             "Hello, textEntryBox!\nYour message text here.";

button draw;
button refresh;
label draw;
label refresh;

while (TRUE) {
  mainWindow inputStream queueInput;
  if (mainWindow inputStream eventPending) {
    e become mainWindow inputStream inputQueue unshift;

    mainWindow subPaneNotify e;

    switch (e eventClass value)
      {
      case EXPOSE:
        button subPaneExpose (button, e);
        label subPaneExpose (label, e);
        break;
      case BUTTONRELEASE:
        textEntryBox showManaged button;
        printf ("returnVal: %d: %s: %s\n", textEntryBox returnVal,
                textEntryBox returnText, textEntryBox entryContents);
        break;
      case WINDELETE:
        mainWindow deleteAndClose;
        exit (0);
        break;
```

```
            }
          } else {
            usleep (1000);
          }
        }

    }
```

Resources

`X11TextEntryBox` objects do not define resources of their own. Instead, it uses the resources defined by the classes of its components:

`X11YesNoBoxPane` See ⟨undefined⟩ `[X11YesNoBoxPane]`, page ⟨undefined⟩.
`X11LabelPane` See ⟨undefined⟩ `[X11LabelPane]`, page ⟨undefined⟩.
`X11ButtonPane` See ⟨undefined⟩ `[X11ButtonPane]`, page ⟨undefined⟩.
`X11TextEntryPane` See ⟨undefined⟩ `[X11TextEntryPane]`, page ⟨undefined⟩.

The top-level window's resources are defined by the `X11YesNoBox` superclass. Each of the component subpanes is defined by an instance variable. This allows each of the subpanes to be managed individually.

`lbutton` Defined by the `X11ButtonPane` class. See ⟨undefined⟩ [X11ButtonPane], page ⟨undefined⟩.

`rbutton` Defined by the `X11ButtonPane` class. See ⟨undefined⟩ [X11ButtonPane], page ⟨undefined⟩.

`label` Defined by the `X11LabelPane` class. See ⟨undefined⟩ [X11LabelPane], page ⟨undefined⟩.

`icon` Defined by the `X11Bitmap` class. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`entryPane`
Defined by the `X11TextEntryPane` class. See ⟨undefined⟩ [X11TextEntryPane], page ⟨undefined⟩.

For example, to change the background color of selected text in the entrybox subwindow, a program would contain a statement like this one.

```
myEntryBox entryPane resources replaceAt "selectionColor", "lightblue";
```

Occasionally, the subpane's class defines its own resources that it propagates to its own subpanes. For example, to change the text that appears on the buttons:

```
myEntryBox resources replaceAt "leftButtonText", "Dismiss";
myEntryBox resources replaceAt "rightButtonText", "Accept";
```

It's necessary to use the `Collection : replaceAt` method, because each subpane's default resources has already been created when the subpane is constructed (generally, this is done by each classes' `new` method).

Instance Variables

`dots` A `Boolean` that causes the entryPane to echo dots instead of the text that the user typed.

`entryContents`
A `String` that contains the text contents of the entry window. This variable is updated whenever the contents of the `entryPane` object are modified.

`entryPane`
The `X11TextEntryPane` object that is constructed when the `show` method is first called, and is updated and displayed via the `show` method's event loop.

Instance Methods

`draw (void`
Draws the main pane and each subpane's controls on each subpane's buffer, as well as the main pane's icon, if any.

`initWidgets (void)`
Constructs the subpanes when the entrybox window is first constructed and mapped to the display. The method sets the `initialized` instance variable (defined in `X11YesNoBoxPane` class) to true, so the pane is only constructed once if it is withdrawn from the display and then remapped.

`new (String` *`paneName`*`)`
The `X11TextEntryBox` constructor. This method calls the constructor of its superclass, `X11YesNoBoxPane`, which defines the resources that this class uses as well. However, this constructor defines the event handlers it needs, in order to make the events available to the `entryPane` subpane when necessary.

`show (Object` *`subPane`*`, InputEvent` *`event`*`)`
This method constructs the `X11TextEntryBox` object when it is first called, and then on the first and each following call, maps the entry box to the display, waits for and dispatches events from the display server, and withdraws the window from the display when the user clicks on a button or on the window's close menu.

`subPaneEnter (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneLeave (Object` *`subPane`*`, InputEvent` *`event`*`)`
Event handlers for enter and leave events, which are generated when the pointer is over one of the subpanes.

3.88 `X11ListBox` Class

A `X11ListBox` object pops up a dialog window that contains a label, a listbox, and "Cancel" and "Ok" buttons.

After the listbox has first appeared and all of its subpanes have been initialized, a program can access any of the items selected in the listbox's list via the `listPane` instance variable.

A `X11ListBox` dialog supports multiple selections. Pressing `Shift` while clicking on a list item adds that item to list's selections. Pressing `Ctrl` while clicking on an item selects all of the items in a range between the first selected item and the last selected item of the list.

Component Widgets

A `X11ListBox` consists of component widgets from the `X11LabelPane`, `X11ListPane`, and `X11ButtonPane` class.

The objects for each component are declared as instance variables, which, except for the list subpane itself, are inherited from the `X11YesNoBoxPane` superclass. See ⟨undefined⟩ [X11YesNoBoxPane], page ⟨undefined⟩.

Component Instance Variable	Class	Inherited From
listPane	X11ListPane	--
lButton	X11ButtonPane	X11YesNoBoxPane
rButton	X11ButtonPane	X11YesNoBoxPane
label	X11LabelPane	X11YesNoBoxPane

More information about the resources, instance variables, and methods that each component uses is contained in each of the component classes' individual sections: See ⟨undefined⟩ [X11ListPane], page ⟨undefined⟩, See ⟨undefined⟩ [X11ButtonPane], page ⟨undefined⟩, and See ⟨undefined⟩ [X11LabelPane], page ⟨undefined⟩

Here is an example program that shows how to display a `X11ListBox` and retrieve the selected list items.

```
/* listbox.ca, a X11ListBox demonstration. -*-c-*- */

#include <ctalk/ctalkdefs.h>

#define N_ITEMS 30

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
  X11ButtonPane new button;
  X11LabelPane new label;
  X11ListBox new listBox;
  InputEvent new e;
  Integer new i;
  List new itemTextOut, itemNOut;
  String new itemText, itemN;

  label textColor = "white";
  label canFocus = false;
  label borderWidth = 0;

  label ftFontVar notifyLevel XFT_NOTIFY_NONE;
```

```
mainWindow backgroundColor = "blue";
label resources replaceAt "backgroundColor", "blue";
button resources replaceAt "backgroundColor", "blue";
listBox resources replaceAt "backgroundColor", "blue";

mainWindow initialize 255, 200;
mainWindow inputStream eventMask =
  EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR

dispatcher attachTo mainWindow;
button attachTo dispatcher, "110x70+73+100";
label attachTo dispatcher, "177x80+34+15";
listBox attachTo dispatcher, "300x400";

mainWindow map;
mainWindow raiseWindow;

mainWindow openEventStream;

mainWindow setWMTitle "X11ListBox Demo";

label ftFontVar initFontLib;

label multiLine "X11ListBox\nDemo";
label resources replaceAt "textColor", "lightgray";
label resources replaceAt "foregroundColor", "blue";
label resources replaceAt "borderColor", "blue";

button label multiLine "Open\nListPane\nDialog";

button label resources replaceAt "highlightForegroundColor", "gray80";

/* The program uses the "replaceAt" method because the key/value
   entry for "backgroundColor" the X11MessageBoxPane : new method
   has alread created an entry for backgroundColor. */
listBox resources replaceAt "backgroundColor", "blue";
listBox resources replaceAt "foregroundColor", "blue";
listBox resources replaceAt "messageColor", "white";
listBox resources replaceAt "messageText",
          "Hello, listBox!\nPlease select an item.";

for (i = 1; i < N_ITEMS; ++i) {
  listBox items push "item " + i asString;
}

button draw;
button refresh;
```

```
  label draw;
  label refresh;

  while (TRUE) {
    mainWindow inputStream queueInput;
    if (mainWindow inputStream eventPending) {
      e become mainWindow inputStream inputQueue unshift;

      mainWindow subPaneNotify e;

      switch (e eventClass value)
{
case EXPOSE:
  button subPaneExpose (button, e);
  label subPaneExpose (label, e);
  break;
case BUTTONRELEASE:
  listBox showManaged button;
  if (listBox listPane nItemsSelected > 1) {
    i = 0;
    listBox listPane selectedItems itemTextOut;
    listBox listPane selectedItemsN itemNOut;
    itemNOut map {
      itemN = self;
      /* This is a convenient way to retrieve
 the i'th item in the itemTextOut list. */
      itemText = *(itemTextOut + i);
      printf ("%d: %s\n", itemN, itemText);
      ++i;
    }
  } else {
    printf ("%d: %s\n",
    listBox listPane selectedItemN,
    listBox listPane selectedItemText);
  }
  break;
case WINDELETE:
    mainWindow deleteAndClose;
  exit (0);
  break;
}
    } else {
      usleep (1000);
    }
  }

}
```

Retrieving List Selections

There are several methods that provide information about which item or list items are currently selected. Generally, if a user has selected more than one item, then the results would be in the form of a list, instead of an `Integer` index of a selection or a `String` containing the selection's contents.

The method, `nItemsSelected`, returns an `Integer` with the number of items that are currently selected. Programs can use this to determine which method to use to retrieve the list selections.

Depending on whether a user has selected a single item or multiple items, a program can use the `selectedItemN` or `selectedItemsN` methods to retrieve selection indexes, and the `selectedItemText` or `selectedItems` methods to retrieve the text of the selected item or items.

The example program above uses this method to print either a single selected item or multiple selections.

```
if (listBox listPane nItemsSelected > 1) {
  i = 0;
  listBox listPane selectedItems itemTextOut;
  listBox listPane selectedItemsN itemNOut;
  itemNOut map {
    itemN = self;
    /* This is a convenient way to retrieve
       the i'th item in the itemTextOut list. */
    itemText = *(itemTextOut + i);
    printf ("%d: %s\n", itemN, itemText);
    ++i;
  }
} else {
  printf ("%d: %s\n",
          listBox listPane selectedItemN,
          listBox listPane selectedItemText);
}
```

Resources

`listFont` A `String` that contains the name of the font that the `X11ListPane` subwidget uses to display the list items. The default is '`sans-serif-12`'.

Instance Variables

`items` A `List` that contains all of the items to be added to the list when the dialog is first popped up. Because the dialog box doesn't construct itself and the list subpane until it first appears, this is a convenient way for programs to define the items that will initially appear in the list before the the dialog first appears.

After the dialog first appears, the list items may be modified by using the list pane's `add` method. For example:

```
myListDialog listPane add "Extra List Item 1";
```

`listPane` A `X11ListPane` object that defines the list subpane. It is initialized when the dialog is first popped up. After that, programs can use any of the methods, resources, and instance variables that are defined in `X11ListPane` class to alter or retrieve the list's items or the user's selections See ⟨undefined⟩ [X11ListPane], page ⟨undefined⟩.

Instance Methods

`draw (void)`
: This calls the indivdual subpane's `draw` and `refresh` methods in order to update the dialog's contents on the display.

`initWidgets (void)`
: Called when the dialog first appears, this initializes each of the component widget objects.

`new (String` *`paneName`*
: The `X11ListBox` constructor. Creates the dialog object with the name given as the argument, and initializes the dialog's event handlers and resources.

`show (String` *`paneName`*
: Displays the dialog window, also calling `initWidgets` if the dialog is being displayed for the first time.

`subPaneEnter (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneLeave (Object` *`subPane`*`, InputEvent` *`event`*`)`
: Event handlers for highlighting the dialog's label or buttons when the pointer is over them.

3.89 `X11FileSelectDialog` Class

A `X11FileSelectDialog` object displays a dialog box with a list of files in the current directory, a preselected target directory, or a subdirectory that a user selects from the current directory's entries.

The class allows users to select multiple entries by holding either the `Shift` or `Ctrl` keys while selecting items. The class provides a number of methods to retrieve the number of items that the user selects, the items' text, and text for a new file that the user enters, if any. These methods are described below. See ⟨undefined⟩ [Retrieving-Selection-Information], page ⟨undefined⟩.

The `X11FileSelectDialog` class can also display a text entry box (from `X11TextEntryPane` class) where users can enter new file names.

Here is an example program that pops up a `X11FileSelectDialog` window.

```
/* fileselectbox.ca, a X11FileSelectBox demonstration. -*-c-*- */
```

```
/*
 *  Uncomment the following #define if you want to build the dialog
 *  with a X11TextEntryPane for creating new file paths.
 */
/* #define FILENAME_ENTRY_BOX */

#include <ctalk/ctalkdefs.h>

int main (void) {
  X11Pane new mainWindow;
  X11PaneDispatcher new dispatcher;
  X11ButtonPane new button;
  X11LabelPane new label;
  X11FileSelectDialog new fileSelect;
  InputEvent new e;
  Integer new i;
  List new itemTextOut, itemNOut;
  String new itemText, itemN;

  label textColor = "white";
  label canFocus = false;
  label borderWidth = 0;

  label ftFontVar notifyLevel XFT_NOTIFY_NONE;

  mainWindow backgroundColor = "blue";
  label resources replaceAt "backgroundColor", "blue";
  button resources replaceAt "backgroundColor", "blue";
  fileSelect resources replaceAt "backgroundColor", "blue";

  mainWindow initialize 255, 200;
  mainWindow inputStream eventMask =
    EXPOSE|ENTERWINDOWNOTIFY|LEAVEWINDOWNOTIFY|BUTTONPRESS|BUTTONRELEASE|KEYPRESS|KEYR

  dispatcher attachTo mainWindow;
  button attachTo dispatcher, "110x70+73+100";
  label attachTo dispatcher, "177x80+34+15";
  fileSelect attachTo dispatcher, "300x400";

  mainWindow map;
  mainWindow raiseWindow;

  mainWindow openEventStream;

#ifdef FILENAME_ENTRY_BOX
  fileSelect resources replaceAt "useEntryBox", TRUE;
```

```
#endif

  mainWindow setWMTitle "X11FileSelectDialog Demo";

  label ftFontVar initFontLib;

  label multiLine "X11FileSelectDialog\nDemo";
  label resources replaceAt "textColor", "lightgray";
  label resources replaceAt "foregroundColor", "blue";
  label resources replaceAt "borderColor", "blue";

  button label multiLine "Open\nFile Select\nDialog";

  button label resources replaceAt "highlightForegroundColor", "gray80";

  /* The program uses the "replaceAt" method because the key/value
     entry for "backgroundColor" the X11MessageBoxPane : new method
     has alread created an entry for backgroundColor. */
  fileSelect resources replaceAt "backgroundColor", "blue";
  fileSelect resources replaceAt "foregroundColor", "blue";
  fileSelect resources replaceAt "messageColor", "white";
  fileSelect resources replaceAt "messageText",
            "Directory:\n";

  button draw;
  button refresh;
  label draw;
  label refresh;

  while (TRUE) {
    mainWindow inputStream queueInput;
    if (mainWindow inputStream eventPending) {
      e become mainWindow inputStream inputQueue unshift;

      mainWindow subPaneNotify e;

      switch (e eventClass value)
{
case EXPOSE:
  button subPaneExpose (button, e);
  label subPaneExpose (label, e);
  break;
case BUTTONRELEASE:
  fileSelect showManaged button;
  if (fileSelect buttonClick == FILESELECT_LBUTTON) {
    printf ("\"%s,\" clicked.\n", fileSelect buttonText);
    /*
```

```
        The button's click should be cleared, because in this
        program, the dialog can be re-opened here (i.e., not
        reinstantiated with the "new" method).

        The, "buttonText," method is equivalent, in this clause,
        to, "fileSelect lbutton text."

        The FILESELECT_LBUTTON #define and the other button
        definitions, are in ctalkdefs.h.
    */
    fileSelect lbutton clearClick;
  } else if (fileSelect entryLength > 0) {
    printf ("%s\n", fileSelect pathEntry);
  } else if (fileSelect nItemsSelected > 1) {
    i = 0;
    fileSelect selectedItems itemTextOut;
    fileSelect selectedItemsN itemNOut;
    itemNOut map {
      itemN = self;
      /* This is a convenient way to retrieve
 the i'th item in the itemTextOut list. */
      itemText = *(itemTextOut + i);
      printf ("%d: %s\n", itemN, itemText);
      ++i;
    }
  } else {
    printf ("%d: %s\n",
    fileSelect selectedItemN,
    fileSelect selectedItemText);
  }
  fileSelect rbutton clearClick;
  break;
case WINDELETE:
    mainWindow deleteAndClose;
  exit (0);
  break;
}
    } else {
      usleep (1000);
    }
  }

}
```

Retrieving Selection Information

The type and number of entries that a `X11FileSelectDialog` can return can differ depending on whether a user selects a single directory entry, multiple directory entries, or enters a new file name.

When returning directory entries, the class returns each entry as a concatenation of the current directory plus the entry name; for example, if the current directory is '`/home/bill`' and the user has selected the file '`message.txt`', the dialog returns the `String`, '`/home/bill/message.txt`'.

In addition, the dialog can provide information about whether the user has clicked the left or right buttons, which are labeled, by default, '`Cancel`' and `Ok`.

Generally, when a program displays the file select dialog, the program waits until the dialog box returns, that is, when the user presses either the '`Cancel`' or `Ok` buttons. On most desktops, the user can also close the dialog using the desktop's close window menu item. In this case, the dialog can return a value indicating that the dialog has returned without the user making a selection or cancelling the selection. These values and the method to retrieve them are described below.

Here is the section of code from the example above (without the comments) that retrieves and displays the user's selection or selections. The file select dialog is defined in the `X11FileSelectDialog` object, `fileSelect`.

```
if (fileSelect buttonClick == FILESELECT_LBUTTON) {
  printf ("\"%s,\" clicked.\n", fileSelect buttonText);
  fileSelect lbutton clearClick;
} else if (fileSelect entryLength > 0) {
  printf ("%s\n", fileSelect pathEntry);
} else if (fileSelect nItemsSelected > 1) {
  i = 0;
  fileSelect selectedItems itemTextOut;
  fileSelect selectedItemsN itemNOut;
  itemNOut map {
    itemN = self;
    itemText = *(itemTextOut + i);
    printf ("%d: %s\n", itemN, itemText);
    ++i;
  }
} else {
  printf ("%d: %s\n",
          fileSelect selectedItemN,
          fileSelect selectedItemText);
}
fileSelect rbutton clearClick;
```

First, the application checks whether the user clicked the '`Cancel`' button with the expresion:

```
if (fileSelect buttonClick == FILESELECT_LBUTTON)
```

In this class, `buttonClick` is a convenience method that checks each of the dialog's buttons. It can return the following values.

```
FILESELECT_LBUTTON
FILESELECT_RBUTTON
FILESELECT_BUTTON_NONE
```

These values are defined in the `ctalkdefs.h` include file, which programs can include with a statement like this:

```
#include <ctalkdefs.h>
```

near the start of the file.

Assuming that the user has pressed the '`Ok`' button, the program next checks if the user has entered a new file name, with the statement:

```
} else if (fileSelect entryLength > 0) {
```

The `entryLength` method returns '`-1`' if the dialog doesn't display an entry box, '`0`' if no text is entered, or the length of the text. If the user has entered text, then the program can retrieve it with the `pathEntry` method, which retrieves and prints the entry with this line.

```
printf ("%s\n", fileSelect pathEntry);
```

Once again, the dialog returns a `String` that is the concatenation of the current directory and the filename that the user entered. To retrieve the entry box's text alone, programs can use the `textEntry` method instead of `pathEntry`.

Finally, the program checks how many items the user selected from the directory entries. The program does this with the `nItemsSelected` method.

```
} else if (fileSelect nItemsSelected > 1) {
```

If `nItemsSelected` returns an `Integer` greater than one, the program can retrieve all of the entries as a list, with the `selectedItems` method, and their position in the list with the `selectedItemsN` method.

```
fileSelect selectedItems itemTextOut;
fileSelect selectedItemsN itemNOut;
```

These methods take one argument, a `List` object, here `itemTextOut` and `itemNOut`, to which the method adds either the list item text or ordinal position of each selected directory entry.

In the case where the user has selected only a single directory entry (i.e., `nItemsSelected` returns '`1`'), the program can use the `selectedItemN` and `selectedItemText` methods, as in this statement, which prints the information about a single entry.

```
printf ("%d: %s\n",
        fileSelect selectedItemN,
        fileSelect selectedItemText);
```

Resources

Further information about modifying resource entries is given in the `resources` instance variable section. See ⟨undefined⟩ [PaneResources], page ⟨undefined⟩.

`listROPx`

`listRWPx` The distance in pixels between the bottom of the file list and the bottom of the window, in pixels. The values determine, respectively whether to display the dialog's buttons directly beneath the list, or to display a text entry box between the file list and the buttons.

`pad` The amount of space in pixels between the dialog window's edges and the component subpane's boundaries. The default is '`0`'.

`useEntryBox`
A boolean value that determines whether the dialog box displays a text entry box. The default is '`false`'.

Instance Variables

In addition to the instance variables defined in superclasses (like `lbutton`, `rbutton`, and `label`, which are defined in `X11YesNoBoxPane` class; `entryPane`, defined in `X11TextEntryBox`; and `listPane`, which is defined in `X11ListBox` class), the `X11FileSelectDialog` class defines the following instance variables.

`dirPattern`
A `String` that contains the file glob of the directory that the dialog displays initially. The default is `.` (i.e., the current directory).

`dotFiles` If true, include hidden files (i.e., files whose names begin with '`.`') in the directory listing.

`targetPath`
A `String` that contains the fully qualified path of the directory that is being displayed. The dialog object fills in the value whenever it compiles a new list of directory contents.

`useEntryBox`
A `Boolean` that determines whether the dialog should display an entry box or check for text entry operations. The default is '`false`'.

`waitCursor`
: A `X11Cursor` item that defines the wait mouse pointer that the dialog can display when building directory lists.

Instance Methods

`buttonClick (void)`
: Returns an `Integer` with one of the following values.

```
FILESELECT_LBUTTON
FILESELECT_RBUTTON
FILESELECT_BUTTON_NONE
```

These values are defined in `ctalkdefs.h`. Programs can include the statement:

```
#include <ctalkdefs.h>
```

near the start of the file.

`buttonText (void)`
: Returns a `String` that contains the text of the button that the user has clicked, or NULL if no button has been clicked.

To reset the button, programs should use the `clearClick` method, which is defined in `X11ButtonPane` class; for example:

```
myDialog lbutton clearClick;

... or ...

myDialog rbutton clearClick;
```

`chDir (ItemBox` *item*`)`
: Changes the working directory of the dialog (and the program that opens it) to the directory whose name is contained in `item`. If `item` contains '`..`', the the new directory is the parent directory of the current directory.

`draw (void)`
: Redraws the dialog's label, list and button subpanes. Redrawing of the text entry pane, if it is displayed, is handled separately.

`entryLength (void)`
: Returns the length of the text entered in the dialog's entry box, or '`-1`' if the dialog does not display a text entry box.

`getExpandedDir (String` *dirPattern*`)`
: Retrieves the contents of the directory named in *dirPattern*, which the method qualifies and stores in the `targetPath` instance variable.

The method then constructs a sorted list of the directory's contents, which it stores in `listPane's items` instance variable. In order to use these items directly, programs could use an expression similar to this one.

```
myFileDialog listPane items
```

`initWidgets (void)`
: Creates the subpane objects and adds them to the dialog's window when the dialog is first opened. Sets the `initialized` instance variable (defined in `X11YesNoBoxPane`) to '`true`'.

`makeItem (String` *`itemText`*`)`
: Creates an `ItemBox` object (which is defined in `X11ListPane` class), with the text of the object, the item's boundaries and position in pixels.

`new (String` *`paneName`*`)`
: The `X11FileSelectDialog` constructor. The argument, a `String`, is the name of the new dialog object.

  Note: The `X11FileSelectDialog` class adds the `fileMode` instance variable to `ItemBox` class, so it's necessary to make sure that the `ItemBox` class, which is defined in `X11ListPane`. In `X11FileSelectDialog` class, the `X11ListPane` class should be loaded automatically when the `listPane` instance variable is created.

`onExpose (Object` *`subPane`*`, InputEvent` *`event`*`)`
`onListClick (Object` *`subPane`*`, InputEvent` *`event`*`)`
`onResize (Object` *`subPane`*`, InputEvent` *`event`*`)`
`subPaneButton (Object` *`subPane`*`, InputEvent` *`event`*`)`
: The `X11FileSelectDialog's` event handlers. If the dialog handles events similarly to the operations in superclasses, then Ctalk uses event handlers defined in `X11FileSelectDialog's` superclasses.

`nItemsSelected (void)`
: Returns an `Integer` with the number of list entries that are currently selected.

`pathEntry (void)`
: If the dialog displays a text entry box, and the user has entered the name of a new file in it, returns the fully qualified path of the entry (i.e., the current directory path plus the name of text entry). If the dialog does not display an entry box, or the entry box contains no text, the method returns an empty string.

`refresh (void)`
: Updates the dialog window with the contents of the dialog's subpanes.

`selectedItems (List` *`selecteItemsOut`*`)`
: Stores the text of each item that the user has selected in *selectedItemsOut*.

`selectedItemN (void)`
: If the user has selected a single directory entry, returns the ordinal position of the item (counting from 0) as an `Integer`.

`selectedItemsN (List selecteItemsOut)`
: Stores the ordinal position of enach item, counting from zero ('0') that the user has selected in *selectedItemsOut*. Each of the elements in in *selectedItemsOut* is an `Integer` object.

`selectedItemText (void)`
: If the user has selected a single directory entry, returns the text of that item as a `String`.

`show (void)`
: The main dialog box loop. This creates the dialog's subpanes and attaches them to the main dialog window. If the dialog has already been popped up, maps the dialog on the display. Contains the event loop. When the dialog is closed, the method returns to the calling program.

`textEntry (void)`
: If the dialog displays a text entry box, and the user has entered a file name in it, returns a `String` that contains the text of the entry. If the dialog does not display an entry box, or if the entry box does not contain any text, returns and empty string.

`updateLabel (String qualPath)`
: Updates the dialog's label with the name of the directory that is being displayed in *qualPath*.

3.90 `X11TextPane` Class

`X11TextPane` objects display text within a X window. The class defines methods to display text when opening or resizing a window, and scrolling through the text.

The Ctalk distribution's, `demos,` subdirectory contains several example applications of `X11TextPane` objects.

Viewing Text

With a `X11TextPane` window open, the class provides the following keyboard commands.

```
Key(s)                  Action
------                  ------
j, Control-N, <Up>      Scroll the text down by one line.
k, Control-P, <Down>    Scroll the text up by one line.
Control-V               Scroll the text down one screenful.
Control-T               Scroll the text up one screenful.
Control-Q               Go to the start of the text.
Control-Z               Go to the end of the text.
Escape, Control-C       Close the window and exit the program.
```

Adding and Formatting Text

To display text, the class provides the methods `addText`, `putStr`, and `printOn`.

The `addText` method wraps long lines to fit within the window's viewing area, and recognizes newline characters as hard line breaks and inserts soft line breaks where necessary. The methods `putStr` and `printOn` can display shorter items of text at specific X,Y positions.

For word wrapped text, each member of the `textList` instance variable (described below) is a `TextBox` object, which is an iternal class that is used exclusively by *__ctalkWrapText* library function and the `displayText` method.

The class supports both X bitmapped fonts and the X11FreeTypeFont outline font libraries. Applications enable outline font support by calling the `X11FreeTypeFont` method `initFontLib` at the start of the program.

In addition, the class supports different typefaces and simple line formatting with a small set of HTML-like formatting tags.

```
Tag(s)                Action
------                ------
<b>, </b>             Start/end boldface type.
<i>, </i>             Start/end italic (or oblique) type.
<center>, </center>   Start/end a centered line of text.
```

Instance Variables

`leading` An `Integer` that contains the extra space between lines, in pixels. Its value depends on the font that the program uses. The dimensions of Xlib bitmap fonts seems to provide enough space without adding any extra space. Outline fonts, however, seem to need this. The default is 2 (pixels), which the program can set whenever it selects a font. Currently, this variable has no effect with Xlib fonts, though.

`lineHeight`
An `Integer` that contains the height of a line in pixels. Its value is set when the program selects a font. The default is 12 pixels, which is the value used whenever a program doesn't specify any fonts. The total lineHeight is the height of the tallest character ascent `+` the longest character descent, and optionally, any leading, either added by the font designer or the X11TextPane object. Refer to the `font` method, below.

`scrollHeight`
An `Integer` that contains the number of lines that the text moves when scrolling by one screenful. Its value is

```
textPaneObject scrollHeight = textPaneObject viewHeight -
                              textPaneObject scrollMargin
```

`scrollMargin`
An `Integer` that contains the number of lines to overlap when scrolling by pagefuls. Its default value is 3.

`text` A `String` that contains the unformmated text to be displayed in the window. Formatting raw text for display is done by the `addText` method, below.

`textList` This `List` object contains the output of the word wrapping routines. Each item of the `List` is a `TextBox` object, an internal class that stores information about each word of the text.

`textLines`
An `Integer` that contains the number of lines of text after it is formatted.

`viewStartLine`
An `Integer` that contains the number of the first text line that is visible in the window.

`requestClose`
A `Boolean` that indicates whether the `X11TextPane` object has requested to be closed, normally in response to an Escape or Control-C keypress.

`viewXOffset`
An `Integer` that contains the left margin of the text when displayed in the window.

`viewHeight`
`viewWidth`
`Integer` objects that contain the width and height of viewable area, in characters. These are normally determined by the font that the program selects, and after the program calculates the line height (refer to the `font` method, below). The `X11TextPane` class adjusts for variable width fonts and faces whenever necessary (and possible, in some cases).

If the program has selected a font, the `X11TextPane` class calculates the width and height of the viewing area like this.

```
self viewHeight = self size y / self lineHeight;
self viewWidth = self size x / self fontVar maxWidth;
```

If the program doesn't select any fonts, the class uses 14 pixels as the height of each character and 12 pixels as the character width.

Instance Methods

`addText (Object` *`text`*`)`
Adds the argument's text to the receiver pane's `text` instance variable, then word-wraps the entire text into the `textList` instance variable.

`attachTo (Object` *`parentPane`*`)`
Attaches the `X11TextPane` object to its parent pane. The parent pane should always be a `X11PaneDispatcher` object.

`cursorPos (Integer` *`x`*`, Integer` *`y`*`)`
Set the pane's cursor to the coordinates given as arguments.

`background (String` *`colorname`*`)`
Set the background color of the text pane. See the note for `X11Bitmap` class's `background` method. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`clear (void)`
: Clear the pane to the background color.

`displayText (void)`
: Update the pane's text. The pane's window is updated at the next `refresh` message (see below).

`faceRegular (void)`
`faceBold (void)`
`faceItalic (void)`
`faceBoldItalic (void)`
: Selects the typeface of the currently selected font. The font should be selected by a previous call to the `font` method (below). The `font` call gathers information about the type variations if the typeface is available.

`font (String` *`font_descriptor`*`)`
: Loads the bitmap font named by *font_descriptor* and the bold, italic, and bold italic typefaces if they are available, and makes the font named by the argument the currently selected the receiver Pane's currently selected font.

  If a program uses outline fonts, it has more freedom to decide when to select the fonts, because the font libraries operate independently of the program's connection to the GUI. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

  Programs that use Xlib bitmap fonts, however, need to wait until the connection to the GUI is opened, with a call to `openEventStream` (class `X11Pane`). See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

  Here's a code snippet from `X11TextPane` class.

```
X11FreeTypeFont new ftFont;
Integer new ftMaxCharWidth;

self fontDescStr = fontDesc;
self fontVar getFontInfo fontDesc;
(X11Bitmap *)self paneBuffer font fontDesc;

if (ftFont libIsInitialized) {
  self lineHeight = ftFont textHeight "ghyl";
  self lineHeight += self leading;
  ftMaxCharWidth = ftFont textWidth "M";
  self viewWidth = self size x / ftMaxCharWidth;
} else {
  /* Note that we don't add the leading to the lineHeight here */
  self lineHeight = self fontVar height;
  self viewWidth = self size x / self fontVar maxWidth;
}
self viewHeight = self size y / self lineHeight;
```

`foreground (String colorname)`
: Set the forground color of the text pane. See the note for `X11Bitmap` class's `foreground` method. See ⟨undefined⟩ [X11Bitmap], page ⟨undefined⟩.

`gotoXY (Integer x, Integer y)`
: Set the pane's cursor to the coordinates given as arguments. The coordinates are the number of horizontal and vertical pixels from the pane's top-left corner.

  This method is a synonym for `cursorPos`, above.

`new (String paneName)`
: Creates a `X11TextPane` object and initializes the pane's event handlers and other instance data.

  If the argument list contains more than one name, create `X11TextPane` objects with the names given by the arguments.

`printOn (char *fmt, ...)`
: Print the argument, which is a *printf(3)* style format string with arguments for the format conversion specifiers, in the pane's buffer at the position given by the pane's software cursor. To update the visible window with the pane buffer's contents, call the `refresh` method (below), after calling this function.

`putChar (Character c)`
: Write a character in the pane's window at the pane's cursor position.

`putStr (String s)`
: Write a string in the pane's window at the pane's cursor position.

`refresh (void)`
: Update the text displayed in the pane's window.

`subPaneDestroy (Object subPaneReference, InputEvent event)`
: The class's destructor method. This method deletes only the data associated with the pane object's window, not the pane object itself, which is treated like any other object.

`subPaneExpose (Object subPaneReference, InputEvent event)`
: The class's `EXPOSE` event handler. Refreshes the main window from the pane's text buffer.

`subPaneKbdInput (Object subPaneReference, InputEvent event)`
: The handler for `KEYPRESS` and `KEYRELEASE` events from the window system.

`subPaneResize (Object subPaneReference, InputEvent event)`
: The handler for `RESIZENOTIFY` events from the window system.

3.91 `X11TextEditorPane` Class

Objects of `X11TextEditorPane` class create a X window which displays text, and editing commands to perform basic text editing operations.

There is an example program that demonstrates a `X11TextEditorPane` object's use in a text editor program at the end of this section.

Editing Commands

The set of editing commands that a `X11TextEditorPane` object uses is given here, along with their key bindings. You can bind them to different keys by modifying the `handleKbdInput` method.

