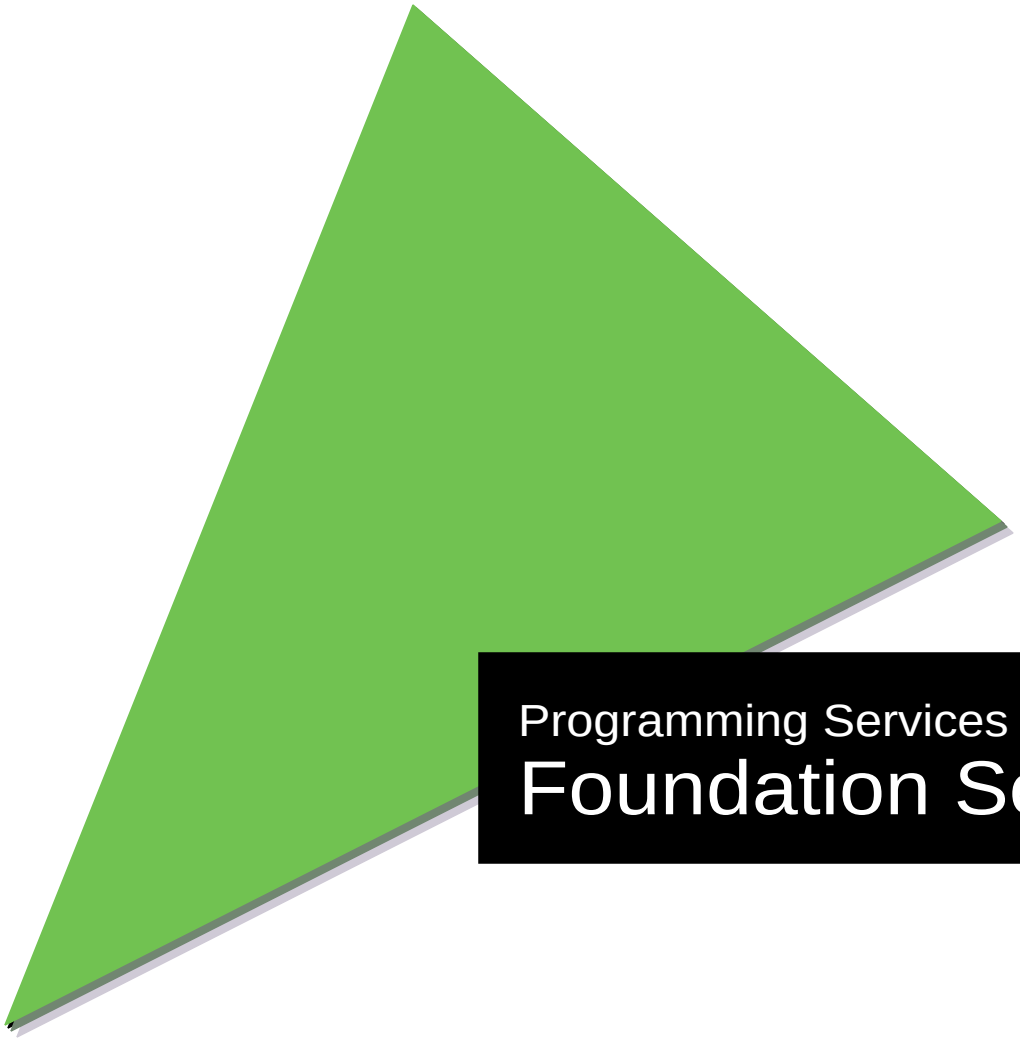


Programming Services Volume I

Foundation Services

AccuSoft Corp.
www.accusoft.com

Chapter 0

Preface

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 0 - Preface

A. About This Book

This volume is intended for the VisiQuest Toolbox programmer who wants to utilize the library routines available in VisiQuest Foundation Services to develop new programs. It is also intended for those who are using VisiQuest as a software integration system, and wish to modify existing code such that it takes full advantage of the capabilities provided by VisiQuest.

Foundation Services is the collective name for the six libraries available in the bootstrap toolbox. The application program developed using Foundation Services will automatically capitalize on the portability derived from the operation system abstraction provided by Foundation Services. Similarly, an application program using Foundation Services will transparently obtain the capability to support the variety of data transport mechanisms handled by Foundation Services. In addition, Foundation Services offers a variety of mathematical functions, a symbolic expression parser, and a wide variety of commonly used functions and utilities.

Foundation Services is one of three major categories into which VisiQuest libraries are divided. Data Services emcompasses the libraries in the dataserv toolbox; these libraries consist of routines for accessing and manipulating data. GUI & Visualization Services, consisting of libraries in the design toolbox, provides routines for visualizing data in the form of images and graphics. Together, Foundation Services, Data Services, and GUI & Visualization Services contain all the libraries that are distributed with VisiQuest. Program Services refers to all three services as a group. Each of three services in Program Services has its own volume in this set. Volume I, this book, deals exclusively with Foundation Services.

B. Assumptions

It is assumed that the reader is well-versed in the C programming language and is familiar with the UNIX operating system. It is assumed that the reader has experience with system programming using the system calls available in libc. Familiarity with routines available in libm is also helpful, but not necessary.

C. Organization

The first chapter gives an introduction to Foundation Services. This introduction includes an overview of Program Services as a whole, in order to establish a context for discussing the different services that are available as part of Foundation Services. Each of the following chapters is devoted to one of the services within Foundation Services. The services included in Foundation Services are:

- Basic Services
- Math Services
- Expression Services
- Operating System Services

D. Conventions

The following conventions are used in this manual.

1. Non-VisiQuest function names, library names, and program names appear in *italics*.
2. VisiQuest function names appear in *courier*.
3. VisiQuest library names appear in *helvetica*.
4. VisiQuest program names appear in helvetica.
5. VisiQuest toolbox names appear in helvetica.
6. Code and file excerpts appear in `courier`.
7. Specific filenames and directory paths appear in `courier`.
8. Command line examples and instructions appear in `courier`.
9. For emphasis, some words appear in *italics*.

E. Related Books

Related books on VisiQuest include:

VisiQuest Installation Guide

Introduction to VisiQuest

Application Toolboxes

Toolbox Programming

Visual Programming

Program Services Volume II, Data Services

Program Services Volume III, GUI & Visualization Services

Table of Contents

This page left intentionally blank

Chapter 1

Introduction

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 1 - Introduction

A. About Foundation Services

Foundation Services is the collective name for the six libraries available in the bootstrap toolbox. The application program developed using these function calls will automatically gain the portability provided by Foundation Services. An application program using Foundation Services will transparently obtain the capability to support a variety of data transport mechanisms, including files, mmap, and shared memory. In addition, Foundation Services offers an assemblage of mathematical functions, a symbolic expression parser, and a wide variety of commonly-used functions and utilities.

VisiQuest libraries are divided into three major categories: Foundation Services, Data Services, and GUI & Visualization Services. *Program Services* refers to all three categories as a group. Foundation Services encompasses several distinct Program Services libraries. Data Services encompasses the libraries in the dataserv toolbox. These libraries consist of routines for accessing and manipulating data. GUI & Visualization Services encompasses the libraries in the design toolbox. These libraries consist of routines for visualizing data in the form of images and graphics. Together, Foundation Services, Data Services, and GUI & Visualization Services contain all the libraries that are distributed within VisiQuest.

The following section provides an overview of Program Services.

B. Overview of Program Services

VisiQuest *Program Services* is a large group of libraries that are layered to provide the software developer with a variety of programming interfaces that trade off reduced complexity against detailed control. Program Services consists of three categories: Foundation Services, Data Services, and GUI Visualization Services. Each Program Services category is comprised of one or more distinct libraries.

While this volume deals exclusively with Foundation Services, an overview of Program Services as a whole follows in order to provide a context for understanding the role of Foundation Services.

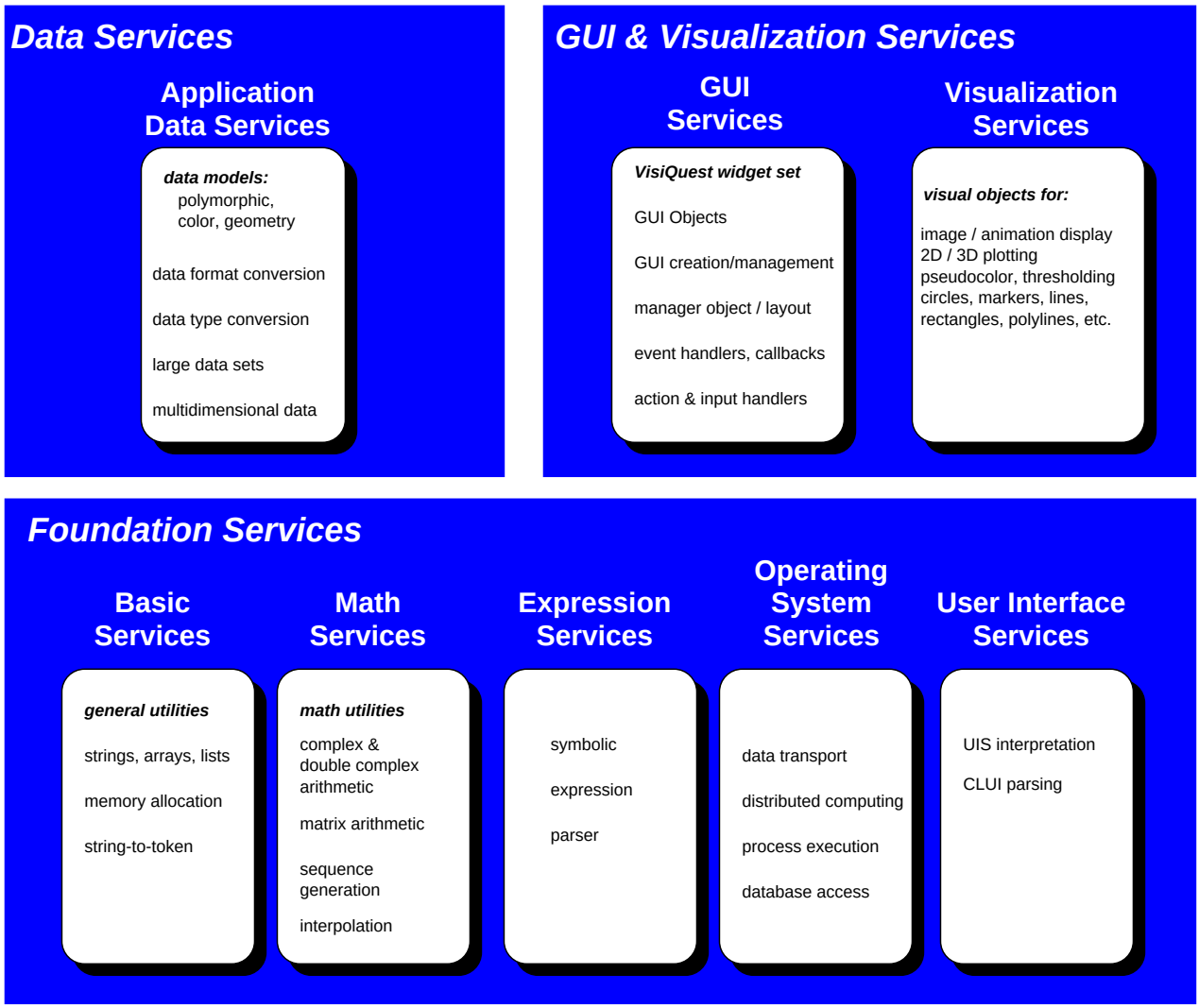


Figure 1: The VisiQuest software infrastructure is composed of three major Program Services categories: Data Services, GUI & Visualization Services and Foundation Services. Each Program Services category is comprised of one or more distinct libraries.

B.1. Foundation Services

Foundation Services encompasses several distinct Program Services. Together, these services fulfill all the requirements of the VisiQuest software infrastructure that do *not* deal with either data processing or data display. Foundation Services provides an extensive Application Programming Interface (API) which ensures portability by introducing a POSIX-compliant layer of abstraction over the operating system. These services also equip the software developer with an extensive range of functionality, ranging from math utilities to distributed computing. Furthermore, use of VisiQuest Foundation Services ensures application capability with the VisiQuest visual programming environment, cantata.

Foundation Services is divided into a number of specific services: Basic Services, Mathematical Services, Operating System Services, Expression Services, and User Interface Services.

Basic Services

Basic Services equips the software developer with a broad range of commonly-used functions and utilities, including a large number of utility functions that are able to handle memory allocation, string manipulation, string parsing, and message reporting.

Mathematical Services

Mathematical Services provides machine-independent implementations of common mathematical operations when not natively available, and offers a variety of useful extensions to the standard mathematical functions. Routines in Mathematical Services are designed to be highly portable and efficient.

Operating System Services

Operating System Services ensures portability by isolating VisiQuest from the operating system. It extends the capabilities of the operating system and allows seamless integration with cantata. It hides the details of data transports such as files, sockets, shared memory, memory mapped, stream, and pipes. It also transparently supports distributed computing. The API is modeled after UNIX function calls and therefore existing applications can be quickly and easily converted to Foundation Services.

Expression Services

Expression Services offers a symbolic expression parser that is used to evalute mathmatical equations and functions. It also supports the definition classes and methods.

User Interface Services

User Interface Services provides low-level support necessary to maintain the standardized Command Line User Interface (CLUI) of VisiQuest 2001 programs. User Interface Services also handles the translation of User Interface Specification (UIS) files into the program CLUI's, and usable data structures to modify GUI and CLUI interfaces at run time.

B.2. Data Services

Data Services comprises a powerful system for accessing and manipulating data. The objective of Data Services is to provide the application programmer with the ability to access and operate on data while remaining independent of the data's file format or its physical characteristics such as size or data type. Data Services is designed to address the needs of a large number of application domains, from image and signal processing to geometry visualization and numerical analysis.

B.3. GUI & Visualization Services

GUI & Visualization Services provides all capabilities related to graphical display using X Windows. It contains all the functions needed to create, interactively change, and maintain a Graphical User Intererface (GUI). GUI & Visualization Services also offers an extensive body of visualization capabilities, including image display and manipulation, colormap control, 2D plotting, 3D plotting, surface rendering, and annotations.

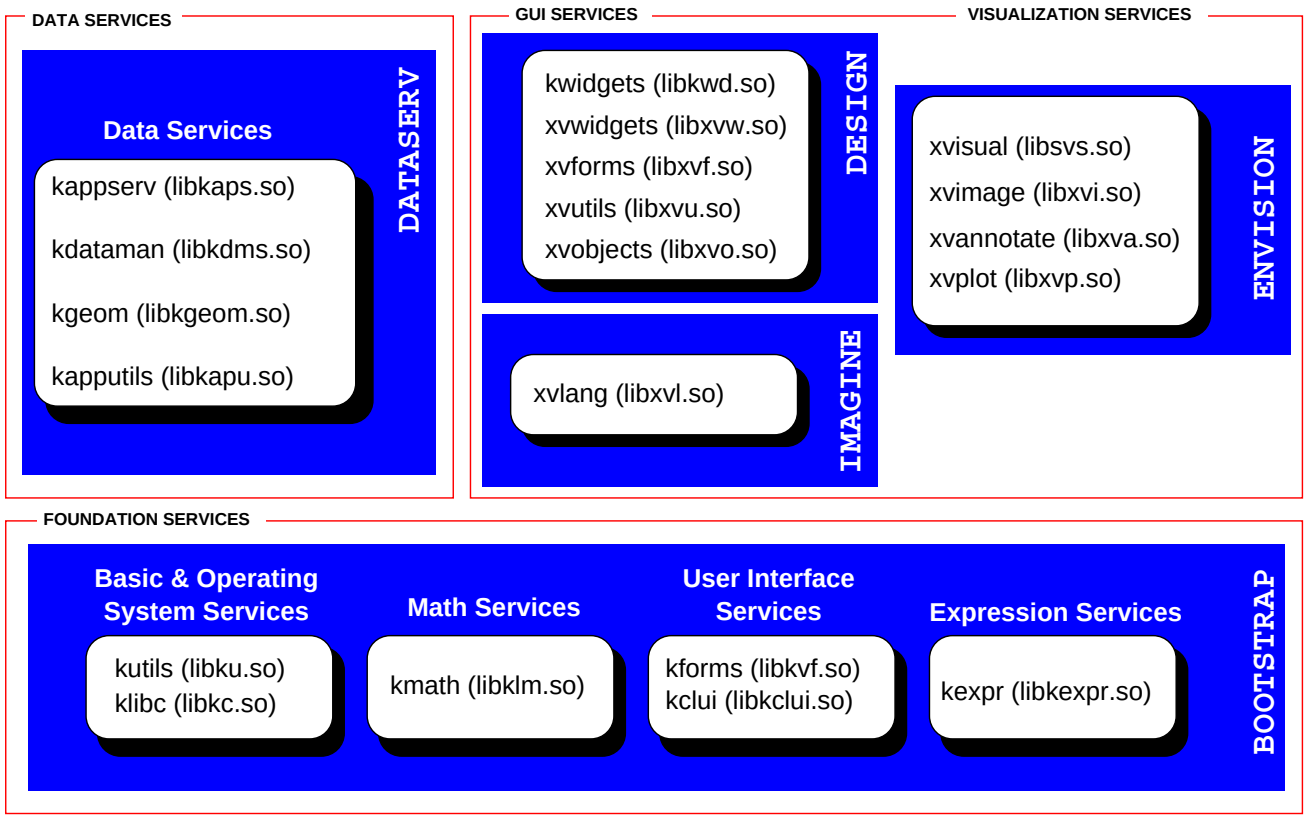


Figure 2: Program Services is comprised of libraries from the various VisiQuest toolboxes, bootstrap, devel, dataserv, design, imagine, and envision. Foundation Services is part of the bootstrap toolbox, and includes the *klibc*, *kutils*, *klibm*, and *kexpr* libraries. Data Services is provided in the dataserv toolbox, and is made up of the *kappserv*, *kapputils*, *kdataccess*, *kdataman*, *kjpg*, *kdatafmt*, and *kgeom* libraries. GUI and Visualization Services is in the design, imagine, and envision toolboxes, and includes the *kwidgets*, *xvforms*, *xvutils*, *xvwidgets*, *xvobjects*, *xvannotate*, *xvgraphics*, *xvimage*, *xvplot*, *xvisual*, *klang*, and *xvlang* libraries.

C. Volume Overview

Program Services is comprised of *libraries* from the three required VisiQuest toolboxes: bootstrap, design and dataserv. The bootstrap toolbox contains all of the libraries that make up Foundation Services. The *klibc* and *kutils* library contains some routines that are part of Basic Services and others that are part of Operating System Services. The *klibm* library contains all the routines included in Mathematical Services. The *kexpr* library implements the symbolic expression parser and the associated utilities which form

Expression Services. In addition, the *kforms* and *kclui* libraries together comprise User Interface Services. User Interface Services is *not* documented in this book. Unlike the other services of Foundation Services, User Interface Services does not contain library routines that have a public Application Programming Interface (API). User Interface Services library routines are subject to change without notice between release, and are used by other routines to manipulate CLUI and GUI's at runtime, and for code generation. They are reserved for internal use by other VisiQuest libraries and are not documented in this or any other volume.

The data structures used by the Foundation Services routines are defined in the include directory of the bootstrap toolbox. The function descriptions contained in this manual can be supplemented with UNIX man pages or with a C Library Reference Manual for further understanding.

A *system call* is a call to a function which requires the operating system to perform a particular task for the application, such as writing to a file (which requires the operating system to write a buffer to disk). Such functions are typically included in libraries which are distributed with the operating system, such as *libc*. In order to provide a layer of abstraction over the operating system and thus ensure portability across a wide variety of computer architectures, Foundation Services provides system calls to replace most of the system calls that are typically used in an application.

A *library call* is a call to a function provided in a library. It does not require the operating system to perform the task, such as the *add* function (which sums two numbers without need for operating system intervention). A number of such functions are provided by libraries which are distributed with the operating system along with the system libraries. An example of such is the *libm* library which provides a number of mathematical functions for use by the application. In order to provide better error checking in addition to portability (some functions that are provided with some operating systems may not be provided with other operating systems), Foundation Services offers library calls to replace a number of the library calls that are typically used in a program.

VisiQuest replacements for both system calls and library calls always begin with a *k* followed by the function name. Thus, the system call to the *libc* function *read()* would become a system call to the VisiQuest function `kread()` when programming with Foundation Services.

The remaining chapters of this volume have been organized to detail the services that comprise Foundation Services.

Chapter 2, Basic Services

This chapter details a number of VisiQuest library calls that are meant to replace *libc* library calls. The VisiQuest counterparts to the *libc* library calls in question have been enhanced with better error checking, more robustness and, in some cases, more flexibility. As well as providing substitutes for a number of the *libc* calls, Basic Services also provides a number of utility functions that go beyond the capabilities of *libc*. Basic Services contains functions that perform string processing and tokenization of strings, retrieve system information and handle message reporting, string parsing, memory allocation, file and directory path manipulation, and array and linked list creation and maintenance.