```
Right, Ctrl-F       Next character
Left, Ctrl-B        Previous character
Up, Ctrl-P          Previous line
Down, Ctrl-N        Next line
PgDn, Ctrl-V        Next page
PgUp, Ctrl-T        Previous page
Home, Ctrl-A        Start of line
End, Ctrl-E         End of line
Ctrl-Q              Start of text
Ctrl-Z              End of text
Ctrl-D              Delete character under cursor
Backspace           Delete previous character.  If selecting text, delete
                    the selection.
Del                 Delete the character under the cursor, or the
                    previous character if at the end of the text.
                    If there is selected text, delete the selection.
Esc                 Close the window and exit the program.

Mouse-1             Move the insertion point cursor to the click.
                    Click and drag the pointer to select text.
Mouse-2             Paste text of the X selection at the insertion point.
```

Cutting and Pasting Text

`X11TextEditorPane` objects can copy text selections to the X primary selection, and paste selected text from other X programs into the program's text.

To select text to be pasted into another application, press the left pointer button and drag the pointer across the text that you want to select. The text should be highlighted with the color defined in the `selectionBackgroundColor` instance variable.

Then, switch to the window that you want to paste the text into, and press the center pointer button at the point where you want to insert the text (or press the left and right buttons simultaneously on machines with two buttons).

Conversely, to paste text into the `X11TextEditorPane` object's contents, select the text in the other application's window, then switch to the `X11TextEditorPane` object's window, and press the center pointer button at the point where you want the text inserted.

If the program has text selected and another program tries to place its contents in the X primary selection, the class will allow the selection ownership to change to the other program. Any text that was selected in the `X11TextEditor` pane's window will no longer be selected.

In general, X programs aren't required to send their to the display's X selection buffers. Many programs only use selected contents internally, and may require another command to

send the content to the X display's primary selection buffer. `X11TextEditorPane` objects maintain the contents of its selection buffer continuously when selecting, but they only send the contents to the X display's primary selection when another program requests it.

Fonts

The demonstration program, `demos/x11/ctedit.ca`, provides options to change the default font and point size. The X11 utility programs *xfontsel(1)* and *fc-list(1)* can display the X bitmap fonts and the Xft library's scalable fonts that are available on the machine, respectively.

Programs configure X fonts within the window's graphics context, using the instance variable `fontVar`, which is inherited from `X11Pane` class.

FreeTypeFonts need to be configured separately from the X window, but the parent `X11Pane` object also defines the `ftFontVar` instance variable, so the program can configure outline fonts before entering its event loop.

If a program is to be configurable for different machines, it should check which of the font libraries are present on the system, and which of the `fontVar` or `ftFontVar` instance variables the program has configured in the program's initialization.

The `X11TextEditorPane` class uses monospaced fonts exclusively. If a program requests a proportionally spaced font, the pane's libraries won't maintain alignment between the displayed text and editing operations.

The sections, `X11FreeTypeFont` and `X11Font` contain more information about how to select fonts. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩, See ⟨undefined⟩ [X11Font], page ⟨undefined⟩.

Instance Variables

`bufLength`

An `Integer` that records the size of the object's text buffer. The class adjusts the buffer's size automatically if necessary.

`button` An `Integer` that records the state of the mouse buttons, i.e., whether they are pressed or not. The values that the the variable might contain are composed of these definitions.

```
#define BUTTON1MASK  (1 << 0)
#define BUTTON2MASK  (1 << 1)
#define BUTTON3MASK  (1 << 2)
```

`foregroundColor`

A `String` that contains the window's foreground color. The variable is included here to facilitate drawing the cursor in reverse video. The `backgroundColor` instance variable is declared in `X11Pane` class. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

An `Integer` that defines the right margin in character columns. Setting this to zero '`0`' sets the line width limit to the line length, so the line width in character columns is calculated as:

```
lineWidth = (window_width_px - left_margin_px - right_margin_px) /
                    character_width_px;
```

`rightMargin`
: The distance in pixels from the right edge of the window, leftward to the right-hand limit of each line's length.

`point`
: An `Integer` that contains the current position in the text where editing occurs.

`selectionBackgroundColor`
: A `String` that contains the background color of selected text. The foreground color is the same as normal text.

`sStart`

`sEnd`

`selecting`
: The `Integer` variables `sStart` and `sEnd` record the beginning and end of selected text as character indexes into the text. The `Boolean` variable `selecting` is true while the pointer is being dragged across text while Button 1 is pressed.

`shiftState`
: An `Integer` that records whether the Shift of Control keys are currently pressed.

`textLength`
: An `Integer` that records the length of the object's text.

Instance Methods

`attachTo (Object` *`parent_pane`*`)`
: Attach a `X11TextEditorPane` object to its parent pane, which is typically a `X11PaneDispatcher` object. This method initializes the size of the pane's window and buffers to the parent pane's dimensions, and positions the pane at the upper left-hand origin of the main window.

`background (String` *`colorName`*`)`
`foreground (String` *`colorName`*`)`
: Sets the foreground and backgrounds color of the pane's window and buffers.

`clearSelection (void)`
: Sets the `sStart` and `sEnd` instance variables to '`0`', cancelling text selection.

`defaultFormat (void)`
: Sets the document-wide margins and text text colors. If the pane is using an `X11FreeTypeFont` object to render text, the font needs to be configured before calling this method. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`displayText (void)`
: Displays the text and editing cursor. Programs should call this method as soon as possible after the program starts the X event loop (which it does by calling the `X11TerminalStream : openEventStream` method), and after every editing operation.

`gotoChar (Integer n)`
: Sets the `point` instance variable to the *n*'th character in the text. If *n* is greater than the length of the text, sets `point` to the end of the text.

`new (String paneName)`
: Initializes the `X11TextEditorPane` object's event handlers, and calls constructors in the `X11TextEditorPane's` superclasses to perform addition initialization. The method `attachTo`, declared in `X11TextPane` class, performs the actual dimensioning of the pane and its buffers. See ⟨undefined⟩ [X11TextPane], page ⟨undefined⟩.

`subPaneKbdInput (X11TextEditorPane subPane, InputEvent event)`
: Handles Keypress and KeyRelease events from the X server. It's possible to reconfigure the editing commands by modifying this method.

  The method works in conjuction with the Ctalk library's editing functions to translate alphanumeric and punctuation characters with the correct shift and control state, and to transmit special keys like arrow keys and Home/End keys untranslated.

`subPaneButtonPress (X11TextEditorPane subPane, InputEvent event)`
: The handler for button press and button release events. This method sets the value of the `button` instance variable, in addition to performing other tasks.

`subPaneMotionNotify (X11TextEditorPane subPane, InputEvent event)`
: The handler method for pointer motion events.

`subPaneResize (X11TextEditorPane subPane, InputEvent event)`
: Handles resizing the `X11TextEditor` pane's dimensions in response to a window resize event. This method is a no-op in the current release.

`subPaneSelectionClear (X11TextEditorPane subPane, InputEvent event)`
: Updates the program's state after receiving a a SelectionClear event. Normally this occurs when another program requests the X primary selection. The method updates this program's state so that it is no longer selecting, and redraws the text.

Example Text Editing Program

```
/*
  ctedit.ca - Basic text editor using X11TextEditorPane class.

  Usage:

    ctedit [<options>] <filename>

  Typing, "ctedit -h" displays a list of options.

  Pressing Esc or selecting "Close" from the window menu
  exits the program and saves the edited text.

  If <filename> exists, ctedit renames the previous version
```

```
  of the file to <filename>.bak.  If <filename> doesn't
  exist, ctedit creates a new file.

  The editing commands are set in the X11TextEditorPane :
  handleKbdInput method.  They are:

     Right, Ctrl-F       Next character
     Left, Ctrl-B        Previous character
     Up, Ctrl-P          Previous line
     Down, Ctrl-N        Next line
     PgDn, Ctrl-V        Next page
     PgUp, Ctrl-T        Previous page
     Home, Ctrl-A        Start of line
     End, Ctrl-E         End of line
     Ctrl-Q              Start of text
     Ctrl-Z              End of text
     Ctrl-D              Delete character under cursor
     Backspace           Delete previous character
     Del                 At the end of the text, delete the
                         previous character.  Otherwise delete
                         the character under the cursor.
     Esc                 Close the window, save the edited text,
                         and exit the program.
 */

#define WIN_WIDTH 500
#define WIN_HEIGHT 340
#define WIN_X 25
#define WIN_Y 30
#define FIXED_FONT "fixed"
#define DEFAULT_BG "white"
#define DEFAULT_FG "black"
#define DEFAULT_FT_FONT "DejaVu Sans Mono"
#define DEFAULT_FT_PTSIZE 12.0

Application new ctEdit;
String new geomString;
String new infileName;
String new xFontName;
String new ftFontName;
Float new ftFontSize;
String new bgColor;
String new fgColor;

Boolean new createFile;

Boolean new useFtFonts;
```

```
X11FreeTypeFont new ftFont;
Boolean new useXFont;

void exit_help () {
  printf ("usage: ctedit [-h] | [-g <geom>] [-fg <color>] "
  "[-bg <color>] [-fn <font> ] <filename>\n");
  printf ("-bg <color>    Set the window background to <color>.\n");
  printf ("-fg <color>    Display the text using <color>.\n");
  printf ("-fn <font>     Use the X <font> to display the text. See xfontsel(1).\n");
  printf ("-ft <font>     Use the FreeType <font> to display the text. See\n");
  printf ("               X11FreeTypeFont class.\n");
  printf ("-g  <geom>     Set the window geometry to <geom>. See XParseGe-
ometry (3).\n");
  printf ("-h             Print this message and exit.\n");
  printf ("-xfonts        Use X bitmap fonts, even if outline fonts are available.\n")
  exit (1);
}

/* UNIX-compatible line ending. */
#define LF 10

X11TextEditorPane instanceMethod writeOutput (String infileName) {
  "Create a backup of the previous version of the file, if any,
   and check that the text ends with a UNIX-standard newline
   (ASCII 10) character."
  WriteFileStream new writeFile;
  Character new c;

  c = self text at (self text length - 1);
  if (c != LF) {
    self text += "\n";
  }

  if (!createFile)
    writeFile renameFile infileName, infileName + ".bak";
  writeFile openOn infileName;
  writeFile writeStream (self text);
  writeFile closeStream;

}

Application instanceMethod commandLineOptions (void) {
  Integer new i, nParams;
  String new param;


  nParams = self cmdLineArgs size;
```

```
  for (i = 1; i < nParams; i++) {

    param = self cmdLineArgs at i;

    if (param  == "-g") {
      ++i;
      geomString = self cmdLineArgs at i;
      continue;
    }
    if (param == "-fn") {
      ++i;
      xFontName = self cmdLineArgs at i;
      continue;
    }
    if (param == "-bg") {
      ++i;
      bgColor = self cmdLineArgs at i;
      continue;
    }
    if (param == "-fg") {
      ++i;
      fgColor = self cmdLineArgs at i;
      continue;
    }
    if (param == "-ft") {
      ++i;
      ftFontName = self cmdLineArgs at i;
      continue;
    }
    if (param == "-xfonts") {
      useXFont = True;
      continue;
    }
    if (param == "-pt") {
      ++i;
      ftFontSize = (self cmdLineArgs at i) asFloat;
      continue;
    }
    if (param == "-h" || param == "--help" || param == "--h" ||
param at 0 == '-') {
      exit_help ();
    }

    infileName = param;

  }
```

```
}

Application instanceMethod winDimensions (void) {
  if (geomString length > 0) {
    self parseX11Geometry geomString;
    if (self winWidth == 0) {
      self winWidth = WIN_WIDTH;
    }
    if (self winHeight == 0) {
      self winHeight = WIN_HEIGHT;
    }
    if (self winXOrg == 0) {
      self winXOrg = WIN_X;
    }
    if (self winYOrg == 0) {
      self winYOrg = WIN_Y;
    }
  } else {
    self winWidth = WIN_WIDTH;
    self winHeight = WIN_HEIGHT;
    self winXOrg = WIN_X;
    self winYOrg = WIN_Y;
  }
}

Application instanceMethod findFtFonts (void) {

  if (useFtFonts && !useXFont) {
    ftFont initFontLib;
    ftFont selectFont ftFontName, 0, 80, 72, ftFontSize;
    ftFont namedX11Color fgColor;
  }

}

int main (int argc, char **argv) {
  X11Pane new xPane;
  X11PaneDispatcher new xTopLevelPane;
  X11TextEditorPane new xEditorPane;
  InputEvent new e;
  Exception new ex;
  X11Cursor new watchCursor;
  ReadFileStream new readFile;
  String new winTitle;

  geomString = "";
  xFontName = FIXED_FONT;
```

```
bgColor = DEFAULT_BG;
fgColor = DEFAULT_FG;
infileName = "";
useFtFonts = True;
useXFont = False;
ftFontSize = DEFAULT_FT_PTSIZE;
ftFontName = DEFAULT_FT_FONT;

ctEdit parseArgs argc, argv;
ctEdit commandLineOptions;
ctEdit winDimensions;

if (ftFont version < 10) {
  useFtFonts = false;
} else {
  ctEdit findFtFonts;
}


if (infileName length == 0) {
  exit_help ();
}

if (!readFile exists infileName) {
  createFile = true;
  winTitle = infileName + "   (New file)";
} else {
  readFile openOn infileName;
  xEditorPane text = readFile readAll;
  readFile closeStream;
  winTitle = infileName;
  createFile = false;
}

xPane initialize ctEdit winXOrg, ctEdit winYOrg,
  ctEdit winWidth, ctEdit winHeight, ctEdit geomFlags, winTitle;

xTopLevelPane attachTo xPane;
xEditorPane attachTo xTopLevelPane;
xPane map;
xPane raiseWindow;
watchCursor watch;

xPane openEventStream;

/* This sets the maximum line width to the width of the window. */
xEditorPane lineWidth = 0;
```

```
  if (!useFtFonts || useXFont) {
    xEditorPane foreground fgColor;
    xEditorPane font xFontName;
    xEditorPane defaultFormat;
  } else {
    xEditorPane defaultFormatFT ftFont;
  }

  xEditorPane background bgColor;
  xEditorPane clear;

  xPane defaultCursor;

  while (TRUE) {
    xPane inputStream queueInput;
    if (xPane inputStream eventPending) {
      e become xPane inputStream inputQueue unshift;
      xPane subPaneNotify e;
      if (ex pending)
  ex handle;
      switch (e eventClass value)
{
  /*
   *  Handle both types of events in case the window
   *  manager doesn't distinguish between them.
   */
case MOVENOTIFY:
  break;
case RESIZENOTIFY:
  break;
case EXPOSE:
  xEditorPane displayText;
  break;
case WINDELETE:
  xEditorPane writeOutput infileName;
    xPane deleteAndClose;
  exit (0);
  break;
default:
  break;
}
    } else {
      if (xEditorPane requestClose) {
xEditorPane writeOutput infileName;
xPane deleteAndClose;
exit (0);
```

```
        }
      }
    }
  }
```

3.92 `Symbol` Class

`Symbol` class objects represent memory locations. They can refer to objects, C variables, buffers, I/O ports, and other data located in memory.

As an alternative, if you receive an `Unimplemented C type` warning, you can store various types of C data in `Symbol` objects. See ⟨undefined⟩ [Objects in Function Arguments], page ⟨undefined⟩.

A unary '`*`' operator behaves similarly to the C operator. That is, it refers to the value of the `Symbol` object, not the object itself. The following example might make this clearer.

```
int main () {

  Symbol new sym1;
  Symbol new sym2;
  Symbol new sym3;
  Integer new i;

  i = 2;

  sym3 = sym1;   /* Save the original value of sym1. */

  sym1 = i;
  printf ("%d\n", sym1);

  *sym2 = i;
  printf ("%d\n", *sym2);

  sym1 = sym3;  /* Restore sym1 to the original object. */

  i = 4;

  *sym1 = i;

  printf ("%d\n", *sym1);
}
```

Instance Variables

`value` The value of a `Symbol` is a hexadecimal memory address.

Instance Methods

`* (void)` The `*`. A shortcut for the `getValue` method, below.

`= (void *v)`
Assign a reference to the argument as the value of the receiver. If *v* is also a `Symbol` object, simply copy the reference. This method is a synonym for `symbolReference`, below. If you want to use multiple levels of object references and dereferences, see the `addressOf` method in `Object` class See ⟨undefined⟩ [Object], page ⟨undefined⟩, and the `deref` method, below.

`asAddrString (void)`
Returns a `String` object with the formatted hexadecimal address of the object pointed to by the receiver.

`asString (void)`
Returns the receiver's value as a `String` object with the value of the `char` string that the `Symbol` receiver object points to. The result object has the same name as the receiver and the class of `String`.

`basicObject (char *name, char *classname, char *superclassname, char *value_expr)`
Create a basic object and make the receiver's value point to it. The new object has the scope '`LOCAL_VAR|VAR_REF_OBJECT`' and a reference count of 1.

Referencing the object with a `Symbol` makes it easy to create objects and then assign them to C variables in the calling method or function. For example,

```
Symbol new s;
OBJECT *int_object;

s basicObject "new_int", "Integer", "Magnitude", "10";

int_object = s getValue;

printf ("%d\n", int_object value);
```

`deref (void)`
Return the object referred to by the receiver. This method is functionally equivalent to the C '`*`' operator.

`getValue (void)`
Return the object that the receiver's value (previously set by `=` or `symbolReference`) refers to. If the address doesn't refer to an object, returns the receiver and generates an exception, which the program can handle in whatever manner is necessary. In some cases, the internals of Ctalk's object-to-C routines can also generate an warning message.

Note that when assigning a non-object data address to a C variable like a `void *`, Ctalk allows both of these expressions:

```
void *myVoidPtr;

myVoidPtr = mySymbol;

myVoidPtr = *mySymbol;   /* Generates an exception. */
```

`name (void)`
: Return a new `String` object containing the receiver's name.

`removeValue (void)`
: Remove the reference to the target object from the receiver. Delete the target object if there are no further references to it.

`symbolReference (void *v)`
: Return a new `String` object containing the receiver's name.

3.93 `Key` Class

Objects of class `Key` are key-value pairs. Other classes and programs can use `Key` objects' names when looking up objects. The value of a key object is a reference to the value object.

Many of the `Collection` subclasses are composed entirely of `Key` objects, which act as the "glue" that maintains references to the the actual contents of the collection. See ⟨undefined⟩ [Collection], page ⟨undefined⟩.

Programs can also manipulate `Key` objects independently. Most of the math operators that work with collections actually work with `Key` objects. So it's important to add the attribute OBJECT_IS_MEMBER_OF_PARENT_COLLECTION to a `Key` object when building collections. This tells Ctalk that the `Key` object can be used independently, as well as part of its parent collection. See ⟨undefined⟩ [Attributes], page ⟨undefined⟩.

Here's a program that manipulates the `Key` objects of a collection (here, an `AssociativeArray`) directly. See ⟨undefined⟩ [AssociativeArray], page ⟨undefined⟩.

```
int main (int argc, char **argv) {

  AssociativeArray new a;
  Key new k;

  a atPut "key1", "value1";
  a atPut "key2", "value2";
  a atPut "key3", "value3";
  a atPut "key4", "value4";

  k = *a;

  while (++k)
    printf ("%s --> %s\n", k name, *k);

}
```

Instance Variables

`value` The value is the formatted representation of a hexadecimal pointer to a memory address.

Instance Methods

`+ (Integer` *n*`)`

Increments the receiver by *n*. For a `Key` object, this sets the receiver to the *nth* successive element in a collection. The increments are numbered with '`1`' pointing to the first member of the collection, and so on. If there are no more elements, the receiver's value is NULL. For an example, refer to `-`, below.

`++` Increments the receiver to point to the next `Key` in a collection. If the receiver is already the last item item in the collection, the value of the receiver after it is incremented is NULL. This method works as both a prefix and postfix method, and increments the receiver either before or after it is referenced, respectively.

Here is an example of how to iterate over an `AssociativeArray` using `++`.

```
AssociativeArray new a;
Key new k;

a atPut "key1", "value1";
a atPut "key2", "value2";
a atPut "key3", "value3";
a atPut "key4", "value4";

k = *a;

while (++k)
  printf ("%s --> %s\n", k name, *k);
```

`- (Integer` *n*`)`

Decrements the receiver by *n*. For a `Key` object, this sets the receiver to the *nth* previous element of the collection that the receiver is a member of. Here is a brief example

```
int main () {

  AssociativeArray new a;
  Key new k;

  a atPut "1", "value1";
  a atPut "2", "value2";
  a atPut "3", "value3";
  a atPut "4", "value4";
```

```
  k = *a;
  printf ("%s --> %s\n", k name, *k);
  k = k + 3;
  printf ("%s --> %s\n", k name, *k);
  k = k - 1;
  printf ("%s --> %s\n", k name, *k);
}
```

Running this program produces the following output.

```
1 --> value1
4 --> value4
3 --> value3
```

`--` Decrements the receiver to point to the previous `Key` in a collection. If the receiver is the first item in the collection, the value of the receiver after it is decremented is NULL. Like `++`, this method works as both a prefix and postfix method, and decrements the receiver either before or after it is referenced, respectively.

`=` If the receiver refers to an object reference (that is, preceded by a '`*`' operator), sets the value of the receiver to the address of the argument. Otherwise, sets the receiver to refer to the argument.

`getKeyObject (void)`
Return the receiver.

`setName (char *`*`key_name`*`)`
Set the receiver's name to the argument, a `String` object.

Note: In some cases, the object's name is the only way that Ctalk can refer to it. In that case, the program needs to maintain an alias to the object, like an `OBJECT *,` so that it can refer to the object later. In the following example, the program can refer to `keyObject` by using `key_alias_ptr,` regardless of the object's name.

```
Key new keyObject;
OBJECT *key_alias_ptr;
...
key_alias_ptr = KeyObject setName keyNameString;
```

`ulink (void)`
Detach the receiver object from its parent collection. The method also removes the OBJECT_IS_MEMBER_OF_PARENT_COLLECTION attribute from the receiver.

3.94 `Vector` Class

Objects of `Vector` class refer to blocks of memory of arbitrary length. The memory area that the `Vector` object points to may contain any data, including NULL bytes.

Whenever a program or method assigns a `Vector` object a new area of memory, the program or method also sets the `Vector` object's `length` instance variable; for example, the method `readVec` (class `ReadFileStream`) records the memory area's size in the `length` instance variable after it has read a chunk of data from a disk file. See ⟨undefined⟩ [ReadFileStream], page ⟨undefined⟩.

Here's an example program that writes a copy of a JPEG image file.

```
int main () {
  ReadFileStream new readF;
  WriteFileStream new writeF;
  FileStream new f;
  Vector new vec;
  LongInteger new size;

  readF openOn "original.jpeg";
  readF statStream;
  size = readF streamSize;

  vec = readF readVec size;

  writeF openOn "copy.jpeg";
  writeF writeVec vec;

  writeF closeStream ;
  readF closeStream;
}
```

Instance Variables

`length` An `Integer` that contains the size in bytes of the memory area that the `Vector` object points to.

Instance Methods

`+ (Vector` *v*`)`
: Return a `Vector` object that is the concatenation of the receiver and the argument.

`+= (Vector` *v*`)`
: Concatenate the receiver with the vector given as the argument.

`asString (void)`
: Returns the value of the receiver as a `String` object terminated with a NUL byte to the value of the receiver's `length` instance variable. Does not check for NULs or non-printing characters in the value, so the returned `String` may still be truncated to less than the receiver's length.

`basicNew (char *name, char *value, int value_length`
`basicNew (char *name, char *classname, char *superclassname, char *value, int value_length)`

Create a new `Vector` object with the name, contents, length, and, optionally, class name and superclass name given as the arguments.

For the five-argument form of `basicNew`, the class should be `Vector` and the superclassname should be `Symbol`, unless the program has subclassed `Vector`.

In the three-argument form, the receiver must be a member of `Vector` class or its subclasses, in which case the method takes the class and superclass from the receiver, as in this example.

```
myBuf = Vector basicNew "memorybuf", memory_buf_ptr, memory_buf_length;
```

The *value* argument may be any memory address that points to data of arbitrary size, and may contain any data, including NULL bytes.

The *value_length* argument supplies the length of the memory segment in bytes, which the returned `Vector` object stores in its `length` instance variable.

Most of the internal work of setting the values and attributes of the returned object, and registering the memory area, is performed by *__ctalkSetObjectValueAddr*, which each of these methods call. See ⟨undefined⟩ [ctalkSetObjectValueAddr], page ⟨undefined⟩.

It's also necessary that these methods take care of other initialization tasks which are necessary for all classes of objects. They're described in general in the description of `basicNew` (`Object` class). See ⟨undefined⟩ [ObjectbasicNew], page ⟨undefined⟩.

`contains (String pattern)`
`contains (String pattern, Integer start_offset)`

With one argument, returns an `Integer` with the byte offset of the first occurence of *pattern* in the receiver, starting from the beginning of the receiver's value, or -1 if the pattern is not found.

If a second argument is given, it contains the `Integer` offset into the buffer where the method begins its search. That allows programs to find multiple matches of the same pattern in the receiver's value.

4 Methods

The following sections outline some of the programming features that Ctalk methods implement. The *Ctalk Tutorial* provides step-by-step instructions on how to write some common method types.

Declaring a method is similar to declaring a C function, but the declaration syntax adds some additional features.

- The declaration specifies the class that the method belongs to.
- The statement allows the method to be identified by an alias, so the method can overload operators. Classes can implement overloaded methods in other ways, too. See ⟨undefined⟩ [Overloading], page ⟨undefined⟩.

To declare an instance method, use the `instanceMethod` keyword, as in the following example.

```
String instanceMethod = set_value (char *s) {

 ... method statements

}
```

This example declares an instance method, `=`, which is recognized by objects that belong to the class `String` and its subclasses. The `=` in the declaration overloads C's `=` operator, so that, instead of assigning a value to a C variable, the method sets the value of its receiver.

In this example, the receiver is an instance of class `String`.

```
newObject = "Hello, World!";
```

If the variable reference immediately preceding `=` refers to a C variable, then `=` behaves as the C operator, `=`.

Ctalk can use most C operators as methods, with the exception of parentheses and prefix operators. Receivers always precede the method message.

For example, the `Integer` methods `invert` and `bitComp` perform the same operations as the C prefix operators `!` and `~`.

```
int i;              /* C variable. */
Integer new myInt;  /* Ctalk object. */

i = 0;
myInt = 0;

/* These two statements are equivalent. */
printf ("%d", !i);
printf ("%d", myInt invert).

/* And so are these two statements. */
printf ("%d", ~i);
printf ("%d", myInt bitComp);
```

4.1 Declaring Methods

The declaration syntax for instance methods is:

classname `instanceMethod` [*alias*] *funcname* (*args*) { *method body* }

and for class methods:

classname `classMethod` [*alias*] *funcname* (*args*) { *method body* }

If *alias* is omitted, Ctalk refers to the method by *funcname*.

In the example at the beginning of this section, if `=` were omitted, a program could refer to the method by the message, `set_value`.

With the exception of the primitive methods `class`, `new`, `classMethod`, and `instanceMethod`, Ctalk declares methods in its class library, or in the program input. See ⟨undefined⟩ [Classes], page ⟨undefined⟩.

4.2 Method Parameters

You can use Ctalk class objects to declare method parameters, but you can also declare parameters using C data types, in order to prevent class libraries from being loaded recursively. This is necessary in many of the basic classes.

Regardless of how you declare method parameters, when the method's message is sent to an object, Ctalk translates C language arguments into class objects.

For example, these two declarations are equivalent.

```
Integer instanceMethod + add (int i) {
...
}

Integer instanceMethod + add (Integer i) {
...
}
```

Ctalk does not use objects in C function arguments, so if you need to use objects as parameters, you must write a method instead of a C function.

4.3 Method Application Programming Interface

This section describes the C functions and macros that methods use to interface with the Ctalk run time libraries and with each other.

At this point, the method API is still developing, and relies heavily on C library functions.

This example shows method `+` of class `Integer`, which adds two Integer objects, creates an object for the result, and returns the result.

```
Integer instanceMethod + add (int i) {

  OBJECT *op1, *op2;

  op1 = self value;
  op2 = i value;
```

```
    methodReturnInteger(atoi(op1->__o_value) + atoi(op2->__o_value))
}
```

The keyword `self` refers to the receiver of the method. See ⟨undefined⟩ [Self and super], page ⟨undefined⟩.

The variables `op1` and `op2` are the `value` instance variables of the receiver and the argument. Ctalk has a method, `value` (class `Object`), which returns the `value` instance variable, but you can also refer to instance variables by sending an instance variable's name as a message to an object.

Internally, the receiver, arguments, and return value of a method are all '`OBJECT *`'. When you use C functions with objects, you use the members of an OBJECT type, a `struct.` (See ⟨undefined⟩ [OBJECT typedef], page ⟨undefined⟩.) If you declare an object as a C `OBJECT *,` then the method uses it like any other C `struct *.`

All methods return an '`OBJECT *`' as the result. The macro `methodReturnInteger` defines the object that the method returns. See ⟨undefined⟩ [Return values], page ⟨undefined⟩.

The `add` method is in `classes/Integer`. The file `include/object.h` contains the declaration of the '`OBJECT`' type.

4.4 `self` and `super`

The keywords `self` and `super` refer to the method's receiver object and a method in a superclass of the receiver's class, respectively.

Refer to the method `+` (class `Integer`) from the previous section. If you call `+` with the statement below, `self` in the `+` method would refer to `i.`

```
j = i + 2;
```

The `super` keyword refers to a method in one of the superclasses of the receiver. It is commonly used in constructors.

For example, if a class implements its own `new` method that performs additional initialization for instances of a class, it can also call `new` implemented by a superclass.

This constructor from the `ReadFileStream` method, `new`, also calls `new` from class `Object.`

```
ReadFileStream instanceMethod new (char *name) {

  ReadFileStream super new name;

  __ctalkInstanceVarsFromClassObject (name);

  ReadFileStream classInit;

  return name;
}
```

The first statement, `ReadFileStream super new name,` calls `new` from class `Object` before performing its own initialization.

4.5 Class Initialization

Some classes that have class variables require initialization when the program is run. For example, class `ReadFileStream` needs to have its class variable `stdinStream` set to the application's `stdin` file stream before the program can read from standard input. For this, you would use the library function *xstdin*, and use its value in `stdinStreamVar`.

```
__ctalkObjValPtr (stdinStreamVar, xstdin ());
```

Classes that require class variable initialization need to define a method, `classInit`, which is called by a constructor when the first object is created. An example is the `ReadFileStream` method `new` from the previous section.

Here is the `classInit` method for class `ReadFileStream`.

```
ReadFileStream classMethod classInit (void) {
  "Initializes the classes' standard input stream.  This
  method is normally called by the ReadFileStream
  constructor.  The method needs to be called only once;
  on further calls to 'new', it is a no-op."
  OBJECT *classObject,
    *stdinStreamVar,
    *classInitVar;

  if (self classInitDone)
    return NULL;

  classObject = __ctalkGetClass ("ReadFileStream");
  stdinStreamVar = __ctalkFindClassVariable ("stdinStream", TRUE);
  __ctalkInstanceVarsFromClassObject (stdinStreamVar);

  __ctalkObjValPtr (stdinStreamVar, xstdin ());

  classInitVar =
    __ctalkCreateObjectInit ("classInitDone", "Integer",
     "Magnitude",
     classObject -> scope, "1");
  __ctalkAddClassVariable (classObject, "classInitDone", classInitVar);

  return NULL;
}
```

A `classInit` method needs to be called only once. The method checks the `classInitDone` instance variable to determine the method has already performed the initialization that the class needs.

4.6 Overloading C Unary Operators

Methods can also overload C unary operators like '`*`', '`&`', '`!`', '`~`', and '`sizeof`'. For most unary operators, if the class defines a method to overload the operator, then Ctalk uses the method; otherwise, it treats the operator as a normal C operator.

Methods that overload unary operators contain the `__prefix__` attribute in their declaration. This simplifies method writing considerably, because that allows a class to define two different method '`*`' methods, for example, one a unary dereference operator, the other a multiplication operator.

In the method's declaration, use the `__prefix__` attribute in place of paramter declarations, as in this example.

```
String instanceMethod * deref_prefix (__prefix__) {
... Method statements here ...
}
```

When Ctalk encounters a '`sizeof`' operator, it checks whether the argument is a Ctalk object, or a C variable or type cast expression. If the argument is a C variable or type cast expression, then Ctalk uses the C `sizeof` operator. Otherwise, it uses the `sizeof` method (`Object` class). This method's implementation is simple: it treats a Ctalk object as an `OBJECT *`, and returns the size in bytes of a pointer on the system. On 32-bit systems, the `sizeof` method always returns 4. However, you can overload the '`sizeof`' operator in your own classes if necessary.

You need to take care when using unary operators in complex statements, due to the precedence of Ctalk's method messages. In the following highly experimental example, you would need to use parentheses to specify the order of evaluation, because the expression needs the result of `(*s) asString` to be a `String` also.