Basic Services also includes routines with which to use the VisiQuest Database Management System. The VisiQuest database system is a simple key/value system similar in API to the dbm API found on most UNIX systems. It should be noted that the kdbm calls, unlike the UNIX dbm calls, are manipulated completely in memory and are only stored on disk when `kdbm_close` is called on the open database pointer.

Chapter 3, Mathematical Services

This chapter covers the routines that control floating-point processing and compute common mathematical functions. The contents of the math libraries that are distributed with various operating systems, such as *libm*, tend to vary with the architecture. Mathematical Services routines, in contrast, have been designed to compensate for the fact that some math routines may be missing or renamed on some architectures. Thus, use of the architecture-independent Mathematical Services routines will ensure portability of an application. Note that in most cases, the k* math calls are the same as the system math call. This is done so that user code does not lose any hardware specific speedups gained by using the vendor math call. So, using the k* call, insures portability, without loss of efficency.

In addition, useful extensions have been made to the standard body of math functions so that Mathematical Services offers a greater range of mathematical utilities than are provided with standard math libraries such as *libm*. Mathematical Services provides single- and double-precision complex arithmetic functions, as well as a set of matrix algebra routines. Data-type conversion, and scaling and normalization are also furnished with Math Services. Functions to generate random numbers and sequences are also provided.

Chapter 4, Expression Services

This chapter explains the use of the VisiQuest symbolic expression parser. This expression parser processes mathematical expressions and evaluates the results. The expression parser is used in cantata for variable parameters and control operators. It is also used in xprism to support interactive evaluation and plotting of 2D and 3D functions.

The expression parser supports a large variety of data types: byte, int, unsigned int, long, unsigned long, double, unsigned double, complex, double complex, and string. New types can also be constructed. Mathematical expressions that use functions supported by Mathematical Services can also be evaluated.

Chapter 5, Operating System Services

This chapter details a number of VisiQuest system calls that are meant to replace *libc* system calls. The VisiQuest counterparts to the *libc* system calls have been expanded to support a variety of data transport mechanisms, including files, sockets, and shared memory, thus providing the transparent capability to support each of these methods of transporting data between processes.

Each chapter begins with a table of contents. In each chapter, routine descriptions are divided into task groupings. For example, in Chapter 2, Basic Services there are sections on string processing, tokenized strings, memory allocation, and so on. Each section begins with an introduction that explains concepts needed to understand and use the functions. The introduction is followed by an alphabetical list of all the functions detailed in the section. Then, each function is detailed with a reference entry that provides a structured guide to the purpose, syntax, and usage of the function. The reference entries for the functions use a standardized template that includes the function name, synopsis, input arguments, output arguments, return values, description, side effects, and restrictions. If an element of the reference entry does not apply, it is left out. For example, if a function has no output arguments, the "Output Arguments" field of the reference entry is omitted. The following template depicts the layout of reference entries for functions:

C.1. function_name() — *short function description*

Synopsis

```
return_data_type function_name(

   data_type input_1,
   data_type input_2,
   data_type output_1,
   data_type output_2)
```

Input Arguments

`input_1`
: input argument 1 description

`input_2`
: input argument 2 description

Output Arguments

`output_1`
: output argument 1 description

`output_2`
: output argument 2 description

Returns

what the function returns

Description

Detailed description of the function

Side Effects

side effects of the function (if any)

Restrictions

restrictions of the function (if any)

Table of Contents

This page left intentionally blank

Chapter 2

Basic Services

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 2 - Basic Services

A. Introduction

Basic Services offers a broad range of commonly-used functions and utilities. All Basic Services routines are located in the *kutils (libku.a)* and *klibc (libku.a)* libraries of the bootstrap toolbox.

The string utilities provided are more robust, flexible versions of those provided in *libc*. Increased error checking allows the string utilities to handle NULL pointers appropriately rather than causing a core dump in the calling program. There are a number of expanded convenience routines as well; these provide functionality that goes beyond the offerings of *libc*, such as the concatenation of three strings, automatic replacement of substrings within a string, deletion of leading and ending whitespace, and case-insensitive string comparisons.

The basic string utilities are augmented by tokenized string utilities; these allow you to greatly increase efficiency of code that involves string manipulation by associating each unique string with a unique *token*. At any time, a string may be translated into its corresponding token and vice versa. Thus, expensive searches involving string comparisons are easily changed to more efficient searches using integer comparisons. The result is a significant increase in speed.

Basic Services also offers a standardized message reporting system. These utilities allow you to use predefined, standard methods of reporting errors, warnings, and general information. Prompting for specific information as well as the special case for prompting before a file over-write are also supported. These routines work consistently with the context of the program from which they are called. If they are called from an *xvroutine* (or from a *kroutine* accessed from the visual programming language cantata), they will be displayed appropriately in a graphical pop-up window format. In contrast, if they are called by a kroutine run from the command line, they will be displayed as text in the terminal.

A set of program information utilities is provided in Basic Services. These utilities allow you to obtain general information about the program that is running, such as the number of command line arguments (argc), the command line argument list (argv), the program name, the name of the toolbox in which the program exists, the environment variable list, and so on. These utilities are most often used by libraries to obtain information about the calling program rather than the programs themselves, which have most of this information readily available in the usual form.

The string parser offered by Basic Services is an integral part of the VisiQuest software development system. In addition to fulfilling general string parsing needs of various VisiQuest programs, the string parser makes it possible for library routine source code headers to be translated into *man pages* and for the code generators to modify files containing *tags*.

The memory allocation utilities standardize the result of zero byte allocation requests and will always return NULL for such requests. Depending on operating system implementation, their *libc* counterparts will sometimes return NULL and other times allocate a single byte. Thus, in order to preserve portability, the memory allocation utilities included in Basic Services should *always* be used in place of their *libc* counterparts.

Another group of features offered by Basic Services involves file path and directory path manipulation. Routines to expand a path, expand a filename, strip a filename from a file path, strip the directory from a file path, create a unique name for a file in TMPDIR, create a new directory, remove a directory, and so on are provided. Environment variables may also be obtained, set, and removed.

The array creation and manipulation routines provided by Basic Services provide a comprehensive set of operations on arrays. These routines support all standard VisiQuest data types, such as KINT, KFLOAT, KDOUBLE, KSTRING, KSTRUCT, etc. A comprehensive set of linked list creation and manipulation routines are also offered by Basic Services.

Using the routines and features offered in Basic Services ensures that new programs will maintain consistency with the entirety of VisiQuest.

Every section begins with a list of the utilities available, followed by an in-depth detailing of each utility.

ALL programs that utilize these routines MUST include the statement:

```
#include "bootstrap.h"
```

Note that this include file is usually automatically included by the public include file that is named after your toolbox. So it is almost never necessary to directly add the above *#include* to your code.

B. String Utilities

The following section details the string utilities that are a part of the *klibc* library.

B.1. Introduction to String Utilities

There is a large collection of utilities for string creation, comparison, modification, and inquiry. The VisiQuest basic string utilities are more robust than those available in *libc*. These string utilities check for NULL. Routines beginning with *kstring_* allocate strings and store the results in a string that you provide.

- *kstrcasecmp()* - do a case insensitive string compare
- *kstrcat()* - concatenate two strings
- *kstrchr()* - find a character in a string
- *kstrcmp()* - compare two strings
- *kstrcpy()* - copy a string
- *kstrcspn()* - return the number of characters not matched
- *kstrdup()* - return a duplicate of the input string
- *kstrlen()* - return the length of a string
- *kstrncasecmp()* - do a case insensitive string compare on n characters
- *kstrncat()* - concatenate up to n characters on a string
- *kstrncmp()* - compare the first n characters of two strings
- *kstrncpy()* - copy the first n characters in a string
- *kstrpbrk()* - find the first occurrence of a character from a set of chararacters
- *kstrrchr()* - reverse scan a string to find a character
- *kstrspn()* - return the number of matched characters

• *kstrstr()* - find a substring within a string
• *kstrtok()* - find a token within a string
• *kchar_replace()* - replace a character with another through a string
• *kstring_capitalize()* - convert a string to its capitalized equivalent
• *kstring_3cat()* - concatenate three strings together
• *kstring_cat()* - concatenate two strings
• *kstring_cleanup()* - remove white space from the ends of a string
• *kstring_copy()* - copy a string
• *kstring_detab()* - remove tabs from a string
• *kstring_lower()* - convert a string to lower case.
• *kstring_ncat()* - concatenate two partial strings
• *kstring_ncopy()* - copy up to n characters of a string
• *kstring_replace()* - replace one substring with another
• *kstring_subcmp()* - compares two sub-strings
• *kstring_upper()* - convert a string to upper case.
• *kstring_seddata()* - perform text changes with one or more sets of substitution rules

B.2. Definitions of String Utilities

B.2.1. kstrcasecmp() — *do a case insensitive string compare*

Synopsis

```
int kstrcasecmp(
   const char *istr1,
   const char *istr2)
```

Input Arguments

`istr1`
The first input string to compare.

`istr2`
The second input string to compare.

Returns

If the two strings are identical, ignoring case differences, the value of 0 is returned. If the two strings differ, the difference of ASCII values between the first character in each string that differs will be returned. If the ASCII value of the differing character in 'istr1' is greater than the one in 'istr2', then the return value is positive, and implies that 'istr1' is greater than 'istr2'. If the difference between the two ASCII values is negative, the return value implies that 'istr2' is greater than 'istr1'.

Description

This function does a case insensitive comparison of two strings. It does a character by character comparison of both input strings, ignoring the case of alphabetic characters, until the current character for one string is not equal the current character in the other string. This continues until the end of one or both of the strings is reached. It protects against NULL on the input strings by replacing NULL

pointers with a reference to an empty string.

Examples

The following example takes two strings that are equal if you ignore case, and passes them to string compare.

```
char *s1 = "John Doe";
char *s2 = "john DOE";
int   result;

result = kstrcasecmp(s1, s2);
kprintf("The result is '%d'\n", result);
```

The output that will be printed is 0.

B.2.2. kstrcat() — *concatenate two strings*

Synopsis

```
char *kstrcat(
   char *ostr,
   const char *istr)
```

Input Arguments

`ostr`
: The first string to concatenate.

`istr`
: The second string to concatenate.

Output Arguments

`ostr`
: The resulting combined output string.

Returns

A pointer to 'ostr' after all the characters in 'istr' are appended onto 'ostr'. It will return NULL if 'ostr' is NULL. If 'istr' is NULL, 'ostr' is returned unchanged.

Description

This function concatenates one string to another. kstrcat appends all the characters from 'istr' to the end of 'ostr', terminating the resulting string with a null character.

Note that the calling routine must make sure that 'ostr' points to a memory buffer large enough to hold the concatenated string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Examples

The following example contenates two strings 's1' and 's2', and to generate a well known sentence.

```
char  s1[KLENGTH] = { 'H', 'e', 'l', 'l', 'o' };
char *s2 = " World.\n";
char *result;

result = kstrcat(s1, s2);
kprintf("%s", result);
```

The output is a single line saying 'Hello World', and the pointer 'result' is pointing to the 's1' array.

B.2.3. kstrchr() — *find a character in a string*

Synopsis

```
char * kstrchr(
   const char *istr,
   int character)
```

Input Arguments

`istr`
The string to scan.

`character`
The character to look for in 'istr'.

Returns

A pointer to the address in the input string where the 'character' character is located. NULL is returned if 'character' is not in the input string, or if the input string is NULL.

Description

This function finds the first occurence of a specified character in a given string. This routine is similar to the system strchr() call. kstrchr() searches the input string, 'istr', for the first occurrence of a character, the search, and can be the specified character to search for.

Examples

The following example takes a sentence, and uses kstrchr() to shorten it for use in a different sentence.

```
char *s1 = "Hello World.";
char *result;

result = kstrchr(s1, 'W');
kprintf("Goodbye Cruel %s\n", result);
```

The output string is 'Goodbye Cruel World.'

B.2.4. kstrcmp() — *compare two strings*

Synopsis

```
int kstrcmp(
   const char *istr1,
   const char *istr2)
```

Input Arguments

`istr1`
A character pointer to the first string.

`istr2`
A character pointer to the second string.

Returns

If the two strings are identical, the value of 0 is returned. If the two strings differ, the ASCII value difference of the first character that differs in the two strings will be returned. If the ASCII value of the differing character in istr1 is greater than the one in istr2, then the return value is positive, and implies that istr1 is greater than istr2. If the difference between the two ASCII values is negative, the return value implies that istr2 is greater than istr1.

Description

This function does a comparison of two strings. It is a replacement for the system strcmp(). It does a character by character case sensitive comparison of both input strings until the current character for one string does not equal the current character of the other string, or until the end of the input strings are reached. It protects against NULL on the input strings by replacing NULL pointers with a reference to an empty string.

Examples

The following example repeats the kstrcasecmp() example.

```
char *s1 = "John Doe";
char *s2 = "john DOE";
int   result;

result = kstrcmp(s1, s2);
kprintf("The result is '%d'\n", result);
```

The result of this compare is -32 instead of 0, since that is the result of the difference 'J' - 'j'.

B.2.5. kstrcpy() — *copy a string*

Synopsis

```
char *kstrcpy(
   char *ostr,
   const char *istr)
```

Input Arguments

`istr`

The string to copy from.

Output Arguments

`ostr`

The string to copy into.

Returns

A pointer to the copied string 'ostr' after 'istr' is copied to 'ostr'. If 'istr' is NULL, 'ostr' is returned without modification. If 'ostr' is NULL, then NULL is returned.

Description

This function copies one string to another. This function is similar to system call strcpy(). kstrcpy() is used used to copy each character in the input string to the output string. The terminating null character of the input string is copied so that the input and output strings are exact copies. Thus, 'ostr' is overwritten.

Note that the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the copied string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Examples

The following example copies strings 's1' to 's2'.

```
char  s2[KLENGTH];
char *s1 = "Hello World.\n";
char *result;

result = kstrcpy(s2, s1);
kprintf("%s", result);
```

The output is a single line saying 'Hello World', and the pointer 'result' is pointing to the 's2' array.

B.2.6. kstrcspn() — *return the number of characters not matched*

Synopsis

```
size_t kstrcspn(
   const char *istr,
   const char *charset)
```

Input Arguments

`charset`

The set of characters to use for counting.

Returns

The number of characters not matched in the input string, from the 'charset', or 0 if the input string or character set string are NULL.

Description

This function returns the length of a portion of the input string not matching any of the characters in specifiec set of characters. It is similiar to the system routine strcspn().

It counts the number of characters at the start of the input string that consist entirely of characters not in list of characters specified by 'charset'. The count stops at the first character in 'string' that is in 'charset'.

Examples

The following example uses kstrcspn() to determine the number of non-vowels that appear before the first vowel.

```
char *str = "    This is my input string";
int non_vowels = 0;

non_vowels = kstrcspn(str, "aeiou");
kprintf("The number of non_vowels = %d\n", non_vowels);
```

The result, stored in 'non_vowels', will be 6.

B.2.7. kstrdup() — *return a duplicate of the input string*

Synopsis

```
char *kstrdup(
   const char *istr)
```

Input Arguments

`istr`

The string to be duplicated.

Returns

A pointer to the duplicated string. This routine will return NULL on a memory allocation error, or if the input string is NULL.

Description

This function allocates memory and copies the input string into that space. This routine is similiar to the system routine strdup. It uses kdupalloc() to create a memory buffer large enough to hold the string passed into this routine and copyies the input string to the buffer. This buffer is then returned to the user.

Examples

This example initialized the a pointer to be a memory buffer with a string in it. This buffer can then be modified by the rest of the routine. Since memory is allocated, a call to kfree_and_NULL() will free the buffer when the buffer is no longer needed.

```
char *str = kstrdup("this sentence needs a capital t at the "
                    "start.");
if (str == NULL)
{
   kerror(NULL, NULL, "Cannot malloc space for the string");
   return;
}
str[0] = 'T';
kprintf("%s\n", str);
```

The resulting print will be 'This sentence needs a capital t at the start.'

B.2.8. kstrlen() — *return the length of a string*

Synopsis

```
size_t kstrlen(
   const char *istr)
```

Input Arguments

`istr`

The string to get the length of.

Returns

The number of characters in the string. On a NULL input string, a value of 0 is returned.

Description

This routine finds the length of a string. This routine is similiar to the system routine strlen(). It counts the number of characters in a string until it reaches a null ('\0') character. This routine is better than many standard unix strlen() routines, because it treats a NULL pointer as an empty string, which returns a length of zero.

Examples

This is a very simple example that prints the length of the input string.

```
char *str = "1234";
kprintf("The length is '%d'\n", kstrlen(str));
```

The resulting length is 4, because the null terminating character is not counted.

B.2.9. kstrncasecmp() — *do a case insensitive string compare on n characters*

Synopsis

```
int kstrncasecmp(
   const char *istr1,
   const char *istr2,
   size_t num)
```

Input Arguments

`istr1`

The first string to compare.

`istr2`

The second string to compare.

`num`

The number of bytes to compare.

Returns

If the two strings are identical up to the specified byte, or end of string the value of 0 is returned. If the

two strings differ, the ASCII value difference of the first character that differs in the two strings will be returned. If the ASCII value of the differing character in 'istr1' is greater than the one in 'istr2', then the return value is positive, and implies that 'istr1' is greater than 'istr2'. If the difference between the two ASCII values is negative, the return value implies that 'istr2' is greater than 'istr1'.

Description

This routine does a case insensitive comparison of two strings for a specified number of bytes. This routine is a replacement for the system strncasecmp(). It does a character by character case insensitive comparison of both input strings until the current character for one string does not equal the current character of the other string, until the end of the input strings are reached, or until num characters have been compared. It protects against NULL on the input strings by replacing NULL pointers with a reference to an empty string.

Examples

This example uses strcasecmp twice on the same strings and different 'num' values.

```
char *s1 = "SAME-DIFFERENT";
char *s2 = "Same-Really Different";
int same, different;

same = kstrncasecmp(s1, s2, 5);
different = kstrncasecmp(s1, s2, 10);
printf("same is %d\ndifferent is %d\n", same, different);
```

the resulting output will be:

same is 0
different is -14

because up to the fifth character they are the same, and on the sixth character the returned result will be 'D' - 'R'.

B.2.10. kstrncat() — *concatenate up to n characters on a string*

Synopsis

```
char *kstrncat(
   char *ostr,
   const char *istr,
   size_t num)
```

Input Arguments

`ostr`
: The base string to concatenate.

`istr`
: The string to concatenate to the base.

`num`
: The number of characters in 'istr' to concat onto 'ostr'. If the specified number is larger than the length of istr, then this function stops at the null character. If 'num' is less than or equal to zero, 'ostr' is left unchanged.

Output Arguments

`ostr`
: The resulting combined output string.

Returns

A pointer to the concatenated string 'ostr' after 'num' characters of 'istr' are appended onto the end of 'ostr'. It will return NULL, if 'ostr' is NULL. If 'istr' is NULL or num is less than or equal to zero, 'ostr' is returned unchanged.

Description

This function concatenates a specified number of characters one string to another. This function is similar to system call strncat(). kstrncat() appends upto 'num' characters from 'istr' to the end of 'ostr'.

Note that the calling routine must make sure that 'ostr' points to a memory buffer large enough to hold the concatenated string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Examples

This example partially combines two strings into a familiar saying.

```
char *s1 = "I'm sorry Dave. ";
char *s2 = "I'm afraid I can't do that.  It is too "
           "complicated.";
char result[KLENGTH];

kstrcpy(result, s1);
```

```
s1 = kstrncat(result, s2, 27);
kprintf("%s\n", s1);
```

The output is 'I'm sorry Dave. I'm afraid I can't do that.'

B.2.11. kstrncmp() — *compare the first n characters of two strings*

Synopsis

```
int kstrncmp(
   const char *istr1,
   const char *istr2,
   size_t num)
```

Input Arguments

`istr1`
: A character pointer to the first string.

`istr2`
: A character pointer to the second string.

`num`
: The number of characters to compare.

Returns

If the two strings are identical or 'num' is less than or equal to zero, the value of 0 is returned. If the two strings differ, the ASCII value difference of the first character that differs in the two strings will be returned. If the ASCII value of the differing character in 'istr1' is greater than the one in 'istr2', then the return value is positive, and implies that 'istr1' is greater than 'istr2'. If the difference between the two ASCII values is negative, the return value implies that 'istr2' is greater than 'istr1'.

Description

This routine is a replacement for the system strcasecmp. It does a character by character case sensitive comparison of both input strings until the current character for one string does not equal the current character of the other string, until the end of the input strings are reached, or until num characters have been compared. It protects against NULL on the input strings by replacing NULL pointers with a reference to an empty string.

Examples

This example is similar to the kstrncasecmp() example. It calls kstrncmp twice on the same two strings with different values of 'num'.

```
char *s1 = "Same-DIFFERENT";
char *s2 = "Same-different";

kprintf("Same is %d\n", kstrncmp(s1, s2, 5));
```

```
kprintf("Different is %d\n", kstrncmp(s1, s2, 10));
```

The output will be:

Same is 0
Different is -32

B.2.12. kstrncpy() — *copy the first n characters in a string*

Synopsis

```
char *kstrncpy(
   char *ostr,
   const char *istr,
   size_t num)
```

Input Arguments

`istr`
The string to copy from.