```
String new s;
String new sFirstChar;
Character new c;
Integer new i;
s = "Hello, ";
sFirstChar = (*s) asString;
```

Ctalk also provides postfix equivalents for many of these operators, like `deref` (class `Symbol`) and `invert` (`Character`, `Integer`, and `LongInteger` classes).

Ctalk can overload these operators even if the class defines a method that uses the operator with an argument. For example, the pointer ('`*`') and unary minus ('`-`') operators can have different methods than the methods that perform multiplication and subtraction.

The subject of method overloading is discussed further in its own section. See ⟨undefined⟩ [Overloading], page ⟨undefined⟩.

4.7 Translating C Variables into Object Values

The `OBJECT` C data type that Ctalk uses internally stores objects' values as C `char *`'s. See ⟨undefined⟩ [OBJECT typedef], page ⟨undefined⟩.

To store numeric values, you must format them as character strings, using, for example, `sprintf`. In many cases, however, Ctalk performs this translation automatically.

`Integer` objects can also translate values with *atoi(3)* or *__ctalkDecimalIntegerToASCII*. Other scalar data types have corresponding functions that translate numbers to character strings.

If for some reason you need to translate an object to its C type or back again, the Ctalk library provides the following functions.

`__ctalkToCArrayElement (OBJECT *o)`
: Translate the value of an `Integer`, `Character`, `String`, or `LongInteger` array element to a `void *` that points to its corresponding C data type.

`__ctalkToCCharPtr (OBJECT *o, int keep)`
: Translate the value of a `String` object into a C `char *`. If *keep* is `FALSE`, delete *o*.

`__ctalkToCIntArrayElement (OBJECT *o)`
: Translate the value of an `Integer` array element to an `int`.

`__ctalkToCInteger (OBJECT *o, int keep_object)`
: Translate the value of an `Integer` to an `int`. If *keep_object* is `FALSE`, then delete the object.

`__ctalkToCInteger (OBJECT *o, int keep_object)`
: Translate the value of an `Integer` to a `long int`. If *keep_object* is `FALSE`, then delete the object.

`__ctalk_to_c_char_ptr (OBJECT *o)`
: Translate the value of a `String` object into a C `char *`. Note: This function is being phased out. Use *__ctalkToCCharPtr ()*, above, instead.

`__ctalk_to_c_double (OBJECT *o)`
: Translate the value of a `Float` object into a C `double`.

`__ctalk_to_c_int (OBJECT *o)`
: Translate the value of an `Integer` object into a C `int`.

`__ctalk_to_c_longlong (OBJECT *obj, int keep)`
: Translate the value of a `LongInteger` object into a C `long long int`. If *keep* is non-zero, does not delete *obj*, or, in compound expressions that need to interface with C variables, (like complex if-statement conditionals), whether to delete the library's internal C variable registrations.

  Formats the arguments of the second-most calling method and prints them on the receiver. This function is meant to be used within a method that is called by another method. One method that uses this function is `String : vPrintOn`. Refer to `String : vPrintOn` for an example of this function's use. See ⟨undefined⟩ [vPrintOn–class String], page ⟨undefined⟩

`__ctalkCBoolToObj (bool b)`
: Create a `Boolean` object with the boolean (either true or false) value of the argument.

`__ctalkCCharPtrToObj (char *s)`
: Create a `String` object with the value *s*.

`__ctalkClassLibraryPath (void)`
: Return a `char *` containing the directories that Ctalk searches for class libraries. When compiling programs Ctalk searches first for directories given as arguments to the '`-I`' option, then directories given by the `CLASSLIBDIRS` environment variable, then the standard class library directory which is defined when Ctalk is built. In addition, for each directory, if a subdirectory named `ctalk` exists, then Ctalk searches that subdirectory also.

  The standard library directory defaults to `/usr/local/include/ctalk`, although if you define a different '`--prefix`' when building and installing Ctalk, the class libraries will be located in *prefix*`/include/ctalk`.

  If it is necessary to look up class libraries at run time, Ctalk first searches the directories listed by the `CLASSLIBDIRS` environment variable, then the default class library directory mentioned above.

`__ctalkClassSearchPath (void)`
: A synonym for *__ctalkClassLibraryPath()*, above.

`__ctalkCDoubleToObj (double f)`
: Create a `Float` object with the value *f*.

`__ctalkCIntToObj (int i)`
: Create an `Integer` object with the value *i*.

`__ctalkCLongLongToObj (long long int l)`
: Create a `LongInteger` object with the value *l*.

`__ctalkCSymbolToObj (unsigned int label)`
: Create a `Symbol` object with the value that is the address of *label*. This is used for expressions where a function name appears by itself, for example. In that case, the result is a `Symbol` object with the address of the function.

The functions that create objects give the objects the scope of the function call, either global or local.

Note: In future versions of Ctalk, the names of these functions are likely to change.

4.8 C Macros

Ctalk also provides many macros that help standardize the Ctalk-to-C conventions. They're defined in the `ctalkdefs.h` include file. To use them, include `ctalkdefs.h` in a source file or class library.

```
#include <ctalk/ctalkdefs.h>
```

Some of the macro definitions in `ctalkdefs.h` are described here.

`ARG`
: With a numeric argument, retrieves the n'th method or template argument from the stack; i.e., `ARG(0)` refers to the first argument on the stack, `ARG(1)` retrieves the second argument, and so on.

`CLASSNAME`
: Returns an object's classname.

Note: You should use this macro in new code. While typing `__o_classname` directly with an `OBJECT *` should work for a while, it's going to be phased out. Using __o_classname with an object and the `->` method is still okay, though. For example:

```
OBJECT *myObjRef;
String new myString;

/* The use of __o_classname as struct member is going away... */
myObjRef -> __o_classname;
/* Instead, write this. */
myObjRef -> CLASSNAME;

/* These are still okay, because -> is a method, not the C operator. */
myString -> __o_classname;
self -> __o_classname;

/* The same is true for the SUPERCLASSNAME definition. */

SUPERCLASSNAME(myObjRef);

self -> SUPERCLASSNAME;
myString ->  SUPERCLASSNAME;
```

`FILEEOF` Writes an fEOF to the `char *` buffer given as its argument.

`FMT_0XHEX`

When used with a function like *sprintf ()*, formats a pointer into its string representation. For example:

```
char buf[64];
OBJECT *my_object_ptr;

....   /* Do stuff. */

sprintf (buf, FMT_0XHEX(my_object_ptr));
```

However, this macro is not very portable and using functions that use stdargs (e.g., printf, scanf, etc.) can be cumbersome. Library functions like *__ctalkGenericPtrFromStr ()* See ⟨undefined⟩ [ctalkGenericPtrFromStr], page ⟨undefined⟩, and *__ctalkFilePtrFromStr ()* See ⟨undefined⟩ [ctalkFilePtrFromStr], page ⟨undefined⟩, might be faster and more reliable.

`IS_OBJECT`

Returns True or False if its argument, a C pointer, refers to a valid object.

`IS_VALUE_INSTANCE_VAR`

Returns True or False if its argument, a C pointer, refers to the value instance variable of an object.

`MEMADDR` Casts its operand to a `void **`, which is what Ctalk's internal function *__xfree ()* uses when freeing memory. Normally you should use __ctalkFree () to free memory, but the MEMADDR macro is here in case you want to call __xfree () directly. Refer to the entry for __ctalkFree () for details. See ⟨undefined⟩ [ctalkFree], page ⟨undefined⟩.

`STR_0XHEX_TO_PTR`
Does the converse of `FMT_0XHEX;` it converts the string representation of a pointer into an actual pointer, when used with a function like *sscanf ()*. For example:

```
OBJECT *my_object;
OBJECT *my_object_ptr;

sscanf (my_object -> __o_value, STR_0XHEX_TO_PTR(my_object_ptr));
```

Again, using stdargs functions can be cumbersome and not very portable. In many cases, *__ctalkObjValPtr ()* accomplishes the same thing. See ⟨undefined⟩ [ctalkObjValPtr], page ⟨undefined⟩.

`SUPERCLASSNAME`
Returns an object's superclass name. This macro should be used only with OBJECT *'s, as it is rather extensive and written in C. Returns an empty string if the object doesn't have a superclass. See ⟨undefined⟩ [CLASSNAMEMacro], page ⟨undefined⟩, above.

`__LIST_HEAD(List *l)`
When given an argument that is a collection like a `List` object, returns the first member of the collection.

`STR_IS_NULL(char *s)`
Evaluates to True if the `char *` argument evaluates to zero; i.e., its value is '`(null)`', '`0`', '`0x0`' or the first character is an ASCII `NUL` ('`\0`') byte.

4.9 Required Classes

The keyword `require` tells `ctalk` to preload the class given as its argument before any other classes or methods.

The following lines appear in the `ctalklib` library.

```
require Object;
require Symbol;
require String;
```

These statements tell `ctalk` to first load the `Object` class and its methods, and then load the class `Symbol` and its methods, and then class `String.` Ctalk loads the class and its methods at that point in the program, before further processing of the source file.

The `require` keyword always occurs in a global scope; that is, outside of any method or function.

4.10 Scope of Objects

In Ctalk, an object can have a number of different scopes, and Ctalk implements many more scopes for objects than for C variables.

All of the scopes are available when creating and modifying objects. In practice, however, you should only need to use a few of them. Ctalk uses many of the scopes internally.

These are the scopes that Ctalk implements.

`GLOBAL_VAR`
: An object that is declared globally; that is, outside of any function or method.

`LOCAL_VAR`
: An object declared within a function or method.

`ARG_VAR`
: An object derived from a C function argument that is used within a Ctalk expression. This scope is normally used only internally.

`RECEIVER_VAR`
: An object created internally for receiver objects that do not already have objects. Ctalk assigns RECEIVER_VAR objects C constants and constant expressions when they are used as receivers.

`PROTOTYPE_VAR`
: Used by the front end when evaluating objects that are declared as method parameters.

`CREATED_PARAM`
: Used mainly for temporary objects that are derived from C constants that are arguments to methods and functions.

`CVAR_VAR_ALIAS_COPY`
: Used for temporary and non-temporary objects that are created from C variables.

`CREATED_CVAR_SCOPE`
: This is a combination of CVAR_VAR_ALIAS_COPY|LOCAL_VAR scopes. Used for C variable objects that are only needed for the duration of an expression. Also used for other values that aren't needed after an expression has been evaluated, like boolean method return values.

`SUBEXPR_CREATED_RESULT`
: Used internally for temporary objects that are the results of subexpressions.

`VAR_REF_OBJECT`
: Used for objects that are referred to by other objects; for example an object referred to by a `Symbol` object may have this scope set.

`METHOD_USER_OBJECT`
: This scope is used mostly internally for objects that are returned by methods and saved as method resources.

`TYPECAST_OBJECT`
: Used internally for temporary objects that are derived from C type cast expressions.

You set a new object's scope with the *__ctalkCreateObject* or *__ctalkCreateObjectInit* functions. To change an existing object's scope, use the *__ctalkSetObjectScope* library function to set an object's scope.

Even though you can set an `OBJECT *`'s `scope` directly, using these functions insures that the object and all of its instance variables maintain the same scope.

When creating an object with a function like *__ctalkCreateObjectInit*, you can declare a scope directly.

```
result = __ctalkCreateObjectInit ("result",
                                  "Symbol", "Object",
                                  LOCAL_VAR, "0x0");
```

When altering the scope of an existing object, however, you should add or subtract only that scope from the object's existing scope.

For example, to add a `VAR_REF_OBJECT` scope to an object:

```
__ctalkSetObjectScope (object, object -> scope | VAR_REF_OBJECT);
```

To remove the scope, use an expression like this.

```
__ctalkSetObjectScope (object, object -> scope & ~VAR_REF_OBJECT);
```

For values of integral classes like `Integer`, `LongInteger` or `Symbol`, `__ctalkCreateObjectInit` tries to convert the *value* parameter to its numeric value. It the function can't figure out a way to convert the argument to its numeric value, it issues a warning message.

```
ctalk: can't convert d to an int.
```

In these cases (and in many others) it is easier to simply use an empty string as the final parameter and then fill in the value after the object is created, as in this example (assuming that the object is an `Integer`)..

```
result = __ctalkCreateObjectInit
  (INTEGER_CLASSNAME, INTEGER_SUPERCLASSNAME, LOCAL_VAR, "");
*(int *)result -> __o_value = int_value;
*(int *)result -> instancevars -> __o_value = int_value;
```

In this case, the INTVAL, LLVAL, and SYMVAL macros can help make the expression more readable, depending on whether the new object is an `Integer`, `LongInteger`, or `Symbol`.

```
INTVAL(result -> __o_value) = int_value;
INTVAL(result -> instancevars -> __o_value) = int_value;
```

4.11 Templates

Templates are simplified methods that are defined as macros and written in C. They provide a method-compatible wrapper to C functions.

Template methods can appear in place of C functions in complex expressions, and they must be used if a C function writes to its arguments (e.g., like *scanf(3)* or *sscanf(3)*). For example, if we have a template defined for the function `myPrintMsg`, then Ctalk subsitutes the method expression wherever `myPrintMsg` appears in an expression like this one.

```
if ((myInt = myPrintMsg (someMsg)) != 0) {
   ... do something...
}
```

After compilation, the expression looks like this.

```
if ((myInt = CFunction cMyPrintMsg (someMsg)) != 0) {
   ...
```

You can also use `myPrintMsg` on the left-hand side of an assignment, like this.

```
myInt = myPrintMsg (someMsg);
```

-or-

```
self = myPrintMsg (someMsg);
```

Additionally, you can use templates on their own, by prefacing the method's name (the template function's alias) with its class object.

```
CFuntion myPrintMsg (someMsg);
```

As the last example implies, templates are class methods in the pseudo-class `CFunction`. Ctalk loads and caches templates where necessary when compiling the input file, so you won't see any methods of the `CFunction` class unless the program calls for one or more of them.

If the template wraps a C function of the same name, then, of course, you can also use the C function on its own. However, templates don't necessarily need to correspond to a C function; they can provide any set of C expressions with a method compatible interface.

Ctalk defines a number of built-in templates for the standard C library. You can determine if Ctalk substitutes a template for a C function by giving the `--printtemplates` option to `ctalk` when it compiles a source file. This displays the templates that the compiler loads and caches (but doesn't necessarily send to the output).

You can define templates for application-specific C functions and routines also. Ctalk retrieves and caches them similarly to its built-in templates, but they are cataloged and stored separately in each user's `~/.ctalk/templates` directory.

As mentioned above, templates don't need to wrap a function of the same name as the template. That is, `myPrintMsg`, above, does not have to be an actual function (although you need to prototype it that way). Templates for C library functions always correspond

to an actual library function. Program specific templates can serve as a wrapper for any function or set of routines.

When compiling templates, the Ctalk compiler checks the user template registry first. If a user defines a template with the same name as one of Ctalk's C library templates, then the compiler uses the user's template instead of Ctalk's built-in template. That means you can define a template that replaces a C library template.

When creating a template for a function, you need to follow these steps.

- Add a prototype of the function to the input file. For example:

```
OBJECT *myPrintMsg (char *text);
```

  Ctalk uses the prototype's argument list to check the number and type of arguments. The prototype's argument list must be the same as the template's argument list.

- Check that any terms in the expression before and after the template are compatible. Ctalk can in most cases distinguish between objects and C variables in expressions. Ctalk will try to warn you if it sees a mixed object/variable expression it can't handle. If an expression causes an error, try breaking it into smaller pieces, with objects and methods in one set of terms, and C variables and operators in another. Also, try making the template function the first term in the expression (or the first term after an assignment operator). This expression, for example, is relatively easy to handle because everything that follows the template is normal C code.

```
myFloat = rand () + 0.33 + (float)my_dbl;
```

- Write a template for the function, and add it to the local directory's template cache with the `template` command. The following sections describe the format that templates use.

Writing Templates

Templates are basically multiple line macros that provide a method selector and method body in a #define preprocessor statement. When Ctalk finds a function name that has a template defined for it, it subsitutes the template name for the function name in the expression, adds the template's body to the output, and adds the template to the CFunction classes' initialization.

Templates can accept arguments similarly to methods, and, like methods, they return a C `OBJECT *`, or NULL.

Here is the template for the *myPrintMsg()* C function. The template provides a wrapper for the *printf(3)* function, and, like *printf(3)*, returns an integer value (as an `Integer` object). The template contains a few features that are part of the template protocol.

```
#define myPrintMsg \n\
cMyPrintMsg (char *text) {\n\
  char buf[255]; \n\
  int result; \n\
```

```
  OBJECT *objresult; \n\
  if (__ctalkIsObject(ARG(0))) {\n\
    result = printf ("myPrintMsg: %s\n", \n\
                     __ctalkToCCharPtr(ARG(0), 1)); \n\
    __ctalkDecimalIntegerToASCII (result, buf); \n\
    return __ctalkCreateObjectInit ("result", "Integer", \n\
                                    "Magnitude", LOCAL_VAR, buf); \n\
  } else { \n\
    __ctalkDecimalIntegerToASCII (-1, buf); \n\
    objresult =  __ctalkCreateObjectInit ("result", "Integer", \n\
                                    "Magnitude", LOCAL_VAR, buf); \n\
    __ctalkRegisterUserObject (objresult); \n\
    return objresult; \n\
  }\n\
}
```

- The template embeds newlines with the string '`\n\`'. When the template is preprocessed, this expression joins the lines of the template but keeps the line endings intact.
- The method selector, '`cMyPrintMsg`' is built from the function name ('`myPrintMsg`') by uppercasing the first letter and prepending a '`c`' to it.
- The template must declare its arguments, like the '`char *text`' declaration above, even though the actual arguments are objects, as described in the next item.
- The expression `__ctalkIsObject(ARG(0))` checks that the argument to the template is a valid object. You can acccess the template's arguments with the `ARG` macro. The first argument is `ARG(0)`, the second argment is `ARG(1)`, and so on.
- The function, *__ctalkToCCharPtr()* translates the argument object into a `char *` string that *printf(3)* expects as its argument.
- Normally templates create a return object manually, using the API function *__ctalkCreateObjectInit ()* or a similar function. Note that *__ctalkCreateObjectInit ()* uses a `char *` to store the value of the object, which is why many templates declare a buffer for the C function's result and use __ctalkDecimalIntegerToASCII (). If the result is more complex, then the template might need to format it with a function like *sprintf (3)*. Templates can also return NULL, but if a program uses such a templated function in an assignment statement, it causes a program to generate a NULL argument object warning when the program is run.
- Whenever a method returns an object it creates, it should register the object, for example by calling *__ctalkRegisterUserObject* as in the template above. That way Ctalk can either maintain the object or clean it up when it needs to be deleted. This function call is optional, but omitting it may cause memory leaks when the program is run.
- A template can use macros, like `LOCAL_VAR`. The most convenient way to define macros is to '`#include <ctalk/ctalkdefs.h>`' somewhere in the input. Many classes already do this, and the template can use the same set of macro definitions as the class libraries.

Cataloging Templates

When looking up templates, Ctalk looks in the template registry file, which is normally `~/.ctalk/templates/fnnames` for the name given in the input. If the function is aliased to another name by a macro substitution, `fnnames` contains the name of the alias also.

For example, on some systems, the function *getc(3)* is a macro that expands to '`_IO_getc`'. The `fnnames` file would then contain '`_IO_getc`' as the templated function's name.

When the compiler finds the function's name or alias in `fnnames`, it looks in the directory for the template file, which is named for the first letter of the function's name. That is, when looking for `myPrintMsg's` template, Ctalk looks for a file named `~/.ctalk/templates/m`.

The C library templates that come with Ctalk use the same scheme, except that the template files are stored in a subdirectory of the class library called `libc`. The registry is part or the run-time library, so C library templates do not need a separate registry file.

The manual page, *fnnames(5ctalk)* contains more information about the `fnnames` file. The *templates(5ctalk)* manual page describes the format of individual templates.

4.12 Return Values

Internally, methods return either an `OBJECT *` or `NULL`. If you write a method that returns a C variable, Ctalk normally translates it into an equivalent object.

Methods that return arrays declared in C generally assign the C array to an equivalent `Array` object. The `Array` object's size is the same as the array declaration, regardless of how it the array's members are initialized.

The `Array` allocation only occurs for arrays declared with a subscript; that is, a variable declared as `int *` is not always stored in an `Array` object, while a variable declared as `int[size]` is.

Ctalk treats C arrays declared as `char[size]` a little differently. If the method's definition says the return class of the method is `Array`, then Ctalk returns an `Array` of `Character` objects; otherwise, it returns a `String` object.

Because Ctalk does not have equivalent classes for multi-dimensioned arrays; that is, arrays with more than one subscript, it does not translate the arrays automatically. In these cases, the method might return a multi-dimensioned array by assigning it to a `Symbol` object.

Occasionally, you might need to return the result of an expression that Ctalk can't translate automatically. In that case, you can use the `eval` keyword to evaluate the expression when the program is run, as in this example.

```
MyClass instanceMethod myMethod (void) {
  ...
  return eval <expression>
}
```

If a method must create an object of its own to return, the object should have the scope CREATED_PARAM, which tells the Ctalk libraries that the the program only needs the object if it's the result of a complete expression; if not, the program cleans up the object automatically when it is no longer needed. See ⟨undefined⟩ [__ctalkRegisterUserObject], page ⟨undefined⟩.

```
return __ctalkCreateObjectInit ("myStringAnswer", "String",
                                 "Character", CREATED_PARAM,
                                 "The contents of the String object.");
```

If the program needs to store a return object for longer than the scope that the object is called in, the method can save the object in its object pool with the library function __ctalkRegisterUserObject, which is described below. See ⟨undefined⟩ [__ctalkRegisterUserObject], page ⟨undefined⟩.

```
String instanceMethod myMethod (void) {

  OBJECT *return_object;

  ... Do stuff. ...

  return_object =  __ctalkCreateObjectInit ("myStringAnswer", "String",
                                 "Character", CREATED_PARAM,
                                 "The contents of the String object.");
  __ctalkRegisterUserObject (return_object);
  return return_object;

}
```

Another case may be when a method needs to retrieve an object reference. In these cases, the method may need to increase the object's reference count. However, such a method can also call `__ctalkRegisterExtraObject` to save the object so its memory isn't lost later. The `__ctalkRegisterExtraObject` function does not, itself, set the object's reference count, and it saves an object (not copies of objects) only once.

4.12.1 Method Return Macros

Alternatively, if a method needs to return an object of a particular class, you can use the following `methodReturn*` statements. These are macros that implement the statements to store and return objects which represent different types or classes, like `int`s as Integer objects, `double`s as Float objects, and so on.

These macros have been superseded in more recent versions of Ctalk, which has the ability to insert the correct return code for any class of object, and many C variables, functions, and expressions. If the compiler doesn't recognize some particular expression, however, these macros may still be useful.

Remember that these return value functions are implemented as macros and contain their own code block, so you can use them in places where normal functions would cause a syntax error.

`methodReturnBoolean (int i)`

> Return a `Boolean` object that Ctalk can evaluate to `TRUE` or `FALSE` depending on the value of its argument. Mostly deprecated; an expression like, `"return TRUE"` is equivalent.

`methodReturnFalse`
: Return a `Boolean` object that evaluates to `FALSE`. Deprecated; using the expression, '`return FALSE;`' has the same effect.

`methodReturnInteger(int i)`
: Return an `Integer` object with the value *i*. Mostly deprecated; an expression like, `"return <int>"` is equivalent.

`methodReturnLongInteger(int l)`
: Return a `LongInteger` object with the value *l*. Mostly deprecated; an expression like, `"return <longlongint>|<longint>"` is equivalent.

`methodReturnNULL`
: Returns the C value `NULL`. Deprecated; simply use the expression, '`return NULL;`' instead.

`methodReturnObject(object)`
: Return *object*. Deprecated; the expression, '`return object`' has the same effect.

`methodReturnObjectName(objectname)`
: Return the object referred to by *objectname*. Also deprecated; like `methodReturObject`, above, the expression, '`return object`' has the same effect.

`methodReturnSelf`
: Returns the method's receiver, `self`. Slightly deprecated; simply using the statement, "`return self`," has the same effect.

`methodReturnString(char *s)`
: Return a `String` object with the value *s*. Mostly deprecated; an expression like, `"return <char *>|<string constant>"` is equivalent.

`methodReturnTrue`
: Return a `Boolean` object that evaluates to `TRUE`. Deprecated; use the expression, '`return TRUE;`' instead.

The macros that return objects use the *__ctalkRegisterUserObject* function to keep track of method's return values, and, if necessary, other objects that the method creates. See ⟨undefined⟩ [__ctalkRegisterUserObject], page ⟨undefined⟩.

4.13 Variable Arguments

C functions like *scanf(3)*, *sscanf(3)*, and *fscanf(3)* have templates that allow you to call them with a variable number of arguments.

If you need to call other C functions that use variable arguments, you must call `__ctalkLibcFnWithMethodVarArgs` with the name of the function, the method that contains the function's template, and the function's return class.

The `readFormat` method (implemented in `String` and `ReadFileStream` classes) can scan a string or input file into the objects that the program gives as arguments. The methods also take care of scalar-to-object translation, memory allocation, and several other tasks.

However, programs can also accomplish the same thing manually.

For example, here is the code of the template method for *fscanf(3)*, `cFscanf` (`CFunction` class), without the preprocessing directives.

```
cFscanf (FILE *s, char *fmt,...) {
  EXPR_PARSER *parser;
  OBJECT *result_object;
  parser = __ctalkGetExprParserAt (__ctalkGetExprParserPtr ());
  result_object =
    __ctalkLibcFnWithMethodVarArgs ((int (*)())fscanf, parser -> e_method, "Integer");
  return result_object;
}
```

At run time, the `e_method` member of an expression parser contains the method and its arguments.

The third argument of `__ctalkLibcFnWithMethodVarArgs` determines the class of `result_object`. For C library functions that use variable arguments, the return class is `Integer`.

The typecast `(int (*)())` in the first argument in front of `fscanf` is not strictly needed because we know that the number and types of arguments to *fscanf(3)* (or *scanf(3)* or any other variable-argument function) might vary from the `__ctalkLibcFnWithMethodVarArgs` prototype, but it tells the compiler not to print a warning message in that case.

If you simply need to print formatted output, the `writeFormat` or `printOn` methods (implemented in `String`, `WriteFileStream` and many other classes classes) perform format-argument translations automatically. Several classes also implement a `readFmt` method, which reads formatted input from a `String` or `ReadFileStream`. See ⟨undefined⟩ [writeFormat–class WriteFileStream], page ⟨undefined⟩, and ⟨undefined⟩ [writeFormat–class String], page ⟨undefined⟩.

4.14 Overloading Methods

You can always implement a method in different classes. For example, you can define a '`+`' method in `Integer`, `LongInteger`, `Character`, and `Float` classes, and Ctalk calls the method defined for that class of the receiver object.

Some operators also have methods implemented in a superclass, like `Magnitude` in this case, which can provide any extra error checking and processing that may be necessary, for example, if you try to use a unary minus ('`-`') operator with a receiver like a `Pen` or `Rectangle` object.

When you overload methods *within* a class, however, Ctalk does some extra checking. The compiler needs to examine the expression to find out how many arguments the statement has, whether the operator is a prefix operator, or whether the method's argument is a block of code or a variable argument list, or whether the method uses a C calling convention.

Ctalk can overload math operators when they are also used as prefix operators. Two examples of these are the unary minus ('`-`') and pointer ('`*`') methods, which have different methods than the operators that perform subtraction and multiplication. Writing methods that are prefix operators is described above. See ⟨undefined⟩ [Prefixes], page ⟨undefined⟩.

4.14.1 Overloading Parameters

Exactly when you should write methods that overload things like parameters and variable arguments is somewhat within the philosophy of programming languages. For example,

a common use of method overloading based on the number of parameters is the, "getter/setter," type of method, which retrieves and sets an object's private data.

In Ctalk, these are much less necessary than in other languages, because Ctalk can address an object's private data with a message that has the same name as an instance or class variable. Since these messages bind more tightly to receiver objects than messages that refer to methods, these types of methods might not work the way you think they would. So be sure that if you write a method of this type, that the program is actually using a method message, and not an instance data message.

Here is an example of overloading parameters. Because the `String` class already has a `concat` method (it overloads the '`+`' operator), we overload the method, `myConcat,` to concatenate one or two strings to the receiver.

As long as the program is relatively simple, it's easy to keep track of which methods already exist in a class. In a bigger application, though, you might want to define a subclass of `String` class for this program.

```
String instanceMethod myConcat (String s1) {

  self = self + s1;

}

String instanceMethod myConcat (String s1, String s2) {

  self = self + s1 + s2;

}

int main () {
  String new s1;
  String new s2;
  String new s3;

  s1 = "Hello, ";
  s2 = "world! ";
  s3 = "Again.";

  s1 myConcat s2;
  printf ("%s\n", s1);

  s1 myConcat s2, s3;
  printf ("%s\n", s1);

}
```

We should mention that the `myConcat` method changes its receiver. So the arguments to the second `myConcat` message simply get added to the receiver again. The output should look something like this.

```
Hello, world!
Hello, world! world! Again.
```

4.15 Variable Method Arguments

Ctalk supports variable arguments lists for methods that follow the `stdarg.h` format for argument lists, where the argument assigned to the last named parameter determines the number and type of the following arguments. (The manual page *stdarg (3)* has more details.)

```
String instanceMethod writeFormat (char *fmt, ...)
```

In addition, Ctalk supports argument lists with *no* named parameters. To implement this, Ctalk interprets an ellipsis as a variable argument list, as usual. It is then up to the method to interpret the arguments as they appear on Ctalk's argument stack.

For this task, the *__ctalkNArgs* library function returns the number of arguments that the run-time libraries place on the stack. The method can then interpret these arguments as necessary.

Here is the slightly abbreviated code for the `List : =` method, which should help illustrate this. See ⟨undefined⟩ [List], page ⟨undefined⟩.

```
List instanceMethod = initEQ (...) {
  int n, i;
  OBJECT *arg;

  self delete;         /* Start with an empty List. */
  n = __ctalkNArgs ();

  for (i = (n - 1); i >= 0; --i) {
    arg = __ctalk_arg_internal (i);
    self push arg;
  }
}
```

4.16 Method Functions

Occasionally an application needs to call a method as a function. One example of this is a `SignalHandler` method that the program installs as the handler for signals from the operating system.

Methods that use a C calling convention need to do several things differently than normal methods.

- The method needs to use a C variable as its argument.

- Because the method can be called with no receivers, Ctalk does not perform any of the normal initialization of local and parameter objects.
- The method body needs to be written almost entirely in C.

In order to make Ctalk interpret a method parameter as a C variable, the method must declare the parameter with the `__c_arg__` attribute.

Note: C functions that are called by the operating system generally need only one argument, and the `__c_arg__` attribute only works for methods with a single parameter.

Additionally, to prevent Ctalk from adding local object initialization code to the method, the method must contain the `noMethodInit` keyword.

Here is an example of a method that is used as a signal handler, and installed by the statements in `main`.

```
SignalHandler instanceMethod myHandler (__c_arg__ int signo) {
  noMethodInit;
  printf ("sigInt handler! Signal %d.\n", signo);
  return NULL;
}

int main () {

  SignalHandler new s;

  s setSigInt;
  s installHandler myHandler;
}
```

The `setSigInt` method (class `SignalHandler`) tells the `SignalHandler` object `s` that it is going to handle `SIGINT` signals. See ⟨undefined⟩ [SignalHandler], page ⟨undefined⟩, for the other signals that the class can handle.

The `__ctalkNewSignalEventInternal` function can generate and queue `SignalEvent` objects, so the signal handler does not need to create objects in order to send signal events to the application. See ⟨undefined⟩ [__ctalkNewSignalEventInternal], page ⟨undefined⟩.

Including C Header Files

Often you will need to use C functions and data types in method functions. However, you need to take care that if you include a C header file, it might not be included later on if a class library requires that file.

Normally Ctalk includes whatever C definitions it needs in the class libraries. However, that can cause the preprocessor to omit those definitions from another source file, should the definitions be needed later.

For example, in the method `handleSignal`, from the `timeclient.c` program, the method needs the definition of the *time(2)* function. If you were to `#include time.h` in the input, as in this example, then `time.h`'s definitions would not be included in the `CTime` class later on.

```
#include <time.h>    /* Could cause errors later. */
```

```
SignalHandler instanceMethod handleSignal (__c_arg__ int signo) {
  time_t t;
  char buf[MAXLABEL];
  noMethodInit;
  t = time (NULL);
  __ctalkDecimalIntegerToASCII (t, buf);
  __ctalkNewSignalEventInternal (signo, getpid (), buf);
  return NULL;
}
```

The best way to avoid omitting dependencies is to include only the definitions that the method needs. In this case, you can include the prototype of *time(2)* in the source file.

```
extern time_t time (time_t *__timer) __THROW;  /* From time.h. */

SignalHandler instanceMethod handleSignal (__c_arg__ int signo) {
  time_t t;
  char buf[MAXLABEL];
  noMethodInit;
  t = time (NULL);
  __ctalkDecimalIntegerToASCII (t, buf);
  __ctalkNewSignalEventInternal (signo, getpid (), buf);
  return NULL;
}
```

The `time_t` type is defined with `ctalklib`, and is available to all programs.

How to resolve multiple library definitions depends on the system's header files, and may vary between different operating systems or compiler versions.

4.17 Exception and Error Handling

There are two ways to handle error conditions in Ctalk. You can simply print an error or warning message in your code. An error message formats the text and data that you provide, the same as in a `printf` statement, and then exits the program. Here is an example.

```
_error ("Program exited with code %d.\n", result_code);
```

A `_warning` message is similar, but it prints the message and continues processing. See ⟨undefined⟩ [errorfuncs], page ⟨undefined⟩.

The other way to handle errors is with exceptions. This is the method you need to use if an error occurs within a method, and the program needs either to print a warning message, or exit.

There are two methods of class `Exception` that handle exceptions in application programs: `pending` and `handle`. There are also other API functions, but they are mostly used internally to translate exceptions into events that application programs can use.

These two methods are generally used together. The method `pending`, if it returns `TRUE`, signals that there is an exception pending. Then the function `handle` handles the event by executing an exception handler.

Generally, events simply issue error messages. It is up to you to determine how the program should handle the exception: by exiting, trying the procedure again, ignoring the condition, or some other procedure. Here is an example.

```
Exception new e;
...
inputStream openOn fileArg;
if (e pending) {
   e handle;
  exit (1);
}
if (inputStream isDir) {
  printf ("Input is a directory.\n");
  exit (1);
}
```

This is simply the way that Ctalk notifies the application if the method `openOn` (class `ReadFileStream`) encountered an error while opening the file named by its argument, `fileArg`.

You should note that the program also checks whether the input is actually a directory, because opening a directory as if it were a file does not necessary cause an error condition. The `isDir` (class `FileStream`) method is one portable way to check if the input path is actually a directory.

The method `openOn`, like other methods, raises an exception if necessary. It does this with `raiseException` (class `SystemErrnoException`).

Ctalk handles most `stdio` error codes in this manner. A program that uses the `ReadFileStream` and `WriteFileStream` classes should rarely need to use the C library's `errno` macro, but it is still available if applications need to check for other errors from the C libraries.

The Ctalk library also provides exceptions for missing arguments, undefined methods, and other error conditions. The file, `include/except.h`, contains the definitions of Ctalk's compile and run time exceptions.

4.18 Cautions when Using Arguments with C Library Calls

The interface for C functions is in development. You can use a C function in a simple expression with any argument, as in the following example.