`num`
The number of characters to copy.

Output Arguments

`ostr`
The string to copy into.

Returns

A pointer to 'ostr' after up to num 'istr' is copied to 'ostr'. If 'ostr' is NULL, then NULL is returned. If 'istr' is NULL or num is less than or equal to zero, then 'ostr' is returned unchanged.

Description

This function copies a specified number of characters from one string to another. This function is similar to system call strncpy(). kstrncopy() copies up to num characters in the input string to the output string.

Note that the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the copied string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

B.2.13. kstrpbrk() — *find the first occurrence of a character from a set of chararacters*

Synopsis

```
char *kstrpbrk(
   const char *istr,
   const char *sub_str)
```

Input Arguments

`istr`
The string to search for a character.

`sub_str`
The string holding the set of characters to look for.

Returns

If one of the characters in 'sub_str' is found in 'istr', a pointer indicating the address of the character it found in 'istr' is returned. If no characters in the 'sub_str' string appear in 'istr', a NULL is returned.

This routine also protects against NULL on both the input and substring variables. If either is NULL, the value of NULL is returned.

Description

This function locates the position of the first occurrence in the input string of any character from a specified set of characters. This routine is a replacement for the system library routine strpbrk() with some extra protection against NULL input strings. It searches the input string, 'istr', for the first occurrence of any character in a set of characters, specified by the 'sub_str' string.

B.2.14. kstrrchr() — *reverse scan a string to find a character*

Synopsis

```
char *kstrrchr(
   const char *istr,
   int character)
```

Input Arguments

`istr`
The input string to scan.

`character`
The character to search for.

Returns

A pointer to the address in the input string, 'istr', where the last occurrence of the 'character' character

is. NULL is returned if 'character' is not in the input string, or if a NULL input string is provided.

Description

This function finds the last occurrence of a specified character in a string and it is similar to the system call strrchr(). kstsrrchr() searches the input string from back to front for a character specified by 'character' in the input string.

B.2.15. kstrspn() — *return the number of matched characters*

Synopsis

```
size_t kstrspn(
   const char *istr,
   const char *charset)
```

Input Arguments

`istr`

The string to be scanned.

`charset`

The string used to determin the character set to use.

Returns

The number of characters matched in the input string, from the 'charset'. If every character in the input string is a member of set of characters defined by 'charset', the length of 'istr' is returned. If either 'istr' or 'charset' are NULL, a value of 0 is returned.

Description

This function returns the length of the input string that consists completely of a specified list of characters. This routine is similiar to the system routine strspn(). It counts the number of characters at the start of the input string that consist entirely of characters from the 'charset' string. The count stops at the first character in the input string that is not in the charset string.

B.2.16. kstrstr() — *find a substring within a string*

Synopsis

```
char *kstrstr(
   const char *istr,
   const char *sub_str)
```

Input Arguments

`istr`
: The string to search.

`sub_str`
: The sub string to look for.

Returns

If the sub_str is found in istr, a pointer indicating the address of the first character of the substring within the istr string is returned. If the substring is not a part of the input string, a NULL is returned.

This routine also protects against NULL on both the input and substring variables. If either is NULL, the value of NULL is returned.

Description

This function locates the first occurrence of one string in another. It is a replacement for the system library routine strstr(). This routine scans the input string, 'istr', for the first occurrence of a substring specified by 'sub_str'.

B.2.17. kstrtok() — *find a token within a string*

Synopsis

```
char *kstrtok(
   char *istr,
   const char *sub_str)
```

Input Arguments

`istr`
: The string to search find tokens in.

`sub_str`
: The string containing the set of character tokens.

Returns

A pointer to the first token in 'istr'. Otherwise, it returns NULL on error, or if the token string 'sub_str' is NULL.

Description

This function gets the next token, or substring, from the input string using a set of characters as delimiters. This routine is a replacement for the system library routine strtok(). kstrtok() searches the input string, 'istr', for a token separator, which is specified in the second string parameter, 'sub_str'. Then, it returns a pointer to the remaining portion of the input string after inserting a '\0' at the token separator. If it is called again with a NULL as the input string, the routine continues to parse the previous string passed in.

Side Effects

This routine adds '\0's to the original string. It also has no way to check if the input string has been freed via a kfree_and_NULL() call between subsequent calls to kstrtok().

B.2.18. kchar_replace() — *replace a character with another through a string*

Synopsis

```
char *kchar_replace(
   const char *istr,
   int scan_char,
   int replace_char,
   char *ostr)
```

Input Arguments

`istr`
: The string to be changed.

`scan_char`
: The scan character to be replaced.

`replace_char`
: The character which replaces the scan character.

Output Arguments

`ostr`
: The string that holds the converted string. If it's NULL, the routine allocates the space necessary to hold the result.

Returns

The converted string 'ostr' if it is not NULL, or a pointer to the resulting allocated string if it is NULL. NULL is returned on an error, or if 'istr' is NULL.

Description

This function performs a global change of character on the input string. It returns a string where every occurrence of the scan character is replaced with the replacement character. If 'ostr' is sent in as NULL, the result will be allocated with kmalloc() for you. If 'ostr' is non-NULL, the result will be stored in 'ostr'. Note that if 'ostr' is non-NULL, it must point at a memory buffer with a sufficient

amount of storage space before this routine is called.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the copied string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Examples

For example, if we call:

```
new_string = kchar_replace("Many fishes", 'f', 'w', NULL);
```

the result will be:

```
new_string = "Many wishes".
```

Side Effects

Allocates the space for the output string if 'ostr' is NULL

B.2.19. kstring_capitalize() — *convert a string to its capitalized equivalent*

Synopsis

```
char *kstring_capitalize(
   const char *istr,
   char *ostr)
```

Input Arguments

`istr`

The string to convert to a capitalized version.

Output Arguments

`ostr`

The string that holds the converted string. If it's NULL, it allocates the space necessary.

Returns

The converted output string, 'ostr' if it is not NULL, or a pointer to the resulting allocated string if 'ostr' is NULL. NULL is returned if 'istr' is NULL, or an error occurs.

Description

This routine checks to see that the first character of each word is capitalized. A word is defined to be something separated by whitespace (i.e. separated by one or more tabs or spaces).

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the capitalized string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

Allocates the space necessary for the output if 'ostr' is NULL.

B.2.20. kstring_3cat() — *concatenate three strings together*

Synopsis

```
char *kstring_3cat(
   const char *istr1,
   const char *istr2,
   const char *istr3,
   char *ostr)
```

Input Arguments

`istr1`
: The first string to concatenate.

`istr2`
: The second string to concatenate.

`istr3`
: The third string to concatenate.

Output Arguments

`ostr`
: The string that holds the concatenated string. If it's NULL, it allocates the space necessary to hold the result.

Returns

The concatenated string in 'ostr' if it is non-NULL, or an allocated string if 'ostr' is NULL. This routine returns NULL if 'istr1', 'istr2', and 'istr3' are NULL, or an error occurs.

Description

This routine concatenates three strings together, into a new string. If the output parameter is not NULL, the concatenated string is put into that string. Otherwise, the resulting string is allocated via kmalloc(). In either case, the resulting NULL-terminated string is returned. The concatenation is as follows; the second string is added to the end of the first, and the third string is added to the end of the first two.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the concatenated string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

This routine uses kmalloc() to create a string that holds the result if 'ostr' is NULL.

B.2.21. kstring_cat() — *concatenate two strings*

Synopsis

```
char *kstring_cat(
   const char *istr1,
   const char *istr2,
   char *ostr)
```

Input Arguments

`istr1`
The first string to concatenate.

`istr2`
The second string to concatenate.

Output Arguments

`ostr`
The string that holds the concatenated string. If it's NULL, the routine allocates the necessary space.

Returns

The concatenated string if 'ostr' is non-NULL, or the allocated string if 'ostr' is NULL. On NULL input strings or an error, a NULL is returned.

Description

This routine concatenates two strings together, into a third. If the output parameter is not NULL, the concatenated string is put into that string. Otherwise, the resulting string is allocated via a call to kmalloc(). In either case, the resulting string is returned.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the concatenated string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

This routine uses kmalloc() to create a string that holds the result, if 'ostr' is non-NULL.

B.2.22. kstring_cleanup() — *remove white space from the ends of a string*

Synopsis

```
char *kstring_cleanup(
   const char *istr,
   char *ostr)
```

Input Arguments

`istr`

The string to cleanup.

Output Arguments

`ostr`

The string that holds the converted string. If it's NULL, it kmallocs the space necessary.

Returns

The cleaned string in 'ostr' if it is not NULL, or a pointer to the resulting allocated string if 'ostr' is NULL. NULL is returned if istr is NULL, or an error occurs.

Description

This routine checks the beginning and end of the string for white space (using the function isspace()) and removes these characters. White space is defined to be a tab, space, carriage return, or line feed.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the cleaned string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

Allocates the space for the output string if 'ostr' is NULL

B.2.23. kstring_copy() — *copy a string*

Synopsis

```
char *kstring_copy(
   const char *istr,
   char *ostr)
```

Input Arguments

`istr`

The string to be copied.

Output Arguments

`ostr`

The string that holds the copied string. If 'ostr' is NULL, the routine allocates the space necessary for the result.

Returns

If 'ostr' is non-NULL, it is returned. Otherwise the allocated string is returned. If 'istr' is NULL, or an error occurs, NULL is returned.

Description

This routine copies a string into another string. If the output parameter is not NULL, the concatenated string is put into that string. Otherwise, the resulting string is allocated via a call to kmalloc(). In either case, the resulting string is returned.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the copied string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

This routine uses kmalloc() to create a string that holds the result if 'ostr' is NULL.

B.2.24. kstring_detab() — *remove tabs from a string*

Synopsis

```
char *kstring_detab(
   const char *istr,
   char *ostr)
```

Input Arguments

`istr`

The string to remove tabs from.

Output Arguments

`ostr`

The string that holds the converted input string. If 'ostr' is NULL, it allocates the space necessary to hold the result.

Returns

The variable 'ostr' if it is non-NULL, or the allocated string if 'ostr' is NULL. If 'istr' is NULL or an error occurs, NULL is returned.

Description

This routine converts tab, ’’, characters into the appropriate number of spaces to make it fall onto modulo eight boundary.

Note that if ’ostr’ is non-NULL, the calling routine must ensure that ’ostr’ points to a memory buffer large enough to hold the detab string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

This routine uses kmalloc to create a string that holds the result, if ’ostr’ is NULL.

B.2.25. kstring_lower() — *convert a string to lower case.*

Synopsis

```
char *kstring_lower(
   const char *istr,
   char *ostr)
```

Input Arguments

`istr`

The string to convert to lowercase.

Output Arguments

`ostr`

The string that holds the converted string. If it’s NULL, it allocates the necessary memory with kmalloc().

Returns

The converted string, ’ostr’ if that variable is not NULL, or a pointer to the resulting allocated string. NULL is returned on an allocation error or if ’istr’ is NULL.

Description

This routine performs a character by character scan for uppercase characters. When an uppercase character is found, it calls the function tolower() to get the lowercase equivalent, and replaces it.

Note that if ’ostr’ is non-NULL, the calling routine must ensure that ’ostr’ points to a memory buffer large enough to hold the converted string plus a null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

Allocates the space for the output if ostr is NULL

B.2.26. kstring_ncat() — *concatenate two partial strings*

Synopsis

```
char *kstring_ncat(
   const char *istr1,
   const char *istr2,
   ssize_t num1,
   ssize_t num2,
   char *ostr)
```

Input Arguments

`istr1`
: The first string to concatenate.

`istr2`
: The second string to concatenate.

`num1`
: The number of characters from 'istr1' to put in the final string.

`num2`
: The number of characters from 'istr2' to put in the final string.

Output Arguments

`ostr`
: The string that holds the concatenated string. If it's NULL, it allocates the necessary memory.

Returns

The concatenated string 'ostr' if the variable is non-NULL, or the allocated string if 'ostr' is NULL. If 'istr1' and 'istr2' are NULL, or if 'num1' and 'num2' are both less then or equal to zero, or an error occurs, then NULL is returned.

Description

This routine concatenates two partial strings together, into a third. If the output parameter, 'ostr', is not NULL, the concatenated string is put into that string. Otherwise, the resulting string is allocated In either case, the resulting NULL-terminated string is returned. The second partial string is added to the end of the partial string first. Also, this routine puts a '\0' on the resulting string.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the concatenated string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

This routine uses kmalloc() to allocate a string that holds the result, if 'ostr' is NULL.

B.2.27. kstring_ncopy() — *copy up to n characters of a string*

Synopsis

```
char *kstring_ncopy(
   const char *istr,
   size_t num,
   char *ostr)
```

Input Arguments

`istr`
: The string to copy characters from.

`num`
: The number of characters to copy.

Output Arguments

`ostr`
: The string that holds the copied string. If it's NULL, it allocates the space necessary.

Returns

The copied string 'ostr', if it is non-NULL, or the allocated string if 'ostr' is NULL. NULL is returned if 'istr' is NULL, or if an error occurs.

Description

This routine concatenates two strings together, into a third. If the output parameter is not NULL, the concatenated string is put into that string. Otherwise, the resulting string is allocated by kmalloc(). In either case, the resulting string is returned. Finally, this routine puts a terminating '\0' on the end of the new string.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the copied string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

This routine uses kmalloc() to create a string that holds the result, if 'ostr' is NULL.

B.2.28. kstring_replace() — *replace one substring with another*

Synopsis

```
char *kstring_replace(
   const char *istr,
   const char *scan_str,
   const char *replace_str,
   const int   icase,
   char *ostr)
```

Input Arguments

`istr`
: The string to be changed.

`scan_str`
: The search string to be replaced.

`replace_str`
: The string that replaces the search string.

`icase`
: TRUE if you want to ignore case while searching for the search string

Output Arguments

`ostr`
: The string that holds the converted output string. If it is NULL, this routine will allocate the space necessary.

Returns

The converted string, 'ostr', if the variable is not NULL, or a pointer to the resulting allocated string. NULL is returned if 'istr' or 'scan_str' are NULL, or if an error occurs.

Description

This function performs a global change of text on the input string. It returns a string where every occurrence of the scan string is replaced with the replacement string. If 'ostr' is sent in as NULL, the result will be allocated for you. If 'ostr' is provided as non-NULL, the result will be stored in it. Note that if the latter use is chosen, ostr must be previously allocated with a sufficient amount of storage space before this routine is called.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the copied string and terminating null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Examples

For example, if we call:

```
new_string = kstring_replace("Welcome to VisiQuest", "VisiQuest",
                             "the Twilight Zone", FALSE, NULL);
```

the result will be:

```
new_string = "Welcome to the Twilight Zone".
```

Side Effects

This routine allocates the space for the output if 'ostr' is NULL.

B.2.29. kstring_subcmp() — *compares two sub-strings*

Synopsis

```
int kstring_subcmp(
   const char *istr1,
   const char *istr2)
```

Input Arguments

`istr1`
: The base string to be compared.

`istr2`
: The sub-string to be compared.

Returns

If the two strings up to the end of the second string are identical, the value of 0 is returned. If the two strings differ, the ASCII value difference of the first character that differs in the two strings will be returned. If the ASCII value of the differing character in 'istr1' is greater than the one in 'istr2', then the return value is positive, and implies that 'istr1' is greater than negative, the return value implies that 'istr2' is greater than 'istr1'.

Description

This routine compares two input strings to determine if 'istr2' is the same string as the beginning of 'istr1'. In kstring_subcmp() the number of characters compared is an implied value which is the length of the second input string 'istr2'. The input strings are compared, character by character, until either the end of one of the input strings is encountered or 'istr1' and 'istr2' no longer match.

B.2.30. kstring_upper() — *convert a string to upper case.*

Synopsis

```
char * kstring_upper(
   const char *istr,
   char *ostr)
```

Input Arguments

`istr`

The string to convert to upper case.

Output Arguments

`ostr`

The string that holds the converted string. If variable is NULL, the routine allocates the necessary space to hold the result.

Returns

The converted string, 'ostr', if it is not NULL, or a pointer to the resulting allocated string if it is NULL. NULL is returned if 'istr' is NULL, or if an error occurs.

Description

This routine performs a character by character scan for lowercase characters. When an lowercase character is found, it calls the function, toupper(), to get the uppercase equivalent, and replaces the lower case character.

Note that if 'ostr' is non-NULL, the calling routine must ensure that 'ostr' points to a memory buffer large enough to hold the converted string plus a null character. If the buffer is not large enough, memory will be overwritten resulting in unpredicable program failure.

Side Effects

Allocates the space for the output if 'ostr' is NULL

B.2.31. kstring_seddata() — *perform text changes with one or more sets of substitution rules*

Synopsis

```
char *kstring_seddata(
   kstring str,
   kvalist)
```

Input Arguments

`str`
: source kstring

`kvalist`
: NULL terminated list of (search pattern, replacement pattern, flags) triples to be used to modify the data as it is being copied.

Returns

pointer to resulting kstring

Description

This function performs a global change of text on a string based on substring or regular expression patterns.

It uses kstring_replace or kstring_regex replace based on the 0 bit of the flags parameter for each search/replace/flags triple specified on the variable argument list. Bit 1 is used to specify a case insensitive match. The following table explains the possibilities:

flag	meaning
TRUE or KRE_ICOMP	case insensitive regex replace
FALSE or KRE_ICOMP	case insensitive string replace
TRUE	case sensitive regex replace
FALSE	case sensitive string replace

Examples

For example, if we call:

```
new_string = kstring_seddata("Welcome to VisiQuest",
                             "VisiQuest", "the Twilight Zone",
                             FALSE, NULL);
```

the result will be:

```
new_string = "Welcome to the Twilight Zone".
```

C. Tokenized String Utilities

The following section details the tokenized string utilities that are a part of the *kutils (libku.a)* library.

C.1. Introduction to Tokenized String Utilities

It is common to use strings frequently in programming. However, the use of strings often implies both an increase in memory use and a decrease in efficiency. For example, an algorithm may involve a string identifier. If the string identifier must be used for searching purposes, the searching algorithm may become inefficient. This situation calls for the use of *tokenized* strings. A *token* is a unique integer representation of a string. The program switches between the string itself and its tokenized representation when appropriate. For example, a program might read in a string, convert it to a token, store it internally as a token, and search on the token when necessary, but it will then print it out as a string again. The tokenized string utilities are:

- *kstring_to_token()* - return the token that is associated with the specified string
- *ktoken_to_string()* - return the string associated with the specified token
- *ktoken_delete()* - delete the token'ized string from the list of tokens
- *ktoken_check()* - check to see if a string has been token'ized

C.2. Definitions of Tokenized String Utilities

C.2.1. kstring_to_token() — *return the token that is associated with the specified string*

Synopsis

```
ktoken
kstring_to_token(
   const char *istr)
```

Input Arguments

`istr`

The string to compute the token for.

Returns

The token computed or KTOKEN_NONE if an error occurs.

Description

This function returns the token that is associated with the supplied string. The idea is that a given string is tagged with a unique token, so that an application which is being slowed down by numerous string comparisons or searches through string attributes, can use the unique tokens for a value comparison instead using a traditional string compare call. Thus, greatly reducing the comparison time.

kstring_to_token() returns unique tokens for different strings, so that calling routines can perform string comparisons using "token1 == token2" instead of the time intensive kstrcmp(str1, str2) function call.

If the string is NULL or if the space to copy the string cannot be allocated, then KTOKEN_NONE is returned.

Examples

For example to instatiate a string, using the function call kstring_to_token(), the programmer uses the following call:

```
token1 = kstring_to_token("test string1");
token2 = kstring_to_token("test string2");
```

This should yield two totally distinct tokens, which when compared (token1 == token2) will result that the two strings are not the same.

C.2.2. ktoken_to_string() — *return the string associated with the specified token*

Synopsis

```
char *
ktoken_to_string(
   ktoken token)
```

Input Arguments

`token`

The token to return the corresponding string for.

Returns

The token's corresponding string or NULL, if the token is not valid.

Description

This function returns the string that is associated with the supplied token. The idea is that a unique

token is given for each string sent to the function kstring_to_token(). This routine is used to retrieve a pointer to the token's associated string.

C.2.3. ktoken_check() — *check to see if a string has been token'ized*

Synopsis

```
ktoken
ktoken_check(
    const char *istr)
```

Input Arguments

`istr`

The string to check for a token.

Returns

If the string is NULL or if the string has not been token'ized, KTOKEN_NONE is returned. Otherwise the token corresponding to the string is returned.

Description

This function checks to see if the input string has a token representation.

C.2.4. ktoken_delete() — *delete the token'ized string from the list of tokens*

Synopsis

```
int
ktoken_delete(
    const char *string)
```

Input Arguments

`string`

The string to delete.

Returns

TRUE (1) on success, FALSE (0) otherwise.