```
op1 = __ctalkGetInstanceVariableByName ("self", "value", TRUE);
```

However, if you want to use a C function in a complex expression, then you must take care that the arguments to the function are C values that correspond to a Ctalk class, unless the function has a template written for it in the class library that performs the translations of specific classes and data types. See ⟨undefined⟩ [Templates], page ⟨undefined⟩.

If you use a function in a method, and the compiler generates an, "implicit declaration," warning, you can include the library function's prototype in either the source file or in `classes/ctalklib`.

4.19 Cautions when using `Array` class elements in C expressions.

If you want to use an `Array` element in a C expression, you need to take care that the value of the element translates to a `void *`, which is the C type that Ctalk returns the values of these elements as.

That means elements of class `LongInteger` might be truncated, and `Float` class array elements cannot be translated in this manner.

If array elements of these classes occur in C expressions, Ctalk prints a warning at run time.

In these cases, it is necessary to convert the values to a compatible pointer type, for example an object of class `String`.

4.20 Method Keywords

`__c_arg__`
: Treat the next method argument as a C argument. See ⟨undefined⟩ [Method functions], page ⟨undefined⟩.

`classMethod`
: The `classMethod` keyword declares a method, as described above.

`classVariable`
: Adds a class variable definition to a class. This method needs to be used globally, when a class is declared. The syntax is:

  parent_class classVariable *name* [*native_class*|*typecast_expr*] [*initial_value*] [*docstring*] ;

  For example:

```
FileStream class ReadFileStream;

...

ReadFileStream classVariable stdinStream Symbol 0x0;
```

  The value of *initial_value* can be either a constant or an expression that evaluates to a constant.

  Ctalk can also translate a typecast into a native class for the variable. Also refer to the entry for `instanceVariable`, below. See ⟨undefined⟩ [InstanceVariableKeyword], page ⟨undefined⟩.

  Similarly, the *docString* element is also optional. See ⟨undefined⟩ [VariableDocStrings], page ⟨undefined⟩.

`eval`
: Do not try to evaluate the following statement until run time. Methods can use this keyword if they need to wait until run time to determine an receiver's class and are not able to alias the object or otherwise inform the front end of the receiver's class before the program is run.

`instanceMethod`
: The `instanceMethod` keyword declares a method, as described above.

`instanceVariable`
: Adds an instance variable definition to a class. This method needs be used when the class is declared. The syntax is:

parent_class instanceVariable *name* [*native_class*|*typecast_expr*] [*initial_value*] [*docString*] ;

For example:

```
FileStream class ReadFileStream;
ReadFileStream instanceVariable pos LongInteger 0L;
```

The value of *initial_value* can be either a constant or an expression that evaluates to a constant.

You can also use a typecast in place of the variable's *native_class*. Ctalk can translate most builtin C types or typedefs to a class, but for less common data types, Ctalk will translate a pointer to the type as a Symbol object.

Similarly, the *docString* element is also optional. See ⟨undefined⟩ [VariableDocStrings], page ⟨undefined⟩.

Note that the `instanceVariable` method does not *create* any variables. Ctalk only creates instance variables for each object when it receives a constructor message (e.g., `new`) by a program.

`noMethodInit`
: Do not include method initialization code for the method. See ⟨undefined⟩ [Method functions], page ⟨undefined⟩.

`require`
: Require a class to be loaded before any other classes or methods.

`returnObjectClass`
: Set the return class of a method to the argument if it is different than the receiver class. See ⟨undefined⟩ [Return values], page ⟨undefined⟩.

`self`
: Return the method's receiver object. In version 0.0.66, you can also use `self` in arguments as a synonym for the receiver of the statement, as in this example.

```
String new path;
path = "/home/user/joe";

printf ("%s", path subString 1, self length - 1);
```

The use of `self` in method arguments is experimental in version 0.0.66, and it should be used with caution.

`super`
: The keyword `super` has two different meanings. It can modify a method, as in the following example.

```
MyReceiverClass super new instanceObject;
```

`super` can also represent the receiver's superclass object, so it can appear as a receiver, as in this example.

```
return super new checksum;
```

4.21 Documenting Methods, Classes, and Variables

Ctalk allows you to add publicly visible comments to methods, classes, and instance and class variables. These comments are different than comments within the code that may have meaning only for specific routines

If you want to add documentation for a class, Ctalk allows documentation string is class declarations (see below). See ⟨undefined⟩ [ClassDocStrings], page ⟨undefined⟩.

You can also document instance and class variables. See ⟨undefined⟩ [VariableDocStrings], page ⟨undefined⟩.

When documenting methods, Ctalk recognizes both C-style comments and C++ comments.

In addition, Ctalk recognizes a character string at the beginning of a method or function as a public documentation string, and it adds a few rules for comments and documentation strings that make it easier to describe methods when browsing the class library.

Basically, if the method contains a comment or documentation string at the start of a method or function body, then that style of comment is used as the method's public documentation when it is referenced by other programs.

```
MyClass instanceMethod myMethod (Object myArg) {
  /* Using a C-style comment at the start of a method body,
     or a series of C-style comments, makes those comments
     available as the method's public documentation. */
  /* The public documentation can span several comments if the
     comments appear before any lines of source code. */

     Method body...


  /* That allows you to add (perhaps temporary) comments elsewhere
     in the method that do not appear as part of the method's
     public documentation. */

     More lines of code...

}

MyClass instanceMethod myMethod (Object myArg) {
  // A series of C++ comments before the first line of
  // code also can appear as the method's public
  // documentation.

     Method body...

  /* A different style of comment anywhere else within
     the method does not appear in the method's public
     documentation. */
}

MyClass instanceMethod myMethod (Object myArg) {
  "A character string at the start of the method also gets
   interpreted as a public documentation string."
```

```
    Method body...
}
```

The Ctalk libraries contain several methods that can be useful when printing documentation. Particularly, the method `methods` (in `Object` class) and `methodSource` (in `Application` class) can retrieve the names of the methods in a class and their source code, and the method `tokenize` (in `String` class) can split the source code into tokens, which you can then process.

Here's a simple application that retrieves a method's source code and splits it into tokens.

```
int main () {

  Application new app;
  String new methodStr;
  List new tokens;

  /* The first argument to methodSource is the class name, and
     the second argument is the method name. */
  methodStr = app methodSource "Application", "methodSource";

  methodStr tokenize tokens;

  tokens map {
    printf ("%s ", self);  /* This example simply prints the method's
                              tokens, but you can perform any processing
                              or formatting that you want here. */
  }
  printf ("\n");
}
```

If you want only the prototype of the method; that is, the declaration and the argument list, feed the output of `methodSource` to `methodPrototypes`, which is also in class `Application`. The `methodPrototypes` method takes a string with the source code of a method or methods as input, which means you can also extract all of the prototypes of a class library.

```
int main () {

  Application new app;
  String new src;
  String new prototypes;

  src = app methodSource "Object", "basicNew";

  prototypes = app methodPrototypes src;

  printf ("%s\n", prototypes);
```

```
}
```

There are a few caveats:

The `methods` method is designed to be quick, so it only finds methods whose declaration appears on one line. If you prefer method declarations spread over several lines, you can read the entire class file using `readAll` (class `ReadFileStream`). See ⟨undefined⟩ [ReadFileStream], page ⟨undefined⟩. Then you can tokenize the entire class file at once, which disregards any line formatting, although tokenizing an entire file takes considerably longer.

Also, The `methodPrototypes` method does not do any special formatting; it simply collects the prototypes into one `String` object.

4.21.1 Class Documentation

Ctalk also allows you add documentation to class declarations. The declaration syntax allows you to add an option character string between the class name and the closing semicolon. The syntax of a class documentation is the following.

```
superclassname class classname <docstring>;
```

For example, here is the class declaration of `WriteFileStream` which contains a documentation string.

```
FileStream class WriteFileStream   "Defines the methods and instance
variables that write data to files. Also defines the class variables
stdoutStream and stderrStream, which are the object representation
of the standard output and standard error streams.";
```

The `classdoc` program can print the documentation string of a class if it provides one. The *classdoc(1)* manual page provides more information.

4.21.2 Instance and Class Variable Documentation

You can add an optional documentation string to an instance or class variable's declaration by enclosing the text within quotes immediately before the final semicolon.

```
WriteFileStream classVariable stdoutStream
"Defines an object that contains the value of the system's standard output";
```

The main thing to watch out for is, syntactically, a documentation string could be mistaken for a variable's initial value if one isn't included in the definition. For example, this definition uses a character string as its initial value.

```
ObjectInspector instanceVariable promptString String "> ";
```

So in this case, if you wanted to add a documentation string, you would also need to include an initial value, otherwise the documentation string would be mistaken for the variable's value.

```
                                                        /* Incorrect! */
    ObjectInspector instanceVariable promptString String
    "The string that is displayed as the object inspector's
    prompt";
```

Instead, you need to add both an an initial value, and the documentation string.

```
                                                          /* Correct. */
    ObjectInspector instanceVariable promptString String "> "
    "The string that is displayed as the object inspector's
    prompt";
```

4.22 Ctalk Library Reference

This section describes some of the Ctalk library functions that you can use in methods and, in many cases, in Ctalk programs generally.

The file `classes/ctalklib` contains the prototypes of the library functions.

`__argvName (char *s)`
: Set the name of the program at run time. Normally this is the same as `argv[0]`.

`__argvFileName (void)`
: Returns the name of the executable program.

`__arg_trace (int stack_index)`
: Prints the object at argument stack index *stack_index*.

`__ctalkAddClassVariable (OBJECT *class_object, char *name, OBJECT *variable_object)`
: Add a class variable *variable_object*to *class_object*.

`__ctalkAddInstanceVariable (OBJECT *object, char *name, OBJECT *variable_object)`
: Add an instance variable to an object. Note that the function adds a copy of *variable_object* to *object*.

`__ctalkAliasObject (OBJECT *rcvr, OBJECT * target)`
: Set the *rcvr* object's label to the *target* object, so the *target* object can be referred to by the *rcvr* object's identifier. This function does not require *rcvr* to be the actual receiver of the method, so the results can be unpredictable if it is used in a context other than where `self` is the first argument. To insure that *rcvr* is the actual receiver of the calling method, use *__ctalkAliasReceiver ()*, below.

`__ctalkAliasReceiver (OBJECT *rcvr, OBJECT * target)`

Like *__ctalkAliasObject ()*, above, but the function checks that *rcvr* is the actual receiver of the calling method and returns `ERROR` (‘`-1`’) if it isn't. Here is an example.

```
String instanceMethod = setEqual (OBJECT *__stringArg) {
  // String assignment method. Assigns the argument  to the
  // receiver label.  Also does some String-specific
  // semantic stuff for different sorts of String objects.
  // Returns the new String object.

  __ctalkStringifyName (self, __stringArg);
  if (__ctalkAliasReceiver (self, __stringArg) != 0) {
    __ctalkAddInstanceVariable (self, "value", __stringArg);
    return self;
  } else {
    return __stringArg;
  }
}
```

`__ctalkANSIClearPaneLine (OBJECT *paneObject, int lineNumber)`

Clear (to spaces) the pane line at *lineNumber*.

`__ctalkANSITerminalPaneMapWindow (Object *childPane)`

Map *childPane* onto the receiver pane. The child pane's upper left-hand corner origin is relative to the receiver pane's origin. The receiver pane should be large enough to completely enclose the child pane. The child pane is displayed at the next `refresh` message.

`__ctalkANSITerminalPaneUnmapWindow (Object *childPane)`

Removes the pane given as the argument from the receiver pane.

`__ctalkANSITerminalPanePutChar (int x, int y, char c)`

Store character *c* at coordinates *x*,*y* in the pane's content region. The character will be displayed after the next `refresh` message.

`__ctalkANSITerminalPaneRefresh (void)`

Display the contents of `ANSITerminalPane` objects on the display, including text and window decorations if any.

`__ctalkANSITerminalPaneUnMapWindow (Object childPane)`

Unmap *childPane* from the receiver pane's visible area. The child pane is not deleted; it is simply not displayed at the next `refresh` message.

`__ctalkARB (void)`

Returns a boolean value of true if the GLEW libraries support the GLEW_ARB_vertex_shader and GLEW_ARB_fragment_shader extensions. Programs must call the *__ctalkInitGLEW* function before using this function.

`__ctalkArgBlkReturnVal (void)`
: Called by the calling function of an argument block to retrieve the block's return value, if any.

`__ctalkArgBlkSetCallerReturn (void)`
: Called by a map method to indicate to a calling method or function that an argument block has requested a return from the function or method that called it. Map-type methods for general use should include a call to this function, which provides argument block support for `return` statements. Refer to *__ctalkRegisterArgBlkReturn*, below, and the `String : map` method for an example of these functions' use.

`__ctalkArrayElementToCCharPtr (OBJECT *array_element)`
`__ctalkArrayElementToCChar (OBJECT *array_element)`
`__ctalkArrayElementToCDouble (OBJECT *array_element)`
`__ctalkArrayElementToCInt (OBJECT *array_element)`
`__ctalkArrayElementToCLongLongInt (OBJECT *array_element)`
`__ctalkArrayElementToCPtr (OBJECT *array_element)`
: Translates the object *array_element*'s value to a C `char *`, `char`, `double`, `int`, `long long int`, or `void *`.

`__ctalkBackgroundMethodObjectMessage (OBJECT *rcvr, OBJECT *method_instance)`
: Perform a method call by sending *rcvr* the message defined by *method_instance*, which is a previously defined `Method` object. See ⟨undefined⟩ [Method], page ⟨undefined⟩.

  The function starts *method_instance* as a separate process, which runs concurrently with the process that launched it. The background process exits when *method_instance* returns.

  The *method instance* argument is a normal method. However, *__ctalkBackgroundMethodObjectMessage* does not save the return object before *method_instance* exits, and *method_instance*, does not take any arguments.

  The function returns the PID of the child process, or '`-1`' on error.

  This function is used by the method `backgroundMethodObjectMessage` (class `Object`). Refer to the description of the method for more information. See ⟨undefined⟩ [Object], page ⟨undefined⟩.

  For examples of method instance calls, See ⟨undefined⟩ [methodObjectMessage], page ⟨undefined⟩.

`__ctalkBackgroundMethodObjectMessage2Args (OBJECT *rcvr, OBJECT *method_instance, OBJECT *arg1, OBJECT *arg2)`
: This function combines a background method instance call with two arguments. Its function is similar to *__ctalkMethodObjectMessage*, below.

  For examples of method instance calls, See ⟨undefined⟩ [methodObjectMessage], page ⟨undefined⟩.

`__ctalkCallMethodFn (METHOD *method)`
: Used internally to perform a method call.

`__ctalkCallingFnObjectBecome (OBJECT *old, OBJECT *new)`
: Used by `Object : become` to translate the receiver when `become` is called within a function.

`__ctalkCallingInstanceVarBecome (OBJECT *old, OBJECT *new)`
: Used by `Object : become` to translate the receiver when the receiver is an instance variable.

`__ctalkCallingMethodObjectBecome (OBJECT *old, OBJECT *new)`
: Used by `Object : become` to translate the receiver when `become's` receiver is an object declared in another method.

`__ctalkCallingReceiverBecome (OBJECT *old, OBJECT *new)`
: Used by `Object : become` to translate the receiver when `become's` receiver also the receiver of the method that calls `become`.

`__ctalkCBoolToObj (bool b)`
: Create a `Boolean` object with the boolean (either true or false) value of the argument.

`__ctalkCCharPtrToObj (char *s)`
: Create a String object from a C `char *`.

`__ctalkConsoleReadLine (OBJECT *string_object, char *prompt_string)`
: Prints the prompt *prompt_string* to standard output of a terminal, then reads a line of text from the standard input, until it encounters a newline, and saves it as the value of *string_object*.

  If Ctalk is built with support for the GNU readline libraries, the function provides the readline libraries' command line editing and history facilities. Otherwise, the function reads input up to a newline using only the basic text input and editing facilities provided by the stdio functions.

`__ctalkCreateArg (OBJECT *receiver, char *methodname, char *arg_expr)`
: Create an argument for the following `__ctalk_method` or `__ctalk_primitive_method` function call. Unlike `__ctalk_arg`, this function always creates a new object. Its primary use is to create local method objects that are fully instantiated into a class by a following `new` method.

`__ctalkCreateArgA (OBJECT *receiver, char *methodname, char *arg_expr)`
: Like *__ctalkCreateArg*, creates local method objects that are instantiated into a class by a following `new` method. The *__ctalkCreateArgA ()* function is more specialized so it can be used when performing method cache fixups.

`__ctalkCDoubleToObj (doubled)`
: Create a Float object from a C `float` or `double`.

`__ctalkCharRadixToChar (char *s)`
: Return the character as a C `char` that is represented by the formatted argument.

  If *s* contains a number of more than one digit, then it is converted from an integer to the ASCII code of a character.

`__ctalkCharRadixToCharASCII (char *s)`
: Return a C string with a lexically correct character - a character enclosed in single quotes - from the formatted argument. If *s* is already a character, then no conversion is done.

  If *s* contains a decimal number of more than one digit, then it is converted from a decimal integer to a character.

`__ctalkCFUNCReturnClass (CFUNC *fn, char *buf)`
: Return in *buf* the name of the class that corresponds to *fn*'s return type.

`__ctalkCIntToObj (int i)`
: Create an `Integer` object from a C `int`.

`__ctalkCLongLongToObj (long long int l)`
: Create a `LongInteger` object from a C `long long int`.

`__ctalkClassMethodInitReturnClass (char *rcvr_class, char *method_name, char *return_class);`
: Set the return class of *method* in *rcvr_class* to *return_class*.

`__ctalkClassMethodParam (char *rcvrclassname, char *methodname, OBJECT *(*selector_fn)(), char *paramclass, char *paramname, int param_is_pointer)`
: Define a method parameter when initializing a class method. Normally the compiler generates this call for inclusion in `__ctalk_init ()` for the method initialization at run time.

`__ctalkClassObject (OBJECT *object)`
: Returns the class object of the argument.

`__ctalkClassVariableObject (OBJECT *var)`
: Return the object that `var` is a class variable of, or NULL.

`__ctalkFree (void *p)`
: This is an API wrapper for Ctalk's memory free routines. Much of the old code in the class libraries still uses __xfree () (which now gets macroized to __ctalkFree () anyway), but you should use __ctalkFree () in new class libraries.

  In cases where you prefer to call __xfree () directly, then you need to use the MEMADDR macro to cast the argument to a `void **`, i.e.,

```
char *my_buf;

... do stuff ...

__xfree (MEMADDR(my_buf));
```

  which is what __ctalkFree () does automagically.

`__ctalkLocalTime (long int utctime, int *seconds_return, int *minutes_return, int *hours_return, int *dom_return, int *mon_return, int *years_return, int *dow_return, int *doy_return, int *have_dst_return)`

Returns the system's local time in the arguments that return the current second, minute, hour, day of the month, month, year, day of the week, day of the year, and (T/F) whether the time uses daylight savings time.

The first argument is the system's UTC time, as returned by the *time ()* library function.

`__ctalkCloseGLXPane (OBJECT *pane_object)`

Releases the pane's GLX context and the context's XVisualInfo struct, and deletes the *pane_object's* X11 window.

`__ctalkCloseX11Pane (OBJECT *pane_object)`

Deletes and closes subpanes of a main X11 window.

`__ctalkCopyPaneStreams (OBJECT *src, OBJECT *dest)`

Copy the *src* `inputHandle` and `outputHandle` values to *dest*.

`__ctalkCopyCVariable (CVAR *c)`

Return a `CVAR *` to a copy of the argument.

`__ctalkCopyObject (OBJREF_T src, OBJREF_T dest)`

Return an `OBJECT *` to a copy of the argument.

Note: When copying normal objects, you can translate them to `OBJREF_T` types with the `OBJREF` macro. When copying method arguments, it is necessary to alias the argument to an object. See `at` (class `Array`) and `atPut` (class `AssociativeArray`) for examples.

`__ctalkCreateArgEntry (void)`

Used internally to create an ARG type, which maintains a method's argument entries.

`__ctalkCreateArgEntryInit (OBJECT *obj)`

Creates an internal method argument entry and initializes the entry to the argument object.

`__ctalkDeleteArgEntry (ARG *arg)`

Delete a method ARG entry.

`__ctalkCreateObject (char *name, char *class, char *superclass, int scope)`

Create an object without instance variables with name *name* of class *class* and superclass *superclass* with scope *scope*.

`__ctalkCreateObjectInit (char *name, char *class, char *superclass, int scope, char *value)`

Create an object with name *name* of class *class* and superclass *superclass* with scope *scope* and value *value*.

For more information about how to use *__ctalkCreateObjectInit*, refer to the examples in See ⟨undefined⟩ [Scoping], page ⟨undefined⟩, and other examples in this manual.

`__ctalkCreateWinBuffer (int x_size, int y_size, int cell_size)`
: Create a buffer for a `Pane` object's window.

`__ctalkCreateGLXMainWindow (OBJECT * pane_object)`
: Normally called by the `initialize` method in class `GLXCanvasPane`, creates a X11 window with a GLX visual. The visual's default attributes are: '`GLX_RGBA`', '`GLX_DEPTH_SIZE`', 24 bits per pixel, and '`GLX_DOUBLEBUFFER`'. These attributes are adjustable via the instance variables in `GLXCanvasPane` class.

  The *self_object* argument must be an instance of `GLXCanvasPane` class. This function fills in the object's `visualInfoPtr` instance variable with a pointer to the `XVisualInfo` structure specified by *self_object*. See ⟨undefined⟩ [GLXCanvasPane], page ⟨undefined⟩.

  Called by the `initialize` methods (class `X11Pane`) to create a X window. Returns the window system's ID for the window, an `int`.

  If a program doesn't specify a window size, then default size is 250x250 pixels. If the program doesn't specify a location for the window, then this function sets the window's *X, Y* origin to 0, although the actual window placement depends on the machine's window manager. See ⟨undefined⟩ [ctalkX11SetSizeHints], page ⟨undefined⟩.

  Application programs can provide a window geometry specification which is used to set the window's size and placement See ⟨undefined⟩ [parseX11Geometry], page ⟨undefined⟩.

  If *x11pane_object* contains a `background` or `backgroundColor` instance variable or resource, the function sets the window's background color to the X11 color named in that value.

  If a program doesn't specify a foreground and background color, the window's background is set to white and the foreground is set to black.

  Refer also to the *__ctalkCreateX11MainWindowTitle* and *__ctalkCreanteX11SubWindow* functions. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

`__ctalkCreateX11MainWindowTitle (OBJECT * x11pane_object, char *title)`
: Similar to *__ctalkCreateX11MainWindow*, except that if *title* is non-null, the function uses it to set the new window's title. Returns the X resource ID of the new window, an `int`. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

`__ctalkCreateX11SubWindow (OBJECT * parent_pane_object, OBJECT * subpane_object)`
: Creates an X subwindow that has the parent window defined by *parent_pane_object's* `xWindowID` instance variable. The dimensions of the subwindow within the parent window are determined by the *subpane_object's* `origin` and `size` instance variables. Creates a new graphics context for the window and saves its address in the *subpane_object's* `xGC` instance variable.

  If *x11pane_object* contains a `background` or `backgroundColor` instance variable or resource, the function sets the window's background color to the X11 color named in that value.

  If a program doesn't specify a foreground and background color, the window's background is set to white and the foreground is set to black.

Then the function initiazes and clears the window to the value of the `backgroundColor` instance variable. Otherwise, the function sets the window's color to black and clears the window.

Also sets the display depth in the `depth` instance variable, and the ID of the subwindow in the `xWindowID` instance variable.

`__ctalkCriticalExceptionInternal (MESSAGE *orig, EXCEPTION ex, char *text)`
Raise a critical exception at run time. The argument *orig* should be `NULL`. This function should be called from a method. This function saves a snapshot of the *calling method's* run-time context.

For an example, see `raiseCriticalException` (class `SysErrnoException`) in the Ctalk class library, which calls `__ctalkCriticalSysErrExceptionInternal`, below.

`__ctalkCriticalSysErrExceptionInternal (MESSAGE *orig, int errno, char *text)`
A convenience function that calls `__ctalkCriticalExceptionInternal` with the Ctalk exception that corresponds to *errno*, the C library's error macro.

`__ctalkCVARReturnClass (CVAR *var, char *buf)`
Return in *buf* the name of the class that corresponds to *var*'s data type.

__ctalkDecimalIntegerToChar C function

`__ctalkDecimalIntegerToChar (int n, char *buf)`
Formats the ASCII 8-bit character representation of *n* as the first character in *buf*, and returns '`(char)n`' as its return value. If *n* is greater than 255 (0xff hex), returns the formatted character '`(char)(n & 255)`'.

`__ctalkDecimalIntegerToASCII (int i, char *s);`
Format an ASCII representation of *i*, a decimal integer, in *s*.

`__ctalkLongLongToDecimalASCII (long long int l, char *buf);`
`__ctalkLongLongToHexASCII (long long int l, char *buf, bool uppercase);`
`__ctalkLongToDecimalASCII (long int l, char *buf);`
`__ctalkLongToHexASCII (long int l, char *buf, bool uppercase);`
Format a decimal or hexadecimal ASCII representation of *l*, a long long int, in *buf*, and return *buf*.

When using *__ctalkLongLongToHexASCII,* or *__ctalkLongToHexASCII,* if *uppercase* is `true`, format the number using uppercase letters; e.g., '`0XFFFF`'; otherwise format the number using lowercase letters: '`0xffff`'.

`__ctalkDefaultSignalHandler (int signo)`
Set the handler of signal *signo* to the system's default handler.

`__ctalkDefineClassMethod (char *class, char *name, OBJECT *(*)(fn), int required_args);`
Define a class method in class *class* with name *name*, which calls the function *fn*, and requires *required_args* arguments.

`__ctalkDefineClassVariable (char *class, char *name, char *varclass, char *init_expr);`
: Define a class variable in class *class* with *name*. The variable is an object of class *varclass* with the initial value *init_expr*.

`__ctalkDefineInstanceMethod (char *class, char *name, OBJECT *(*)(fn), int required_args);`
: Define an instance method in class *class* with name *name*, which calls the function *fn*, and requires *required_args* arguments.

`__ctalkDefineInstanceVariable (char *class, char *name, char *varclass, char *init_expr);`
: Define an instance variable in class *class* with *name*. The variable is an object of class *varclass* with the initial value *init_expr*.

`__ctalkDefinedClassMethodObject (OBJECT *rcvr, char *classname, char *method_name)`
: Initialize a `Method` object from the class method *method_name* from class *classname*. Note that this function only works for methods that are already defined. See ⟨undefined⟩ [Method], page ⟨undefined⟩.

`__ctalkDefinedInstanceMethodObject (OBJECT *rcvr, char *classname, char *method_name)`
: Initialize a `Method` object from the instance method *method_name* from class *classname*. Note that this function only works for methods that are already defined. See ⟨undefined⟩ [Method], page ⟨undefined⟩.

`__ctalkDefineTemplateMethod (char *classname, char *name, OBJECT *(*cfunc)(), int required_args, int n_args)`
: Defines a template method. First calls __ctalkDefineClassMethod (), then performs additional initialization needed for templates.

`__ctalkDeleteLastExceptionInternal (void)`
: Deletes the most recent exception from Ctalk's internal exception list.

`__ctalkDeleteObject (OBJECT *object)`
: Delete an object, and any objects it refers to, if they are not referenced elsewhere. It's a good idea to call `__objRefCntZero` first. The object then won't be showing any extra references.

  Zeroing the reference count first causes the function to delete the object *completely*. It bypasses Ctalk's internal mechanisms that decide when objects are no longer needed. Don't use these functions unless you know where the object and any objects it refers to are declared.

  It should seldom be necessary to remove an object from a particular scope manually anyway. But in that case, call `__objRefCntDec` instead, and then let Ctalk try to clean up the object when it goes out of scope.

  For example, to delete a scratch object using C,

```
Object new tmpObject;
OBJECT *tmpObject_alias;
```

```
tmpObject_alias = tmpObject;

__objRefCntZero (OBJREF(tmpObject_alias));
__ctalkDeleteObject(tmpObject_alias);
```

Translating the object to C should work in all cases, regardless of the surrounding code. Of course, you can also use the `delete` method (class `Object`) with many simple C expressions.

```
Object tmpObject;
OBJECT *tmpObject_alias;

tmpObject delete;

..or..

tmpObject_alias delete;

..or even this expression..

tmpObject_alias -> instancevars delete;
```

Deletes *obj* if there are no references to it, or if this is the last reference to the object. In this case, the function works similarly to *__ctalkDeleteObject*. Otherwise, the function decreases the object's reference count by 1.

`__ctalkDeleteObjectList (OBJECT *object)`
: Delete an object and any objects that link to it.

`__ctalkDeleteWinBuffer (OBJECT *paneBuffer_instance_var)`
: Deletes a `paneBuffer` backing store allocated when a subclass of `Pane` creates an object.

`__ctalkDocDir (void)`
: Returns a `char *` that contains the path where the Ctalk-specific documentation is installed on the system (i.e., documentation other than man pages and Texinfo manuals).

`__ctalkDoubleToASCII (double d, char *s);`
`__ctalkFloatToASCII (float f, char *s);`
`__ctalkLongDoubleToASCII (long double f, char *s);`
: Format an ASCII representation of the first argument in *s*. Returns *s*.

__ctalkEnterArgBlockScope C function

`__ctalkEnterArgBlockScope (void)`
: Ctalk inserts this call at the beginning of argument blocks. Checks that the block call is the result of an inline method call and sets the run-time stack's

RTINFO structure to indicate a block call with the correct stack frame for the block's scope.

__ctalkEntryIconXPM C function

`__ctalkEntryIconXPM (int iconID)`

Returns a `char **` with the contents of an eye or slashed eye icon used when X11TextEntryPane objects echo dots. The argument, *iconID*, may be one of the following.

```
#define ENTRY_ICON_EYE_NON       0
#define ENTRY_ICON_EYE_OK        1
#define ENTRY_ICON_EYE_SLASHED   2
```

The definitions are included in `x11defs.h` and `ctalkdefs.h`, which should be included in classes or programs with a statement like this.