Description

This function deletes the token'ized representation of a string from the token list. If the string has a token representation then it is removed from the list.

If the string is NULL, or the string or the token'ized string cannot be deleted, then FALSE is returned.

D. Time String Utilities

The following section details the time string utilities that are a part of the *klibc* library.

D.1. Introduction to the Time String Utilities

The time string utilities are:

• *kstrftime()* - generate formatted time information
• *kget_date()* - get the current time and date in a string

D.2. Definitions of Time String Utilities

D.2.1. kstrftime() — *generate formatted time information*

Synopsis

```
size_t kstrftime(
   char *s,
   size_t maxsize,
   const char *format,
   const struct tm * timeptr)
```

Input Arguments

`maxsize`
: The size of the 's' output array.

`format`
: The string that describes how the time structure should be formatted.

`timeptr`
: A pointer to a time structure, which is set to the time you want to format.

Output Arguments

`s`
: formatted output time string

Returns

If the total number of resulting characters including the terminating null character is not more than maxsize, the kstrftime() function returns the number of characters placed into the array pointed to by s

not including the terminating null character. Otherwise, zero is returned and the contents of the array are indeterminate.

Description

The following description is transcribed verbatim from the December 7, 1988 draft standard for ANSI C. This draft is essentially identical in technical content to the final version of the standard.

The kstrftime() function places characters into the array pointed to by 's' as controlled by the string pointed to by 'format'. The format shall be a multibyte character sequence, beginning and ending in its initial shift state. The format string consists of zero or more conversion specifiers and ordinary multibyte characters. A conversion specifier consists of a % character followed by a character that determines the behavior of the conversion specifier. All ordinary multibyte characters (including the terminating null character) are copied unchanged into the array. If copying takes place between objects that overlap the behavior is undefined. No more than 'maxsize' characters are placed into the array. Each conversion specifier is replaced by appropriate characters as described in the following list. The appropriate characters are determined by the LC_TIME category of the current locale and by the values contained in the structure pointed to by 'timeptr'.

%a is replaced by the locale's abbreviated weekday name.

%A is replaced by the locale's full weekday name.

%b is replaced by the locale's abbreviated month name.

%B is replaced by the locale's full month name.

%c is replaced by the locale's appropriate date and time representation.

%d is replaced by the day of the month as a decimal number (01-31).

%H is replaced by the hour (24-hour clock) as a decimal number (00-23).

%I is replaced by the hour (12-hour clock) as a decimal number (01-12).

%j is replaced by the day of the year as a decimal number (001-366).

%m is replaced by the month as a decimal number (01-12).

%M is replaced by the minute as a decimal number (00-59).

%p is replaced by the locale's equivalent of the AM/PM designations associated with a 12-hour clock.

%S is replaced by the second as a decimal number (00-61).

%U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00-53).

%w is replaced by the weekday as a decimal number [0 (Sunday)-6].

%W is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (00-53).

%x is replaced by the locale's appropriate date represen- tation.

%X is replaced by the locale's appropriate time represen- tation.

%y is replaced by the year without century as a decimal number (00-99).

%Y is replaced by the year with century as a decimal number.

%Z is replaced by the time zone name or abbreviation, or by no characters if no time zone is determinable.

%% is replaced by %.

If a conversion specifier is not one of the above, the behavior is undefined.

D.2.1.1. Extensions

The following sections describe conversion specifiers that are available as extensions to the standard set.

D.2.1.1.1. Non-ANSI Extensions

If SYSV_EXT is defined when the routine is compiled, then the following additional conversions will be available. These are borrowed from the System V cftime(3) and ascf- time(3) routines.

%D is equivalent to specifying %m/%d/%y.

%e is replaced by the day of the month, padded with a blank if it is only one digit.

%h is equivalent to %b, above.

%n is replaced with a newline character (ASCII LF).

%r is equivalent to specifying %I:%M:%S %p.

%R is equivalent to specifying %H:%M.

%T is equivalent to specifying %H:%M:%S.

%t is replaced with a TAB character.

If SUNOS_EXT is defined when the routine is compiled, then the following additional conversions will be available. These are borrowed from the SunOS version of kstrftime.

%k is replaced by the hour (24-hour clock) as a decimal number (0-23). Single digit numbers are padded with a blank.

%l is replaced by the hour (12-hour clock) as a decimal number (1-12). Single digit numbers are padded with a blank.

D.2.1.1.2. POSIX 1003.2 Extensions

If POSIX2_DATE is defined, then all of the conversions available with SYSV_EXT and SUNOS_EXT are available, as well as the following additional conversions:

%C The century, as a number between 00 and 99.

%u is replaced by the weekday as a decimal number [1 (Monday)-7].

%V is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (01-53). The method for determining the week number is as specified by ISO 8601 (to wit: if the week contain- ing January 1 has four or more days in the new year, then it is week 1, otherwise it is week 53 of the pre- vious year and the next week is week 1). The text of the POSIX standard for the date utility describes %U and %W this way:

%U is replaced by the week number of the year (the first Sunday as the first day of week 1) as a decimal number (00-53). All days in a new year preceding the first Sunday are considered to be in week 0.

%W is replaced by the week number of the year (the first Monday as the first day of week 1) as a decimal number (00-53). All days in a new year preceding the first Monday are considered to be in week 0.

In addition, the alternate representations %Ec, %EC, %Ex, %Ey, %EY, %Od, %Oe, %OH, %OI, %Om, %OM, %OS, %Ou, %OU, %OV, %Ow, %OW, and %Oy are recognized, but their normal represen- tations are used.

D.2.1.1.3. VMS Extensions

If VMS_EXT is defined, then the following additional conver- sion is available:

%v The date in VMS format (e.g. 20-JUN-1991).

D.2.2. kget_date() — *get the current time and date in a string*

Synopsis

```
kstring
kget_date(
```

```
void)
```

Returns

A string containing the current time and date, in the format used by all the tools.

Description

This function determines the current time and date, and returns a fixed format string.

E. Standardized Error Messages & Prompting

The following sections detail the error message and prompting utilities that are a part of the *klibc (libku.a)* library.

E.1. Introduction to Message/Prompting Utilities

There are a number of utilities available in VisiQuest to ensure consistent and standardized prompting. By default, these routines will produce output which conforms to the context in which they occur. When data processing routines (kroutines) are executed from the command line, these routines output to stderr or stdout. The output is then converted and appears in a pop-up window when used by an X-based application (xvroutine), or when a data processing routine is accessed via cantata. These routines are provided through Basic Services so that a central facility may be employed. Since the routines are provided at the lowest level, Basic Services, they can be used to control how, why, and when information is reported back to the user. This is critical to allow routines to be encapsulated within larger applications. The defines for these routines are found in $BOOTSTRAP/include/klibc/knotify.h. The routines are:

- *kannounce()* - report or announce a message in a standardized format
- *kchoose()* - prompt the user to select from a list of items
- *kerror()* - print error messages in a standardized format
- *kget_notify()* - get the VisiQuest notify level
- *kinfo()* - print information messages in a standardized format
- *koverwrite()* - request an acknowledgement for overwriting files
- *kprompt()* - request an acknowledgement from the user
- *ksave()* - request an acknowledgement for quitting an application
- *kquit()* - request an acknowledgement for quitting an application
- *kset_announcehandler()* - set the announce handling routine used by kannounce()
- *kset_choosehandler()* - set the choose handling routine used by kchoose()
- *kset_errorhandler()* - set the error handling routine used by kerror()
- *kset_infohandler()* - set the information handling routine used by kinfo()
- *kset_notify()* - set the VisiQuest notify level
- *kset_prompthandler()* - set the prompt handling routine used by kprompt()
- *kset_quithandler()* - set the quit handling routine used by kquit()
- *kset_savehandler()* - set the save handling routine used by ksave()
- *kset_warnhandler()* - set the warning handler routine used by kwarn()
- *kwarn()* - print warning messages in a standardized format

If you use the routines above and would like to *force* the output to be in a particular form, you may use the kset_* routines to do so. These routines allow you to force output to the *tty* in standard VisiQuest form. They also allow you to force output to a standard VisiQuest pop-up window or to force output to be displayed in the manner which is defined by your own handler.

The utilities defined in this section are: Message/Prompting Utilities, Variable Argument Functions, Routines To Set/Get Notify Level, and Routines to Set Handlers.

E.2. Definitions of Message/Prompting Utilities

E.2.1. kannounce() — *report or announce a message in a standardized format*

Synopsis

```
int kannounce(
   char *library,
   char *routine,
   char *format,
   kvalist)
```

Input Arguments

`library`
: name of library

`routine`
: name of routine

`format`
: grammatically correct, clear explanation of what should be announced to the user

Returns

TRUE if the announcement was successfully delivered, otherwise FALSE is returned.

Description

kannounce produces standardized messages for VisiQuest library routines and applications. It should be called when the programmer wants give generalized updates or reports to the user. This typically used by the larger applications, such as VisiQuest, composer, craftsman, etc to give progress or status reports to the user.

E.2.2. kchoose() — *prompt the user to select from a list of items*

Synopsis

```
char *kchoose(
   int notify_type,
   char **list_of_options,
   int num_options,
   int default_index,
   char *return_string,
   int *return_index,
   char *format,
   kvalist)
```

Input Arguments

`notify_type`
: the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSSE

`list_of_options`
: an array of strings containing the items to select from.

`num_options`
: The number of items in the list_of_options.

`default_index`
: The index number to the default item, must start at 1.

`format`
: grammatically correct, clear explanation of the error that occurred. This can be formatted like a (void) printf statement.

Output Arguments

`return_string`
: string that holds the selected item. If it's NULL, it kmallocs the space necessary, and returns the string.

`return_index`
: This is the index of the item selected.

Returns

return_string if it is not NULL, or a pointer to the resulting kmalloc'ed string if it is NULL. NULL is returned upon error.

Description

kchoose will call the specified choose handler to request the user to make a selection from a list of items. This utility can operate in several different modes.

If the notify_type variable is set to KFORCE, then the prompt will always appear regardless of the setting of the environment variable KHOROS_NOTIFY.

If the notify_type variable is set to KSTANDARD and the user has the environment variable KHOROS_NOTIFY set to either KSTANDARD or KVERBOSE the prompt will appear.

And finally, if the notify_type variable is set to KVERBOSE and the environment variable KHOROS_NOTIFY set to KVERBOSE, the prompt will appear.

Here is a summary table:

```
notify_type = FORCE     always prompt, ignore the setting of
                        the environment variable KHOROS_NOTIFY

notify_type = STANDARD  only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        STANDARD or VERBOSE.

notify_type = VERBOSE   only prompt when the environment
                        variable KHOROS_NOTIFY is set to
```

E.2.3. kerror() — *print error messages in a standardized format*

Synopsis

```
int kerror(
   char *library,
   char *routine,
   char *format,
   kvalist)
```

Input Arguments

`library`
: name of library (NULL if not applicable)

`routine`
: name of routine

`format`
: grammatically correct, clear explanation of the error that occurred. This can be formatted like a printf statement

Returns

TRUE if the error was successfully acknowledge, otherwise if the message was not acknowledged FALSE is returned.

Description

kerror produces standardized error messages for VisiQuest library routines and applications. It should be called in EVERY instance of error messaging by EVERY VisiQuest function, subroutine, or main program. If library is NULL then the program name will be used as the source of the error.

E.2.4. kinfo() — *print information messages in a standardized format*

Synopsis

```
int kinfo(
   int notify_type,
   char *format,
   kvalist)
```

Input Arguments

`notify_type`

the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSE

`format`

grammatically correct, clear explanation of the information. Note the message can be formatted like a printf statement.

Returns

TRUE if the message was successfully printed, otherwise FALSE is returned.

Description

kinfo produces standardized information messages for VisiQuest library routines and VisiQuest applications. It should be called in EVERY instance of information messaging by EVERY VisiQuest function, subroutine, or main program.

If the notify_type variable is set to KFORCE, then the prompt will always appear regardless of the setting of the environment variable KHOROS_NOTIFY.

If the notify_type variable is set to KSTANDARD and the user has the environment variable KHOROS_NOTIFY set to either STANDARD or VERBOSE the prompt will appear.

And finally, if the notify_type variable is set to KVERBOSE and the environment variable KHOROS_NOTIFY set to VERBOSE, the prompt will appear.

Here is a summary table:

```
notify_type = FORCE     always prompt, ignore the setting of
                        the environment variable KHOROS_NOTIFY
```

```
notify_type = STANDARD  only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        STANDARD or VERBOSE.

notify_type = VERBOSE   only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        VERBOSE
```

E.2.5. koverwrite() — *request an acknowledgement for overwriting files*

Synopsis

```
int koverwrite(
   int notify_type,
   char *filename)
```

Input Arguments

`notify_type`
: the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSSE

`filename`
: the filename in question to overwrite

Returns

TRUE if the prompt was successfully acknowledged, otherwise if the message was not acknowledged FALSE is returned. In the event of an error TRUE is returned. If the file does not exist, TRUE is returned.

Description

koverwrite will call the specified prompt handler to request or demand an acknowledgement from the user. This utility will as the user if it is ok to overwrite the file in question, and can operate in several different modes.

If the notify_type variable is set to KFORCE, then the prompt will always appear regardless of the setting of the environment variable KHOROS_NOTIFY.

If the notify_type variable is set to KSTANDARD and the user has the environment variable KHOROS_NOTIFY set to either STANDARD or VERBOSE the prompt will appear.

And finally, if the notify_type variable is set to KVERBOSE and the environment variable KHOROS_NOTIFY set to VERBOSE, the prompt will appear.

Here is a summary table:

```
notify_type = FORCE     always prompt, ignore the setting of
                        the environment variable KHOROS_NOTIFY

notify_type = STANDARD  only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        STANDARD or VERBOSE.

notify_type = VERBOSE   only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        VERBOSE
```

E.2.6. kprompt() — *request an acknowledgement from the user*

Synopsis

```
int kprompt(
   int notify_type,
   char *yes_response,
   char *no_response,
   int default_val,
   char *format,
   kvalist)
```

Input Arguments

`notify_type`
: the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSSE

`yes_response`
: name of "yes" response string ("Yes" if NULL)

`no_response`
: name of "no" response string ("No" if NULL)

`default_val`
: the default value to list when prompting

`format`
: grammatically correct, clear explanation of the error that occurred. This can be formatted like a printf statement.

Returns

TRUE if the prompt was successfully acknowledged, otherwise if the message was not acknowledged FALSE is returned. In the event and error occurs the default value is returned.

Description

kprompt will call the specified prompt handler to request or demand an acknowledgement from the user. This utility can operate in several different modes.

If the notify_type variable is set to KFORCE, then the prompt will always appear regardless of the setting of the environment variable KHOROS_NOTIFY.

If the notify_type variable is set to KSTANDARD and the user has the environment variable KHOROS_NOTIFY set to either STANDARD or VERBOSE the prompt will appear.

And finally, if the notify_type variable is set to KVERBOSE and the environment variable KHOROS_NOTIFY set to VERBOSE, the prompt will appear.

Here is a summary table:

```
notify_type = FORCE     always prompt, ignore the setting of
                        the environment variable KHOROS_NOTIFY

notify_type = STANDARD  only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        STANDARD or VERBOSE.

notify_type = VERBOSE   only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        VERBOSE
```

E.2.7. ksave() — *request an acknowledgement for quitting an application*

Synopsis

```
int ksave(
   int notify_type,
   char *format,
   kvalist)
```

Input Arguments

`notify_type`

the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSE message - the message prompting the user for exiting.

Returns

TRUE if the user wants to quit the application, otherwise if the user doesn't want to quit then FALSE is returned. In the event of an error FALSE is returned.

Description

ksave will call the specified prompt handler to request or demand an acknowledgement from the user. This utility will as the user if it is ok to overwrite the file in question, and can operate in several different modes.

If the notify_type variable is set to KFORCE, then the prompt will always appear regardless of the setting of the environment variable KHOROS_NOTIFY.

If the notify_type variable is set to KSTANDARD and the user has the environment variable KHOROS_NOTIFY set to either STANDARD or VERBOSE the prompt will appear.

And finally, if the notify_type variable is set to KVERBOSE and the environment variable KHOROS_NOTIFY set to VERBOSE, the prompt will appear.

Here is a summary table:

```
notify_type = FORCE     never prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        FORCE.

notify_type = STANDARD  only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        STANDARD or VERBOSE.

notify_type = VERBOSE   only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        VERBOSE
```

E.2.8. kquit() — *request an acknowledgement for quitting an application*

Synopsis

```
int kquit(
   int notify_type,
   char *format,
   kvalist)
```

Input Arguments

`notify_type`

the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSE message - the message prompting the user for exiting.

Returns

TRUE if the user wants to quit the application, otherwise if the user doesn't want to quit then FALSE is returned. In the event of an error FALSE is returned.

Description

kquit will call the specified prompt handler to request or demand an acknowledgement from the user. This utility will as the user if it is ok to overwrite the file in question, and can operate in several different modes.

If the notify_type variable is set to KFORCE, then the prompt will always appear regardless of the setting of the environment variable KHOROS_NOTIFY.

If the notify_type variable is set to KSTANDARD and the user has the environment variable KHOROS_NOTIFY set to either STANDARD or VERBOSE the prompt will appear.

And finally, if the notify_type variable is set to KVERBOSE and the environment variable KHOROS_NOTIFY set to VERBOSE, the prompt will appear.

Here is a summary table:

```
notify_type = FORCE     never prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        FORCE.

notify_type = STANDARD  only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        STANDARD or VERBOSE.

notify_type = VERBOSE   only prompt when the environment
                        variable KHOROS_NOTIFY is set to
                        VERBOSE
```

E.2.9. kwarn() — *print warning messages in a standardized format*

Synopsis

```
int kwarn(
   char *library,
   char *routine,
   char *format,
   kvalist)
```

Input Arguments

`library`
: name of library (NULL if not applicable)

`routine`
: name of routine

`format`
: grammatically correct, clear explanation of the warning that occurred. This can be formatted like a printf statement

Returns

TRUE if the error was successfully acknowledge, otherwise if the message was not acknowledged FALSE is returned.

Description

kwarn produces standardized warning messages for VisiQuest library routines and applications. It should be called in EVERY instance of warning messaging by EVERY VisiQuest function, subroutine, or main program. If library is NULL then the program name will be used as the source of the warning.

E.3. Definitions of Routines To Set/Get Notify Level

E.3.1. kget_notify() — *get the VisiQuestnotify level*

Synopsis

```
int kget_notify(void)
```

Returns

the current notify value

Description

This routine gets the notify level. The possible values are KFORCE, KSTANDARD, KVERBOSE, KXVLIB, KSYSLIB, KHOSTILE, and KDEBUG.

E.3.2. kset_notify() — *set the VisiQuest notify level*

Synopsis

```
int kset_notify(
   int notify_type)
```

Input Arguments

`notify_type`

the notify level specified by the programmer KFORCE, KSTANDARD, KVERBOSE, KXVLIB, KSYSLIB, KHOSTILE, KDEBUG

Returns

the current notify value before overriding it with the new value

Description

This routine sets the notify level. The possible values are KFORCE, KSTANDARD, KVERBOSE, KXVLIB, KSYSLIB, KHOSTILE, and KDEBUG.

E.4. Definitions of Routines To Set Handlers

E.4.1. kset_announcehandler() — *set the announce handling routine used by kannounce()*

Synopsis

```
int (*kset_announcehandler(
   int (*new_handler)(char *, char *, char *, char *, char *)))
                              (char *, char *, char *, char *, char *)
```

Input Arguments

`new_handler`

the announce handler to be called.

Returns

the previously installed announce handler, or NULL if the default announce handler was installed.

Description

Sets the announce handler routine to be used by the kannounce() reporting facility. When set to NULL (the default), the VisiQuest announce handler will be used, which simply prints the announcement to kstderr.

If a different announce handler is set, the announce handler must be declared as follows, and should return TRUE.

```
int announce_handler(
   char *toolbox,
   char *program,
   char *library,
   char *routine,
   char *message)
```

toolbox - name of toolbox
program - name of program
library - name of library
routine - name of routine
message - grammatically correct, clear announcement message

E.4.2. kset_choosehandler() — *set the choose handling routine used by kchoose()*

Synopsis

```
int (*kset_choosehandler(
   int (*new_handler)(char **, int, int, int *, char **, char *)))
         (char **, int, int, int *, char **, char *)
```

Input Arguments

`new_handler`

the choose handler to be called. Specify xvu_choose() for the VisiQuest pop-up choice dialog (xvroutines & hybrid routines only), NULL for the VisiQuest default choice handler which prints to the tty, or your own choice handler.

Returns

the previously installed choose handler, or NULL if the default choose handler was installed.

Description

Sets the choose handler routine to be used by the kchoose() facility. When set to NULL (the default) the standard VisiQuest choose handler will obtain the user's choice via interaction through kstderr and kstdin.