```
#include <ctalk/ctalkdefs.h>
```

__ctalkExec C function

`__ctalkExec(char *cmdline, OBJECT *strObject)`

Executes the command line given as the argument and waits for the subprocess to finish. If the *strObject* argument is non-NULL, the function saves the subprocesses' standard output as the object's `value`. Normally, *strObject* should be a `String` object.

If the *strObject* argument is NULL, *__ctalkExec* reads and prints the subprogram's standard output to the terminal's standard output.

__ctalkExec supports the redirection operators '`>`' or '`>>`', which send the subprocess's standard output to the file given as the redirection operator's target. If *cmdLine* is a shell script, it is executed by a sub-shell using the *system(3)* library function.

__ctalkExitArgBlockScope C function

`__ctalkExitArgBlockScope (void)`

Ctalk inserts this function, which is currenly a no-op, as the last function call in an argument block.

__ctalkExpandPath C function

`__ctalkEpandPath (char *dirGlobPattern, char *expandedPathOut)`

Expands *dirGlobPattern* into the full path name, and stores it in *expanedPathOut*. Returns the value of *expandedPathOut*.

`__ctalkErrorExit (void)`

Cleans up the program's immediate object environment before a call to the C *exit(3)* function.

`__ctalkEvalExpr (char *expr)`

Evaluate *expr* and return an `OBJECT *` to the result.

`__ctalkEvalExprU (char *expr)`

Like *__ctalkEvalExpr*, above, but *__ctalkEvalExprU* returns a C NULL instead of a null result object when an expression evaluates to 0.

`__ctalkExceptionInternal (MESSAGE *msg, EXCEPTION e, char *text)`

Generate an exception of type *e*. Exceptions are listed in `except.h`, and in the description of `Exception` class. See ⟨undefined⟩ [Exception], page ⟨undefined⟩. If *msg* is non-NULL, which is usually the case when the compiler generates and exception, the function records the line and column number of the exception. If *text* is non-NULL, the exception handler adds the text to the exception record, so it can be displayed when the program handles the exception.

Programs should handle run-time exceptions as soon as possible after returning from a method. The function `__ctalkHandleRunTimeExceptionInternal` is the normal exception handler, and `__ctalkTrapExceptionInternal` determines whether an exception has occurred.

`__ctalkFindClassVariable (char *varname, int warn)`

Search the class library for *first occurrence* of the class variable *varname*.

If *warn* is `TRUE`, issues a warning if the variable is not found.

__ctalkFindMethodByName C function

`__ctalkFindMethodByName (OBJECT **object, const char *name, int warn)`

Returns the first instance or class method *name* if it exists in *object*'s class or superclasses, or NULL if a method with that name isn't found. If *warn* is true, prints a warning message if the method isn't found.

`__ctalkFindPrefixMethodByName (OBJECT **rcvr, const char *name, int warn)`

Looks up a classes' instance method that matches *name* which has the `prefix` attribute set. If *warn* is true, prints a warning if the method is not found.

`__ctalkExceptionNotifyInternal (I_EXCEPTION *i)`

The handler function of most exceptions. Prints a message including any text provided as an argument to `__ctalkExceptionInternal`.

`__ctalkFilePtrFromStr (char *sformatted_ptr)`

Like *__ctalkGenericPtrFromStr ()*, below, converts a string containing a formatted hexadecimal number to a pointer, and also checks that the pointer is a valid file.

The function returns a `void *` in order to keep Ctalk's header inclusion simple, or NULL if the argument does not contain a valid hexadecimal number, or if the pointer does not point to a valid file.

This function can also set `errno` in case of an error.

Because the function returns a `void *` an app or method must perform the cast from `void *` to `File *` when it calls this function.

`__ctalkFindClassMethodByFn (OBJECT **rcvr_p, OBJECT *(*fn)(), int warn)`

Finds a class method by calling `__ctalkGetClassMethodByFn` for the method's receiver and then the receivers of previous method calls. On success, returns the method, and *rcvr_p* contains the address of the method's receiver.

If *warn* is `TRUE`, issues a warning if the method is not found.

`__ctalkFindClassMethodByName (OBJECT **rcvr_p, char *name, int warn)`
: Finds a class method by by calling `__ctalkGetClassMethodByName` for the method's receiver and then the receivers of previous method calls. On success, returns the method, and *rcvr_p* contains the address of the method's receiver.

  If *warn* is `TRUE`, issues a warning if the method is not found.

`__ctalkFindInstanceMethodByFn (OBJECT **rcvr_p, char *name, int warn)`
: Finds an instance method by calling `__ctalkGetInstanceMethodByFn` for the method's receiver and then the receivers of previous method calls. On success, returns the method, and *rcvr_p* contains the address of the method's receiver.

  If *warn* is `TRUE`, issues a warning if the method is not found.

`__ctalkFindInstanceMethodByName (OBJECT **rcvr_p, char *name, int warn)`
: Finds an instance method by by calling `__ctalkGetInstanceMethodByName` for the method's receiver and then the receivers of previous method calls. On success, returns the method, and *rcvr_p* contains the address of the method's receiver.

  If *warn* is `TRUE`, issues a warning if the method is not found.

`__ctalkSaveCVARArrayResource (char *name, int initializer_size, void *var)`
: Saves an `Array` object with the contents of `var` in a method's object pool.

`__ctalkSaveCVARResource (char *name)`
: Saves the contents of C variable *name* in a method's object pool.

`__ctalkSaveOBJECTMemberResource (OBJECT *object)`
: Saves an `OBJECT *` member to a method's resource pool.

`__ctalkSleep (int usecs)`
: Put a program to sleep for *usecs* microseconds. The operating system restores the program to a running state no sooner than that amount of time has elapsed.

`__ctalkSort (OBJECT *collection, bool sortdescending)`
`__ctalkSortByName (OBJECT *collection, bool sortdescending)`
: Sorts a collection's members into ascending or descending order. *__ctalkSort* performs the sort using the collection members' values, while *__ctalkSortByName* uses the members' names.

  The algorithm used is very simple minded, although due to the mechanics of finding earlier/later collection members, it is as fast as divide-and-conquer algorithms for small or medium sized collections. For large collections it is probably more practical to sort the collection as an `OBJECT **` array and create a new list based on the collection members' sorted order.

  However, it is almost always faster to add members to collections in the order you want them sorted rather than trying to re-arrange the collection later. For this, refer to the methods in the `SortedList` class See ⟨undefined⟩ [SortedList], page ⟨undefined⟩.

`__ctalkStrToPtr (char *ptr)`
: Converts a formatted hexadecimal number to a `void *`. The function *__ctalkGenericPtrFromStr*, below, performs some extra validation.

`__ctalkGenericPtrFromStr (char *s)`
: A wrapper function for *__ctalkStrToPtr ()* that performs some extra validation of the string argument. These functions convert a string containing a formatted hexadecimal number (e.g., `"0xnnnnnn"` or "`"0XNNNNNN"` into a `void *`. Both of the functions return a `void *`, or NULL if the string does not contain a valid hexadecimal number.

`__ctalkGetCallingFnObject (char *name, char *classname)`
: When used within `Object : become`, returns the object with the name *name* in the class *classname* from `become's` calling function. Also adjusts its caller indexes if `become` is called within an argument block.

`__ctalkGetCallingMethodObject (char *name, char *classname)`
: When used within `Object : become`, returns the object with the name *name* in the class *classname* from `become's` calling method. Also adjusts its caller indexes if `become` is called within an argument block.

`__ctalkGetCArg (OBJECT *obj)`
: Retrieves the CVAR of the C variable named by *obj*.

`__ctalkGetInstanceMethodByFn (OBJECT *class_object, OBJECT *(*fn)(void), int warn)`
: Returns the method that defines function *fn* from *class_object*, or NULL if the method doesn't exist. If *warn* is true, prints a warning message if the method isn't found.

`__ctalkGetInstanceMethodByName (OBJECT *class_object, const char *name, int warn)`
: Returns the method named *name* from *class_object*, or NULL if the class doesn't define a method with that name. If *warn* is true, prints a warning message if the method isn't found.

`__ctalkGetReceiverPtr (void)`
: Returns an `int` with the current value of the receiver stack pointer.

`__ctalkGetRS (void)`
: Returns a `char` with the current record separator. The record separator determines, among other uses, how regular expression characters act at line endings. See ⟨undefined⟩ [RecordSeparator], page ⟨undefined⟩.

`__ctalkGetRunTimeException (void)`
: Remove the first exception from the exception queue and return the exception's message as a char *.

`__ctalkGetTemplateCallerCVAR (char * name)`
: If called from within a function template, looks up the `CVAR` *name* in the calling function or method. This function returns a temporary object with the name and class, superclass, and value that correspond to the `CVAR`'s data type. The return object persists until the next time this function is called.

`__ctalkGetClass (char * classname)`
: Get the class object for *classname*.

`__ctalkGetClassMethodByFn (OBJECT *rcvr, OBJECT *(*fn)(void), int warn)`
: Return a class method of *rcvr's* class with the run-time function *fn*.

  If *warn* is `TRUE`, issue a warning if the method is not found.

`__ctalkGetClassMethodByName (OBJECT *rcvr, char *name, int warn)`
: Return a class method of *rcvr's* class with the name *name*.

  If *warn* is `TRUE`, issue a warning if the method is not found.

`__ctalkGetExprParserAt (int idx)`
: Return the expression parser, which is a struct '`EXPR_PARSER`' typedef, at stack index *idx*.

`__ctalkGetExprParserPtr (void)`
: Return the expression parser pointer, and `int`.

`__ctalkGetClassVariable (OBJECT *receiver, char *varname, int)`
: *warn*) Return the class variable named *varname* from the receiver's class object, or `NULL` if the variable does not exist. If *warn* is `TRUE`, issue a warning message if the variable is not found.

`__ctalkGetExceptionTrace (void)`
: Return `TRUE` if a program has enabled exception walkbacks, `FALSE` otherwise.

`__ctalkGetInstanceVariable (OBJECT *receiver, char *varname, int)`
: *warn*) Return the instance variable named *varname* from the receiver, or `NULL` if the variable does not exist. If *warn* is `TRUE`, issue a warning message if the variable is not found.

`__ctalkGetInstanceVariableByName (char *receiver_name, char *varname, int)`
: *warn*) Return the instance variable named *varname* from the object named by *receiver_name*, or `NULL` if the variable does not exist. If *warn* is `TRUE`, issue a warning message if the variable is not found.

`__ctalkGetPrefixMethodByName (OBJECT *class_object, const char *name, int warn)`
: Returns the method named *name* from *class_object* that has the `prefix` attribute set, or NULL if the class doesn't define a prefix method with that name. If *warn* is true, prints a warning message if the method isn't found.

`__ctalkGetTypeDef (char * name)`
: Return the `CVAR` of the typedef *name*.

`__ctalkGetX11KeySym (int keycode, int shift_state, int keypress)`
: Returns an `int` with the keyboard mapping of a keypress in X applications. This allows programs to distinguish between modifier keypresses (e.g., shift, control, and alt), and alphanumeric keypresses.

  The first and second parameters are taken from an XKeyPressEvent or XKeyReleaseEvent structure. The third parameter, *keypress*, should be true for Keypress events and false for keyrelease events.

  For alphanumeric keys, this function does not automatically modify the ASCII code of a key that is pressed when the shift key (or any other modifier key) is pressed. That is, pressing '`A`' and '`a`' both return the ASCII value 97. It is up

to the program to record whether the shift key is pressed at the same time, and to provide the shifted character itself if necessary.

Refer to the `run` method in `GLXCanvasPane` class for an example.

This function uses *XGetKeyboardMapping(3)* internally.

`__ctalkGLEW20 (void)`

Returns a boolean value of true if the GLEW library supports version 2.0 extensions, mainly for OpenGL programs that use shaders. Programs must call the *__ctalkInitGLEW* function before using this function.

`__ctalkGlobalObjectBecome (OBJECT *old, OBJECT *new)`

Called when the receiver of `Object : become` is a global object.

`__ctalkGlobFiles (char *pattern, OBJECT *list)`

If the system's C libraries support file globbing with the *glob* library function, *__ctalkGlobFiles* returns the file and directory pathnames that match *pattern* in the `List` object given as the *list* argument.

For information about how the C library matches file patterns, refer to the *glob(3)* and related manual pages.

```
__ctalkGLUTVersion (void)
__ctalkGLUTCreateMainWindow (char *title)
__ctalkGLUTInitWindowGeometry (int x, int y, int width, int height)
__ctalkGLUTInit (int argc, char **argv)
__ctalkGLUTRun (void)
__ctalkGLUTInstallDisplayFn (void (*fn)())
__ctalkGLUTInstallReshapeFn (void (*fn)(int, int))
__ctalkGLUTInstallIdleFn (void (*fn)())
__ctalkGLUTInstallOverlayDisplayFunc (void (*fn)())
__ctalkGLUTInstallKeyboardFunc (void (*fn)(unsigned char, int, int))
__ctalkGLUTInstallMouseFunc (void (*fn)(int, int, int, int))
__ctalkGLUTInstallMotionFunc (void (*fn)(int, int))
__ctalkGLUTInstallPassiveMotionFunc (void (*fn)(int, int))
__ctalkGLUTInstallVisibilityFunc (void (*fn)(int))
__ctalkGLUTInstallEntryFunc (void (*fn)(int))
__ctalkGLUTInstallSpecialFunc (void (*fn)(int, int, int))
__ctalkGLUTInstallSpaceballMotionFunc (void (*fn)(int, int, int))
__ctalkGLUTInstallSpaceballRotateFunc (void (*fn)(int, int, int))
__ctalkGLUTInstallSpaceballButtonFunc (void (*fn)(int, int))
__ctalkGLUTInstallButtonBoxFunc (void (*fn)(int, int))
__ctalkGLUTInstallDialsFunc (void (*fn)(int, int))
__ctalkGLUTInstallTabletMotionFunc (void (*fn)(int, int, int, int))
__ctalkGLUTInstallMenuStatusFunc (void (*fn)(int, int, int))
__ctalkGLUTInstallMenuStateFunc (void (*fn)(int))
__ctalkGLUTInstallMenuStateFunc (void (*fn)(int))
__ctalkGLUTSphere (double, int, int, int);
__ctalkGLUTCube (double, int);
__ctalkGLUTCone (double, double, int, int, int);
__ctalkGLUTTorus (double, double, int, int, int);
__ctalkGLUTDodecahedron (int);
__ctalkGLUTOctahedron (int);
__ctalkGLUTTetrahedron (int);
__ctalkGLUTIcosahedron (int);
__ctalkGLUTTeapot (double, int);
__ctalkGLUTFullScreen (void);
__ctalkGLUTPosition (int, int);
__ctalkGLUTReshape (int, int);
__ctalkGLUTWindowID (char *window_name)
```

The functions that make up Ctalk's glue layer for the GLUT API. For their use, refer to the methods in GLUTApplication class.

```
__ctalkGLXAlpha (float alpha)
```

Sets the alpha (opacity) channel for outline text rendering. Values should be between 0.0 (transparent) and 1.0 (opaque). The Ctalk library's default value is 1.0 (opaque).

`__ctalkGLXDrawText (char *text)`
: This is another convenience function that draws text on a `GLXCanvasPane` using a X font that the program registered with the pane's GLX context via a previous call to *__ctalkGLXUseXFont*.

  The `GLXCanvasPane` class defines several methods that facilitate drawing with X fonts when using GLX. See ⟨undefined⟩ [GLXCanvasPane], page ⟨undefined⟩.

`__ctalkGLXDrawTextFT (char *text, float x, float y)`
: Draws *text* at the matrix coordinates given by the *x*,*y* arguments. Programs should call at least *__ctalkGLXUseFTFont* before calling this function.

`__ctalkGLXExtensionsString (void)`
: Returns a `char *` containing the extensions supported glX.

`__ctalkGLXExtensionSupported (char *extName)`
: Returns a boolean value of True if the system's glX extension supports *extName*, False otherwise.

`__ctalkGLXFrameRate (void)`
: Returns a `float` with the frames per second of the calling program. The function averages the rate over each interval of approximately five seconds.

`__ctalkGLXFreeFTFont (void)`
: Frees the font and library data from a previous call to *__ctalkGLXUseFTFont*.

`__ctalkGLXFreeXFont (void)`
: Frees X font data that was allocated by a previous call to *__ctalkGLXUseXFont*. The `GLXCanvasPane` class defines several methods that facilitate drawing with X fonts when using GLX. See ⟨undefined⟩ [GLXCanvasPane], page ⟨undefined⟩.

`__ctalkGLXFullScreen (OBJECT *selfObject, char *winTitle)`
: Toggles the window's full screen mode on and off.

`__ctalkGLXNamedColorFT (char *colorname)`
: Sets the foreground color for drawing text with Freetype fonts to the named X11 color given as the argument.

`__ctalkGLXPixelHeightFT (int pxheight)`
: Sets the height of the current Freetype face in use to the pixel height given as the argument.

`__ctalkGLXRefreshRate (void)`
: Returns a `float` with the display's refresh rate if the OpenGL installation supports the `GLX_OML_sync_control` extension. If OpenGL doesn't support `GLX_OML_sync_control`, the function prints a warning message on the terminal and returns -1.

`__ctalkGLXSwapBuffers (OBJECT *glxpane_object)`
: This is an API-level wrapper for the GLXSwapBuffers library function.

`__ctalkGLXSwapControl (int interval)`
: Sets the swap buffer synchronization to 1/interval. If interval is 0, disables buffer swap synchronization. If the machine's OpenGL does not support the

`GLX_MESA_swap_control` extension, the function is a no-op. Returns 0 on success, -1 if the extension is not supported.

`__ctalkGLXTextWidth (char *text)`
: Returns an `int` with the width in pixels of *text* rendered in the current font. The program must first have selected a X font using *__ctalkGLXUseFon*. If no font is selected, the function returns '`-1`'.

`__ctalkGLXUseFTFont (String fontfilename)`
: Initializes the Freetype library and loads the font from the file given as the argument. Use *__ctalkGLXFreeFTFont* to release the font data before calling this function again when changing fonts.

`__ctalkGLXUseXFont (OBJECT *glxCanvasPaneObject, char *fontname)`
: This is a convenience function that registers the X font, *fontname* for use with *glxCanvasPaneObject*, first by retrieving the X font data for *fontname*, then registering the font using *glXUseXFont(3)*.

  After the program has finished drawing with the font, the program should call *__ctalkGLXFreeXFont*.

  The `GLXCanvasPane` class defines several methods that facilitate drawing with X fonts when using GLX. See ⟨undefined⟩ [GLXCanvasPane], page ⟨undefined⟩.

`__ctalkGLXFullScreen (void)`
: Returns a boolean value of true if the window is using Freetype fonts, false otherwise.

`__ctalkGLXWindowPos2i (int x, int y)`
: This is a wrapper for the *glWindowPos2i* function, which several methods in `GLXCanvasPane` class use.

  Because *glWindowPos2i* is an extension in many GL implementations, Ctalk checks for the function's presence when compiling the libraries.

  If the GL implementation does not provide *glWindowPos2i*, then any Ctalk program that tries to use this function (or one of the methods that call it), prints an error message and exits.

`__ctalkGLXWinXOrg (void)`
`__ctalkGLXWinYOrg (void)`
`__ctalkGLXWinXSize (void)`
`__ctalkGLXWinYSize (void)`
: These functions return an int with the window's current origin and size.

`__ctalkGUIPaneDrawCircleBasic (void *display, int window_id, int gc, int center_x, int center_y, int radius, int fill, int pen_width, int alpha char *fg_color_name, char *bg_color_name)`
: Draws a circle centered at *center_x*,*center_y* with radius *radius*. The dimensions are given in pixels. If *filled* is true, then the function draws a filled circle; otherwise, the circle's edge has the width *pen_width*.

  This function is a synonym for *__ctalkX11PaneDrawCircleBasic*.

`__ctalkGUIPaneClearRectangle (OBJECT *pane_object, int x, int y, int width, int height)`

Clear a rectangular region in a GUI Pane object. Also clear the region in any buffers associated with the object.

`__ctalkGUIPaneClearWindow (OBJECT *pane_object)`

Clear a pane object's window.

`__ctalkGUIPaneDrawLine (OBJECT *pane_object, OBJECT *line_object, OBJECT *pen_object)`

Draw a line specified by *line_object* (an instance of `Line` class) using *pen_object* (an instance of `Pen` class).

`__ctalkGUIPaneDrawLineBasic (void *display, int drawable_id, int gc_ptr, int x_start, int y_start, int x_end, int y_end, int pen_width, int alpha, char *pen_color)`
`__ctalkX11PaneDrawLineBasic (void *display, int drawable_id, int gc_ptr, int x_start, int y_start, int x_end, int y_end, int pen_width, int alpha, char *pen_color)`

Draw a line between the points (x_start,y_start) and (x_end, y_end) with the color, and transparency using the drawable ID, graphics context, and pen color, width, and transparency given as arguments.

This function is a synonym for *__ctalkX11PaneDrawPointBasic*.

`__ctalkGUIPaneDrawPoint (OBJECT *pane_object, OBJECT *point_object, OBJECT *pen_object)`

Draw a point on *pane_object* specified by *point_object* using *pen_object*.

`__ctalkGUIPaneDrawRectangle (OBJECT *pane_object, OBJECT *rectangle_object, OBJECT *pen_object, int fill)`

Draw a rectangle on *pane_object* specified by *rectangle_object* using *pen_object*. If *fill* is non-zero, draw a filled rectangle.

`__ctalkGUIPaneDrawRoundedRectangle (OBJECT *pane_object, OBJECT *rectangle_object, OBJECT *pen_object, int fill, int radius)`

This is similar to *__ctalkGUIPaneDrawRectangle*, except that it takes an extra argument, *radius*, which specifies the radius of the arcs that are used to draw the corners.

`__ctalkGUIPanePutStr (OBJECT *pane_object, int x, int y, char *string)`

Display `String` object *string* at coordinates *x*,*y* on *pane_object*. You can select the font with the `font` method in class `X11Pane`. This function relies on instance variables defined in `X11Pane` class. The `__ctalkX11PanePutStrBasic` function, below, provides a more flexible interface to the X libraries.

`__ctalkGUIPaneRefresh (OBJECT *pane_object, int srcX, int srcY, int srcWidth, int srcHeight, int destX, int destY)`

Refresh the Pane object by updating the visible window with *pane_object*'s buffers and, if necessary, notifying the GUI library that the window has been updated.

`__ctalkGUISetBackground (OBJECT *pane_object, char *color)`
: Set the background of a pane's window to *color*. This function is intended only for objects that have an actual window; e.g., `X11Pane` objects. For all other visual types, like pixmaps, use __ctalkX11SetBackgroundBasic.

`__ctalkX11SubWindowGeometry (OBJECT * parentpane, char * geomstr, int *x_out, int *y_out, int *width_out, int *height_out)`
: Parses a string that contains the geometry specification of a subwindow, and returns the width and height of the subwindow and its X,Y position within the parent pane's window.

  A geometry specification has the form:

  ```
  width[%]xheight[%][+x_org[%]+y_org[%]]
  ```

  The *x, y, width, and height* parameters are interpreted as the actual origin and size of the subwindow, unless a parameter is followed by a percent sign ('`%`'). In that case, the dimension is interpreted as a fraction of the parent window's corresponding vertical or horizontal dimension.

  For some `Pane` classes, like dialog windows, if *x_org* and *y_org* are missing, then the class positions the dialog window centered over the parent window.

`__ctalkX11TextFromData (void *display, int drawable_id, int GD_ptr, char *text)`
: Displays *text* on *drawable_id*.

`__ctalkX11TextWidth (char *fontDesc, char *text)`
: Returns an `int` with the width in screen pixels of the *text* argument when rendered in the font named by *fontDesc*. There is more information about how to use fonts in the sections that discuss the X graphics classes. See ⟨undefined⟩ [X11Font], page ⟨undefined⟩.

`__ctalkX11WxHGeometry (int parentWidth, int parenthHeight, char *geomspec, int xOut, int yOut, int widthOut, int heightOut)`
: Calculate the dimensions specified by *geomspec* within *parentWidth* and *parentHeight*, and return the results in *xOut*, *yOut*, *widthOut*, and *heightOut*. For information about the format of *geomspec*, refer to the *Window Geometry* subsection of the `X11PaneDispatcher` class. See ⟨undefined⟩ [X11PaneDispatcher], page ⟨undefined⟩.

`__ctalkHandleRunTimeException (void)`
: Execute the exception handler for a pending exception. The `handle` method (class `Exception`) calls this function. See ⟨undefined⟩ [Exception], page ⟨undefined⟩.

`__ctalkHandleRunTimeExceptionInternal (void)`
: Execute the exception handler for a pending exception. If the exception is generated by an expression, execute the exception handler only if further expressions or subexpressions need to be evaluated.

`__ctalkHaveFTFaceBasic (void)`

Returns TRUE if an application has created a new FreeType2 font face, FALSE otherwise. This is a lower level library function that apps should not need to use directly, and may go away in the future.

`__ctalkHexIntegerToASCII (unsigned int` *ptr*`, char *`*s*`)`

Format a hexadecimal representation of *ptr* in *s*. The return value is the formatted string in *s*.

On 64-bit machines, the prototype is:

```
char *__ctalkHexIntegerToASCII (unsigned long long int ptr, char *buf)
```

`__ctalkIconXPM (int` *iconID*`)`

Returns a `char **` with the XPM data for the dialog icon given by *iconID*. The library defines the following icon IDs.

```
ICON_NONE
ICON_STOP
ICON_CAUTION
ICON_INFO
```

`__ctalkIgnoreSignal (int` *signo*`)`

Set the handler for *signo* to ignore the signal.

`__ctalkIncKeyRef (OBJECT *`*object*`, int` *inc*`, int` *op*`)`
`__ctalkIncStringRef (OBJECT *`*object*`, int` *idx*`, int` *op*`)`

Increment the reference to the value of *object*, a `String` or `Key` object, or one of its subclasses, by *idx*. If *idx* is negative, decrements the reference to the value of the receiver. If the reference is before or after the start or end of the receiver's value, further uses of the object return NULL.

The argument *op* can be one of the following constants, which are defined in `ctalkdefs.h`.

`TAG_REF_PREFIX`

Increments (or decrements) the value of the receiver immediately. Normally this is used for prefix `++` and `--` operators, and also `+=` and `-=` operators.

`TAG_REF_POSTFIX`

Increments (or decrements) the value of the receiver after its value is accessed. Used normally for postfix `++` and `--` operators.

`TAG_REF_TEMP`

Adds a temporary reference that is cleared after the receiver is next read. Normally you would use this for expressions that assign the reference to another object, as in this example.

```
String new str1;
```

```
String new str2;

str1 = "Hello, world!";

str2 = str1 + 3;
```

The object `str2` is assigned the calculated reference. The value of `str1` is unaffected.

`__ctalkInitGLEW (void)`
: Initialize the GLEW library. Programs must call this function before performing any operations that use OpenGL extensions.

`__ctalkInlineMethod (OBJECT *rcvr, METHOD *method, int n_args, ...)`
: Call a method or block of code that is an argument to another method. The class of *rcvr* and the class of *method* do not need to be the same. Currently, only the `map` method uses inline method calls. For an example of the `__ctalkInlineMethod ()`'s use, see `map` (implemented by the `List`, `Array`, and `AssociativeArray` classes). This function can (and should) be used to implement inline method messages or code blocks when streaming over collections.

  The *n_args* argument specifies the number of arguments to be passed to the target method. Currently `__ctalkInlineMethod ()` supports 0 - 6 arguments.

`__ctalkIntRadixToDecimalASCII (char *intbuf)`
: Return a C string with the integer formatted in *intbuf* formatted as a decimal (base 10) integer.

`__ctalkInstallHandler (int signo, OBJECT *(*method_c_function)())`
: Set the handler of signal *signo* to *method_c_function*. The prototype of *method_c_function* is similar to the intermediate C prototype of Ctalk's methods. Signal handlers installed with this function reset the handler to the default after each use, except for handlers on DJGPP platforms.

`__ctalkInstallPrefix (void)`
: Returns a `char *` with the top-level directory where Ctalk is installed. Ctalk's installation uses this directory as the top-level directory of its installation layout; for example, in relative terms, this is where Ctalk's various components get installed:

```
Executables:          prefixdir/bin
Libraries:            prefixdir/lib
Class Libraries:      prefixdir/include/ctalk
Texinfo Manuals:      prefixdirshare/info
Manual Pages:         prefixdir/share/man
Searchable Docs:      prefixdir/share/ctalk
```

`__ctalkIntanceMethodInitReturnClass (char *rcvrclassname, char *methodname, char *returnclassname)`

Set the return class of method *methodname* of class *rcvrclassname* to *returnclassname*.

`__ctalkInstanceVarsFromClassObject (OBJECT *obj)`

Add the instance variables defined by *obj's* class object.

`__ctalkInstanceVarIsCallersReceiver (void)`

Used by `Object: become`. Returns True if the receiver object is an instance variable, False otherwise.

`__ctalkInitFTLib (void)`

Initialize the system's FreeType2 library. Returns 0 if successful, ERROR ('`-1`') if unsuccessful or if the library isn't available. This is a lower level function that should not normally be needed by apps directly, and may go away in the future.

`__ctalkInstanceMethodParam (char *rcvrclassname, char *methodname, OBJECT *(*selector_fn)(), char *paramclass, char *paramname, int param_is_pointer)`

Define a method parameter when initializing a method. Normally the compiler generates this call for inclusion in `__ctalk_init ()` for the method initialization at run time.

`__ctalkIntFromCharConstant (char *str)`

Returns the `int` value of the character constant *str*. Recognizes all of the escape sequences that Ctalk uses, whether the constant is enclosed in single quotes or not. Also recognizes backslash escape sequences and the following control character constants.

Escape Sequence	Int Value
\0	0
\a	1
\b	2
\e	27
\f	6
\n	10
\r	13
\t	9
\v	11

The '`\e`' escape sequence is an extension to the C language standard.

`__ctalkIsClassVariableOf (char *class, char *varname)`

Returns `TRUE` if *varname* is a class variable of *class*, `FALSE` otherwise.

`__ctalkIsCallersReceiver (void)`

Used by `Object : become` to determine if an object is the calling method's receiver.

`__ctalkIsDir (char *path)`

Returns `TRUE` if *path* is a directory, `FALSE` otherwise.

`__ctalkIsInstanceMethod (OBJECT *self_object, char *method_name)`
`__ctalkIsClassMethod (OBJECT *self_object, char *method_name)`
: The functions return True if the method given by *method_name* is an instance or class method, respectively, in *self_object's* class.

`__ctalkIsInstanceVariableOf (char *class, char *varname)`
: Returns `TRUE` if *varname* is an instance variable of *class*, `FALSE` otherwise.

`__ctalkIsObject (OBJECT *o)`
: Return `TRUE` if if *o* is a valid object, `FALSE` otherwise.

`__ctalkIsSubClassOf (char *classname, char *superclassname)`
: Return `TRUE` if *classname* is a subclass of *superclassname*, `FALSE` otherwise.

`__ctalkLastMatchLength (void)`
: Return the length of the match from the last call to *__ctalkMatchText*, below.

`__ctalkLibcFnWithMethodVarArgs (int (*libcfn)(), METHOD *method, char *libcfn_return_class)`
: Call the C library function *libcfn* using with its template method *method*. For C library functions that use `stdarg.h` variable arguments, *libcfn_return_class* should be `Integer`.

  When evaluating an expression, the currently executing method is contained in the current `EXPR_PARSER`. See ⟨undefined⟩ [__ctalkGetExprParserAt], page ⟨undefined⟩.

  Note: This version of Ctalk only supports variable-argument functions on 32-bit Intel platforms. If you try to use a variable-argument function on another hardware platform, Ctalk issues a warning and returns `NULL`.

`__ctalkLogMessage (char *, ...)`
: Formats the message given as the argument and writes the message to the system's syslog facility.

`__ctalkMatchAt (Integer n)`
`__ctalkMatchIndexAt (Integer n)`
: Returns, respectively, the text, or the character index matched by the *n'th* parenthesized subexpression during a previous call to *__ctalkMatchText* (i.e., a backreference). The argument, *N*, is '`0`' for the first parenthesized subexpression, '`1`' for the next subexpression, and so on. If the *n*'th pattern didn't match any text, returns NULL. See ⟨undefined⟩ [Pattern Matching], page ⟨undefined⟩.

`__ctalkMatchText (char *pattern, char *text, long long int *offsets)`
: Find the occurences of *pattern* in *text*. Returns the index of each match in the *offsets* array, with the list terminated by -1. Returns the number of matches, or -1 if there are no matches.

`__ctalkMatchPrintToks (bool printToks)`
: If *printToks* is `true`, then Ctalk prints the regular expression tokens and the matching text for every regular expression match, which can be useful for debugging regular expressions.

`__ctalkMapGLXWindow (OBJECT *glxcanvaspane_object)`

Maps a `GLXCanvasPane's` window to the display and creates a `GLXContext` for the window, and makes the GLXContext current.

Saves the GLXContext pointer in the receiver's `glxContextPtr` instance variable. See ⟨undefined⟩ [GLXCanvasPane], page ⟨undefined⟩.

`__ctalkMapX11Window (OBJECT *x11pane_object)`

The X library interface of the `map` (class `X11Pane`) method See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩. This function is a wrapper for the *XMapWindow* and *XMapSubwindows* Xlib functions.

`__ctalkMethodObjectMessage (OBJECT *rcvr, OBJECT *method_instance)`

Perform a method call by sending *rcvr* the message defined by *method_instance*, which is a previously defined `Method` object. See ⟨undefined⟩ [Method], page ⟨undefined⟩.

The function returns '`0`' on success, '`-1`' on error.

For examples of `Method` object calls, See ⟨undefined⟩ [methodObjectMessage], page ⟨undefined⟩.

`__ctalkMethodObjectMessage (OBJECT *rcvr, OBJECT *method_instance, OBJECT *arg1, OBJECT *arg2)`

Perform a method call by sending *rcvr* the message defined by *method_instance*, which is a previously defined `Method` object.

The parameters *arg1* and *arg2* are the arguments to the method instance. `Method` objects with two arguments are commonly used in graphical event dispatchers, particularly in `X11PaneDispatcher` class. This helps simplify the event dispatcher methods.

The function returns '`0`' on success, '`-1`' on error.

For examples of `Method` object calls, See ⟨undefined⟩ [methodObjectMessage], page ⟨undefined⟩.

`__ctalkMethodPoolMax (void)`
`__ctalkSetMethodPoolMax (int new_size)`

Get or set a program's method pool size, in the number of objects that each method's pool retains. The default pool size is set when the Ctalk libraries are built, and is displayed in the configure program's status report during the build process. When a method's pool size reaches this number of objects, the pool deletes the oldest object in the pool to make room for the new object.

Generally, the default pool size is suitable for the language tools and demonstration programs that come packaged with Ctalk. Some of the test programs in the `test/expect` subdirectory that run through many iterations (i.e., thousands of iterations) require a larger pool size. This is especially true if a program uses many C variables when iterating through its operations, and whether the C variables are simply scalar or constant values (e.g., ints, doubles, and literal strings), and whether the variables are pointers to objects in memory.

`__ctalkMethodReturnClass (char *classname)`

Set the return class of an instance or class method during method initialization.

`__ctalkNArgs (void)`
: Returns an `int` with the number of arguments passed to the current method.

`__ctalkNMatches (void)`
: Returns an `int` with the number of matches from the last call to __ctalkMatchText.

`__ctalkNewFTFace (void)`
: Initialize a new FreeType2 face object. This is a lower level library function that apps should not need to use directly, and may go away in the future.

`__ctalkNewSignalEventInternal (int signo, int pid, char *data)`
: Generate and queue a `SignalEvent` object for signal *signo* with process ID *pid*. The *data* argument is a `String` object that the program can use to pass information back to the application.

`__ctalkNonLocalArgBlkReturn (void)`
: Returns a `bool` value of true or false to an argument block's parent method to indicate whether the argument block executed a `return` statement.

`__ctalkObjValPtr (OBJECT *o, void *ptr)`
: Set the value of the object *o* to *ptr*.

`__ctalkPaneResource (OBJECT *paneObject, char *resourceName, bool warn)`
: Returns an `OBJECT *` with the value corresponding to *resourceName* from *paneObject's* `resource` instance variable. If *warn* is true, displays a warning if the resource isn't found.

`__ctalkPeekExceptionTrace (void)`
: Returns a `char *` with the text of the most recent exception and its stack trace.

`__ctalkPeekRunTimeException (void)`
: Returns a `char *` with the text of the most recent exception.

`__ctalkPendingException (void)`
: A convenience method for `__ctalkTrapException`. Returns `TRUE` if an exception is pending, `FALSE` otherwise.

`__ctalkPrintExceptionTrace (void)`
: Print a walkback of the current exception's copy of the program call stack.

`__ctalkPrintObject (OBJECT *object)`
: Print the object given by the argument, and its instance variables, to standard output.

`__ctalkPrintObjectByName (OBJECT *object_name)`
: Print the object named by *object_name* to the standard output.

`__ctalkProcessWait (int child_processid, int *child_return_value_out, int *child_term_sig_out, int *errno_out)`
: Checks the status of the child process specified by *child_processid*.

  If the return value of *__ctalkProcessWait* is 0, then there is no change in the child processes' status to report. A return value equal to *child_processid* indicates that the child process has exited. If the return value is -1, then there was an

error either in the process that called *__ctalkProcessWait*, the child process, or both.

When *__ctalkProcessWait's* return value is equal to *child_processid*, the function returns the child processes' return value in *child_return_value_out*. If the child process was terminated by an uncaught signal, the signal number is returned in *child_term_sig_out*.

If the function's return value is -1, then function returns the system's error code in *errno_out*.

`__ctalkRaiseX11Window (OBJECT *x11pane_object)`
The X library interface of the `raise` (class `X11Pane`) method.

`__ctalkReceiverReceiverBecome (OBJECT *object)`
Used by `become` (class `Object`) to change the calling method's receiver to the object given as the argument.

`__ctalkReferenceObject (OBJECT *obj, OBJECT *reffed_obj)`
Sets *obj's* value to *reffed_obj's* hexadecimal address. Also incrememts *reffed_obj's* reference count by 1 and adds VAR_REF_OBJECT to its scope.

`__ctalkRegisterArgBlkReturn (int return_code, OBJECT *return_object)`
This function gets called when Ctalk encounters a return statement in an argument block. The first argument is the return code of the argument block itself (typically an `Integer` object with a value of -2, which signals the `map` method that the argument block has requested a return from the parent function or method), and the second argument is the object that is to be returned by the caller.

The following example should hopefully explain how these functions work together. The comments indicate where the compiler inserted these functions.