If a different choose handler is set, the choose handler must be declared as follows:

```
choose_handler(
  char **list_of_options,
  int    num_options,
  int    default_index,
  int   *return_index,
  char **return_string,
  char  *message)
```

list_of_options - an array of strings containing the items to select from.
num_options - The number of items in the list_of_options
default_index - The index number to the default item.
return_index - This is the index of the item selected.
return_string - string that holds the selected item. If it is NULL, it kmallocs the space necessary, and returns the string.
message - grammatically correct, clear explanation of the error that occurred. This can be formatted like a (void) printf statement.

E.4.3. kset_errorhandler() — *set the error handling routine used by kerror()*

Synopsis

```
int (*kset_errorhandler(
   int (*new_handler)(char *, char *, char *, char *, char *, char *)))
         (char *, char *, char *, char *, char *, char *)
```

Input Arguments

`new_handler`

the error handler to be called. Specify xvu_error() for the VisiQuest pop-up error message (xvroutines & hybrid routines only), NULL for the VisiQuest default error handler which prints to the tty, or your own error handler.

Returns

the previously installed error handler, or NULL if the default error handler was installed.

Description

Sets the error handler routine to be used by the kerror() reporting facility. When set to NULL (the default) the default VisiQuest error handler is used, which simply prints the error to kstderr.

If a different error handler is set, the error handler must be declared as follows, and should return

TRUE.

```
int error_handler(
   char *toolbox,
   char *program,
   char *library,
   char *routine,
   char *category,
   char *message)
```

toolbox - name of toolbox in which error occured
program - name of program in which error occured
library - name of library in which error occured
routine - name of routine in which error occured
category - error message category
(see $BOOTSTRAP/include/klibc/knotify.h)
message - grammatically correct, clear error message

E.4.4. kset_infohandler() — *set the information handling routine used by kinfo()*

Synopsis

```
int (*kset_infohandler(
   int (*new_handler)(char *)))(char *)
```

Input Arguments

`new_handler`
the info handler to be called. Specify xvu_info() for the VisiQuest pop-up info message (xvroutines & hybrid routines only), NULL for the default info handler which prints to the tty, or your own info handler.

Returns

the previously installed information handler, or NULL if the default information handler was installed.

Description

Sets the information handler routine to be used by the kinfo() reporting facility. When set to NULL (the default) the standard VisiQuest information handler will be used, which reports to standard error.

If a different info handler is set, the info handler must be declared as follows, and should return TRUE.

```
info_handler(
   char *message )
```

message - message to give the user

E.4.5. kset_prompthandler() — *set the prompt handling routine used by kprompt()*

Synopsis

```
int (*kset_prompthandler(
   int (*new_handler)(char *, char *, int, char *)))
         (char *, char *, int, char *)
```

Input Arguments

`new_handler`

the prompt handler to be called. Specify xvu_prompt() for the VisiQuest pop-up prompt window (xvroutines & hybrid routines only), NULL for the VisiQuest default prompt handler which prints to the tty, or your own prompt handler.

Returns

the previously installed prompt handler, or NULL if the default prompt handler was installed.

Description

Sets the prompt handler routine to be used by the kprompt() reporting facility. When set to NULL (the default) the default VisiQuest prompt handler prompt the user for a response using kstderr and kstdin.

If a different prompt handler is set, the prompt handler must be declared as follows:

```
prompt_handler(
   char *yes_response,
   char *no_response,
   int  default_val,
   char *message)
```

yes_response - string to put in the affirmative field of a prompt
no_response - string to put in the negative field of a prompt
default - string to put in the default field of a prompt
message - grammatically correct, clear prompt

E.4.6. kset_quithandler() — *set the quit handling routine used by kquit()*

Synopsis

```
int (*kset_quithandler(
   int (*new_handler)(char *)))(char *)
```

Input Arguments

`new_handler`

the quit handler to be called. Specify xvu_quit() for the VisiQuest pop-up quit message (xvroutines & hybrid routines only), NULL for the default quit handler which returns TRUE.

Returns

the previously installed quit handler, or NULL if the default information handler was installed.

Description

Sets the quit handler routine to be used by the kquit() reporting facility. When set to NULL (the default) the default VisiQuest quit handler is used, which does not prompt the user but simply returns yes, the user wants to quit.

If a different quit handler is set, the quit handler must be declared as follows. It should return TRUE if the user indicates that they do want to quit the program, FALSE if the user indicates that they don't want to quit after all.

```
int quit_handler(
      char *message)
```

char *message - message to ask the user if they are ready to quit

E.4.7. kset_savehandler() — *set the save handling routine used by ksave()*

Synopsis

```
int (*kset_savehandler(
   int (*new_handler)(char *)))(char *)
```

Input Arguments

`new_handler`

the save handler to be called. Specify xvu_save_wait() for the VisiQuest pop-up quit message (xvroutines & hybrid routines only), NULL for the default quit handler which returns TRUE that the user wants to save changes.

Returns

the previously installed save handler, or NULL if the default save handler was installed.

Description

Sets the save handler routine to be used by the ksave() reporting facility. When set to NULL (the default) the default VisiQuest save handler is used, which does not prompt the user but simply returns yes, the user wants to save changes made to the application.

If a different save handler is set, the save handler must be declared as follows. It should return 2 if the user wants to save changes, 1 if the user wants to discard changes, 0 if the user wants to cancel the operation.

```
int save_handler(
   char *message)
```

char *message - message asking user if they want to save

changes to (some file) that were made during the run of the program.

E.4.8. kset_warnhandler() — *set the warning handler routine used by kwarn()*

Synopsis

```
int (*kset_warnhandler(
   int (*new_handler)(char *, char *, char *, char *, char *)))
         (char *, char *, char *, char *, char *)
```

Input Arguments

`new_handler`

the warning handler to be called. Specify xvu_warn() for the VisiQuest pop-up error message (xvroutines & hybrid routines only), NULL for the VisiQuest default error handler which prints to the tty, or your own error handler.

Returns

the previously installed warning handler, or NULL if the default warning handler was installed.

Description

Sets the warning handler routine to be used by the kwarn() reporting facility. When set to NULL (the default) the default VisiQuest warn handler is used, which simply prints the warning to kstderr.

If a different warning handler is set, the warning handler must be declared as follows, and should return TRUE.

```
int warn_handler(
   char *toolbox,
   char *program,
   char *library,
   char *routine,
   char *message)
```

toolbox - name of toolbox containing code issuing warning
program - name of program containing code issuing warning
library - name of library containing code issuing warning
routine - name of routine issuing warning
message - grammatically correct, clear warning

F. A Dyamic Errno System

F.1. Introduction to Generalized VisiQuestErrno Facility

There are a number of utilities under UNIX that use an error number or *errno* value to indicate the general category of an error in a low-level library. This method of error reporting has allowed for consistent error reporting by low-level routines from a top-level library or program. The drawback to this method of error reporting is that on most UNIX systems, this list of errno numbers and strings is statically-defined by the company that wrote the operating system. In VisiQuest, an errno is reasonable for low-level libraries but it is not reasonable to set a static list of numbers and strings. This section describes the functions necessary to dynamically add new error numbers and strings to the VisiQuest errno system at run-time. This errno information is used by the information and error-reporting utilities to print the category of the error or warning. The routines are located in the *klibc (libku.a)* library and are as follows:

- *kerrno_init_errors()* - initialize errors to be used with khoros errno
- *kerrno_check()* - check to see if an errno is within a given error list.
- *kerrno_lookup()* - lookup the error message associated with a errno.
- *kerrno_class()* - return the class number for a given errno.
- *kset_errno()* - set an errno with a debug message

F.2. Errno Initialization and lookup routines

F.2.1. kerrno_init_errors() — *initialize errors to be used with khoros errno*

Synopsis

```
int kerrno_init_errors(
   kerrlist * errlist,
   size_t num_errs)
```

Input Arguments

`errlist`
: the array of errno-message string pairs to be added to the global errno list.

`num_errs`
: number of errno's in the errlist array

Returns

new error class identifier on success, or 0 on failure

Description

kerrno_init_errors initialize errors to be used with the khoros error reporting facility.

F.2.2. kerrno_check() — *check to see if an errno is within a given error list.*

Synopsis

```
int kerrno_check(
   int errnum,
   kerrlist * errlist,
   size_t num_errs)
```

Input Arguments

`errnum`
: the error number to be looked up

`errlist`
: the error list to check against

`num_errs`
: the number of errors

Returns

TRUE (1) if the errno is a member of the errlist passed in, FALSE (0) othersize.

Description

This routine looks up a errno to see if it is within a supplied list of errors.

F.2.3. kerrno_lookup() — *lookup the error message associated with a errno.*

Synopsis

```
char *kerrno_lookup(
   int errnum)
```

Input Arguments

`errnum`

the error number to be looked up

Returns

a pointer to the error message associated with the errnum or NULL if the error number does not exist.

Description

This routine finds the message string associated with an errno.

F.2.4. kerrno_class() — *return the class number for a given errno.*

Synopsis

```
int kerrno_class(int errnum)
```

Input Arguments

`errnum`

the error number to be looked up

Returns

error class if the error is defined

Description

This routine looks up a errno and returns the class of error it is associated with.

F.2.5. kset_errno() — *set an errno with a debug message*

Synopsis

```
void kset_errno(int num)
```

Input Arguments

`num`

errno value to set

Description

This function is a helper function which can be used to set an errno value as well as printing a debug message indicating the file and line number at which the warning was set. The message will only be printed if KHOROS_NOTIFY is set to KDEBUG.

G. Program Attributes

G.1. Introduction to Program Statistic Utilities

The Program Statistic Utilities get and set attributes of the program itself, such as the command line arguments, the program environment, the program object, and the toolbox in which the program is installed. These routines include:

- *kprog_get_argc()* - get the number of arguments in the argv structure
- *kprog_get_argv()* - get the arguments in the argv structure
- *kprog_get_command()* - gets the command string in which this program was executed with.
- *kprog_get_envp()* - gets the environment variable parameter structure
- *kprog_get_program()* - gets the name of the program
- *kprog_get_toolbox()* - gets the toolbox in which this program belongs.
- *kprog_set_argc()* - set the number of commandline parameters
- *kprog_set_argv()* - set the command line argument array
- *kprog_set_envp()* - set the number of environment variable parameters
- *kprog_set_program()* - set the name of the program
- *kprog_set_toolbox()* - set the toolbox in which this software object belongs.
- *khoros_initialize()* - initialize khoros system (old version for VisiQuest 2.1p1)
- *khoros_init()* - initialize khoros system (new version for VisiQuest 2.1p2)
- *khoros_imprint()* - imprint the khoros toolbox

G.2. Definitions of Utilities To Get Program Statistics

G.2.1. kprog_get_argc() — *get the number of arguments in the argv structure*

Synopsis

```
int kprog_get_argc(void)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Gets the number of arguments in the program argv structure.

G.2.2. kprog_get_argv() — *get the arguments in the argv structure*

Synopsis

```
char **kprog_get_argv(void)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

returns the array of commandline arguments.

Restrictions

This routine returns a pointer to the internal argv array. Do not change or free any data in this array, use for read only purposes only.

G.2.3. kprog_get_command() — *gets the command string in which this program was executed with.*

Synopsis

```
char *kprog_get_command(void)
```

Description

Gets the command used to originally excute the current program.

Restrictions

This routine returns a pointer to the private string used to store the toolbox variable. Use this as a read only resource; do NOT change or free this pointer.

G.2.4. kprog_get_envp() — *gets the environment variable parameter structure*

Synopsis

```
char **kprog_get_envp(void)
```

Returns

returns the envp structure or NULL upon failure

Description

Gets the environment variable parameters structure, which is the envp.

Restrictions

This routine returns a pointer to the internal environment array pointer. Do not change or free any data in this array, Use as a read only resource only.

G.2.5. kprog_get_program() — *gets the name of the program*

Synopsis

```
char *kprog_get_program(void)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Gets the name of the program. This is usually the basename of the first argument in the argv structure.

Restrictions

This routine returns a pointer to the private string used to store the program variable. Use this as a read only resource; do NOT change or free this pointer.

G.2.6. kprog_get_toolbox() — *gets the toolbox in which this program belongs.*

Synopsis

```
char *kprog_get_toolbox(void)
```

Returns

the current toolbox or NULL if not initialized

Description

Gets the name of the toolbox in which this program belongs.

Restrictions

This routine returns a pointer to the private string used to store the toolbox variable. Use this as a read only resource; do NOT change or free this pointer.

G.3. Definitions of Utilities To Set Program Statistics

G.3.1. kprog_set_argc() — *set the number of commandline parameters*

Synopsis

```
int kprog_set_argc(
   int argc)
```

Input Arguments

`argc`

The current list in which we will be adding the

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Sets the number of arguments in the program argv structure.

G.3.2. kprog_set_argv() — *set the command line argument array*

Synopsis

```
int kprog_set_argv(
   char *argv[])
```

Input Arguments

`argv`

The array to set the program argv list to.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Set the argv list that the program uses.

G.3.3. kprog_set_envp() — *set the number of environment variable parameters*

Synopsis

```
int kprog_set_envp(
   char *envp[])
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Sets the number of environment variable parameters structure from the program envp structure.

G.3.4. kprog_set_program() — *set the name of the program*

Synopsis

```
int kprog_set_program(
   const char *program)
```

Input Arguments

`program`

The current name of this program

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Sets the name of the program. This is usually the basename of the first argument in the argv structure.

G.3.5. kprog_set_toolbox() — *set the toolbox in which this software object belongs.*

Synopsis

```
int kprog_set_toolbox(
   const char *toolbox)
```

Input Arguments

`toolbox`
: toolbox name in which the program exists

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Sets the name of the toolbox in where the software belongs.

Side Effects

the toolbox name is also placed into the environment variable list so that "$TOOLBOX" will be defined.

G.4. Definitions of Utilities To Initialize VisiQuest

G.4.1. khoros_initialize() — *initialize khoros system (old version for Khoros 2.1p1)*

Synopsis

```
void khoros_initialize(
   int argc,
   char **argv,
   char *toolbox)
```

Input Arguments

`argc`
: the number of arguments on the command line

`argv`
: the list of command line arguments

`toolbox`
: the name of the toolbox the program is associated with.

Description

Initializes some khoros global variables used in the khoros system. Old version for Khoros 2.1p1 and earlier, this routine will be obsolete in the next release; it is provided here for backwards compatibility only. Initializes the following global variables:

G.4.2. khoros_init() — *initialize khoros system (new version for Khoros 2.1p2)*

Synopsis

```
void khoros_init(
#ifdef KOPSYS_WIN32
   void (*pkexitfunc) PROTO((int)),
#endif
   int argc,
   char **argv,
   char *toolbox,
   char *date,
   char *product,
   char *version,
   char *major,
   char *minor,
   char *path)
```

Input Arguments

`argc`
: the number of arguments on the command line

`argv`
: the list of command line arguments

`toolbox`
: the name of the toolbox the program is associated with.

`date`
: the product release date

`product`
: the product release name

`version`
: the product version number

`major`
: the product major release number

`minor`
: the product minor release number

`path`
: the path to the application home directory

Description

Initializes some khoros global variables used in the khoros system. Initializes the following global

variables:

G.4.3. khoros_imprint() — *imprint the khoros toolbox*

Synopsis

```
int khoros_imprint(void)
```

Description

Imprint license information into the khoros toolbox.

H. The String Parser

H.1. Introduction to String Parser

This section contains a set of general string and VisiQuest data transport utilities that do regular expression *string parsing*. The regular expression syntax understood by these parsing routines is a subset of the syntax used by the *sed* and *awk* parsers. These routines are wrappers around the VisiQuest Regular Expression parser detailed in the next section of this chapter. All keys used by these routines are expected to be keys under stood by *kre_comp* and *kre_icomp*.

The routines included in these string utilities are:

- *kparse_string_search_delimit()* - break up a line data into an array of strings based on some set of delimiters
- *kparse_string_delimit()* - break a string into an array of strings based on some set of delimiters.
- *kparse_string_search()* - match a search key in a data string
- *kparse_string_scan()* - scan a data string for a specific section
- *kparse_string_scan_delimit()* - Break a string into an array of strings
- *kparse_file_search()* - search a file for a specific key
- *kparse_file_search_delimit()* - break up a line data into an array of strings based on some set of delimiters
- *kparse_file_scan()* - scan a VisiQuest Data Transport Stream for a specific section of data
- *kparse_file_scan_delimit()* - break a section of a VisiQuest Data Transport Stream into an array of strings

H.2. Definitions of String Parsing Utilities

H.2.1. kparse_string_search_delimit() — *break up a line data into an array of strings based on some set of delimiters*

Synopsis

```
char **kparse_string_search_delimit(
   char *data,
   char *key,
   int mode,
   char *delimiters,
   char *cont,
   char **key_format,
   ssize_t *num,
   int *status)
```

Input Arguments

`data`
input string to search and delimit.

`key`
regular expression key to search for.

`mode`
tells the parser which mode to work in:

```
KIGNORE_CASE     Ignore case on a-z and A-Z
KLITERAL         Use case on a-z and A-Z
```

`delimiters`
a string containing the delimiter characters.

`cont`
a string containing the continuation characters.

Output Arguments

`key_format`
the address of a pointer to hold the returned key that was matched. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key matched will not be returned.

`num`
returns the number of items in the array, -1 on error

`status`
error status of the search. It can be one of the following:

```
KPARSE_OK          (parse ok, return data valid)
```

```
KPARSE_NOKEY      (couldn't find key)
KPARSE_PARTKEY    (data ends on a partial match)
KPARSE_NULLKEY    (key was NULL)
KPARSE_SYNTAXKEY  (key had an illegal syntax)
```

Returns

This routine returns a pointer to an array of items we just broke apart from the input data string. NULL is returned when the kparse_string_search() or kparse_string_delimit() fails.

Description

This routine looks for a user specified key with a call to kparse_string_search(); then, it parses the rest of the line, up to a '\0, according to the user's list of delimiters by calling kparse_string_delimit(). The user can specify a set of continuation characters. Continuation characters must appear as the last character of the line. Line continuation can be chained together at the end of each consecutive line you want this routine to parse as a single line.

Side Effects

This routine mallocs data and sets the value of key_format parameter. Thus, the user should pass in an address of an unused character pointer. The calling routine is responsible for freeing the space malloc'ed for the key_format parameter.

This routine creates a new array of strings, and the calling routine is responsible for freeing the space allocated while creating the array via a call to karray_free().

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

Search keys and data strings should not contain the values '\001', '\002', '\003', or '\004', because these values are used as special search parameters by the parser.

H.2.2. kparse_string_delimit() — *break a string into an array of strings based on some set of delimiters.*

Synopsis

```
char **kparse_string_delimit(
   char *data,
   char *delimiter,
   int mode,
   ssize_t *num)
```

Input Arguments

`data`
: the data string to delimit

`delimiter`
: the list of delimiters to use

`mode`
: the delimit mode

```
KLITERAL     - delimit on delimiters, leave
               extra whitespace, and leave
               NULL entries in the array
               if two delimiters are next
               to each other.
KDELIM_CLEAN - eliminate whitespace on
               strings, and ignore two
               delimiters next to each other
```

Output Arguments

`num`
: the number of items in the list

Returns

This routine returns a pointer to an array of items we just broke apart from the input data string.

Description

This routine parses the input data string according to a set of single character delimiters. These delimiters can be escaped by a '\' inside of the data string if the delimiter must appear as an item; hence, you cannot use a '\' as a delimiter. The new array is created via calls to the karray_add() library call, and a pointer to the new array is returned back to the calling routine

Side Effects

This routine creates a new array of strings, and the calling routine is responsible for freeing the space allocated while creating the array via a call to karray_free().

H.2.3. kparse_string_search() — *match a search key in a data string*

Synopsis

```
char * kparse_string_search(
   char *data,
   char *key,
   int mode,
   char **key_format,
   int *status)
```

Input Arguments

`data`
: the data string to search through

`key`
: the regular expression key to search for

`mode`
: tells the parser which mode to work in:

```
KIGNORE_CASE Ignore case on a-z and A-Z
KLITERAL     Use case information on a-z and A-Z
```

Output Arguments

`key_format`
: the address of a pointer to hold the returned key that was matched. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key matched will not be returned.

`status`
: error status of the search. It can be one of the following:

```
KPARSE_OK        (parse ok, return data valid)
KPARSE_NOKEY     (couldn't find key)
KPARSE_PARTKEY   (data ends on a partial match)
KPARSE_NULLKEY   (key was NULL)
KPARSE_SYNTAXKEY (key had an illegal syntax)
```

Returns

This function returns a character pointer set to the address of the character following the matched search key. If an error occured, NULL is returned and status is set to the appropriate error status.

Description

This routine performs a regular expression search of an input data string for the first occurance of a specific search string. The search string is specified as a unix regular expression string (see syntax listed below), and it returns a pointer to the portion of the string that follows the search string. It also returns the exact format of the search string that the regular expression has matched. The formatted return string is kmalloc'ed for you.

Side Effects

This routine mallocs data and sets the value of key_format parameter. Thus, the user should pass in an address of an unused character pointer. The calling routine is responsible for freeing the space malloc'ed for the key_format parameter.