```
int main () {
  String new str;

  str = "Hello, world!";

  str map {
    if (self == 'o') {
      break;
    }
    printf ("%c", self);
  }
  printf ("\n");

  str map {
    switch (self)
      {
      case 'a':
      case 'e':
```

```
      case 'i':
      case 'o':
      case 'u':
if (self == 'o') {
  printf ("\n");
  return 11;          /* __ctalkRegisterArgBlkReturn inserted   */
}                    /* here.  The String map method, which is */
break;                /* the argument block's direct caller,    */
      }                      /* contains a __ctalkArgBlkSetCallerReturn*/
    (Character *)self -= 32;  /* function call.                        */
    printf ("%c", self);
  }
  printf ("\n");
}


/* After the argument block call, the compiler inserts a
   construct like the following:

    if (__ctalkNonLocalArgBlkReturn ()) {
        return __ctalkToCInteger (__ctalkArgBlkReturnVal (), 1);
    }

  This retrieves the argument block's return value if any,
  and returns from the calling function.
*/
```

The `String : map` method contains an example of how an argument block can signal a return from the function or method that called it. Refer also to the *__ctalkArgBlkSetCallerReturn* and *__ctalkArgBlkClearCallerReturn* functions above.

`__ctalkRegisterBoolReturn (int t-or-f-arg)`

Returns a boolean object with a true or false value depending on the value of *t-or-f-arg*. If the Boolean class variables `boolTrue` or `boolFalse` are defined, returns one of those objects. Otherwise, creates a `Boolean` object with the value true or false.

`__ctalkRegisterCharPtrReturn (char *var)`

Saves a C `char *` method return value to the method's resource pool.

`__ctalkRegisterCharPtrReturn (char var)`

Saves a C `char` method return value to the method's resource pool.

`(char *type, char *qualifier, char *qualifier2, char *qualifier3, char *qualifier4, char *storage_class, char *name, int n_derefs, int attrs, int is_unsigned, int scope)`

Register a C typedef with an application. This function is typically used by *__ctalk_init* to register typedefs defined in C include files and elsewhere.

`__ctalkRegisterExtraObject (OBJECT *created_object)`
: Save an object retrieved by a function so it may be referred to later. This function registers each object only once and does not adjust the object's reference count or scope. The `__ctalkRegisterExtraObject` function silently ignores request to register global and class objects. Refer to the entry for `__ctalkRegisterUserObject,` below.

`__ctalkRegisterFloatReturn (double d)`
: Registers a C `double` return value as a `Float` method resource object. Note that the C libraries do not automatically convert C `floats` to `doubles`, so if you register a C `float` as a method resource, you need to cast it to a `double` first.

`__ctalkRegisterIntReturn (int returnval)`
: Registers a C `int` method return value as an `Integer` method resource object.

`__ctalkRegisterIntReturn (long long int returnval)`
: Registers a C `long long int` method return value as a `LongInteger` method resource.

`__ctalkRegisterUserFunctionName (char *name)`
: Registers the names of C functions in the program, mainly for diagnostic messages. This function is added automatically to `__ctalk_init` whenever a C function in the source code is parsed and is called at the start of a program.

`__ctalkRegisterUserObject (OBJECT *created_object)`
: Save objects created by a method so they may be referred to later. New objects registered by this function have a reference count of 1, and have the additional scope `METHOD_USER_OBJECT`. This function is also used by many of the `methodReturn*` macros, and if necessary you can included it in a method if you need to register an object in some non-standard manner. See ⟨undefined⟩ [Returning method values], page ⟨undefined⟩.

  Note that global objects and class objects do not need to be registered. In fact, registering such objects as method resources can confuse the object's entries in their respective dictionaries, because method resources have a separate dictionary of their own. If a method tries to register a class object or global object, `__ctalkRegisterUserObject` silently ignores the request.

`__ctalkReplaceVarEntry (VARENTRY *varentry, OBJECT *new_object)`
: This function has been superceded. If you want to attach an Object to another tag, it's only necessary to use an assignment statement. See *__ctalkAliasReceiver ()* for an example

`__ctalkRtGetMethod (void)`
: Returns the currently executing method as a `METHOD *` from the call stack, or NULL if called from within a C function.

`__ctalkRtReceiver (OBJECT *receiver_object)`
: Sets the call stack's receiver to *receiver_object*. The function however, does not alter the currently executing method's receiver on the receiver stack.

`__ctalkRtReceiverObject (void)`
: Returns the currently executing method's receiver object from the call stack.

`__ctalkRtSaveSourceFileName (char *fn)`
: Called during the initialization of a function or method to store the name of its source file.

`__ctalkRtGetMethodFn (void)`
: Returns the C function pointer (an `OBJECT *(*)()` of the currently executing method, or NULL if called from within a C function.

`__ctalkRtMethodClass (OBJECT *class_object)`
: Sets the class object of the currently executing method to *class_object*.

`(OBJECT *class_object)`
: Returns the class object of the currently executing method.

`__ctalkSearchBuffer (char *pattern, char *buffer, long long *offsets)`
: Finds all occurrences of *pattern* in *buffer*, and returns the positions of the matches in *offsets*, terminated by -1.

`__ctalkSelectXFontFace (void *display, int drawable_id, int gc_ptr, int face)`
: Selects the typeface of the currently selected font, if available, which should have been loaded with a call like *__ctalkX11UseFontBasic*, or the equivalent calls for FreeType fonts.

  The argument, *face*, may be one of the following.

```
X_FACE_REGULAR
X_FACE_BOLD
X_FACE_ITALIC
X_FACE_BOLD_ITALIC
```

  Because these functions use shared memory to manage each X typeface's metrics, it is generally necessary to call this function after calling *__ctalkOpenX11InputClient* in order to display multiple faces with the correct character spacing.

`__ctalkSelfPrintOn (void)`
: Print the calling method's arguments to the receiver. This function is called directly by `printOn` (class `String`) and similar methods. See ⟨undefined⟩ [String], page ⟨undefined⟩.

`__ctalkSetExceptionTrace (int val)`
: Enable or disable exception method traces in `handle` (class `Exception`) and other methods. See ⟨undefined⟩ [Exception], page ⟨undefined⟩.

`__ctalkSetObjectAttr (OBJECT *object, unsigned intattribute)`
`__ctalkObjectAttrAnd (OBJECT *object, unsigned intattribute)`
`__ctalkObjectAttrOr (OBJECT *object, unsigned intattribute)`
: These methods sets the `attr` member of *object* to *attribute*.

Note that when setting or clearing attributes on complex objects, it is better to use *__ctalkObjectAttrAnd* or *__ctalkObjectAttrOr*, because complex objects can contain instance variables that use different attributes.

For example, an instance variable that is a `Symbol` would have the additional attribute '`OBJECT_VALUE_IS_BIN_SYMBOL`', so if a method or function contained an expression like this:

```
__ctalkSetObjectAttr (myObj, myObj -> attrs | OBJ_HAS_PTR_CX);
```

This would have the effect of setting the parent object's attributes as well as the instance variables, which would clear any additional attributes that the instance variables have.

What would actually happen is that the parent object `myObj`, and its instance variables would have their attributes set like this.

```
<obj> -> attrs = (myObj -> attrs | OBJ_HAS_PTR_CX);
```

so it is better to use an expression like this one.

```
__ctalkObjectAttrOr (myObj, OBJ_HAS_PTR_CX);
```

This has the effect of applying the following to the parent object and each instance variable:

```
<obj> -> attrs |= OBJ_HAS_PTR_CX);
```

Conversely, to clear a single attribute, a function or method would contain an expression like this.

```
__ctalkObjectAttrAnd (myObj, ~OBJ_HAS_PTR_CX);
```

`__ctalkSetObjectName (OBJECT *`*object*`, char *`*name*`)`
Sets the name of *object* to *name*.

`__ctalkSetObjectScope (OBJECT *`*object*`, int `*scope*`)`
Set the scope of *object* to *scope*. Note that many of Ctalk's scopes are only used internally. The scopes that are useful in methods are defined in `ctalkdefs.h`. Those definitions are listed here along with their values. See ⟨undefined⟩ [Scoping], page ⟨undefined⟩.

```
GLOBAL_VAR          (1 << 0)
LOCAL_VAR           (1 << 1)
CREATED_PARAM       (1 << 6)
CVAR_VAR_ALIAS_COPY (1 << 7)
```

```
VAR_REF_OBJECT      (1 << 9)
METHOD_USER_OBJECT  (1 << 10)
```

`__ctalkSetObjectValue (OBJECT *object, char *value)`
: This is a wrapper for *__ctalkSetObjectValueVar ()*, below, which was used in earlier versions of the class libraries. You should use *__ctalkSetObjectValueVar ()* instead.

`__ctalkSetObjectValueAddr (OBJECT *object, void *mem_addr, int data_length)`
: Set *object's* value to a pointer to the memory area *mem_addr*. The object must be a member of Vector class or one of its subclasses. The function also sets the object `length` instance variable, and adds OBJECT_VALUE_IS_MEMORY_VECTOR to its attributes, and registers the vector * address.

`__ctalkSetObjectValueBuf (OBJECT *object, char *buf)`
: Set the `value` instance variable to the buffer *buf*. Unlike *__ctalkSetObjectValue ()* and *__ctalkSetObjectValueVar ()*, this function replaces the value of *object* with *buf*, even if *buf* is empty, so you can add a random-length buffer to *object*.

`__ctalkSetObjectValueVar (OBJECT *object, char *value)`
: Set the value of *object* to *value*. If *value* is `NULL`, sets *object*'s value to Ctalk's '`(null)`' string.

`__ctalkSetObjPtr (OBJECT *object, void *p)`
: Save the pointer *p* in *object*.

`__ctalkSetRS (char record_separator_char)`
: Set's the current program's record separator character, which determines, among other things, how regular expression metacharacters work with line endings. See ⟨undefined⟩ [RecordSeparator], page ⟨undefined⟩.

`__ctalkSignalHandlerBasic (int signo)`
: Provides a basic signal handler that is more robust than the methods in `SignalHandler` class, but less flexible. Causes the application to terminate and print a walkback trace if enabled.

  Applications can use `__ctalkInstallHandler ()` to install the signal handler. In this case it works similarly to a method with a C calling protocol. Here is the `installExitHandlerBasic` method from `Application` class.

```
Application instanceMethod installExitHandlerBasic (void) {
  __ctalkInstallHandler
    (__ctalkSystemSignalNumber ("SIGINT"),
     (OBJECT *(*)())__ctalkSignalHandlerBasic);

  return NULL;
}
```

`__ctalkSpawn (char *command, int restrict_io)`

The *__ctalkSpawn* function launches the program named by *command* as a daemon process, and then returns to the parent program and continues execution of the parent.

The function returns the process id of the child process.

The daemon process runs as a true daemon - that is, without a controlling terminal, and without the standard input, output, or error channels. All communication between the daemon and the parent program should take place with UNIX interprocess communication facilities.

If *restrict_io* is non-zero, the program changes the daemon processes' working directory to '`/`' and sets its umask to '`0`'.

Traditionally, a parent program exits immediately after spawning a daemon process. But *__ctalkSpawn* maintains the session process - the process that handles the session and I/O initialization before it launches the daemon. The session process stays active until the parent process exits and orphans it. Then the session process exits also, leaving the daemon to run completely in the background until it is killed. That means, while the parent program is running, there can be *three* entries in the system's process table, when viewed with a program like `ps` or `top`. However, it also mimimizes the possibility of causing zombie processes should any part of the program quit unexpectedly.

You should note that *__ctalkSpawn* does not use a shell or any shell facilities to exec the daemon process, which means the function doesn't support I/O redirection or globbing. If you want the parent process to handle the child processes' I/O, refer to the *__ctalkExec* function. See ⟨undefined⟩ [ctalkExec], page ⟨undefined⟩.

`__ctalkStringifyName (OBJECT *src, OBJECT *dest)`

When called by a function like `String : =`, performs some munging of different types of `String` objects in order to keep the API consistent for different types of `String` objects.

`__ctalkSplitText (char *text, OBJECT *list_out)`

Splits a text buffer into word tokens, and returns the tokens as members of *list_out*. This function preserves newlines and spaces, and places HTML-style format tags in their own tokens. This is used by classes like `X11TextPane` to split its text buffer before displaying the wrapped text. See ⟨undefined⟩ [X11TextPane], page ⟨undefined⟩.

`__ctalkStrToPtr (char *s)`

If *s* is a C string formatted as a hexadecimal number with the format `0x`*nnnnnnn*, return a C `void *` pointer with that address.

`__ctalkSysErrExceptionInternal (MESSAGE *orig, int errno, char *text)`

Generates an exception base on *errno* with the text *text*. Ctalk translates *errno* in an exception that represents the C library's `errno` error definitions. The *orig* argument provides the line and column number where the exception occurred. If NULL, the exception doesn't record the line and column information.

`__ctalkSystemSignalName (int signo)`
: Returns a string containing a mnemonic name like `SIGINT` or `SIGHUP` that corresponds to *signo*. Includes the mnemonics of the common signals defined by POSIX standards.

`__ctalkSymbolReferenceByName (OBJECT *object)`
: Used in `Symbol : =` and similar methods returns a boolean value of `true` if the object (the argument to the method normally) was retrieved by its name, or false if the argument is the result of pointer math or indirection. This allows the method to determine whether it needs to perform additional indirection or pointer math on the argument before assigning it the the receiver.

`__ctalkSystemSignalNumber (char *signame)`
: For a signal named *signame*, return the number system-dependent number of the signal. The function defines names POSIX 1990 signals on most systems. Refer to the system's *signal(2)* (or similar) manual page for information.

`__ctalkTemplateCallerCVARCleanup (void)`
: Cleans up after a *__ctalkGetTemplateCallerCVAR* call. See ⟨undefined⟩ [__ctalkGetTemplateCallerCVAR], page ⟨undefined⟩. Ctalk calls this function internally; you should not need to use it in your own programs.

  (macro) Note that this function does not know about parameter substitution. If you want to print an object that is an argument to a method, use the `ARG(n)` macro, and reference the `name` member. See ⟨undefined⟩ [ARG macro], page ⟨undefined⟩.

```
__ctalkPrintObject(ARG(0)->__o_name);
```

`__ctalkTerminalHeight (void)`
`__ctalkTerminalWidth (void)`
: Returns the height and width of the terminal in character rows and columns. If the terminal does not support reporting its size, these functions return 0.

`__ctalkToCArrayElement (OBJECT *o)`
: Translate the value of an `Integer`, `Character`, `String`, or `LongInteger` array element to a `void *` that points to its corresponding C data type.

`__ctalkToCCharPtr (OBJECT *obj, int keep)`
: Returns the value of *obj* as a C `char *`. If *keep* is zero, deletes *obj* if possible.

`__ctalkToCDouble (OBJECT *obj)`
: Returns the value of *obj* as a C `double`.

`__ctalkToCIntArrayElement (OBJECT *obj)`
: Returns the value of *obj* as a C `int`. This function has mostly been superceded by *__ctalkToCInteger* (below).

`__ctalkToCInteger (OBJECT *obj)`
: Returns the value of *obj* as a C `int`. The value can be a binary, octal, decimal, or hexadecimal number. Prints a warning message if the value is not a valid number or is out of range.

`__ctalkTrapException (void)`
: If there is a run-time exception pending, returns the first exception in Ctalk's internal format. Otherwise, returns NULL.

`__ctalkTrapExceptionInternal (void)`
: Similar to *__ctalkTrapException*, except that it works with the passes in the compiler as well as the run time library.

`__ctalkObjectPrintOn (OBJECT *object)`
: Print the calling method's arguments to the argument's `value` instance variable. This function is called directly by `printOn` (class `ANSITerminalStream`) and similar methods. See ⟨undefined⟩ [ANSITerminalStream], page ⟨undefined⟩.

`__ctalkOpenX11InputClient (OBJECT *X11TerminalStream_object)`
: Start a GUI program's input client in the background. The input client receives input events, like mouse motion and keypresses, and window events, like resize notifications from the X display server, and sends the information to the application program so that it can queue `InputEvent` objects which the app can then process.

  The argument is a `X11TerminalStream` object, which is normally created with a `X11Pane` object, and which programs can refer to by the `X11Pane` object's `inputStream` instance variable.

  This is the lower-level function that the `openEventStream` (class `X11Pane`) method uses to begin communicating with the X display server. For an example, refer to the `X11TerminalStream` section. See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`__ctalkUNIXSocketOpenReader (char *socketpath)`
: Opens a UNIX domain socket, binds the socket to the path given by *socketpath*, and places the socket in listening mode.

  Returns the file descriptor of the new socket on success, or -1 if an error occured, in which case the C library sets the variable `errno`.

`__ctalkUNIXSocketOpenWriter (char *socketpath)`
: Opens a UNIX domain socket and connects to the socket given by *socketpath*.

  Returns the file descriptor of the new socket on success, or -1 if an error occurs, in which case the C library sets the variable `errno`.

`__ctalkUNIXSocketRead (int sockfd, void * buf_out)`
: Reads data from the socket given by *sockfd*. On success, returns the data read in *buf_out* and the return value is the number of bytes read. On error returns -1 and the C library sets the variable *errno*.

`__ctalkUNIXSocketShutdown (int sockfd, int how)`
: This function is a wrapper for the C library's `shutdown` function. Shuts down the socket identified by the *sockfd* argument. The second argument, *how*, can be either `SHUT_RD, SHUT_WR, or SHUT_RW.` These constants are defined in `UNIXNetworkStream` class and described in the *shutdown(1)* manual page.

  The function returns 0 on success, or -1 if an error occurred.

`__ctalkUNIXSocketWrite (int sockfd, void * data, int length)`
: Writes *length* bytes of *data* to the socket given by *sockfd*.

  On success returns the number of bytes written, or returns -1 on error, in which case the C library sets the variable `errno`.

`__ctalkUTCTime (void)`
: Returns an `int` with the system's UTC time. This function is currently a wrapper for the *time(2)* function. Because *time(2)*'s argument is an `int *`, it can often be more reliable to use *__ctalkUTCTime* and let the library worry about the argument's storage. There is also a template for *time(2)* if you want to use the function directly in complex expressions.

`__ctalkWarning (char *fmt, ...)`
: Prints a formatted message to the terminal. Unlike *_warning* and other functions, does not add line numbers or input file information to the output.

`__ctalkWrapText (unsigned int drawable, unsigned int gc_ptr, OBJECT *text_list, int pane_width, int lmargin)`
: Formats the text in *text_list* to be displayed between *lmargin* and *pane_width* (the right edge of the drawing surface given as the first argument.. The *text_list* list should have been generated by *__ctalkSplitText*. The *__ctalkWrapText* function uses the current typeface to determine character widths. If no font or typeface is selected, uses the default font, "fixed" to format the text.

`__ctalkX11CloseClient (OBJECT *pane_object)`
: Closes the main program's connection to the X11 client and exits the client process.

`__ctalkX11CloseParentPane (OBJECT *pane_object)`
: Closes and deletes an application's main X11 window, and its buffers and other data. Does not delete subpanes - see __ctalkCloseX11Pane () (above) to delete subpanes. Applications should delete subpanes before closing and deleting the main window. For an example of the functions' use, refer to the method `X11Pane : deleteAndClose`. See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩.

`__ctalkX11ClearRectangleBasic (void *display, int visual_id, int gc_ptr, int x, int y, int width, int height)`
: Clear a rectangle of a visual type like a pixmap to the background color.

`__ctalkX11Colormap (void)`
: Returns the X resource ID of the display's default colormap. It's contained in a library function because the X headers define some of the DefaultColormap's dependent macros after DefaultColormap, which is not compatible with the ctpp preprocessor.

`__ctalkX11CopyPixmapBasic (void *display, int dest_drawable_id, int dest_gc_ptr, int src_drawable_id, int src_x_org, int src_y_org, int src_width, int src_height, int dest_x_org, int dest_y_org)`
: Copies the drawable *src_drawable_id* to *dest_drawable_id*, with the dimensions of the source graphic given by *src_x_org*, *src_y_org*, *src_width*, and *src_height*. The image is drawn with its upper left-hand corner positioned at *dest_x_org*, *dest_y_org* on the destination drawable.

This function is called by `X11CanvasPane : copy`. For an example, refer the `X11CanvasPane` classes' description. See ⟨undefined⟩ [X11CanvasPane], page ⟨undefined⟩.

`__ctalkX11CreateGC (void *display, int drawable)`
Create a X Graphics Context and return its address as a `void *`. The GC is created with the following values:

```
foreground      white
background      black
fill_style      FillSolid
function        GXcopy
font            fixed
```

`__ctalkX11CreatePixmap (void *display, int x_drawable, int width, int height, int depth)`
reate a X pixmap and return its X resource ID as an unsigned int.

`__ctalkX11CreatePaneBuffer (OBJECT *pane_object, int width, int height, int depth)`
Create the buffers for a pane object's X window. Applications normally call this function when the pane object is created or when a subpane is attached to a parent pane. This function sets the *pane_object's* `paneBuffer` and `paneBackingStore` instance variables.

`__ctalkX11DeletePixmap (int drawable_id)`
Delete the server-side pixmap whose ID is given as the argument.

`__ctalkX11Display (void)`
Return a pointer to the X display, opening the display if necessary.

`__ctalkX11DisplayHeight (void)`
Returns an `int` with the display's height in pixels.

`__ctalkX11DisplayWidth (void)`
Returns an `int` with the display's width in pixels.

`__ctalkX11FontCursor (OBJECT *cursor_object, int cursor_id)`
Set *cursor_object*'s value to a X11 *cursor_id*. Cursor ID's are defined by the X server in the include file `X11/cursorfont.h`. See ⟨undefined⟩ [X11Cursor], page ⟨undefined⟩.

`__ctalkX11FreeGC (int gc_addr)`
Free the X11 graphics context pointed to by *gc_addr*. The address of the GC is given as an `int` which does not require any special handling by methods; the library function casts *gc_addr* to a `GC *`.

`__ctalkX11FreePaneBuffer (OBJECT *pane_object)`
Release the server-side buffers used by *pane_object*. Note that this function is being phased out; programs should use *__ctalkX11DeletePixmap*, which does not rely on hard-coded instance variable names.

`__ctalkX11FreeSizeHints (void)`
: Frees the data allocated by *__ctalkX11SetSizeHints ()*, below.

`__ctalkX11GetSizeHints (int win_id, int *x_org_return, int *y_org_return, int *width_return, int *height_return, int *win_gravity_return, int *flags_return)`
: Get the actual size and placement of the window, as reported by the X server, after the window is created, normally with *__ctalkCreateX11MainWindow ()*.

`__ctalkX11InputClient (OBJECT *streamobject int fd)`
: The `X11TerminalStream` input client. This function is not used directly by any method but is a process of the *__ctalkOpenX11InputClient* function, above.

`__ctalkX11MakeEvent (OBJECT *eventobject_value_var, OBJECT *inputqueue)`
: Encapsulates much of the function of the `X11TerminalStream : queueInput` method: receives the data for an X event from the X11 input client and saves it in an `InputEvent` object, then queues the `InputEvent` object in the `X11TerminalStream`'s `inputQueue`.

`__ctalkX11MoveWindow (OBJECT *pane_object, int x, int y)`
: Move *pane_object*'s window so that its origin is at *X*,*Y*.

`__ctalkX11OpenInputClient (OBJECT *streamobject)`
: The library interface of the `X11TerminalStream` class's input client. This function is called by `openInputClient` (class `X11TerminalStream`). The *streamobject* argument is a `X11TerminalStream` object, generally the stream created by `new` (class `X11Pane`). See ⟨undefined⟩ [X11TerminalStream], page ⟨undefined⟩.

`__ctalkX11ParseGeometry (char *geomString, int* x, int* y, int* y, int* width, int* height)`
: Parses a X11 geometry string and returns the values specified in the *x*, *y*, *width*, or *height* variables. If the geometry string does not specify one of these values, sets the corresponding variable to zero.

  For information about the format of a X11 geometry specification, refer to the *XParseGeometry(3)* manual page.

`__ctalkX11PaneDrawCircleBasic (void *display, int window_id, int gc, int center_x, int center_y, int radius, int fill, int pen_width, int alpha, char *fg_color_name, char * bg_color_name)`
: Draws a circle centered at *center_x*,*center_y* with radius *radius*. The dimensions are given in pixels. If *filled* is true, then the function draws a filled circle; otherwise, the circle's edge has the width *pen_width*.

  This function is a synonym for *__ctalkGUIPaneDrawCircleBasic*.

`__ctalkX11PaneDrawRectangleBasic (void *display, int drawable_id, unsigned long intgc_ptr, int xOrg, int yOrg, int xSize, int ySize, int fill, int pen_width, char *pen_color, int corner_radius)`
: Draw a rectangle on the drawable with the ID *drawable_id*, with the dimensions givent in the arguments. If *fill* is non-zero, draws a filled rectangle; otherwise uses the line width given by the *pen_width* argument. If *corner_radius* is non-zero, draws a rectangle with rounded corners with the radius in pixels given by *corner_radius*.

This function is a synonym for *__ctalkGUIPaneDrawRectangleBasic*.

`__ctalkX11PanePutStr (OBJECT *pane_object, int x, int y, char *str)`
Displays *str* at window coordinates *x*,*y* on *pane_object*'s drawable in the pane's current font. If `pane_object` is buffered, writes the string to the pane's buffers, and the string is displayed at the next `refresh` method call.

Note that this method is slowly being superceded because it relies on instance variable names that are defined in several class libraries. If the application uses different drawables than *pane_object*'s window surface or its buffers, use *__ctalkX11PanePutStrBasic* instead.

`__ctalkX11SetSizeHints (int x, int y, intp width, int height, int geom_flags)`
Set the window size hints based on the window dimensions set by the application. The *geom_flags* argument has the format provided by the *__ctalkX11ParseGeometry ()* function, above. Normally this function is called by a `X11Pane*` class when initializing a window.

If an application calls this function, it must also call *__ctalkX11FreeSizeHints ()*, above.

`__ctalkX11ResizePaneBuffer (OBJECT *pane_object, int width, int height)`
Resize *pane_object*'s buffers to width *width* and height *height*. New programs should use *__ctalkX11ResizePixmap*, which does not rely on hard-coded instance variable names.

`__ctalkX11ResizePixmap (void *display, int, parent_drawable_id, int self_xid, int gc, int old_width, int old_height, int new_width, int new_height, int depth, int *new_pixmap_return)`
Create a new Pixmap with the dimensions *new_width* and *new_height* that contains the contents of the original pixmap. Returns the X ID of the new pixmap in *new_pixmap_return*.

`__ctalkX11ResizeWindow (OBJECT *pane_object, int width, int height, int depth)`
Resize a pane object's X window. Returns '`1`' on success, '`0`' if the window's new size is <= its current size, and '`-1`' if there is an error.

`__ctalkX11PaneClearRectangle (OBJECT *pane_object, int x, int y, int width, int height)`
Clears a rectangle in *pane_object*'s window. Note that this function is deprecated - it relies on the Pane object having specific instance variables. New programs should use *__ctalkX11PaneClearRectangleBasic* instead.

`__ctalkX11PaneDrawLine (OBJECT *pane_object, OBJECT *line_object, OBJECT *pen_object)`
Draw a line on the drawable identified by *pane_object* See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩, with the endpoints given by *line_object* See ⟨undefined⟩ [Line], page ⟨undefined⟩, with the line width and color defined in *pen_object* See ⟨undefined⟩ [Pen], page ⟨undefined⟩. This function is a synonym for *__ctalkGUIPaneDrawLine* on systems with a X Window System display.

`__ctalkX11PaneDrawLineBasic (int drawable_id, int gc_ptr, int x_start, int y_start, int x_end, int y_end, int pen_width, int alpha, char *pen_color)`

Draw a line between the points (x_start,y_start) and (x_end, y_end) with the color, and transparency using the drawable ID, graphics context, and pen color, width, and transparency given as arguments.

`__ctalkX11PaneDrawPoint (OBJECT *pane_object, OBJECT *point_object, OBJECT *pen_object)`

Draw a point on the drawable id given in *pane_object* See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩, with the location given by *point_object* See ⟨undefined⟩ [Point], page ⟨undefined⟩, with the radius and color given by *pen_object* See ⟨undefined⟩ [Pen], page ⟨undefined⟩. This function is a synonym for *__ctalkGUIPaneDrawPoint* on systems that use the X Window system.

`__ctalkX11PaneDrawPointBasic (void *display, int drawable_id, int gc_ptr, int x, int y, int pen_width, int alpha, char *pen_color)`

Draw a point of the size, position, color, and transparency using the drawable ID, graphics context, and Pen color, transparency, and width given as arguments.

`__ctalkX11PaneClearWindow (OBJECT *pane_object)`

Clears *pane_object*'s window. Note that this function is deprecated - it relies on the Pane object having specific instance variables. New programs should use

`__ctalkX11PaneDrawRectangle (OBJECT *pane_object, OBJECT *rectangle_object, OBJECT *pen_object, Integer fill)`

Draw a rectangle on the drawable identified by *pane_object* See ⟨undefined⟩ [X11Pane], page ⟨undefined⟩, with the dimensions given by *rectangle_object* See ⟨undefined⟩ [Rectangle], page ⟨undefined⟩, and with the line width and color given by *pen_object* See ⟨undefined⟩ [Pen], page ⟨undefined⟩. If *fill* is true, draw a filled rectangle.

`__ctalkX11PanePutStrBasic (void *display, int visual_id,int gc_ptr, int x, int y, char *text)`

Write the string *text* on the drawable named by *visual_id* at *X*,*Y* using the graphics context pointed to by *gc_ptr*. If the drawable is a `X11CanvasPane` buffer, the text will not be visible until the next call to the pane's `refresh` method.

`__ctalkX11PaneRefresh (OBJECT *pane_object, int src_x_org, int src_y_org, int src_width, int src_height, int dest_x_org, int dest_y_org)`

If *pane_object* is a buffered pane, copy the contents of the pane buffer(s) within the rectangle given by *src_x_org*, *src_y_org*, *src_width*, *src_height* to the visible window at *dest_x*, *dest_y*.

`__ctalkX11QueryFont (OBJECT *font_object, char *xlfd)`

Fills in *font_object*'s '`ascent`', '`descent`', '`maxWidth`', '`height`', and '`fontDesc`' instance variables with the font metrics returned by the X server for the font given by *xlfd*.

`__ctalkX11SetBackground (OBJECT *pane_object, char *color_name)`
: Set *pane_object*'s background color to *color_name*. This function is being phased out because it uses named instance variables of *pane_object*. Programs should use *__ctalkX11SetBackgroundBasic ()*, below, instead.

`__ctalkX11SetBackgroundBasic (void *display, int visual_xid, int gc_ptr, char *color)`
: Sets the background color of any class with a X11 visual and graphics context.

`__ctalkX11SetForegroundBasic (void *display, int visual_xid, int gc_ptr, char *color)`
: Sets the foreground color of any class with a X11 drawable and graphics context.

`__ctalkX11SetResource (void *display, int drawable_id, char *resource_name, char *resource_value)`
: Sets the X11 resource *resource_name* to *resource_value* for the drawable identified by *drawable_id*.

`__ctalkX11SetWMNameProp (OBJECT *pane_object, char *name)`
: Sets the WMName property *pane_object*'s window to *name*. This is the window property that window managers use to set the window frame's title.

`__ctalkX11UseCursor (OBJECT *pane_object, OBJECT *cursor_object)`
: Sets the X11 cursor of *pane_object* to *cursor_object*. See ⟨undefined⟩ [X11Cursor], page ⟨undefined⟩.

`__ctalkX11UseXRender (bool b)`
: If *b* is true, draw using the X Render entension if it is available. If *b* is false, use Xlib for drawing even if X Render is available.

`__ctalkX11UseFontBasic (void *display, int drawable_id, int gc_ptr, char *font_desc)`
: Sets the font of the graphics context *gc_ptr* and drawable *drawable_id* for further string printing operations. See use in `X11Bitmap` class, where the GC pointer, which is an opaque object, is encoded as an `int`, in order to avoid confusion with `OBJECT *'s`.

`__ctalkX11UsingXRender (void)`
: Returns a boolean value of True if the program is using the X Render extension for drawing, False otherwise. To use the X Render extension, the extension and its supporting libraries must be available when the Ctalk libraries are built, and the program has not changed the default setting, normally via *__ctalkX11UseRender*, above.

`__ctalkX11XPMFromData (void *display, int drawable_id, int gc_ptr, int x_org, int y_org, char **xpm_data)`
: Draw a X pixmap at the x,y position on the drawable named by *drawable_id*. The *xpm_data* argument is the declaration of the data given in a XPM file, and has the C data type `char **`.

`__ctalkX11XPMInfo (void *display, char **xpm_data, int *width_ret, int *height_ret, int *n_colors_ret, int *chars_per_color_ret)`

Returns the width, height, number of colors, and characters per color of the XPM data referred to by *xpm_data*

`__ctalkXPMToGLTexture (char **xpm_data, unsigned short int alpha, int *width_out, int *height_out, void **texel_data_out)`
`__ctalkXPMToGLXTexture (char **xpm_data, unsigned short int alpha, int *width_out, int *height_out, void **texel_data_out)`

Read the XPM data pointed to by *xpm_data*, and return the OpenGL texture data pointed to by *texel_data_out*.

The *alpha* parameter defines the texture data transparency and should be in the range 0 - 0xffff. The alpha channel's effect may not be apparent in the image that is displayed, because OpenGL has its own set of functions to perform texture blending.

For Mesa OpenGL implementations, like those found on Linux systems, textures have the format GL_RGBA and the data type GL_UNSIGNED_INT_8_8_8_8. To define a basic 2-dimensional texture to the OpenGL API, use an OpenGL function like this.

```
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, my_width, my_height, 0,
              GL_RGBA, GL_UNSIGNED_INT_8_8_8_8, my_texel_data);
```

Apple OpenGL implmentations use a different internal format, so a program would define a texture from the *__ctalkXPMToGLXTexture* function's output like this.