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

Search keys and data strings should not contain the values '\001', '\002', '\003', or '\004', because these values are used as special search parameters by the parser.

H.2.4. kparse_string_scan() — *scan a data string for a specific section*

Synopsis

```
char *kparse_string_scan(
    char *data,
    char *key1,
    char *key2,
    int mode,
    char **key1_format,
    char **key2_format,
    int *status)
```

Input Arguments

`data`
: the data string to search through

`key1`
: the regular expression begin key to search for

`key2`

the regular expression end key to search for

`mode`

tells the parser which mode to work in:

```
KIGNORE_CASE   Ignore case on a-z and A-Z
KLITERAL       Use case information on a-z and A-Z
```

Output Arguments

`key1_format`

the address of a pointer to hold the returned begin key that was matched. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key1 matched will not be returned.

`key2_format`

the address of a pointer to hold the returned end key that was matched. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key2 matched will not be returned.

`status`

error status of the search. It can be one of the following:

```
KPARSE_OK          (parse ok, return data valid)
KPARSE_NOKEY       (couldn't find begin key)
KPARSE_NOEND       (couldn't find end key)
KPARSE_DATAERR     (data string was invalid)
KPARSE_PARTKEY     (data ended with partial match)
KPARSE_PARTEND     (data ended with partial match on end key)
KPARSE_NULLKEY     (key was NULL)
KPARSE_NULLEND     (end key was NULL)
KPARSE_SYNTAXKEY   (key had an illegal syntax)
KPARSE_SYNTAXEND   (end key had an illegal syntax)
```

Returns

This routine returns a pointer to a malloc'ed string containing the text between the two matched keys. If an error occured during the search, it will return NULL, and the error status is set appropriately.

Description

This routine finds a section of data, marked by begin and end keys, out of a data string. When the begin and end keys are matched, this routine allocates a string big enough to hold the data between the keys, and then copies the data into that space. A pointer to this string is then returned by this routine. This routine makes calls to kparse_string_search() which handles the regular expression searching for the begin and end match keys.

Side Effects

This routine mallocs data and sets the value of the key1_format and key2_format parameters. Thus,

the user should pass in addresses of an unused character pointers for them. The calling routine is responsible for freeing the space malloc'ed for the key1_format and key2_format parameters.

This routine mallocs the space for the return string; and hence, is responsible for freeing the that space via kfree_and_NULL() when they are done with it.

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

H.2.5. kparse_string_scan_delimit() — *Break a string into an array of strings*

Synopsis

```
char **kparse_string_scan_delimit(
   char *data,
   char *key1,
   char *key2,
   int mode,
   char *delimiters,
   char **key1_format,
   char **key2_format,
   ssize_t *num,
   int *status)
```

Input Arguments

`data`
: the data string to search through

`key1`
: the regular expression begin key to search for

`key2`
: the regular expression end key to search for

`mode`
: tells the parser which mode to work in:

```
KIGNORE_CASE (match reguardless of case)
KLITERAL     (match with case)
```

`delimiters`
: a string containing the delimiter characters.

Output Arguments

`key1_format`

the address of a pointer to hold the returned begin key that was matched. If key1 was KPARSE_BOF, this address will set to NULL. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key1 matched will not be returned.

`key2_format`

the address of a pointer to hold the returned end key that was matched. If key2 was KPARSE_EOF, this address will be set to NULL. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key2 matched will not be returned.

`num`

returns the number of items in the array, -1 on error

`status`

error status of the search. It can be one of the following:

```
KPARSE_OK         (parse ok, return data valid)
KPARSE_NOKEY      (couldn't find begin key)
KPARSE_NOEND      (couldn't find end key)
KPARSE_DATAERR    (data string was invalid)
KPARSE_PARTKEY    (data ended with partial match)
KPARSE_PARTEND    (data ended with partial match on end key)
KPARSE_NULLKEY    (key was NULL)
KPARSE_NULLEND    (end key was NULL)
KPARSE_SYNTAXKEY (key had an illegal syntax)
KPARSE_SYNTAXEND (end key had an illegal syntax)
```

Returns

This routine returns a pointer to an array of items that were just broke apart from the input data string. NULL is returned when the kparse_string_scan() or kparse_string_delimit() fails.

Description

This routine looks in a data string for an area of text between two user specified match keys, then it delimits the section of text in to an array of smaller strings based on a set of character delimiters. Delimiters can be escaped by a '\' if they need to appear in the text section. This routine is a combination of the calls kparse_string_scan() and kparse_string_delimit(). This routine cleans up the entries in the array via a kstring_cleanup() call.

Side Effects

This routine mallocs data and sets the value of the key1_format and key2_format parameters. Thus, the user should pass in addresses of an unused character pointers for them. The calling routine is responsible for freeing the space malloc'ed for the key1_format and key2_format parameters.

This routine mallocs the space for the return string; and hence, is responsible for freeing the that space via kfree_and_NULL() when they are done with it.

This routine creates a new array of strings, and the calling routine is responsible for freeing the space allocated while creating the array via a call to karray_free().

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

Search keys and the data file should not contain the values '\001', '\002', '\003', or '\004', because these values are used as special search parameters by the parser.

H.2.6. kparse_file_search() — *search a file for a specific key*

Synopsis

```
int kparse_file_search(
   kfile *file,
   char *key,
   int mode,
   char **key_format)
```

Input Arguments

`file`
: a pointer to the transport file opened for reading

`key`
: The regular expression key to search for.

`mode`
: tells the parser which mode to work in:

```
KIGNORE_CASE  Ignore case on a-z and A-Z
KLITERAL      Use case information on a-z and A-Z
```

Output Arguments

`key_format`
: the address of a pointer to hold the returned key that was matched. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key matched will not be returned.

Returns

return status of the search. It can be one of the following:

```
KPARSE_OK          (parse ok, return data valid),
```

```
KPARSE_NOKEY       (couldn't find key)
KPARSE_DATAERR     (something was wrong with the data)
KPARSE_PARTKEY     (data ended with a partial match)
KPARSE_NULLKEY     (key was NULL)
KPARSE_SYNTAXKEY   (key had an illegal syntax)
```

Description

This routine is a VisiQuest Transport Interface to the string parser provided in the kparse_string_search routine. It returns a malloc'ed copy of the key that was matched in the data file. It also sets the current file position, with the transport call kfseek(), to the character following the matched key.

Side Effects

This routine mallocs data and sets the value of key_format parameter. Thus, the user should pass in an address of an unused character pointer. The calling routine is responsible for freeing the space malloc'ed for the key_format parameter.

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

Search keys and the data file should not contain the values '\001', '\002', '\003', or '\004', because these values are used as special search parameters by the parser.

H.2.7. kparse_file_search_delimit() — *break up a line data into an array of strings based on some set of delimiters*

Synopsis

```
char **kparse_file_search_delimit(
   kfile *file,
   char *key,
   int mode,
   char *delimiters,
   char *cont,
   char **key_format,
   ssize_t *num,
   int *status)
```

Input Arguments

```
file
```

a pointer to the transport file opened for reading

`key`

The regular expression key to search for

`mode`

tells the parser which mode to work in:

```
KIGNORE_CASE   Ignore case on a-z and A-Z
KLITERAL       Use case information on a-z and A-Z
```

`delimiters`

a string containing the delimiter characters

`cont`

a string containing the continuation characters

Output Arguments

`key_format`

the address of a pointer to hold the returned key that was matched. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key matched will not be returned.

`num`

returns the number of items in the array, -1 on error

`status`

return status of the search. It can be one of the following:

```
KPARSE_OK         (parse ok, return data valid),
KPARSE_NOKEY      (couldn't find key)
KPARSE_PARTKEY    (data ended with a partial match)
KPARSE_NULLKEY    (key was NULL)
KPARSE_SYNTAXKEY (key had an illegal syntax)
KPARSE_DATAERR    (routine had a a data error)
```

Returns

This routine returns a pointer to an array of items we just broke apart from the input data string. NULL is returned when the kparse_file_search() or kparse_string_delimit() fails.

Description

This routine looks for a user specified key with a call to kparse_file_search(); then, it reads in the rest of the line into a string, up to a '\0. Once the line is read in, it breaks up the line according to the user's list of delimiters by calling kparse_string_delimit(). The user can specify a set of continuation characters. Continuation characters must appear as the last character of the line. Line continuation can be chained together at the end of each consecutive line you want this routine to parse as a single line.

Side Effects

This routine mallocs data and sets the value of key_format parameter. Thus, the user should pass in an address of an unused character pointer. The calling routine is responsible for freeing the space malloc'ed for the key_format parameter.

This routine creates a new array of strings, and the calling routine is responsible for freeing the space allocated while creating the array via a call to karray_free().

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

Search keys and the data file should not contain the values '\001', '\002', '\003', or '\004', because these values are used as special search parameters by the parser.

H.2.8. kparse_file_scan() — *scan a VisiQuest Data Transport Stream for a specific section of data*

Synopsis

```
char *kparse_file_scan(
   kfile *file,
   char *key1,
   char *key2,
   int mode,
   char **key1_format,
   char **key2_format,
   int *status)
```

Input Arguments

`file`
: a pointer to an open file pointer

`key1`
: the regular expression begin key to search for. If this key is the #define KPARSE_BOF, it will use a file offset of 0 for the returned data's starting point.

`key2`
: the regular expression end key to search for. If this key is the #define KPARSE_EOF, it will set the file offset to the end of the file for the returned data's ending point.

`mode`
: tells the parser which mode to work in:

```
KIGNORE_CASE  Ignore case on a-z and A-Z
KLITERAL Use case information on a-z and A-Z
```

Output Arguments

`key1_format`

the address of a pointer to hold the returned begin key that was matched. If key1 was KPARSE_BOF, this address will set to NULL. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key1 matched will not be returned.

`key2_format`

the address of a pointer to hold the returned end key that was matched. If key2 was KPARSE_EOF, this address will be set to NULL. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key2 matched will not be returned.

`status`

error status of the search. It can be one of the following:

```
KPARSE_OK          (parse ok, return data valid),
KPARSE_NOKEY       (couldn't find begin key)
KPARSE_NOEND       (couldn't find end key)
KPARSE_DATAERR     (data string was invalid)
KPARSE_PARTKEY     (data ended with partial match)
KPARSE_PARTEND     (data ended with partial match on end key)
KPARSE_NULLKEY     (key was NULL)
KPARSE_NULLEND     (end key was NULL)
KPARSE_SYNTAXKEY   (key had an illegal syntax)
KPARSE_SYNTAXEND   (end key had an illegal syntax)
```

Returns

This routine returns a pointer to a malloc'ed string containing the text between the two matched keys. If an error occured during the search, it will return NULL, and the error status is set appropriately. This current file position is set to the

Description

This routine finds a section of data, marked by begin and end keys, out of a VisiQuest Data Transport Stream that was opened for input. When the begin and end keys are matched, this routine allocates a string big enough to hold the data between the keys, and then copies the data into that space. A pointer to this string is then returned. This routine makes calls to kparse_file_search() which handles the regular expression parsing for the begin and end match keys. On a successful search, the current position in the VisiQuest Data Transport Stream will be set to the character directly following the last character matched by the end key.

Side Effects

This routine mallocs data and sets the value of the key1_format and key2_format parameters. Thus, the user should pass in addresses of an unused character pointers for them. The calling routine is responsible for freeing the space malloc'ed for the key1_format and key2_format parameters.

This routine mallocs the space for the return string; and hence, is responsible for freeing the that space via kfree_and_NULL() when they are done with it.

Restrictions

It does not support the following regular expression constructs: or'ing '|', grouping of regular expressions '()', match one or more times '+', or match n to m times '\{n,m\}'. Finally, the '\0mber' and '\(\)' constructs have no meaning for these routines, so they are not supported either.

H.2.9. kparse_file_scan_delimit() — *break a section of a VisiQuest Data Transport Stream into an array of strings*

Synopsis

```
char **kparse_file_scan_delimit(
   kfile *file,
   char *key1,
   char *key2,
   int mode,
   char *delimiters,
   char **key1_format,
   char **key2_format,
   ssize_t *num,
   int *status)
```

Input Arguments

`file`
: a pointer to an open file pointer

`key1`
: the regular expression begin key to search for if this key is the #define KPARSE_BOF, it will make file offset 0 the returned data starting point.

`key2`
: the regular expression end key to search for if this key is the #define KPARSE_EOF, it will make file offset at the end of the file the returned data ending point.

`mode`
: tells the parser which mode to work in:

```
KIGNORE_CASE   Ignore case on a-z and A-Z
KLITERAL       Use case information on a-z and A-Z
```

`delimiters`
: a string containing the delimiter characters.

Output Arguments

`key1_format`
: the address of a pointer to hold the returned begin key that was matched. If key1 was KPARSE_BOF, this address will set to NULL. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the

string that key1 matched will not be returned.

`key2_format`

the address of a pointer to hold the returned end key that was matched. If key2 was KPARSE_EOF, this address will be set to NULL. Sufficient space for the returned string will be allocated if you pass in a valid pointer. Note that if this parameter is passed in as NULL, this routine will ignore it, and the string that key2 matched will not be returned.

`num`

returns the number of items in the array, -1 on error

`status`

error status of the search. It can be one of the following:

```
KPARSE_OK         (parse ok, return data valid)
KPARSE_NOKEY      (couldn't find begin key)
KPARSE_NOEND      (couldn't find end key)
KPARSE_DATAERR    (data string was invalid)
KPARSE_PARTKEY    (data ended with partial match)
KPARSE_PARTEND    (data ended with partial match on end key)
KPARSE_NULLKEY    (key was NULL)
KPARSE_NULLEND    (end key was NULL)
KPARSE_SYNTAXKEY (key had an illegal syntax)
KPARSE_SYNTAXEND (end key had an illegal syntax)
```

Returns

an array of of strings that were delimited.

Description

This routine looks for an area of text between two user specified keys in a VisiQuest Data Transport Stream; then, copies the section into a string, and parses it according to the user's set of delimiters. Delimiters can be escaped by a '\' character if a delimiter must appear in the text. This routine is a combination of the routines kparse_file_scan() and kparse_string_delimit(). This routine cleans up the strings to be returned via the kstring_cleanup() routine.

Side Effects

This routine mallocs data and sets the value of the key1_format and key2_format parameters. Thus, the user should pass in addresses of an unused character pointers for them. The calling routine is responsible for freeing the space malloc'ed for the key1_format and key2_format parameters.

This routine mallocs the space for the return string; and hence, is responsible for freeing the that space via kfree_and_NULL() when they are done with it.

This routine creates a new array of strings, and the calling routine is responsible for freeing the space allocated while creating the array via a call to karray_free().

I. Regular Expression Pattern Matching & Replacement

I.1. Introduction to Regular Expression Utilities

VisiQuest includes the *regex* package, a public domain set of routines for regular expression pattern matching and replacement provided by Ozan S. Yigit (oz) of the Dept. of Computer Science at York University. The following excerpt is from the header of the *regex.c* file:

```
/*
 * regex - Regular expression pattern matching
 *         and replacement
 *
 *
 * By:  Ozan S. Yigit (oz)
 *      Dept. of Computer Science
 *      York University
 *
 *
 * These routines are the PUBLIC DOMAIN equivalents
 * of regex routines as found in 4.nBSD UN*X, with minor
 * extensions.
 *
 * These routines are derived from various implementations
 * found in software tools books, and Conroy's grep. They
 * are NOT derived from licensed/restricted software.
 * For more interesting/academic/complicated implementations,
 * see Henry Spencer's regexp routines, or GNU Emacs pattern
 * matching module.
 *
 *
 * Acknowledgments:
 *
 *      HCR's Hugh Redelmeier has been most helpful in various
 *      stages of development. He convinced me to include BOW
 *      and EOW constructs, originally invented by Rob Pike at
 *      the University of Toronto.
 *
 * References:
 *              Software tools                  Kernighan & Plauger
 *              Software tools in Pascal        Kernighan & Plauger
 *              Grep [rsx-11 C dist]            David Conroy
 *              ed - text editor                Un*x Programmer's Manual
 *              Advanced editing on Un*x        B. W. Kernighan
 *              RegExp routines                 Henry Spencer
 *
 * Notes:
 *
 *      This implementation uses a bit-set representation for character
 *      classes for speed and compactness. Each character is represented
 *      by one bit in a 128-bit block. Thus, CCL or NCL always takes a
 *      constant 16 bytes in the internal dfa, and re_exec does a single
 *      bit comparison to locate the character in the set.
 *
 * Examples:
 *
```

```
*      pattern:        foo*.*
*      compile:        CHR f CHR o CLO CHR o END CLO ANY END END
*      matches:        fo foo fooo foobar fobar foxx ...
*
*      pattern:        fo[ob]a[rz]
*      compile:        CHR f CHR o CCL 2 o b CHR a CCL bitset END
*      matches:        fobar fooar fobaz fooaz
*
*      pattern:        foo\+
*      compile:        CHR f CHR o CHR o CHR \ CLO CHR \ END END
*      matches:        foo\ foo\\ foo\\\  ...
*
*      pattern:        \(foo\)[1-3]\1  (same as foo[1-3]foo)
*      compile:        BOT 1 CHR f CHR o CHR o EOT 1 CCL bitset REF 1 END
*      matches:        foo1foo foo2foo foo3foo
*
*      pattern:        \(fo.*\)-\1
*      compile:        BOT 1 CHR f CHR o CLO ANY END EOT 1 CHR - REF 1 END
*      matches:        foo-foo fo-fo fob-fob foobar-foobar ...
*
*/
```

It should be noted that the original code has been heavily modified to include some new features. Please see the table below for a complete list of the regular expressions understood by this regular expression parser.

Regular Expression Special Symbols

Symbol	Description of Symbol
.	Match any single character except newline
^	If this is the first character of the regular expression, it matches the beginning of the line.
$	If this is the last character of the regular expression, it matches the end of the line.
[...] or [^..]	Matches any one character contained within the brackets. If the first character after the '[' is the ']', then it is included in the characters to match. If the first character after the '[' is a '^', then it will match all characters NOT included in the []. The '-' will indicate a range of characters. For example, [a-z] specifies all characters between and including the ASCII values 'a' and 'z'. If the ´-' follows the '[' or is right before the ']' then it is interpreted literally. There are special symbols that can be used as short hand: \w will expand to '0-9a-z_A-Z', \d expands to '0-9', and \s expands to ' \t\n\r\f'
*****	Match the preceding regular expression 0 or more times. The matching includes items within a [...] or (...).
+	Match the preceding character or range of characters 1 or more times. The matching includes items within a [...] or (...).

Regular Expression Special Symbols	
Symbol	Description of Symbol
(..)	Tagged boundary region. This pattern indicates the begin and end of a tagged region in the regular expression. It can be used to match the exact same pattern later in the regular expression via a reference \1 through \127. It can also be used in kre_subs to substitute the substring that was matched by this part of the regular expression. The final use of the (..) notation is for grouping of the or'ing function. This allows a pattern like (regexp1|regexp2|regexp3). As indicated by the example, you can nest grouping if necessary. Only 127 of these tagged regions are allowed in the regular expression.
\b	Word boundary. This pattern will match the empty char before the start and after the end of a word. By default, a word character contains 0-9a-z_A-Z. This can be modified by the kre_modw routine.
\B	Non-word boundary. This pattern will match the empty character between two characters in a word.
\1-\127	These symbols are used to reference the 1st through 9th \(\) region.
\h	Stored the ASCII character '\b' since \b is used to define word boundary (See above).
\A	If it is the first character of the regular expression, it matches the empty character at the beginning of the string.
\Z	If it is the last character of the regular expression, it matches the empty character at the end of the string.
\c@-\cZ	These symbols are translated into control-@ through control-Z. Any other values, and the \c part is ignored.
\d	Same as [0-9].
\D	Same as [^0-9].
\s	Same as [\t\n\r\f].
\S	Same as [^ \t\n\r\f].
\w	Same as [a-zA-Z_0-9].
\W	Same as [^a-zA-Z_0-9].
\Q..\E	A section enclosed in these symbols it taken literally. Inside these sections, meta characters and special symbols have no meaning. If a \E needs to appear in one of these sections, the \ must be escaped with a \.
\	This escapes the meaning of a special character.

Table 1: This table describes all the special characters supported by the parser. All other characters are matched directly.

The *regex* function that have been integrated into VisiQuest carry the *k* prefix in their names. They are as follows:

• *kre_comp()* - compile a regular expression
• *kre_debug()* - prints a DFA for debug purposes

• *kre_exec()* - execute dfa to find a match.
• *kre_icomp()* - compile a case insensitive regular expression
• *kre_modw()* - modify kre_exec's work table
• *kre_pos()* - begin and end pointers of regular expression group
• *kre_status()* - return a parse status code
• *kre_subs()* - substitute the matched portions of the src in dst
• *kregex_replace()* - replace an input string given a regular expression input and output string

I.2. Definitions of Regular Expression Utilities

I.2.1. kre_comp() — *compile a regular expression*

Synopsis

```
char *kre_comp(

    char *pat)
```

Input Arguments

`pat`
regular expression pattern to be compiled

Returns

NULL on success, a string indicating the error otherwise

Description

Compiles a regular expression.

```
.              Match any single character except newline

^              If this is the first character of the
               regular expression, it matches the beginning
               of the line.

$              If this is the last character of the
               regular expression, it matches the end of
               the line.

[...] or [^..]  Matches any one character contained within
               the brackets. If the first character after
               the '[' is the ']', then it is included in
               the characters to match.  If the first
               character after the '[' is a '^', then it
               will match all characters NOT included in
```

the []. The '-' will indicate a range of characters. For example, [a-z] specifies all characters between and including the ascii values 'a' and 'z'. If the '-' follows the '[' or is right before the ']' then it is interpreted literally. There are special symbols that can be used as short hand: \w will expand to '0-9a-z_A-Z', \d expands to '0-9', and \s expands to ' \t\0r\f'

{n,m} Match between n and m times the DFA directly
{n,} before this range syntax. Thus, 'a{2,10}'
{n} will match a minimum of 2 a's and a maximum 10. The {n,} syntax tells the parser to match n or more times., and the {n} syntax tells it to match exactly n times.

* Match the preceding character or range of characters 0 or more times. This is equivalent to the range syntax {0,}

+ Match the preceding character or range of characters 1 or more times. This is equivalent to the range syntax {1,}

? Match the preceding character or range of characters 0 or 1 times. This is equivalent to the range syntax {0,1}

| This symbol is used to indicate where to separate two sub regular expressions for a logical OR operation.

(..) Group boundaries. This pattern indicates an area of a memory tagged region of the regular expression that can be used to match the exact same pattern later in the regular expression via a reference, or used in kre_subs to bring in this part of the matched string. It can also be used to indicate areas where the or symbol '|' should be applied. Note, only 127 groups are allowed.

\b Word boundary. This pattern will match the empty char before the start and after the end of a word. By default, a word character contains 0-9a-z_A-Z. This can be modified by the kre_modw routine.

\B Non-word boundary. This pattern will match the empty character between two characters

	in a word.
\1-\127	These symbols are used to reference the 1st through 127th () region.
\h	Stored the ASCII character '\b'
\A	If it is the first character of the regular expression, it matches the empty character at the beginning of the string.
\Z	If it is the last character of the regular expression, it matches the empty character at the end of the string.
\c@-\cZ	These symbols are translated into control-@ through control-Z. Any other values, and the \c part is ignored.
\d	Same as [0-9].
\D	Same as [^0-9].
\s	Same as [\t\0r\f].
\S	Same as [^ \t\0r\f].
\w	Same as [a-zA-Z_0-9].
\W	Same as [^a-zA-Z_0-9].
\Q..\E	A section enclosed in these symbols it taken literally. In side these sections, meta characters and special symbols have no meaning. If a \E needs to appear in one of these sections, the \ must be escaped with \.
\	This escapes the meaning of a special character.

I.2.2. kre_debug() — *prints a DFA for debug purposes*

Synopsis

```
void kre_debug(
```

```
char  *s,
kfile *out)
```

Input Arguments

`s`
: the pattern to be debugged

`out`
: output kfile pointer to write to. It defaults kstdout if out is NULL

Description

Prints a dfa to kstdout for debugging purposes.

I.2.3. kre_exec() — *execute dfa to find a match.*

Synopsis

```
int kre_exec(

    char *lp)
```

Input Arguments

`lp`
: string to exec the DFA on

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Execute DFA to match a pattern.

special cases: (dfa[0])

```
BOS - Match only once, starting from the beginning of the string.
CHR - First locate the character without calling pmatch, and if
      found, call pmatch for the remaining string.
END - re_comp failed, poor luser did not check for it. Fail fast.
```

If a match is found, bopat[0] and eopat[0] are set to the beginning and the end of the matched fragment, respectively.

Restrictions

kre_comp or kre_icomp must have been called previously to calling this routine.

I.2.4. kre_icomp() — *compile a case insensitive regular expression*

Synopsis

```
char *kre_icomp(

    char *pat)
```

Input Arguments

`pat`

regular expression pattern to be compiled

Returns

NULL on success, a string indicating the compile error otherwise

Description

This routine is an extension to kre_comp, such that the characters a-z and A-z are treated as though they were all of the same case.

I.2.5. kre_modw() — *modify kre_exec's work table*

Synopsis

```
void kre_modw(

    char *s)
```

Input Arguments

`s`

input string

Description

Adds new characters into the word table to change the re_exec's understanding of what a word should look like. Note that we only accept additions into the word definition.

If the string parameter is 0 or null string, the table is reset back to the default, which contains A-Z a-z

0-9 _.

Restrictions

This routine only allows you to add characters to the table, or reset to the default table. It canot delete characters from the table.

I.2.6. kre_pos() — *begin and end pointers of regular expression group*

Synopsis

```
int kre_pos(

    int  num,
    kaddr *begin,
    kaddr *end)
```

Input Arguments

`num`

the actual desired group number

Output Arguments

`begin`

if not NULL then returns the actual pointer to the beginning of the group

`end`

if not NULL then returns the actual pointer to the end of the group

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Returns the start and end pointers to a regular expression group. This returns the beginning and end pointers of the matched pattern. The num parameter indicates which tag field to return, 0 implies that it should return pointers to the whole matched pattern

I.2.7. kre_status() — *return a parse status code*

Synopsis

```
int kre_status(void)
```

Returns

error status of the search. It can be one of the following:

```
KPARSE_OK          (parse ok, return data valid),
KPARSE_NOKEY       (couldn't find begin key)
KPARSE_NOEND       (couldn't find end key)
KPARSE_DATAERR     (data string was invalid)
KPARSE_PARTKEY     (data ended with partial match)
KPARSE_PARTEND     (same as above, but on end key)
KPARSE_NULLKEY     (key was NULL)
KPARSE_NULLEND     (end key was NULL)
KPARSE_SYNTAXKEY   (key had an illegal syntax)
KPARSE_SYNTAXEND   (end key had an illegal syntax)
```

Description

Returns an integer status code that can be used by other library routines to determine the error that occured during the parse.

I.2.8. kre_subs() — *substitute the matched portions of the src in dst*

Synopsis

```
int kre_subs(

    char *src,
    char *dst)
```

Input Arguments

`src`

source string

Output Arguments

`dst`

destination string

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Substitutes the matched portions of the source string in the in destination string.

```
'&'      - substitute the entire matched pattern.
'\digit' - substitute a subpattern, with the given
           tag number. Tags are numbered from 1 to
           127. If the particular tagged subpattern
           does not exist, null is substituted.
```

Other symbols can be used to modify the substitution patterns:

```
\Q..\E - Ignore all special characters between the
         \Q and \E symbols
\l     - Convert next character to lower case.
\L..\E - Convert all text between \L and \E to lower
         case
\u     - Convert next character to upper case.
\U..\E - Convert all text between \U and \E to upper
         case
```

Restrictions

kre_exec must be called before this routine will do any substitutions.

I.2.9. kregex_replace() — *replace an input string given a regular expression input and output string*

Synopsis

```
char * kregex_replace(
   const char *istr,
   const char *scan_pat,
   const char *replace_pat,
   const int   icase,
   char *ostr)
```

Input Arguments

`istr`
: the input string

`scan_pat`
: the source pattern to subsititute on

`replace_pat`
: the destination pattern to subsititute for

`icase`
TRUE if you want to ignore case while searching for the search pattern

Output Arguments

`ostr`
the output string

Returns

NULL on failure, the newly replaced string on success

Description

Replace an input string given a regular expression input and output string. This routine is used to do regular expression substitution given the source and destination patterns as replacement. So for example if the input string is:

Old Man in an Old Boat

and the source pattern is:

^Old

and the replace pattern is:

Young

then the output would be:

Young Man in an Old Boat

J. Memory Allocation Utilities

J.1. Introduction to Memory Utilities

The VisiQuest utilities *kcalloc(), kmalloc(), krealloc(),* and *kfree()* are more robust than their *libc* counterparts. These utilities check for requests to allocate 0 bytes.

- *kbcopy()* - copies bytes from src to dest
- *kbzero()* - zeros out 'num' bytes (BSD style)
- *kbcmp()* - compare bytes from src1 and src2 (BSD style)
- *kcalloc()* - allocate memory and initialize it
- *kdupalloc()* - duplicates a piece of memory
- *kfree()* - free allocated memory
- *kfree_and_NULL()* - free memory previously allocated
- *kmalloc()* - allocate a contiguous piece of memory

• *krealloc()* - re-allocate a piece of memory to a new size
• *kmemchr()* - find the first occurence of 'c' in an character array
• *kmemcmp()* - compare bytes from src1 and src2
• *kmemcpy()* - copies bytes from src to dest
• *kmemccpy()* - restricted copy of bytes from src to dest
• *kmemmove()* - copy a block of memory to another block
• *kmemset()* - initialize bytes in dest to the character value 'c'

J.2. Definitions of Memory Utilities

J.2.1. kbcopy() — *copies bytes from src to dest*

Synopsis

```
void kbcopy(

  char *src,
  char *dest,
  int  num)
```

Input Arguments

`src`
the source pointer to copy "num" bytes from

`num`
the number of bytes to be copied

Output Arguments

`dest`
the destination pointer to copy the bytes to

Description

This function is a replacement for the BSD system call bcopy(). The kbcopy() call copies "num" bytes from src to dest using kmemcpy(), which makes sure that overlapping strings are handled correctly.

J.2.2. kbzero() — *zeros out 'num' bytes (BSD style)*

Synopsis

```
void kbzero(

  char *dest,
```

```
  int  num)
```

Input Arguments

`num`
the number of bytes to be zero'ed

Output Arguments

`dest`
the destination pointer to be zero'ed

Description

This function is a replacement for the BSD system call bzero(). The kbzero() zeros out "num" bytes from ptr. kbzero() will really use kmemset() with a c value of 0, so it will be checked to make sure that the "dest" array is not NULL and "num" is greater than 0.

J.2.3. kbcmp() — *compare bytes from src1 and src2 (BSD style)*

Synopsis

```
void kbcmp(

  char *src1,
  char *src2,
  int  num)
```

Input Arguments

`src1`
the first source pointer to compare from

`src2`
the second source pointer to compare from

`num`
the number of bytes to be compared

Returns

0 if up to num characters in src1 equals the corresponding characters in src2. The positive difference of the first character that differs between src1 and src2 if src1 is lexicographically greater than src2. The negative difference of the first character that differs, if src2 is lexicographically greater than src1.

Description

This function is a replacement for the BSD system call bcmp() which does not exist on many systems. The kbcmp() routine compares "src1" and "src2" for "num" bytes. kbcmp() really uses kmemcmp(), so it will be checked to make sure that both source arrays are not NULL and "num" is greater than 0.

kbcmp() compares its arguments, looking at the first num bytes, and returns an integer less than, equal to, or greater than 0, according as src1 is lexicographically less than, equal to, or greater than src2.

J.2.4. kcalloc() — *allocate memory and initialize it*

Synopsis

```
kaddr kcalloc(
    size_t nelem,
    size_t elsize)
```

Input Arguments

`nelem`
: number of elements to calloc

`elsize`
: number of bytes in an element

Returns

The address of the allocated memory. NULL is returned if the allocation fails, or if either nelem or elsize are 0

Description

This macro calls calloc() to allocate the requested data. However, it checks nelem and elsize to make sure both are greater than 0. If either parameter is 0, then NULL is returned.

J.2.5. kdupalloc() — *duplicates a piece of memory*

Synopsis

```
kaddr kdupalloc(
    const kaddr src,
    size_t size)
```

Input Arguments

`src`
: the source pointer to duplicate

`size`
: the number of characters to duplicate from the source.

Returns

The newly duplicated memory or NULL upon failure.

Description

The routine allocates new memory and then copies the contents from the ptr to the new memory. This routine is really a convience routine that uses kmalloc() and kmemcpy() in which to make a duplicate.

J.2.6. kfree() — *free allocated memory*

Synopsis

```
void kfree(
   kaddr ptr)
```

Input Arguments

`ptr`

A pointer to memory to be freed.

Description

This routine calls free() to free previously allocated data. However, it also checks to make sure the pointer argument is not NULL.

J.2.7. kfree_and_NULL() — *free memory previously allocated*

Synopsis

```
(void) kfree(

  kaddr ptr)
```

Description

This function is a replacement for the khoros kfree() routine. The only difference is that kfree_and_NULL() is an inlined macro that will also NULL your memory pointer.

J.2.8. kmalloc() — *allocate a contiguous piece of memory*

Synopsis

```
kaddr kmalloc(
    size_t size)
```

Input Arguments

`size`
: number of bytes to allocate

Returns

a pointer to malloc'ed data on success, or NULL on size 0 or malloc failure

Description

Calls malloc() to allocate the requested data. However, it checks to make sure the size parameter is greater then zero. If the size paramter is 0, it returns NULL.

J.2.9. krealloc() — *re-allocate a piece of memory to a new size*

Synopsis

```
kaddr krealloc(
    kaddr ptr,
    size_t size)
```

Input Arguments

`ptr`
: the pointer to reallocate

`size`
: the number of characters to reallocate to.

Returns

The address of the allocated memory. NULL is returned if the allocation fails, or if size is 0. If ptr is NULL, malloc will be called instead of realloc.

Description

The macro calls realloc() to re-allocate the block of memory with size bytes. The realloc will automatically copy the data from the first memory block to the newly allocated one.

J.2.10. kmemchr() — *find the first occurence of 'c' in an character array*

Synopsis

```
kaddr kmemchr(
   const kaddr src,
   int c,
   size_t num)
```

Input Arguments

`src`
: the source pointer to be searched

`c`
: the character value to be searched for within src

`num`
: the number of bytes to be searched

Returns

pointer in which "c" occurs, NULL upon failure

Description

This function is the same as the system call memchr(). Except kmemchr() will make sure that the source and the the number of bytes to be searched are greater than 0.

memchr() searches the bytes in the "src" character array for the character value "c". The number of bytes to be searched is determined by the parameter "num". Upon loction of "c" in the source array, the pointer pointing to "c" within the source array is returned. If "c" does not occur within the first "num" bytes then NULL is returned.

J.2.11. kmemcmp() — *compare bytes from src1 and src2*

Synopsis

```
int kmemcmp(
   const kaddr src1,
   const kaddr src2,
   size_t num)
```

Input Arguments

`src1`
: the first source pointer to compare from

`src2`
: the second source pointer to compare from

`num`
the number of bytes to be compared

Returns

0 if up to num characters in src1 equals the corresponding characters in src2. The positive difference of the first character that differs between src1 and src2 if src1 is lexicographically greater than src2. The negative difference of the first character that differs, if src2 is lexicographically greater than src1.

Description

This function is the same as the system call memcmp(). Except kmemcmp() will make sure that the src1 and src2 pointers are not NULL before calling memcmp().

memcmp() compares its arguments, looking at the first num bytes, and returns an integer less than, equal to, or greater than 0, according as src1 is lexicographically less than, equal to, or greater than src2.

J.2.12. kmemcpy() — *copies bytes from src to dest*

Synopsis

```
kaddr kmemcpy(
   kaddr dest,
   const kaddr src,
   size_t num)
```

Input Arguments

`src`
the source pointer to copy "num" bytes from
`num`
the number of bytes to be copied

Output Arguments

`dest`
the destination pointer to copy the bytes to

Returns

dest on a successful memory copy. If dest or src are NULL, NULL is returned. If num is 0, NULL is returned.

Description

This routine copies num bytes of memory from src to dest in a similar manner as memcpy. However, this routine checks that the src, dest, and num values are not NULL or 0.

J.2.13. kmemccpy() — *restricted copy of bytes from src to dest*

Synopsis

```
kaddr kmemccpy(
   kaddr dest,
   const kaddr src,
   int c,
   size_t num)
```

Input Arguments

`src`
: the source pointer to copy "num" bytes from

`c`
: the character value to terminate copy

`num`
: the number of bytes to be copied

Output Arguments

`dest`
: the destination pointer to copy the bytes to

Returns

returns dest if "c" was encountered, NULL otherwise

Description

This function is the same as the system call memccpy(). Except kmemccpy() will make sure that the src and dest pointers are not NULL before calling memccpy().

memccpy() copies the bytes in the "src" character array to the "dest" array. The number of bytes to be copied is determined by the "c" character value and the parameter "num". Bytes will be copied from "src" to "dest" until the "c" character value is encountered or num bytes have been copied. If "c" is encountered and copied before "num" bytes then the "dest" array is returned. Otherwise if "num" bytes are copied then NULL is returned to indicate the "c" was not encountered.

J.2.14. kmemmove() — *copy a block of memory to another block*

Synopsis

```
kaddr kmemmove(
   kaddr dest,
   const kaddr src,
   size_t num)
```

Input Arguments

`src`
: the source pointer to move the data from

`num`
: the number of bytes to be moved

Output Arguments

`dest`
: the destination pointer to move the data to

Returns

returns the dest pointer on success, NULL otherwise

Description

This function is the same as the system call memmove(). Except kmemmove() will make sure that the source and the destination are not NULL.

memmove() copies n bytes from memory areas src to dest. Copying between objects that overlap will take place correctly. It returns dest.

J.2.15. kmemset() — *initialize bytes in dest to the character value 'c'*

Synopsis

```
kaddr kmemset(
   kaddr dest,
   int c,
   size_t num)
```

Input Arguments

`c`
: the character value to set in dest

`num`
: the number of bytes to be initialized

Output Arguments

`dest`
: the destination pointer to be initialized

Returns

returns dest on success, NULL upon failure

Description

This function is the same as the system call memset(). Except kmemset() will make sure that the dest

and the the number of bytes to be initialized are greater than 0.

memset() initializes the bytes in the "dest" character array to the character value "c". The number of bytes to be set is determined by the parameter "num".

K. File Path Utilities

K.1. Introduction to Path Utilities

The utilities below expand a path, expand a filename, strip a filename from a file path, strip the directory from a file path, get a name of a file in TMPDIR and so on. These utilities are used frequently throughout the VisiQuest system and are found in the *klibc (libku.a)* library.

• *kbasename()* - return the filename component of a pathname
• *kdirname()* - find directory component of a given pathname
• *kexpandpath()* - Expand a path to it <I>true</I> path
• *kfullpath()* - Expand a environment variables in a path
• *ktbpath()* - Expand a environment variables local to a toolbox
• *ktempnam()* - create a name for a temporary khoros transport
• *kfindpath()* - find the path to an executable

K.2. Definitions of Path Utilities

K.2.1. kbasename() — *return the filename component of a pathname*

Synopsis

```
char *kbasename(
   const char *pathname,
   char *return_base)
```

Input Arguments

`pathname`
A valid pathname

Output Arguments

`return_base`
if return_base is not NULL, then this will be where the result is placed. Otherwise, it will malloc the space required.

Returns

filename component of pathname, NULL on error

Description

This routine searches an input pathname for the final occurance of a '/', and returns the string existing after it. If a '/' is not found, it looks for a '~', as in ~username. If neither a '/' or a '~' is found and the input string is not empty, then it will return the original string.

Side Effects

This routine kmalloc's the return string, and will remove whitespace (as defined by isspace() from the end of the string)

K.2.2. kdirname() — *find directory component of a given pathname*

Synopsis

```
char *kdirname(
   const char *pathname,
   char *return_path)
```

Input Arguments

`pathname`

A valid pathname

Output Arguments

`return_path`

A string that will hold the resulting directory portion of the string. If this parameter is NULL, it will automatically malloc space for the result.

Returns

directory component of pathname, NULL on error

Description

This routine searches an input pathname for the final occurance of a '/', and returns the string existing before the '/'. If a '/' is not found, it looks for a '~', as in ~username. If neither a '/' or a '~' are found, it will assume the pathname does not have a directory component, and returns the string ".".

Side Effects

This routine kmalloc's the return string, and will remove whitespace (as defined by isspace()) from the end of the input string

K.2.3. kexpandpath() — *Expand a path to it true path*

Synopsis

```
char *kexpandpath(
   const char *filename,
   const char *directory,
   char *expandpath)
```

Input Arguments

`filename`

the file to be expanded

Output Arguments

`expandpath`

a string to put the expanded path into. If it is NULL, the result will be kmalloc'ed

Returns

expandpath if it is not NULL, the kmalloc'ed output string if expandpath is NULL, or NULL on error

Description

This routine takes an input pathname, and expands environment variables, khoros variables, ~'s , and logical paths to return the true path from / to the filename specified in the input. It does this by calling kfullpath to expand the first three expansions, and then changing the current working directory to the one containing the specified file and calling getcwd.

Side Effects

if expandpath is NULL, the output string is kmalloc'ed

Restrictions

kexpandpath will fail if the directory specified does not already exist.

K.2.4. kfullpath() — *Expand a environment variables in a path*

Synopsis

```
char *kfullpath(
   const char *filename,
   const char *directory,
   char *fullpath)
```

Input Arguments

`filename`

the file to be expanded

Output Arguments

`fullpath`

if the return file is not NULL then this will be where it place the fullpath to the expanded file. Otherwise, it will kmalloc the space required.