```
glTexImage2D (GL_TEXTURE_2D, 0, GL_RGBA, my_width, my_height, 0,
              GL_BGRA, GL_UNSIGNED_INT_8_8_8_8_REV, my_texel_data);
```

Note that the function does not make any changes for 1-dimensional or 3-dimensional textures, nor for textures that might be rendered as mipmaps. The specific texture parameters depend on the nature of the texture and the surface that it's applied to... in most cases, though OpenGL works best with a XPM that has a geometry that is an even multiple of 2; e.g., 512x512 or 1024x1024 pixels.

Both functions are similar, and generic enough to work with any OpenGL toolkit - the main difference is that either of the functions can be implemented for either a GLUT or GLX development environemnt independently, and that the GLX version is compatible with 64-bit machines.

`__ctalkXftAscent (void)`

Returns an `int` with the currently selected font's height in pixels above the baseline.

`__ctalkXftDescStr (void)`

Returns a string with the font descriptor of the currently selected font that containing the attributes that the Ctalk Xft library uses: family, point size, slant, and weight.

For the complete descriptor string that the Freetype library derives from the selected font's pattern, refer to the *__ctalkXftSelectedFontDescriptor* function.

`__ctalkXftAscent (void)`

Returns an `int` with the currently selected font's height in pixels below the baseline.

`__ctalkXftFgRed (void)`
`__ctalkXftFgGreen (void)`
`__ctalkXftFgBlue (void)`
`__ctalkXftFgAlpha (void)`

Returns `int` values 0-0xffff for the selected font's red, green, blue, and alpha values.

`__ctalkXftRed (unsigned short val)`
`__ctalkXftGreen (unsigned short val)`
`__ctalkXftBlue (unsigned short val)`
`__ctalkXftAlpha (unsigned short val)`

Set the current Xft font's foreground color channels individually. The value of the argument must be between 0 and 65535 (0xffff hex).

`__ctalkXftFontPathFirst (char *pattern)`

Return the path of the first font file that matches *pattern*. If pattern is '`*`' or an empty string ('`""`'), return the first path of all the fonts that are available to the library.

`__ctalkXftFontPathNext (void)`

Return a string containing the path of the next font file that matches the pattern given to *__ctalkXftFontPathFirst ()*, above.

`__ctalkXftGetStringDimensions (char *str, int *x, int *y, int *width, int *height, int * rbearing)`

Return the x and y origin, and width and height of *str* in the currently selected FreeType font. Because the dimensions are absolute, *x* and *y* are normally 0. If the Xft library is not initialized or not available, the function returns 0 for all of the dimensions. If the library is initialized but the application hasn't yet specified a font, then the value returned for *rbearing* is '`0`'.

`__ctalkXftHeight (void)`

Returns an `int` with the font's height above and below the baseline in pixels, and including any additional vertical spacing.

`__ctalkXftInitLib (void)`

Initializes the Freetype outline font library. If Ctalk isn't built with libXft or Fontconfig support, this function returns `ERROR` ('`-1`'. This is the only Xft-related function to return silently; all other functions exit with an error message. However applications can check the `X11Pane : haveXft` instance variable to

determine if the libraries were initialized. See ⟨undefined⟩ [haveXft], page ⟨undefined⟩.

If the system is not configured to use outline fonts See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩, then the function prints a message and exits the program. If the library is already initialized, then this function returns `SUCCESS` '`0`'.

`__ctalkXftInitialized (void)`
Returns TRUE if the FreeType font library is available and initialized, FALSE otherwise.

`__ctalkXftIsMonospace (void)`
A read-only function that returns '`true`' if the program's selected font is monospace, false otherwise.

`__ctalkXftListFontsFirst (char *xftpattern)`
Initializes the FreeType library to list fonts and returns a `char *` that contains the first font descriptor that contains the string *xftpattern*. If *xftpattern* is emtpy ('`""`') or '`*`', then the function returns the first font, and the following calls to *__ctalkXftListFontsNext ()* match all of the fonts available to the FreeType library.

`__ctalkXftListFontsNext (void)`
Returns a `char *` with the next matching font descriptor of a font listing initialized by *__ctalkXftListFontsFirst ()*.

`__ctalkXftListFontsNext (void)`
Cleans up after a series of list fonts function calls.

`__ctalkXftMajorVersion (void)`
Returns an `int` with the Xft library's major version number. Like all libXft library functions, *__ctalkXftMajorVersion* causes the program to exit with an error if Ctalk is built without libXft support.

`__ctalkXftMaxAdvance (void)`
Returns an `int` with the maximum horizontal dimension in pixels of any of the selected font's characters.

`__ctalkXftMinorVersion (void)`
Returns an `int` with the Xft library's minor version number. Returns an `int` with the Xft library's major version number. Like all libXft library functions, *__ctalkXftMinorVersion* causes the program to exit with an error if Ctalk is built without libXft support.

`__ctalkXftQualifyFontName (char *pattern)`
Return the qualified font name string for *pattern*.

`__ctalkXftRequestedFamily (void)`
`__ctalkXftRequestedPointSize (void)`
`__ctalkXftRequestedSlant (void)`
`__ctalkXftRequestedWeight (void)`
`__ctalkXftRequestedDPI (void)`
Returns the font attributes requested after parsing a Fontconfig string by *__ctalkXftSelectFontFromFontConfig* and *__ctalkXftSelectFontFromXLFD*.

`__ctalkXftRevision (void)`
Returns an `int` with the Xft library's revision number. Returns an `int` with the Xft library's major version number. Like all libXft library functions, *__ctalkXftRevision* causes the program to exit with an error if Ctalk is built without libXft support.

`__ctalkXftSelectFontFromFontConfig (char *fontDesc)`
Parses a Fontconfig font descriptor and selects the font. For information about selecting fonts using Fontconfig descriptors, refer to the `X11FreeTypeFont` class. See ⟨undefined⟩ [X11FreeTypeFont], page ⟨undefined⟩.

`__ctalkXftShowFontLoad (int warningLevel)`
Enables or disables the display on standard output of fonts that the library is loading. The Ctalk library defines the following constants.

```
XFT_NOTIFY_NONE
XFT_NOTIFY_ERRORS
XFT_NOTIFY_LOAD
XFT_NOTIFY_VERBOSE
```

See also *__ctalkXftVerbosity*.

Note that programs should call this function before launching any processes. Generally, this is before the program calls the `X11Pane : openEventStream` method. Otherwise, the program will only display message for the process that this function was called from.

`__ctalkXftSelectedFamily (void)`
Returns a `char *` string that contains family of the selected font.

`__ctalkXftSelectedPointSize (void)`
Return the point size of the selected font as a C *double*.

`__ctalkXftSelectedSlant (void)`
Returns an `int` that contains slant of the selected font.

`__ctalkXftSelectedSlant (void)`
Returns an `int` that contains slant of the selected font.

`__ctalkXftSelectedFontDescriptor (void)`
Returns a `char *` string that contains the descriptor of the selected font.

`__ctalkXftSelectedFontDescriptor (void)`
Returns a `char *` string that contains the file path descriptor of the selected font.

`__ctalkXftSelectFont (char *family, int slant, int weight, int dpi, double point_size)`

Selects the FreeType font that matches the arguments. Returns 0 if successful in matching the font given by the arguments. If no matching font is found, the current font does not change, and the method returns -1. The `selectFont` method (class `X11FreeTypeFont` contains a description of the parameters recognized by the function. See ⟨undefined⟩ [X11FreeTypeFont-selectFont], page ⟨undefined⟩.

`__ctalkXftSelectFontFromXLFD (char *xlfd)`

Selects fonts in the FreeType font library using a XLFD specification. When selecting outline fonts: the libraries use the fields: family, weight, slant, dpi-x, and pixel height. An example XLFD would be the following.

```
-*-Nimbus Sans L-medium-r-*-*-12-72-*-*-*-*-*-*
```

Note that the function does not translate between bitmap and outline font families - the font libraries pick the closest match to the font metrics given in the XLFD, regardless of type style.

Also, the outline font libraries use a single dpi metric for both the vertical and horizontal dot pitch of the display, so only the '`resx`' field of the XLFD is actually used.

`__ctalkXftSetForegrounc (int red, int green, int blue, int alpha)`

Sets the red, green, blue, and alpha values for the selected font. The values are `int`s and have a range of 0-0xffff.

`__ctalkXftSetForegroundFromNamedColor (String colorName)`

Sets the selected outline font's red, green, and blue values from the named X11 color given as the argument.

`__ctalkXftSelectedFontDescriptor (void)`

Return the font descriptor of the selected font as a `char *`.

`__ctalkXftVerbosity (void)`

Returns the current Xft message reporting level. The values that are defined in `ctalk/ctalkdefs.h` are:

```
XFT_NOTIFY_NONE
XFT_NOTIFY_ERRORS
XFT_NOTIFY_LOAD
XFT_NOTIFY_VERBOSE
```

See also *__ctalkXftShowFontLoad*.

`__ctalkXftVersion (void)`

Returns an `int` with the Xft library's version number, which is (__ctalkXftMajorVersion () * 10000) + (__ctalkXftMinorVersion * 100) + __ctalkXftRevsion ().

`__ctalk_arg (char *rcvr, char *method,void *arg)`
: Define an argument for the following `__ctalk_method ()` call.

`__ctalk_arg_cleanup (OBJECT *result)`
: Remove an argument used by the previous method call. If used within an expression, then *result*, the return object of the previous method call, may not be NULL. If used after a `__ctalk_method ()` call, then *result* may be NULL.

`__ctalk_arg_internal (int n_th_arg)`
: Return the *n_th_arg* that method was called with, as an `OBJECT *.` The first argument's index on the stack is 0 within the caller's argument frame, and the last argument is *method* -> *n_params* - 1.

`__ctalk_arg_value_internal (int n_th_arg)`
: Return the value object of the *n_th_arg* that method was called with, as an `OBJECT *,` if available. If it isn't (for example, if the value instance variable is used as an argument alone, and not the parent object), then the function returns the argument object.

  As with `__ctalk_arg_internal (),` The first argument's index on the stack is 0 within the caller's argument frame, and the last argument is *method* -> *n_params* - 1.

`__ctalk_arg_pop (void)`
`__ctalk_arg_pop_deref (void)`
: Removes and returns the last object pushed onto the argument stack. *__ctalk_arg_pop_deref* also decreases the object's reference count by one.

`__ctalk_class_initialize (void)`
: Called by __ctalk_init () to perform any neeeded initialization the before any classes are defined.

`__ctalk_define_class (ARG**args)`
: The primitive method that Ctalk executes when it encounters the '`class`' keyword.

`__ctalk_dictionary_add (OBJECT*object)`
: Add an object to the Ctalk global dictionary, or, if the object is a class object, to the class library.

`__ctalk_exitFn (int app_exit)`
: Called just before a `return` statement when returning from a C function. If the function is `main`, then *app_exit* should be non-zero, to indicate that the program is finished, and to clean up the global objects and the class library.

`__ctalk_initFn (void)`
: Called at the beginning of a function to register the function's name.

`__ctalk_initLocalObjects (void)`
: Called during method or function initialization to delete old local objects before creating new objects.

`__ctalk_get_object (char *name, char *classname)`
`__ctalk_get_object_return (char *name, char *classname)`
Retrieves the object *name*. If *classname* is non-null, retrieves the object by name and class.

The `__ctalk_get_object_return` function is similar, but it is used only as an operand to a `return` keyword. The function checks the call stack's local object cache, in case an expression cleanup removed the most recent call's local variable list during recursive or successive calls of the same method.

`__ctalk_init (char *program_name)`
Initialize the Ctalk class libraries when the program starts. The argument, *program_name* is normally `argv[0]`.

`__ctalk_method (char *rcvr_object_name, OBJECT *(method_fn *)(), char *method_name)`
Perform a simple Ctalk method call, in places where the call can be used after one or more calls to *__ctalk_arg* and followed by *__ctalk_arg_cleanup*. For complex expressions or expressions occurring within control structures, Ctalk normally uses *__ctalkEvalExpr* instead.

`__ctalk_new_object (ARG **args)`
The primitive function that is called by the '`new`' method.

`__ctalk_primitive_method (char *rcvr_name, char *method_name, int attrs)`
Call primitive method *method_name* with receiver *rcvr_name*. The *attrs* argument can be METHOD_SUPER_ATTR, which uses the receiver's superclass as the receiver redirects the method's arguments to a superclass method.

`__ctalk_process_exitFn (int app_exit)`
Similar to *__ctalk_exitFn*, above, except that the function is meant to be invoked by child processes on exit, so it does not try to manage other child processes.

`__ctalk_receiver_pop (void)`
Pops the most recent receiver object off the receiver stack and returns it.

`__ctalk_receiver_push (OBJECT *object)`
Push *object* onto the receiver stack, without changing its reference count. This can be useful in conjunction with *__ctalk_receiver_pop* to quickly retrieve the current receiver object.

```
currentReceiver = __ctalk_receiver_pop ();
__ctalk_receiver_push (currentReceiver);
```

`__ctalk_register_c_method_arg (char *decl, char *type, char *qualifier, char *qualifier2, char *storage_class, char *name, int type_attrs, int n_derefs, int initializer_size, int scope, int attrs, void *var)`
Registers a C variable for use as an argument in the following method call. The arguments provide the information given by the variable's declaration so that it can be re-created by Ctalk. The last parameter, *var*, contains the address of the actual C variable in memory.

There are also abbreviated versions of *__ctalk_register_c_method_arg*, __ctalk_register_c_method_arg_b|c|d, which work similarly but don't try to handle syntax elements that aren't present in the variable's declaration.

`__ctalk_self_internal (void)`
Return the current method's receiver object as an `OBJECT *`.

`__ctalk_self_internal (void)`
Return the current method receiver's `value` instance variable as an `OBJECT *` if it exists, or the receiver object otherwise (for example, if the receiver is the `value` instance variable itself instead of the parent object).

`__ctalk_set_global (char *name, char *classname)`
Adds the object *name* of class *classname* to ctalk's global dictionary. This function is normally called during program initialization.

`__ctalk_set_local (OBJECT *obj)`
`__ctalk_set_local_by_name (char *obj_name)`
Make *obj* a method- or function-local object. This function is normally called during a method or function's initialization when the local objects are created with a `new` method. The *__ctalk_set_local_by_name* function is similar, except that it retrieves the global object by looking up its name in the global dictionary.

`__ctalk_to_c_char (OBJECT *obj)`
Returns the value of *obj* as a C `char` if possible. If the value is an ASCII code, converts the value to the actual `char`.

`__ctalk_to_c_char_ptr (OBJECT *obj)`
Returns the value of *obj* as a C `char *`. This function has been mostly replaced by *__ctalkToCCharPtr*, which you should use instead.

`__ctalk_to_c_double (OBJECT *obj)`
Returns the value of *obj* as a C `double`. This function performs the conversion with the C library function *strtod*. Currently only supports the translation of base 10 values.

`__ctalk_to_c_int (OBJECT *obj)`
Returns the value of *obj* as a C `int`. This function handles values in hexadecimal, octal, using the C library function *strtol*, and binary, using Ctalk's internal routines. Also handles ASCII-to-char conversions if the argument is a `Character` object.

`__ctalk_to_c_longlong (OBJECT *obj)`
Returns the value of *obj*, which is normally an instance of `LongInteger` class, as a C `long long int`. This function uses the C library function *strtoll* to perform the conversion. Currently only handles `LongInteger` object values in base 10.

`__ctalk_to_c_ptr (OBJECT *o)`
Translate the value of an object into a C `void *`. If the value isn't a pointer, then return the address of the value in memory.

`__ctalk_to_c_ptr_u (OBJECT *o)`
Translate the value of an object into a C `void *`. This is an unbuffered verson of *__ctalk_to_c_ptr ()*, above. That is, if the value of the object is an empty

string, '(NULL)', or '0x0', it returns NULL. This allows you to compare objects to NULLs in C expressions without the generating compiler warnings.

(X11TextEditorPane class)

```
__edittext_delete_char (OBJECT *x11texteditorpane_object)
__edittext_insert_at_point (OBJECT *x11texteditorpane_object, int keycode,
int shift_state, int keypress)
__edittext_line_end (OBJECT *x11texteditorpane_object)
__edittext_line_start (OBJECT *x11texteditorpane_object)
__edittext_xk_keysym (int keycode, int shift_state, int keypress)
__edittext_next_char (OBJECT *x11texteditorpane_object)
__edittext_next_line (OBJECT *x11texteditorpane_object)
__edittext_next_page (OBJECT *x11texteditorpane_object)
__edittext_prev_char (OBJECT *x11texteditorpane_object)
__edittext_prev_line (OBJECT *x11texteditorpane_object)
__edittext_prev_page (OBJECT *x11texteditorpane_object)
__edittext_scroll_down (OBJECT *x11texteditorpane_object)
__edittext_scroll_up (OBJECT *x11texteditorpane_object)
__edittext_text_start (OBJECT *x11texteditorpane_object)
__edittext_text_end (OBJECT *x11texteditorpane_object)
__edittext_point_to_click (OBJECT *x11texteditorpane_object, int pointer_x,
int pointer_y)
__edittext_index_from_pointer (OBJECT *x11texteditorpane_object, int
pointer_x, int pointer_y)
__edittext_insert_str_at_click (OBJECT *x11texteditorpane_object, int
click_x, int click_y, char *)
__edittext_insert_str_at_point (OBJECT *x11texteditorpane_object, char *)
__edittext_get_primary_selection (OBJECT *x11texteditorpan_eobject, void
**buf_out, int *size_out)
__edittext_set_selection_owner (OBJECT *x11texteditorpane_object)
__edittext_recenter (OBJECT *x11texteditorpane_object)
```

Text editing functions used by X11TextEditorPane objects. For more information, refer to X11TextEditorPane class.. See ⟨undefined⟩ [X11TextEditorPane], page ⟨undefined⟩.

```
__entrytext_set_selection_owner (void *displayPtr, unsigned int win_id
unsigned long int gc_ptr)
__entrytext_get_primary_selection (OBJECT *entrypane_object)
__entrytext_update_selection (void *displayPtr, unsigned int win_id,
unsigned long int gc_ptr)
__entrytext_send_selection (void *displayPtr, void * xEvent)
```

These functions handle requesting ownership of the X display's primary selection, storing current values of the selection text that might be requested by other programs, and requesting the contents of the primary selection when another X application owns the selection. The functions are normally called by methods in X11TextEntryPane class that handle X events.

`_error (char *fmt, ...)`
: Display an error message and exit the program.

`__inspector_trace (int stack_index)`
`__inspect_get_arg (int stack_index)`
`__inspect_get_receiver (int stack_index)`
`__inspect_get_global (char *obj_name)`
`__inspect_get_local (int stack_index, char *obj_name)`
`__receiver_trace (int stack_index)`
`__inspect_globals (void)`
`__inspect_locals (void)`
`__inspect_short_help (void)`
`__inspect_long_help (void)`
: Functions used internally by Ctalk's object inspector. The function parameters are designed to be consistent with the syntax of the inspector commands, although not every function makes use of them. For details refer to the *inspect(3ctalk)* manual page, the *inspectors* section of the `ctalktools.info` Texinfo manual, and the `inspect` method in `ObjectInspector` class.

`__rt_init_library_paths (void)`
: Initialize Ctalk's library paths. The function first checks the value of the '`CLASSLIBDIRS`' environment variable for a colon-separated list of directories, then adds the location of *classlibdir* which is configured when Ctalk is built, and *classlibdir*`ctalk`.

`__warning_trace (void)`
: A generic stack trace function that prints a trace of the call stack wherever it is inserted in a program.

`__xalloc (int size)`
: Allocates a block of memory of *size* characters and returns a pointer to the memory. If the alloc call fails, generates an `_error` message and exits the program.

`__xfree (void **addr)`
: Frees the block of memory pointed to by **addr*, then sets the *addr* to to NULL. **addr* must be a block of memory previously allocated by *__xalloc*, above, or a similar *malloc* call. When used in a program, wrapping *addr* in the '`MEMADDR()`' macro provides the correct dereferencing for the allocated memory and its pointer. The example shows how the '`MEMADDR()`' macro is used.

```
char *my_ptr;

my_ptr = (char *)__xalloc (BUF_SIZE);

... do stuff ...

__xfree (MEMADDR(my_ptr));
```

`__xrealloc (void **addr, int size)`
: Re-allocates the block of memory pointed to by **addr* to size. If *size* is larger than the original block of memory, the contents of *addr* are preserved. As with *__xfree*, above, the '`MEMADDR`' macro provides the correct dereferencing for the reference to *addr*.

`_warning (char *fmt, ...)`
: Print a warning message.

`__objRefCntInc (obj_ref)`
: Increment the reference count of an object and its instance variables. This function takes an object reference as its argument. You can use the `OBJREF` macro to cast the object to an object reference.

`__objRefCntDec (obj_ref)`
: Decrement the reference count of an object and its instance variables. As with all of the `__objRefCnt*` functions, `__objRefCntDec` takes an object reference as its argument. You can use the `OBJREF` macro to cast the object to an object reference.

`__objRefCntSet (obj_ref, int refcount)`
: Set the reference count of an object and its instance variables to *refcnt*. As with all of the `__objRefCnt*` functions, `__objRefCntDec` takes an object reference as its argument. You can use the `OBJREF` macro to cast the object to an object reference.

  Note: `__objRefCntSet` will not set an object's reference count to zero. That is, if you give '`0`' as the second argument, the call has no effect. Use `__objRefCntZero` instead. That might sound silly, but it's much more reliable overall, in a programming-by-contract way.

`__objRefCntZero (obj_ref)`
: Set the reference count of an object and its instance variables to 0 (zero). As with all of the `__objRefCnt*` functions, `__objRefCntZero` takes an object reference as its argument. You can use the `OBJREF` macro to cast the object to an object reference.

  You should not need to use this function unless you're *completely* deleting an object. Refer to the `__ctalkDeleteObject` function for more information.

`obj_ref_range_chk (OBJECT **hostObj, OBJECT **targetObject)`
: Returns a `bool` value of '`true`' or '`false`' depending on whether the host object and the target object it references occupy the same data segment.

`__refObj (OBJECT **obj1, OBJECT **obj2)`
: Assign *obj2* to *obj1*. If *obj1* already points to an object, decrement its reference count. Increment the reference count of *obj2* by 1. When calling functions that use `OBJECT **` arguments, they correspond to the OBJREF_T typedef, and Ctalk defines the macro OBJREF() to cast an object to an OBJREF_T.

`_store_int (OBJECT *receiver, OBJECT *arg)`
: This is a primitive that stores the value of *arg* in *receiver*, which is an `Integer`. Checks the class of `arg`, and if *arg* is not an `Integer`, converts it to an `Integer`

value. If *receiver* a pointer to a value, then it stores *arg* as a fully-fledged object.

`BOOLVAL(`*`IntegerOrBoolValue`*`)`
: This macro returns the value of an `Integer` or `Boolean` object, or any of `Boolean's` subclasses (or just about any other scalar value), as a C `bool`. For an example of its use, refer to the entry for the See ⟨undefined⟩ [INTVAL_Macro], page ⟨undefined⟩.

`INTVAL(`*`IntegerObjectValue`*`)`
: A macro that returns the value of an `Integer` object, or any of `Integer's` subclasses, as a C `int`. Uses an `OBJECT *'s __o_value` member directly, as in this example.

```
OBJECT *my_int_object_alias = myInt;

int a = INTVAL(my_int_object_alias -> __o_value);
```

`is_zero_q (char *str)`
: Returns a `bool` if the number represented by the string evaluates to zero, false otherwise.

`LLVAL(`*`LongIntegerValue`*`)`
: A macro that returns the value of a `LongInteger` object, or any of `LongInteger's` subclasses, as a C long long int. For an example of its use, refer to the entry for the See ⟨undefined⟩ [INTVAL_Macro], page ⟨undefined⟩.

`obj_ref_str (char *`*`str`*`)`
: If *str* contains a formatted hexadecimal number of the format '`0x`*`nnnnnn`*' that points to an object, return an `OBJECT *` reference to the object, `NULL` otherwise.

`str_is_zero_q (char *`*`str`*`)`
: Like *is_zero_q*, above, except that it also checks for an empty string (and returns true), as well as a string that contains only the digit '`0`', which causes the function to return false.

`substrcat (char *`*`dest`*`, char *`*`src`*`, int `*`start_index`*`, int `*`end_index`*`)`
: Concatenate the substring of *src* from *start_index* to *end_index* to *dest*.

`substrcpy (char *`*`dest`*`, char *`*`src`*`, int `*`start_index`*`, int `*`end_index`*`)`
: Copy a substring of *src* from *start_index* to *end_index* to *dest*.

`SYMTOOBJ (`*`SymbolValue`*`)`
: This is another macro that converts a `Symbol's` reference to an `OBJECT *`. This macro can be used on the right-hand side of an assignment statement.

```
if ((value_object =
    SYMTOOBJ((self_object -> instancevars) ?
             (self_object -> instancevars -> __o_value) :
             (self_object -> __o_value))) != NULL) {
```

```
    return value_object;
}
```

`SYMVAL(SymbolValue)`

A macro that returns the value of a `Symbol` object, or any of `Symbol's` subclasses, as a C `uintptr_t *`, which is guaranteed to be valid for 32- and 64-bit machines.

However, due to the way that pointers work in C, `SYMVAL` only works on the left-hand side of an assignment; you just need a cast (for example to `OBJECT *`) in order to avoid compiler warnings when it appears on the right-hand side of an assignment. Here is an example:

```
SYMVAL(object_alias->__o_value) = (OBJECT *)some_ptr;
```

However, if he label on the right-hand side is also a `Symbol`, the `Symbol` class duplicates the address that the operand points to, not the operand itself.

```
SYMVAL(object_alias->__o_value) = SYMVAL(some_ptr -> __o_value);
```

For another, perhaps more basic, example of the macro's use, refer to the entry for the See ⟨undefined⟩ [INTVAL_Macro], page ⟨undefined⟩.

`OBJREF(obj)`

Creates a reference to an object. This macro returns an `OBJECT **`, but using the macro allows the definition of an object reference to change without affecting too much code. `OBJREF` is used with the `__objRefCnt*` functions, and in other places.

`TRIM_LITERAL(s)`

A macro that trims the quotes from a literal string. It expands to `substrcpy (s, s, 1, strlen (s) - 2)`.

`TRIM_CHAR(c)`

A macro that trims the quotes from a literal character. It expands to `substrcpy (c, c, 1, strlen (c) - 2)`

`TRIM_CHAR_BUF(s)`

A macro that trims nested single quotes from a literal character. `TRIM_CHAR_BUF` also checks whether a single quote (`''`) is the actual character.

```
xfopen (const char *path, const char *mode)
xfprintf (FILE *stream, const char *fmt, ...)
xfscanf (FILE *stream, const char *fmt, ...)
xmemcpy (void *s, const void *s, size_t)
xmemmove (void *s, const void *s, size_t)
xmemset (void *s, int c, size_t n)
xsprintf (char *s, const char *fmt, ...)
xstrcat (char *d, const char *s)
xstrncat (char *d, const char *s, size_t)
xstrcpy (char *d, const char *s)
xstrncpy (char *d, const char *, size_t)
```

These are portable wrappers for systems that #define their own (possibly more secure) versions of library functions. For the exact prototype and definition, you should consult the system's manual page for the corresponding library function (e.g., *strcpy(3)* for the definition of *xstrcpy*).

```
xstdin (void)
xstdout (void)
xstderr (void)
```

These functions return a `FILE *` with the `stdin, stdout,` or `stderr` file stream. They're useful if your compiler redefined either *stdin, stdout,* or `stderr` internally. The functions provide a consistent interface and leave any implementation details in the library, where they belong.

5 Ctalk Language Featues

This chapter describes `ctalk`'s language features and its low-level application programming interface.

5.1 `ctpp`, the `ctalk` Preprocessor

Information about `ctpp`, the Ctalk preprocessor, is contained in its Texinfo manual, `ctpp.info`.

The preprocessor is compatible with GNU `cpp` and supports ISO C99 preprocessing features. This allows you to include C header files in Ctalk programs and class libraries. Ctalk caches macros from include files, so it can use, in the GNU compiler's terminology, *include once* header files.

If you have a header file called, for example, `myheader.h`, you would wrap the definitions with the following preprocessor directives.

```
#ifndef _MYHEADER_H
#define _MYHEADER_H
.
. <Your definitions appear here.>
.
#endif /* _MYHEADER_H */
```

This makes certain that the preprocessor defines macros, data types, and other library definitions only once, no matter how many times the input includes the header file.

5.2 Pragmas

Ctalk recognizes GCC, G++, and C99 pragmas.

Pragmas that apply to floating point operations and code generation are ignored and elided, unless the '`--keeppragmas`' command line option is given. See ⟨undefined⟩ [Invoking], page ⟨undefined⟩.

Inclusion of a file that contains G++ pragmas causes the preprocessor to issue a warning if the '`-v`' option is given, and `ctalk` ignores the file. See ⟨undefined⟩ [Invoking], page ⟨undefined⟩.

Here is the effect of the GCC and C99 pragmas.

`#pragma interface`
`#pragma implementation`
: The include file is not processed.

`#pragma GCC dependency` *`file`*
: Issues a warning if the source file is more recent than *file*.

`#pragma GCC poison` *`identifier ...`*
: `ctalk` issues an error and discontinues processing if the source file contains an identifier given as an argument.

`#pragma GCC system header`
: The `ctalk` preprocessor processes all input files in the same manner and ignores this pragma.

`#pragma GCC pack`
`#pragma STDC FP_CONTRACT`
`#pragma STDC FENV_ACCESS`
`#pragma STDC CX_LIMITED_RANGE`
: `ctalk` ignores and elides these pragmas, which apply to floating point and code generation options, unless the '`--keeppragmas`' option is used. See ⟨undefined⟩ [Invoking], page ⟨undefined⟩.

5.3 C Expressions

In version 0.0.66, you can use simple constant expressions as receivers, as in this example.

```
printf ("%s", "/home/users/joe" subString 1, self length - 1);
```

Warning - This use of `self` is experimental in version 0.0.66 and should be used with caution.

You can use a C constant in place of any receiver whose class corresponds to a basic C data type. These classes include `Character`, `String`, `Float`, `Integer`, and `LongInteger`.

Expressions like the following work.

```
if (99 == myInt)
 ...

if ('c' == myInt asCharacter)
 ...

if (3.1416 == pi)
 ...
```

The following code examples are equivalent.

```
myString = "This is a string.";
printf ("%d\n", myString length);
```

and,

```
printf ("%d\n", "This is a string" length);
```

However, if you try to use a C variable on the left side of a method that overloads a C operator, the expression might simply be interpreted as C code, as in this example.

```
String new progName;

progName = "myprog";

if (argv[0] == progName) {
  ...
}
```

This is because Ctalk does not interpret `argv[0]` as a receiver object, and then interprets `==` as a C operator.

5.4 Objects in Function Parameters

Programs cannot, at this time, use objects as parameters to C functions. If you need to use an object as a parameter, you need to use a method instead of a function, or translate the object's value to C. See ⟨undefined⟩ [Translating], page ⟨undefined⟩.

5.5 Objects in Function Arguments

You can use of Ctalk expressions as C function arguments, but the values should be treated as read-only, as in this example.

```
Integer new myIndex;
char buf[255];

myIndex = 0;

/*
 *   This statement works correctly.
 */
sprintf (buf, "%d", __ctalk_to_c_int (myIndex));

/*
 *   This statement does not work correctly.
 */
sscanf (myString, "%s %s", mySubString1, mySubString2);
```

If you need to read a string into objects, try `readFormat` (class `String`) instead.

If you receive an `Unimplemented C type` warning, it means that Ctalk does not implement a class that corresponds to the data type. In these cases, you can generally assign the C variable to an instance of class `Symbol`, and use that as the argument to a function.

The Classes that implement C data types are described in the next section.

5.6 C Data Type Classes

These classes correspond to the basic C types.

`Array` `char **`

`Character`
`char`

`Float` `float` and `double`

`Integer` `int` and `long int`

`LongInteger`
`long long int`

`String` `char *`

5.7 Typedefs in Function Arguments

Ctalk resolves many of the derived types defined in C99, as well as incomplete types; however, variables that are of derived types can still cause unpredictable results, if the variable is of an unusual or complex type.

If you encounter a case where a derived type confuses the parser or run-time library, the workaround is to declare the type as an equivalent C type. For example, if a variable is of type `time_t,` you could equivalently declare it as type `long long int.`

5.8 C Functions in Complex Expressions

You can use C functions in complex expressions within conditionals, as in this example.

```
int int_fn1 (void) {
  return 10;
}

char *add_int_fn_as_string (int a, int b, int c) {
  static char buf[30];
  sprintf (buf, "%d", a + b + c);
  return buf;
}

int main () {

  String new myString1;

  if ((myString1 = add_int_fn_as_string (int_fn1 (), 20, 30)) != "60")
    exit(1);

  printf ("%s\n", myString1);

}
```

As long as your function returns one of the C data types `int, char, char *,` or `double,` Ctalk can translate the function output to an object, as well as call the function at run time using a method call.

If you try to use a C function that returns a complex or derived type, Ctalk prints a warning and uses `Integer` as the default return class. In these cases, you should consider writing a method instead.

Note: When you use functions in complex expressions, the function's arguments must also be C variables or expressions. If you want to use objects as the arguments to a function, then you must perform the object-to-C translation manually.

5.9 Debugging

5.9.1 Object Inspectors

Ctalk provides a basic set of methods that can inspect and print the contents of objects.

The `inspect` method in `Object` class is an interactive utility that lets you examine a program's objects as the program is running.

To inspect an object, simply send it the message, `inspect` - it's a shortcut for the `inspect` method in `ObjectInspector` class, which a program can also call directly.

```
String new globalString;

int main () {
  Integer new i;

  globalString = "global string";

  i inspect;
}
```

In either case, the program stops execution when it reaches the `inspect` method, and presents a prompt where you can type commands.

Here's a transcript of a brief inspector session.

```
$ ./inspect
> p
p
name:       i
class:      Integer (0x48bf4958)
superclass: Magnitude (0x48bf29f0)
value:      (null) (Integer)

> p g globalString
p g globalString
name:       globalString
class:      String (0x48cce8d0)
superclass: Character (0x48c8acc0)
value:      global string (String)

> c
c
$
```

At the inspector prompt, '`> `', the command '`p`' prints the inspector's receiver object, and '`p g globalString`' prints the named global object, `globalString`. The '`c`' command exits the inspector and continues running the program.

There are several commands that the inspector recognizes. Typing '`?,`' '`h,`' or '`help`' at the prompt displays a list of them.

The inspector uses the method `formatObject` to print the contents of individual objects.

If you want to print a formatted object directly, without stopping the program, Ctalk also has the method `dump` in `Object` class, which simply calls `formatObject` with its receiver object to print the object and then returns so the program can continue running.

5.9.2 Using `gdb` for Debugging

The GNU `gdb` debugger allows you to trace through Ctalk applications as well as the compiler and the run-time libraries, at the level of Ctalk's source code.

In order to debug Ctalk programs with `gdb`, the source must be compatible with the debugger; that means that you can debug Ctalk programs using the intermediate C code to get source level debugging within Ctalk apps.

You can also examine the contents of objects and their run-time environment with the `inspect` method (in `Object` class), which doesn't use line number information. See ⟨undefined⟩ [Object_inspect], page ⟨undefined⟩. There's a tutorial on using object inspectors in the *ctalktools* Texinfo manual, and yet more information in the *inspect.3ctalk* manual page.

The '`-P`' command line option disables line numbering. You can then debug the intermediate output, which is normal C that `gdb` can interpret correctly.

This means that method line numbers are calculated from the start of all of the input, which includes all other classes and header files. So when you give the '`-P`' option to Ctalk, it reports the line if possible, although without line number information, the compiler can't track line numbers of preprocessor output; like for example, by adjusting line numbers after including a file with the '`#include`' directive.

The `ctdb` command builds Ctalk programs with the correct arguments for debugging. Then you can use `gdb` to debug the program.

```
$ ctdb -k myprog.c -o myprog
```

If you need to debug either `ctalk` or the `libctalk` library, then you need to build and install Ctalk without optimization. You can do that by adding the '`--without-optimization`' option to `configure` when building Ctalk. Compiler optimization often removes lines of code (and variables) from the binary, so the output often doesn't correspond to the source code. Also, it often helps to add the '`--without-inline-functions`' option to `configure`.

```
$ ./configure --without-inline-functions --without-optimization
```

Then build and install Ctalk with `make` and `make install`.

Ctalk also provides other build options. Typing

```
$ ./configure --help
```

at the shell prompt prints a list of them.

Ctalk is compatible at the machine code level with C programs. That means you use most of `gdb's` functions, like peek into a running program and examine core dumps. The `gdb` documentation describes the debugger's extensive set of options.

5.10 Externs

Ctalk provides a few facilities to help when compiling code in several input modules.

There are also a few caveats when dealing with C variables in multiple input modules, which are described below.

Ctalk allows you to prototype methods. That is, you can declare a method in a source code module before compiling another module later where the method is actually defined.

Prototypes are similar to method declarations, except that the prototype omits the method body. For example, a method prototype before the method is first used would look like this.

```
String instanceMethod myLength (void);
```

You can also define a different return class in the prototype, as in this example.

```
String instanceMethod myLength (void) returnClass Integer;
```

For example, if the input file module1.ca looks like this:

```
String instanceMethod trimStrLength (void) returnObjectClass Integer;

int main () {

  String new myStr;

  myStr = "Hello, world!";

  printf ("%s\n", myStr subString 0, myStr trimStrLength 2);
}
```

and the file module2.ca, which contains the definition of `trimStrLength`, looks like this:

```
String instanceMethod trimStrLength (void) {
  returnObjectClass Integer;
  return self length - 1;
}
```

Then you can build the program with a command line like the following, and Ctalk will have the definition of `trimStringLength` while compiling module1.ca, before it actually compiles the method in module2.ca.

```
$ ctcc module1.ca module2.ca -o myprog
```

C Variables and `extern` Declarations

When using a global C variable in several input modules, you only need to declare it once, before it is first used. Ctalk combines the C code of all of the input files with one copy of the class libraries, so it isn't necessary to declare a C variable in the first module and then declare it as extern in the modules that get compiled later.

5.11 Class casting

In many cases, it's obvious which class an object is, even when the object's defition is removed from the place where a program needs to perform an operation on it, or the object is aliased to `self` or to a C variable, or you need to use a different type of language semantics with an object.

If a program has a set of expressions, as in this hypothetical example:

```
Integer new myInt;
myList new myList;
Key new myKey;
Symbol new *intPtr;

*intPtr = Integer new "Int 1", "1";
myList push *intPtr;
*intPtr = Integer new "Int 2", "2";
myList push *intPtr;

myKey = myList + 1;

myInt = *myKey;

myInt += 3;

... do stuff with myInt ...

myList map {
  printf ("%d\n", self value);
}
```

When run, the program would produce output like this.

```
$ ./myProg
1
2
```

That's because the changes to `myInt` would not take effect for the member of '`myList`', because `Integer` objects, when a program assigns values to them, normally assigns the value of one `Integer` to another. However, in the example above, you might want to work on the original list member - that is, you want the assignment to treat `myInt` as if it were a reference.

One way to notify Ctalk of this is to use an `Object` to refer to the list element, and use a *class cast* to notify Ctalk that the `Object` is actually an `Integer`.

Then the program example above looks like this.

```
Object new myIntObject;
```

```
myList new myList;
Key new myKey;
Symbol new *intPtr;

*intPtr = Integer new "Int 1", "1";
myList push *intPtr;
*intPtr = Integer new "Int 2", "2";
myList push *intPtr;

myKey = myList + 1;

myIntObject = *myKey;

(Integer *)myIntObject += 3;  /* The cast tells Ctalk to treat myIntObject,
                                 which is declared as an Object,
                                 as an Integer, so it can work correctly
                                 with the first element of myList. */

... do stuff with myIntObject ...

myList map {
  printf ("%d\n", self value);
}
```

Other places that you can use class casting is when a program uses a C `OBJECT *`. In that case, you can tell Ctalk what class the object is. Here's an abbreviated example from a `map` method in `TreeNode` class.

```
  OBJECT *t, *list_elem;

  /* rcvr_obj is a TreeNode object. */
  for (t = __LIST_HEAD(rcvr_obj), have_break = NULL;
       t && !have_break; t = t -> next) {
    list_elem = obj_ref_str ((t -> instancevars) ?
     t -> instancevars -> __o_value :
     (IS_VALUE_INSTANCE_VAR(t) ?
      t -> __o_p_obj -> instancevars -> __o_value :
      "0x0"));

  ... do stuff ...

  (TreeNode *)list_elem children __mapChildren methodfn;

  (TreeNode *)list_elem siblings __mapSiblings methodfn;

  }
```

This is a convenient way for a program to tell Ctalk that `list_elem` is a `TreeNode` object. It's up to the program to ensure that the C variable actually does point to an object of that class, or the program won't work correctly at run time.

Programs can also cast `self` to a class, in cases where Ctalk cannot determine `self's` class from its context, like in this example

```
myList map {

  (Float *)self = 0.01f;

}
```

This feature is still experimental, and you should use it with caution; in particular, it's up to the program to insure that the object actually is a member of the class that you cast it to. However, on the occasions when a program needs to excercise some control over a set of expressions' semantics, then class casting can be useful.

5.12 Control Structures

Generally, Ctalk objects work the same as C variables when they appear in `if`, `for`, `while`, `switch`, and `do-while` statements.

If Ctalk cannot figure out a way to resolve an expression that contains both C variables or functions and objects, it will try to warn you.

One exception to these rules are the methods that perform logical negation operator, which generally overload the '`!`' math operator. When you place a '`!`' operator at the beginning of a conditional, Ctalk checks whether the class of the expression's result overloads the operator. In that case, Ctalk treats '`!`' as a method. If a class does not overload the operator, then Ctalk treats it as a C operator.

That way, you can overload '`!`' in classes that define complex objects, which provides a flexible way to determine if an object has been initialized or contains valid data.

For example, in `X11Font` class, you can overload the '`!`' operator to check whether or not an object's `fontId` instance variable is zero to determine whether or not the object refers to a valid font.

If a class doesn't overload '`!`', then Ctalk uses the C semantics for the operator - that is, it simply checks whether an operand is zero (or NULL) or non-zero, and inverts the logical true or false value of the operand.

In addition, Ctalk provides many methods to iterate over collections of objects. These methods include `map`, `mapInstanceVariables`, `mapClassVariables`, and overloaded math operators like those in `Key` class.

For a complete description of the control structures Ctalk uses, refer to the *Ctalk Tutorial*.

`OBJECT` typedef

At the lowest level, ctalk declares objects as pointers to an `OBJECT` struct. You can access an object's members if you assign an object's value to a C variable of the type `OBJECT *`, as in this example.

```
Object new myObject;
OBJECT *myObjectValue;

myObjectValue = myObject value;

if (!strcmp (myObjectValue -> CLASSNAME, "Object"))
   printf ("myObjectValues class is, \"Object.\"\n");
```

The declaration of the OBJECT type is contained in `include/object.h`.

```
typedef struct _object OBJECT;
. . .
struct _object {
  char sig[16];
  char __o_name[MAXLABEL];
  char __o_classname[MAXLABEL];
  OBJECT *__o_class;
  char __o_superclassname[MAXLABEL];
  OBJECT *__o_superclass;
  OBJECT *__o_p_obj;
  VARTAG *__o_vartags;
  char *__o_value;
  METHOD *instance_methods,
    *class_methods;
  int scope;
  int nrefs;
  int attrs
  struct _object *classvars;
  struct _object *instancevars;
  struct _object *next;
  struct _object *prev;
};
```

Note that `__o_name`, `__o_classname`, `__o_superclassname`, and `__o_value` are all `char *`, even if the object belongs to a class like `Integer` or `Float`. The struct members `__o_class` and `__o_superclass` contain pointers to the library class and superclass entries, which are also objects.

For numeric classes, the `value` member contains a formatted representation of a numeric value. Examples of directly assigning values to objects are given in the section about writing methods. See ⟨undefined⟩ [Method API], page ⟨undefined⟩.

Ctalk uses the members `instance_methods, class_methods, classvars` for class objects.

The `sig` member contains a numeric stamp that verifies that the `OBJECT *` refers to a valid object.

The `scope` member describes an object's scope. The scope can be one of `GLOBAL_VAR`, `LOCAL_VAR`, `ARG_VAR`, `RECEIVER_VAR`, `PROTOTYPE_VAR`, or `BLOCK_VAR`.

The `nrefs` member keeps track of the number of references that exist to an object at run time. Every time ctalk creates a reference to an object, internally ctalk increments `nrefs`.

When ctalk deletes a reference, it decrements `nrefs`. When `nrefs` drops to zero, `ctalk` deletes the object.

The `attrs` member is a combination of one or more object attributes. The next section describes object attributes in more detail.

The `tag` member is an abbreviation for the `__o_vartags -> tag -> var_decl -> name` member; that is, the object's primary label.

5.13 Object Attributes

The Ctalk API defines a number of object attributes. The attributes help identify the context that the object appears in. Many of the attributes are only meaningful internally; some are also useful in the method API.

The attributes are defined in `ctalkdefs.h`, which you can include in class libraries. To set an object's attribute, it's generally convenient to use the *__ctalkSetObjectAttr ()* library function, which has the prototype:

```
__ctalkSetObjectAttr (OBJECT *object, int attr)
```

Many attributes are only used by Ctalk internally. The attributes that are useful in methods are defined in `ctalkdefs.h`. Those attributes, and their values and uses, are listed here.

`OBJECT_IS_VALUE_VAR (1 << 0)`
: The object is the `value` instance variable of its parent object.

`OBJECT_VALUE_IS_C_CHAR_PTR_PTR (1 << 1)`
: Used to indicate that an object refers to a `char **` C array.

`OBJECT_IS_NULL_RESULT_OBJECT (1 << 2)`
: Indentifies an object that is created when an operation produces a NULL result.

`OBJECT_HAS_PTR_CX (1 << 3)`
: Set when an object appears on the left-hand side of an equation with a pointer reference; e.g.,

```
*mySymbol = __ctalk_self_internal ().
```

`OBJECT_IS_GLOBAL_COPY (1 << 4)`
: Set when a program copies a global object.

`OBJECT_IS_I_RESULT (1 << 5)`
: Identifies temporary objects that are the result of an operation that sets the object tag's pointer references.

`OBJECT_IS_STRING_LITERAL (1 << 6)`
: Used to identify an object created to represent a string literal.

`OBJECT_IS_MEMBER_OF_PARENT_COLLECTION (1 << 7)`
: Indicates that an object (generally a `Key` object) is a member of a parent collection. Normally used to identify individual collection members.

`OBJECT_HAS_LOCAL_TAG (1 << 8)`
: Indicates that an object's tag was created as a placeholder for an ad-hoc object; for example, objects created by a `basicNew` method. The local tag is not necessarily the primary tag - the object can also acquire other tags when being assigned. Normally this attribute is set by the *__ctalkAddBasicNewTag ()* library function.

`OBJECT_IS_DEREF_RESULT`
: Set by the `Object : ->` method. The attribute is used to indicate that the receiver of `->` is the result of a previous call to `->`; i.e., the expression contains several dereference operators; for example, '`myObject -> instancevars -> __o_value`'.

5.14 C Library Functions

You can assign the result of a C library function to a Ctalk object, provided that the return type of the function has a corresponding Ctalk class.

All C functions must have prototypes. The library functions in this section already have their prototypes defined in the C library headers. If a function in a program does not have a prototype, Ctalk prints an error and exits.

Generally, if Ctalk has a method that is analogous to a C library function, you can use the method with Ctalk objects, although in many cases you can mix objects and C variables. Of course, you can still use any C library function with C data types.

There are some incompatibilities with more specialized libraries. For example, you should take care when using the X Window System Xt widgets in Ctalk programs, because the widget classes use the some of the same class names as the Ctalk library.

The following sections describe the C library functions that Ctalk can use directly.

`abs` Function

```
#include <stdlib.h>

Integer new i;

i = abs (3);
```

`acos` Function

```
#include <math.h>

Integer new i;

i = acos (0.5);
```

Ctalk does not check if the argument is outside the range -1 to 1 and does not check for an error. Refer to the, `acos`, manual page.

`acosh`, `asinh`, and `atanh` Functions

```
#include <math.h>

Float new myFloat;

myFloat = acosh (2.0);
```

Ctalk does not check the range of the arguments or errors. Refer to the, *acosh(3)*, *asinh(3)*, and *atanh(3)* manual pages.

`asctime` Function

```
#include <time.h>

time_t t;
struct tm *time_struct;
String new s;

time (&t);
time_struct = localtime (&t);
s = asctime (time_struct);
```

Note: Ctalk does not have a class corresponding to a `struct tm *`. An explicit assignment of a `struct tm *` to a `Symbol` passes the address to `asctime`. See ⟨undefined⟩ [Objects in Function Arguments], page ⟨undefined⟩.

`asin` Function

```
#include <math.h>

Float new f;

f = asin (0.5);
```

Note that ctalk does not perform any range checking of the argument.

`atexit` Function

```
#include <stdlib.h>

Integer new i;

i = atexit (exitfunc);
```

`atof` Function

```
#include <stdlib.h>
```

```
Float new pi;

pi = atof ("3.1416");
```

`atoi` Function

```
#include <stdlib.h>

Integer new i;

i = atoi ("35");
```

`atol` Function

```
#include <stdlib.h>

Integer new i;

i = atol ("35");
```

Note that, `Integer`, class corresponds to the C types, `int`, and, `long int`.

`atoll` Function

#include <stdlib.h>

LongInteger new i;

i = atoll ("35");

Note: The *atoll(3)* function is not implemented by the OS X or DJGPP C libraries. Use *atol(3)*, *strtoll(3)*, or a method instead.

`calloc` Function

```
#include <stdlib.h>

Integer new n;
Integer new size;
int *intbuf;

n = 10;
size = sizeof (int);

intbuf = calloc (n, size);
```

`cbrt` Function

#include <math.h>

Float new f;

f = cbrt (9.0);

`ceil` Function

#include <math.h>
Float new f;
f = ceil (3.5);

`chdir` Function

Integer new result; String new dirName;
dirName = "mySubdir";
result = chdir (mySubdir);

`clock` Function

```
#include <time.h>

Integer new i;

i = clock ();
```

`copysign` Function

```
#include <math.h>

Float new f;

f = copysign (3.0, -1.0);
```

`cos` Function

```
#include <math.h>

Float new f;

f = cos (45.0);
```

`cosh` Function

```
#include <math.h>

Float new f;

f = cosh (45.0);
```

`ctime` Function

```
#include <time.h>

time_t t;
String new s;

time (&t);
```

```
s = ctime (&t);
```

Note: There is not a portable way to take the address of an object with '`&`', although it may be possible to perform the translation manually in some cases. If the argument must be an object, then use the `cTime` (class `CTime`) method instead.

`difftime` Function

```
#include <time.h>

Float new f;

f = difftime (time1, time0);
```

`erf` Function

```
#include <math.h>

Float new f;

f = erf (0.5);
```

`erfc` Function

```
#include <math.h>

Float new f;

f = erfc (0.5);
```

`exp` Function

```
#include <math.h>

Float new f;

f = exp (2);
```

`expm1` Function

```
#include <math.h>

Float new f;
```

```
f = expm1 (1.05);
```

fabs Function

```
#include <math.h>

Float new f;

f = fabs (-1.05);
```

fclose Function

```
#include <stdio.h>

Integer new i;

i = fclose (fileHandle);
```

fegetround Function

```
#include <fenv.h>

Integer new i;

i = fegetround ();
```

feholdexcept Function

```
#include <fenv.h>

Integer new i;

i = feholdexcept (fe_envp);
```

feof Function

```
#include <stdio.h>

Integer new i;
```

```
i = feof (fileStream);
```

ferror Function

```
#include <stdio.h>

Integer new i;

i = ferror (fileStream);
```

fesetround Function

```
#include <math.h>

Integer new i;

i = fesetround (mode);
```

fetestexcepts Function

```
#include <math.h>

Integer new i;

i = fetestxcept (exceptions);
```

fflush Function

```
#include <stdio.h>

Integer new i;

i = fflush (fileStream);
```

fgetc Function

```
#include <stdio.h>

Integer new i;

i = fgetc (fileStream);
```

fgetpos Function

```
#include <stdio.h>

Integer new i;

i = fgetpos (fileStream, pos);
```

fgets Function

```
#include <stdio.h>

String new s;

s = fgets (s, s length, fileStream);
```

floor Function

```
#include <math.h>

Float new f;

f = floor (3.01);
```

fmod Function

```
#include <math.h>

Float new f;

f = fmod (3.0, 2.0);
```

fopen Function

```
#include <stdio.h>

FILE *f

String new path;
String new mode;
```

```
path = "/home/user/.profile";
mode = "r";

f = fopen (path, mode);
```

`fprintf` Function

See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

`fputc` Function

```
#include <stdio.h>

Integer new myInput;

myInput = fgetc (xstdin ());
fputc (myInput, xstdout ());
```

`fputc` Function

```
#include <stdio.h>

String new myInput;

fgets (myInput, 255, xstdin ());
fputs (myInput, xstdout ());
```

`fread` Function

```
#include <stdio.h>

String new myInput;

myInput = "";

fread (myInput, 255, sizeof (char), xstdin ());
```

`free` Function

Do not use `free` with objects. Use the `__ctalkDeleteObject` library function instead.

Also, calling *__objRefCntZero* before *__ctalkDeleteObject* insures that the object will be completely deleted. You can do this in two ways: first, by sending the object a `delete` message; or by casting the object to a C `OBJECT *` and then giving the `OBJECT *` as an argument to *__objRefCntZero* and *__ctalkDeleteObject*.

```
Object new myObject;
OBJECT *myObject_alias;
```

```
myObject_aliast = myObject;

__objRefCntZero (OBJREF (myObject_alias));
__ctalkDeleteObject (myObject_alias);
```

`freopen` Function

```
#include <stdio.h>

FILE *f;

String new path;
String new mode;

path = "/home/user/.profile";
mode = "r";

f2 = freopen (path, mode, xstdin ());
```

`frexp` Function

```
#include <math.h>

int i_exp_val;
Integer new expInt;
Float new myFloat;
Float new myFraction;

myFloat = "2.5";

myFraction = frexp (myFloat, &i_exp_val);

expInt = i_exp_val;
```

`fscanf` Function

See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

Ctalk wraps `stdin`, `stdout`, and `stderr` in the functions *xstdin*, *xstdout*, and *xstderr*, which gives the streams a constant interface for methods to use regardless of the streams' internal implementation.

`fseek` Function

`fsetpos` Function

```
#include <stdio.h>

FILE *f;
String new path;
String new mode;
```

```
Integer new offset;

path = "/home/user/.profile.new";
mode = "r";


f = fopen (path, mode);

offset = 0L;

fsetpos (f, offset);
```

`fstat` Function

```
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

String new path;
Integer new result;
struct stat statbuf;

path = "/home/user/.profile";

result = fstat (path, &statbuf);
```

`ftell` Function

```
#include <stdio.h>

Integer new filePos;

filePos = ftell (file);
```

`fwrite` Function

```
#include <stdio.h>

FILE *f;
String new path;
String new mode;
String new promptLine;

path = "/home/user/.profile.new";
mode = "a";

f = fopen (path, mode);
```

```
promptLine = "PS1=#";

fwrite (promptLine, promptLine length, sizeof (char), f);
```

getc Function

```
#include <stdio.h>

Integer new myInput;

myInput = getc (xstdin ());

printf ("%c", myInput asCharacter);
```

getchar Function

```
#include <stdio.h>

Integer new myInput;

myInput = getchar ();

printf ("%c", myInput asCharacter);
```

getcwd Function

```
#include <stdio.h>
String new myString;
getcwd (myString, FILENAME_MAX);
```

Note: The argument `myString` must already be initialized to hold the entire directory path. If in doubt, use `getCwd` (class `DirectoryStream`) instead. See ⟨undefined⟩ [DirectoryStream], page ⟨undefined⟩.

getenv Function

```
String new envPath;

envPath = getenv ("PATH");

printf ("%s", envPath);
```

getpid Function

gmtime, localtime Functions

index Function

See ⟨undefined⟩ [strchr], page ⟨undefined⟩.

`isalnum`, `isalpha`, `isascii`, `isblank`, `iscntrl`, `isdigit`, `isgraph`, `islower`, `isprint`, `ispunct`, `isspace`, `isupper`, `isxdigit` Functions

Ctalk does not support `ctype.h` macros portably. Use the corresponding method of class `Character` instead. See ⟨undefined⟩ [Character], page ⟨undefined⟩.

```
Ctype.h Macro          Ctalk Method
-------------          ------------
isalnum                isAlNum
isalpha                isAlpha
isascii                isASCII
isblank                isBlank
iscntrl                isCntrl
isdigit                isDigit
isgraph                isGraph
islower                isLower
isprint                isPrint
ispunct                isPunct
isspace                isSpace
isupper                isUpper
isxdigit               isXDigit
```

`labs` Function

```
#include <stdlib.h>

Integer new myValue;
Integer new myAbsValue;

myAbsValue = labs (myValue);
```

`llabs` Function

```
#include <stdlib.h>

LongInteger new myValue;
LongInteger new myAbsValue;

myAbsValue = llabs (myValue);
```

`lrint`, `lrintf`, `lrintl`, `llrint`, `llrintf`, `llrintl` Functions

```
#include <math.h>

Float new myFloat;
LongInteger new myLongInt;

myFloat = 2.5;

myLongInt = llrint (myFloat);
```

Consult the manual page for *lrint(3)*, etc., for detailed information about each function.

`llround` Function

```
#include <math.h>

Float new myFloat;
LongInteger new myLongInt;

myFloat = 2.5;

myLongInt = llround (myFloat);
```

`log` Function

```
#include <math.h>

Float new myFloat;
Float new myLog;

myFloat = 2.5;

myLog = log (myFloat);
```

`log10` Function

```
#include <math.h>

Float new myFloat;
Float new myLog;

myFloat = 2.5;

myLog = log10 (myFloat);
```

`log1p` Function

```
#include <math.h>

Float new myFloat;
Float new myLog;

myFloat = 2.5;

myLog = log1p (myFloat);
```

`lrint` Function

```
#include <math.h>

Float new myFloat;
Integer new myInt;
```

```
myFloat = 2.5;

myLog = lrint (myFloat);
```

lround Function

```
#include <math.h>

Float new myFloat;
Integer new myInt;

myFloat = 2.5;

myLog = lround (myFloat);
```

malloc Function

```
#include <stdlib.h>

Integer new size;
int *intbuf

size = sizeof (int) * 10;

memblk = (int *)malloc (size);
```

memchr Function

```
#include <string.h>

#define BUFLENGTH 1024

Integer new searchChar;
Integer new length;
char buf[BUFLENGTH], *charptr;

length = BUFLENGTH

strcpy (buf, "Some text.");

searchChar = '.';

charptr = (char *)memchr ((void *)buf, searchChar, length);
```

memcmp Function

```
#include <string.h>

#define BUFLENGTH 1024
```

```
Integer new length;
Integer new result;
char buf1[BUFLENGTH], buf2[BUFLENGTH];

length = BUFLENGTH

strcpy (buf1, "Some text.");
strcpy (buf2, "Some other text.");

result = memcmp ((void *)buf1, (void *)buf2, length);
```

memcpy Function

```
#include <string.h>

#define BUFLENGTH 1024

Integer new length;
Integer new result;
char buf1[BUFLENGTH], buf2[BUFLENGTH], *charptr;

length = BUFLENGTH

strcpy (buf1, "Some text.");

result = (char *)memcpy ((void *)buf1, (void *)buf2, length);
```

memmove Function

```
#include <string.h>

#define BUFLENGTH 1024

Integer new length;
Integer new result;
char buf1[BUFLENGTH], buf2[BUFLENGTH], *charptr;

length = BUFLENGTH

strcpy (buf1, "Some text.");

charptr = (char *)memmove ((void *)buf1, (void *)buf2, length);
```

memset Function

```
#include <string.h>

#define BUFLENGTH 1024

Integer new length;
Integer new fillChar;
char buf[BUFLENGTH], *charptr;

length = BUFLENGTH
fillChar = 0;

charptr = (char *)memset ((void *)buf1, fillChar, length);
```

mkdir Function

```
Integer new r;
String new myDirName;

myDirName = "myDir";

r = mkdir (myDirName);
```

modf Function

```
#include <math.h>

double dptr;
Float new x;
Float new frac;

x = 2.54;

frac = modf (x, &dptr);
```

nearbyint Function

```
#include <math.h>

Float new x;
Float new result;

x = 2.53;

result = nearbyint (x);
```

perror Function

```
#include <stdio.h>
```

```
String new message;

message = "Program error";

perror (message);
```

pow Function

```
#include <math.h>

Float new x;
Float new exp;
Float new result;

x = 2.5;
exp = 2;

result = pow (x, exp);
```

printf Function

```
#include <stdio.h>

String new message;
String new fmt;

printf (fmt, message);
```

raise Function

```
#include <signal.h>

Integer new signal;
Integer new result;

signal = SIGTERM;

result = raise (signal);
```

rand Function

```
#include <stdlib.h>

Integer new random;

random = rand ();
```

realloc Function

```
#include <stdlib.h>
```

```
int *intptr;
Integer new size;

size = sizeof (int *);

intptr = (int *)realloc (NULL, size);
```

remove Function

```
#include <stdio.h>

String new path;
Integer new result;

path = ``/home/user'';

result = remove (path);
```

rename Function

```
#include <stdio.h>

String new oldPath;
String new newPath;
Integer new result;

oldPath = "/home/user";
newPath = "/home/joe";

result = rename (oldPath, newPath);
```

rindex Function

See ⟨undefined⟩ [strrchr], page ⟨undefined⟩.

rint Function

```
#include <math.h>

Float new myFloat;
Float new myIntValue;

myFloat = 2.54;

myIntValue = rint (myFloat);
```

rmdir Function

```
r = rmdir (dirToRemove);
```

round Function

```
#include <math.h>

Float new myFloat;
Float new myIntValue;

myFloat = 2.54;

myIntValue = round (myFloat);
```

scanf Function

See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

sin Function

```
#include <math.h>

Float new x;
Float new sinX;

x = 2.5;

sinX = sin (x);
```

sinh Function

```
#include <math.h>

Float new x;
Float new sinX;

x = 2.5;

sinX = sinh (x);
```

snprintf Function

See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

sprintf Function

See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

sqrt Function

```
#include <math.h>
```

```
Array instanceMethod printSquareRoot (void) {

  Float new squareRoot;
  WriteFileStream classInit;

  /*
   *  Use Ctalk C API library function calls within a C function.
   */
  squareRoot = sqrt(__ctalk_to_c_double(__ctalk_self_internal ()));
  stdoutStream writeStream squareRoot;

  return NULL;
}

int main () {

  Array new floatArray;

  floatArray atPut 0, 1.0;
  floatArray atPut 1, 4.0;
  floatArray atPut 2, 9.0;
  floatArray atPut 3, 16.0;
  floatArray atPut 4, 25.0;

  floatArray map printSquareRoot;

}
```

`srand` Function

```
#include <stdlib.h>

Integer new seed;

seed = 2;

srand (seed);
```

`sscanf` Function

See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

Note: The C99 standard requires that `stdin`, `stdout`, and `stderr` should be implemented as macros, which on some systems (notably Solaris) causes problems with C-to-object translation. If Ctalk cannot register these macros as C variables, then either call *sscanf(3)* C function with only C variables, or use a method with `stdoutStream` or `stderrStream` See ⟨undefined⟩ [WriteFileStream], page ⟨undefined⟩, or `stdinStream` See ⟨undefined⟩ [ReadFileStream], page ⟨undefined⟩.

`strcat`, `strcasecmp`, `strcmp`, and `strcpy` Functions

The *strcat(3)*, *strcasecmp(3)*, *strcmp(3)*, and *strcpy(3)* functions work in most statements. When necessary, Ctalk uses `cStrcat`, `cStrcasecmp`, `cStrcmp`, and `cStrcpy` (class `CFunction`). See ⟨undefined⟩ [CFunction], page ⟨undefined⟩.

`strchr` Function

```
#include <string.h>

int main () {

  String new s;
  String new result;

  s = "s1";

  if ((result = strchr (s, '1')) == "1") {
    printf ("Pass\n");
  } else {
    printf ("Fail\n");
  }

  exit(0);
}
```

`strcoll` Function

```
#include <string.h>

String new s1;
String new s2;
Integer new result;

result = strcoll (s1, s2);
```

`strspn` Function

`strerror` Function

`strftime` Function

`strlen` Function

`strncat`, `strncmp`, and `strncpy` Functions

The *strncat(3)*, *strncasecmp(3)*, *strncmp(3)*, and *strncpy(3)* functions work in most statements. When necessary, Ctalk uses `cStrncat`, `cStrncasecmp`, `cStrncmp`, and `cStrncpy` (class `CFunction`). See ⟨undefined⟩ [CFunction], page ⟨undefined⟩.

`strpbrk` Function

strrchr Function

```
#include <string.h>

String new s1;
Character new searchChar;
String new charptr;

searchChar = '/';
s1 = "/home/user";

charptr = strrchr (s1, searchChar);
```

strstr Function

strtod Function

strtok Function

strtoll Function

strxfrm Function

system Function

```
#include <stdlib.h>

String new commandLine;

commandLine = "ls -lR";

system (commandLine);
```

tan Function

tanh Function

tmpnam Function

tolower Function

The `tolower` function may be implemented in a non-portable manner. Use the `toLower` method instead.

toupper Function

The `toupper` function may be implemented in a non-portable manner. Use the `toUpper` method instead.

trunc Function

`ungetc` Function

`vfprintf` Function

This version of Ctalk does not support the `va_list` data type. You should use *printf(3)* or `writeStream` (class `WriteFileStream`) instead.

`vfscanf` Function

This version of Ctalk does not support the `va_list` data type. You should use *scanf(3)* instead. See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

`vprintf` Function

This version of Ctalk does not support the `va_list` data type. You should use *printf(3)* or `writeStream` (class `WriteFileStream`) instead.

`vscanf` Function

This version of Ctalk does not support the `va_list` data type. You should use *scanf(3)* instead. See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

`vsnprintf` Function

This version of Ctalk does not support the `va_list` data type. You should use *sprintf(3)* instead. See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

`vsprintf` Function

This version of Ctalk does not support the `va_list` data type. You should use *sprintf(3)* instead. See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

`vsscanf` Function

This version of Ctalk does not support the `va_list` data type. You should use *sprintf(3)* instead. See ⟨undefined⟩ [Variable arguments], page ⟨undefined⟩.

5.15 Using GNU Tools with Ctalk

If you want to build Ctalk programs using the GNU configuration tools; i.e, the '`./configure,`' '`make,`' '`make install`' sequence of commands, you need to tell the build tools about Ctalk.

Doing this mostly involves telling the utility `automake,` which helps write `Makefiles` for the `make` program, how to build a Ctalk source file into an executable.

The `make` program allows Makefiles to define rules to build different types of input files into programs, libraries, and other types of data files.

In order to distinguish a Ctalk file, we give it the file extension '`.ca`'. This lets the build tools know that the Ctalk program isn't a standard C input file.

Then, in `Makefile.am` (consult the `automake` manual if you're not certain what this is), you can define a rule to build a '`.ca`' file into an '`.o`' object file.

```
SUFFIXES=.ca .o
```

```
.ca.o:
$(top_builddir)/src/ctalk -I $(top_builddir)/classes $< \
  -o `basename $ .o`.i ; \
$(CC) -c $(AM_CFLAGS) $(AM_CPPFLAGS) $(DEFS) -o $ `basename $< .ca`.i
```

Then, add another line to link the object file into a program.

```
methods$(EXEEXT) : methods.o
$(CC) methods.o $(AM_LDFLAGS) $(LDFLAGS) $(LIBS) -o methods$(EXEEXT)
```

Note that this example comes from the `methods` program in the Ctalk distribution, where, "methods," is the canonical name of the output file, as defined in the '`bin_PROGRAMS`' macro. That allows `make` to install the program normally when you type, '`make install.`'

If you're using Ctalk for another package, you'll almost certainly want to change the paths to something that uses an already-installed Ctalk. In that case, `Makefile.am` might contain lines like these.

```
SUFFIXES=.ca .o

.ca.o:
/usr/local/bin/ctalk -I /usr/local/include/classes $< \
  -o `basename $ .o`.i ; \
$(CC) -c $(AM_CFLAGS) $(AM_CPPFLAGS) $(DEFS) -o $ `basename $< .ca`.i
```

Cleaning Up Extra Files

Note that the `basename` command in these examples handles the translation of the `make` targets into an intermediate Ctalk file.

This way `make` doesn't need to worry about any intermediate files, except that the `Makefile` should clean them up.

So to define rules to clean up the extra files after the build, include `make` targets like these in `Makefile.am`.

```
clean-local:
rm -f *.i

distclean-local:
rm -f *.i
```

Running Ctalk Utilities in an Emacs Window

The `doc/` subdirectory of the Ctalk source code distribution contains the Emacs Lisp programs `classes.el`, `methods-brief.el`, and `methods-full.el`. They define simple Emacs Lisp functions that let you capture the output of the Ctalk utilities in an Emacs window.

The documentation file, `ctalktools.info` contains descriptions of these functions, and the files also contain instructions to install and use them.

6 Copying

Ctalk is free software. You can copy, distribute, and modify Ctalk under the terms of the GNU General Public License, Version 3 (see, `COPYING`, in the Ctalk distribution). You can also distribute executable programs which link to the `ctalk` run time libraries under the terms of the GNU Lesser General Public License, Version 3 (`COPYING.LIB` in the Ctalk distribution).

7 GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you."

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with

modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque."

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For

works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy

of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant

Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements." Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties-for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end

of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications." You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the

documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other

attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright (C) YEAR YOUR NAME.
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License."

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we

```
recommend releasing these examples in parallel under your choice of
free software license, such as the GNU General Public License, to
permit their use in free software.
```

8 Index

(Index is nonexistent)