Returns

pointer to fullpath if it is not NULL, or a pointer to a kmalloc'ed string containing the result.

Description

This function returns the full path to the user supplied file. "kfullpath" expands all environment variables in a path, and returns a full path back the user. If the file specified with a ˜username, then the path will be expanded.

Side Effects

if fullpath is NULL, it kmalloc's the return string

K.2.5. ktbpath() — *Expand a environment variables local to a toolbox*

Synopsis

```
char *ktbpath(
   const char *filename,
   char *tbpath)
```

Input Arguments

`filename`

the file to be expanded

Output Arguments

`tbpath`

if the return file is not NULL then this will be where it place the path to the expanded file. Otherwise, it will kmalloc the space required.

Returns

pointer to tbpath if it is not NULL, or a pointer to a kmalloc'ed string containing the result.

Description

This function returns the full path to the user supplied file. "ktbpath", like "kfullpath", expands all environment variables in a path, and returns a full path back the user. If the file specified with a ~username, then the path will be expanded. The difference between "ktbpath" and "kfullpath" is that "ktbpath" tries to expand the relative to the list of toolboxes. So for example if the path handed in is:

/usr/data/test/tb/repos/Aliases

So if the path is inside the toolbox TBTEST which has the path "/usr/data/test/tb", then the resulting path will be handed back as:

$TBPATH/repos/Aliases

Side Effects

if tbpath is NULL, it kmalloc's the return string

K.2.6. ktempnam() — *create a name for a temporary khoros transport*

Synopsis

```
char * ktempnam(
   const char *identifier,
   const char *ktemplate)
```

Input Arguments

`identifier`

The transport identifier

`ktemplate`

template of the file to be created.

Returns

a unique temporary filename on success, NULL otherwise

Description

This module is used to create temporary files. The application program calls "ktempnam" with a template filename which is used to help identify the application routine. The type of temporary created depends on the identifier. If a unique temporary shared memory key is desired then the template would look like:

"shm=XXXX"

If a file was desired then either

"XXXXXXXXX"
"file=XXXX"

would work. The convention should be as follows, an identifier followed by an "=" equals indicator, and finally followed by an optional template.

K.2.7. kfindpath() — *find the path to an executable*

Synopsis

```
char *kfindpath(
   const char *program,
   char *fullpath)
```

Input Arguments

`program`

program to find fullpath for

Output Arguments

`fullpath`

the fullpath to the program, if not NULL then this will be where it place the path. Otherwise, it will kmalloc the space required.

Returns

pointer to fullpath if it is not NULL, or a pointer to a kmalloc'ed string containing the result.

Description

This function returns the full path to the user specified executable. The user's PATH environemnt variable is searched, and the first executable occurance of the program is returned. Paths in . are expanded to be fullpaths.

If the executable provided is already a path, then it is returned unchanged. An exception to this is if the provided path is relative to . or .. then it is expanded.

Side Effects

if fullpath is NULL, it kmalloc's the return string

L. Directory Utilities

L.1. Introduction to Directory Utilities

The following Basic Service routines are used to create and remove directories throughout VisiQuest.

- *kmake_dir()* - make a directory and all parent directories if necessary
- *kremove_dir()* - remove a directory and it's contents
- *kchdir()* - library call to change the current working directory
- *kgetcwd()* - library call to get the current working directory
- *kmkdir()* - library call to create a directory
- *krmdir()* - remove a directory

L.2. Definitions of Directory Utilities

L.2.1. kmake_dir() — *make a directory and all parent directories if necessary*

Synopsis

```
int kmake_dir(
   const char *path,
   mode_t mode)
```

Input Arguments

`path`

directory to create

`mode`

octal permission mode of the new directory

Returns

TRUE (1) on success, FALSE (0) on failure

Description

This routine calls kmkdir as many times as needed to build a new directory. This means that if making a directory, and the parent directories do not exist, it will first make all the parent directories that are needed, and then create the directory requested by the calling routine.

L.2.2. kremove_dir() — *remove a directory and it's contents*

Synopsis

```
int kremove_dir(
    const char *path)
```

Input Arguments

`path`

the pathname to remove

Returns

0 on success and -1 on failure

Description

This routine removes a specified path. If it is a directory, it recursively calls itself on all pathnames under it.

Restrictions

Hard links to other directory trees are not recognized by this routine, so if a hard link exists to a directory, it will follow it as a normal directory.

L.2.3. kchdir() — *library call to change the current working directory*

Synopsis

```
int kchdir(
    const char *path)
```

Input Arguments

`path`

pathname of new directory to create

Returns

(0) on success, (-1) otherwise

Description

This routine acts the same as the system routine chdir(), with one minor change. It makes an internal call to kfullpath to expand environment variables, toolbox variables, and ~'s. NOTE: errno is set by the internal call to chdir()

L.2.4. kgetcwd() — *library call to get the current working directory*

Synopsis

```
char *kgetcwd(
   char *path,
   size_t length)
```

Input Arguments

`length`

the length of the path string

Output Arguments

`path`

the buffer that holds the return string

Returns

path on success, NULL otherwise

Description

This routine acts the same as the system routine getcwd(), with one minor change. It actually returns the KHOROS_PWD environment variable.

L.2.5. kmkdir() — *library call to create a directory*

Synopsis

```
int kmkdir(
   const char *path,
   mode_t mode)
```

Input Arguments

`path`

pathname of new directory to create

`mode`

octal permission mode of the new directory e.g.: 0777. The 0700 part defines owner's permissions, the

0070 part defines group's permissions, and the 0007 part defines the permissions for everyone else. Within each of those parts, the 04 bit is READ, 02 is WRITE and 01 is EXECUTE.

Returns

(0) on success, (-1) otherwise

Description

This routine acts the same as the system routine mkdir(), with one minor change. It makes an internal call to kfullpath to expand environment variables, toolbox variables, and ~'s. NOTE: errno is set by the internal call to mkdir()

L.2.6. krmdir() — *remove a directory*

Synopsis

```
int krmdir(
   const char *path)
```

Input Arguments

`path`

directory name to delete

Returns

0 on success, -1 otherwise

Description

This routine is a replacement for the system routine rmdir. It has the same actions, error status, and return values of rmdir, but it will expand ~'s, environment variables, and VisiQuest variables.

M. Environment Variable Utilities

M.1. Introduction to Environment Variable Utilities

The utilities below work with environment variables that are set by the user prior to execution of a program.

- *kgetenv()* - get an environment variable from the environ list.
- *kputenv()* - put an environment variable into the environ list.
- *kremenv()* - remove an environment variable from the environment

M.2. Definitions of Environment Variable Utilities

M.2.1. kgetenv() — *get an environment variable from the environ list.*

Synopsis

```
char *kgetenv(
   const char *name)
```

Input Arguments

`name`

the name of the environament variable to look for

Returns

the environment string upon success, NULL on failure

Description

This module is used to get an environment variable from the environ list. For example, if the user has their TMPDIR environment variable set to /usr/var/tmp, and this routine is called with (name = "TMPDIR"), then "/usr/var/tmp" will be returned.

Restrictions

The string returned is simply an address into the environ list, so it should **NOT** be freed, or changed in any way by the calling routine.

M.2.2. kputenv() — *put an environment variable into the environ list.*

Synopsis

```
int kputenv(
   const char *name)
```

Input Arguments

`name`

string to be added into putenv

Returns

(0) on success, (-1) otherwise

Description

This module is used to put an environment variable in the environ list. If "PUTENV" is defined then we call putenv() otherwise we do our own putenv. The only difference is that kputenv() will malloc space for the incoming environment variable.

Side Effects

the input "name" is copied into the environment list

M.2.3. kremenv() — *remove an environment variable from the environment*

Synopsis

```
int kremenv(
   const char *name)
```

Input Arguments

`name`

a string containing the environment variable to remove.

Returns

(0) on success, (-1) otherwise

Description

This module is used to remove an environment variable from the environ list. Since there does not seem to be a standard routine for this on many machines, we wrote our own.

Side Effects

Does not kfree_and_NULL string associated with environment variable deleted, since we cannot know how it was created.

Restrictions

This routine will not work if the environ list is not stored as an array of strings.

N. Variable Argument Utilities

N.1. Introduction to Variable Argument Utilities

There are three functions available to help write functions or subroutines that take a variable number of arguments.

• *kva_arg()* - gets an argument off the variable argument list
• *kva_end()* - sets the end of the variable argument list
• *kva_start()* - sets the start of the variable argument list

N.2. Definitions of Variable Argument Utilities

N.2.1. kva_start() — *sets the start of the variable argument list*

Synopsis

```
void kva_start(

   kva_list vararg_list,
   last_param)
```

Input Arguments

`last_param`
the last argument on the parameter list before variable arguments begin

Output Arguments

`vararg_list`
the variable argument pointer is set to the beginning of the variable argument list

Description

This routine sets the "vararg_list" pointer to the start of the variable argument list.

Example variable argument routine definition:

```
var_arg_routine(int p1, int p2, double pN, kva_alist)
{
   short    var;          // the next parameter after pN
                          // is expected to be short
```

```
   kva_list *vararg_list; // the variable argument list

   kva_start(vararg_list, pN); // pN is the last parameter
                               // before variable args begin
   :
   var = kva_arg(vararg_list, short);
   :
   kva_end(vararg_list);
}
```

The execution stack when procedure var_arg_routine is executed; when kva_start is called, the "vararg_list" pointer is set to the address indicated below.

```
                            | byte0   int parameter p1
                            |  :
                            | byte4   int parameter p2
                            |  :
                            | byte8   double parameter pN
                            |  :
 args stack address ---->   | byte16  short parameter A
  "vararg_list"             | byte17
                            | byte18  int parameter B
                            | byte19
                            | byte20
                            | byte21
                            |  :
```

N.2.2. kva_arg() — *gets an argument off the variable argument list*

Synopsis

```
type kva_arg(

   kva_list vararg_list,
   type)
```

Input Arguments

`vararg_list`
: the argument list

`type`

The data type of the parameter. If the type specified is not the same as the actual type of the value on the stack, the behavior is undefined.

In standard C, arguments that are char or short are converted to int and should be accessed as int. Arguments that are unsigned char or unsigned short are converted to unsigned int and should be accessed as unsigned int. Arguments that are float are converted to double and should be accessed as double.

Returns

The next paramter on the variable argument list cast to the type specified by the type parameter

Description

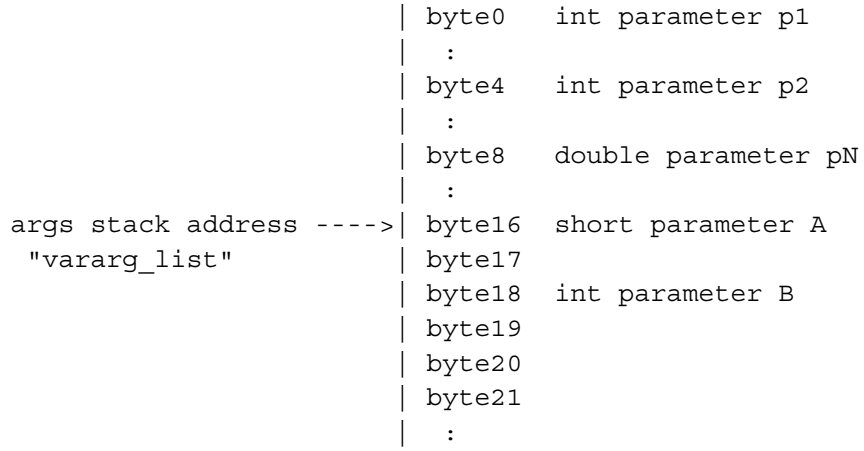
This routine gets an argument off the variable argument list.

Example variable argument routine definition:

```
var_arg_routine(int p1, int p2, double pN, kva_alist)
{
    short    var;          // the next parameter after pN
                           // is expected to be short
    kva_list *vararg_list; // the variable argument list

    kva_start(vararg_list, pN); // pN is the last parameter
                                // before variable args begin
    :
    var = kva_arg(vararg_list, short);
    :
    kva_end(vararg_list);
}
```

The execution stack when procedure var_arg_routine is executed; before kva_arg is called, the "vararg_list" pointer is set to the address indicated below.



Now, a call to kva_arg(vararg_list, short) moves the "vararg_list" pointer to byte18, and returns the pointer to byte16 as a short, so that var_arg_routine() can obtain the value of short parameter A.

the resulting execution stack is now:

```
                            | byte0   int parameter p1
                            |  :
                            | byte4   int parameter p2
                            |  :
                            | byte8   double parameter pN
                            |  :
    args stack address -+   | byte16  short parameter A
     "vararg_list"      |   | byte17  (assigned to "var")
                        +-->| byte18  int parameter B
                            | byte19
                            | byte20
                            | byte21
                            |  :
```

N.2.3. kva_end() — *sets the end of the variable argument list*

Synopsis

```
void kva_end(

    kva_list vararg_list)
```

Input Arguments

`vararg_list`
the variable argument list to be terminated

Output Arguments

`vararg_list`
the variable argument list pointer set to NULL

Description

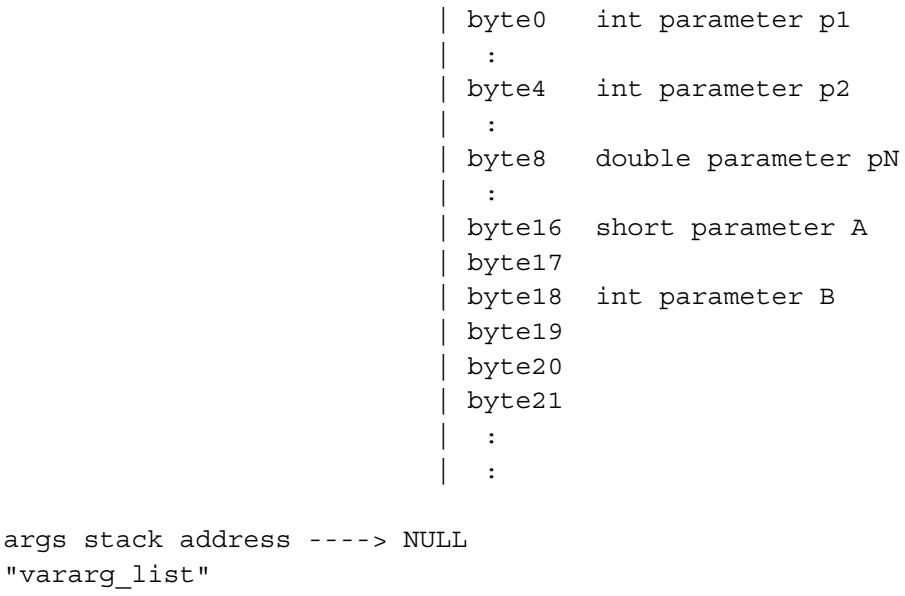
This routine clears the "vararg_list" pointer and shuts down the variable argument access mechanism.

Example variable argument routine definition:

```
var_arg_routine(int p1, int p2, double pN, kva_alist)
{
   short    var;          // the next parameter after pN
                          // is expected to be short
   kva_list *vararg_list; // the variable argument list

   kva_start(vararg_list, pN); // pN is the last parameter
                               // before variable args begin
   :
   var = kva_arg(vararg_list, short);
   :
   kva_end(vararg_list);
}
```

The execution stack when procedure var_arg_routine is executed; when kva_end is called, the "vararg_list" pointer is set to NULL



Restrictions

On some architectures, this routine is mapped to va_end(), which may or may not have any effect on the vararg_list pointer.

O. Array Creation & Manipulation

O.1. Introduction to Array Utilities

The *karray* utilities provide a collection of functions for array manipulation. These routines support all standard VisiQuest data types, such as KINT, KFLOAT, KDOUBLE, KSTRING, KSTRUCT, etc. The copy, sort, merge, dirlist, and filelist utilities will work only on arrays of strings. The functions included within in this category are:

- *karray_add()* - add an entry into the array list
- *karray_copy()* - copy an array of strings
- *karray_delete()* - delete an entry from an array
- *karray_dirlist()* - create an array of strings reflecting directory contents
- *karray_filelist()* - create an array of strings reflecting the contents of a file
- *karray_free()* - free memory used by an array
- *karray_insert()* - insert an entry into an array
- *karray_locate()* - locate an entry in an array
- *karray_merge()* - merge two arrays into one
- *karray_sort()* - sort an array
- *karray_to_list()* - convert an array into a linked list
- *karray_to_string()* - convert a string array into a single big string
- *knumber()* - the number of items in an array

O.2. Definitions of Array Utilities

O.2.1. karray_add() — *add an entry into the array list*

Synopsis

```
<type> *karray_add(

    <type> *array,
    int     type,
    <type>  entry,
    size_t  num)
```

Input Arguments

`array`

The current array in which we will be adding the entry to (if NULL then return the newly malloc'ed array list).

`type`

The type of the array and the entry.

`entry`
The entry identifier to be added to the array list

`num`
The number of entries in the list; ignored if list is NULL

Returns

returns the modified array list.

Description

Adds an entry to the array list. This is done by adding the entry to the end of the array list (if the array currently exists). If the array is currently NULL then the new item is returned as the first entry of the array list. If the array list is not NULL then the entry is added to the end of the list and the original array is passed back to the calling routine.

The routine first scans the array to make sure the entry is not already on the list, if so then we don't change original array.

The type field can be one of the following settings:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

O.2.2. karray_copy() — *copy an array of strings*

Synopsis

```
<type> *karray_copy(

   <type> *array,
   int     type,
```

```
size_t  num,
int     copy_entries)
```

Input Arguments

`array`
: the array that is to be sorted

`type`
: The type of the array.

`num`
: the number of entries in the array

`copy_entries`
: whether to kmalloc and copy each entry or not

Returns

a pointer to the new array, or NULL on error

Description

This module is used to copy the input array of strings into a new array. If the "copy_entries" parameter is TRUE, then each entry will also be copied rather than just copying the pointers.

The type field can be one of the following settings:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

Side Effects

kmallocs the space for the new array

O.2.3. karray_delete() — *delete an entry from an array*

Synopsis

```
<type> *karray_delete(

  <type> *array,
   int    type,
  <type>  entry,
   size_t num)
```

Input Arguments

`array`
: The array from which to delete the entry

`entry`
: The entry to be deleted from the array

`num`
: The number of entries currently in the array (before the entry is deleted).

Returns

The modified array when there are still entries in the array, NULL when the array becomes empty.

Description

Deletes an entry from the array, and returns the modified array to the calling routine.

The array utilities, including karray_delete(), support arrays of a variety of data types. The data type of the array being used is specified with the 'type' parameter; supported data types include:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

Note that the array and the entry passed in MUST both be of the type specified by the type parameter, or the results of this routine are unpredictable.

O.2.4. karray_dirlist() — *create an array of strings reflecting directory contents*

Synopsis

```
char **karray_dirlist(
   char *basename,
   char *directory,
   char *filter,
   int list_mode,
   int format,
   size_t * num)
```

Input Arguments

`basename`
: Basename of the entries in the directory to be included as strings in the array.

`directory`
: The directory path to the basename

`filter`
: Filter to be used in matching the directory entries. If filter is NULL then all entries are accepted (unless otherwise indicated by basename).

`list_mode`
: A flag indicating which directory entries are to be included (eg. files, directories, dot files, symbolic links, etc.)

`format`
: Whether the files should be postfixed with the symbol indicating the type of entry (eg. "@" for symbolic links, "/" for subdirectories, etc)

Output Arguments

`num`
: The number of entries in the resulting string array

Returns

The array of strings representing the directory contents on success, NULL on failure

Description

Translates the contents of a directory into an array of strings, where each string in the array is the name of an entry in the specified directory.

The 'directory' parameter specifies the directory path to use for creating of the string array. This directory path may include toolbox names (for example, $BOOTSTRAP) or tildas (for example, ~fred).

Passing NULL implies that the ’basename’ parameter will be used to specify the directory.

The ’basename’ parameter allows you to specify a set of letters which must be matched by the entries in the directory before they are included in the resulting array. For example, providing "img" as the basename will cause only those entries in the directory beginning with the letters "img" to be included in the array. Passing NULL indicates that ALL entries in the directory specified by (unless otherwise specified using the ’filter’ parameter).

For your convenience, the values of both the ’directory’ and the ’basename’ parameters may be combined into the value of the ’basename’ parameter. For example, if you wanted to create the string array from all entries in the ˜fred/data directory beginning with the letters "d3", then you could pass "˜fred/data/d3" as the ’basename’ and NULL as the ’directory’.

More powerful and flexible than the ’basename’ parameter, as well as more difficult to use, the ’filter’ parameter allows you to specify a more complicated filter that entries in the directory must match before they are included in the array. The regular expression syntax is used to specify the filter; see *kre_comp()* for the syntax required to specify the desired pattern matching.

The ’list_mode’ parameter indicates which entries in the directory are to be included in the resulting array of strings. The following list modes are supported:

```
KPATH    - prepend the path to each file
KFILE    - list plain text files
KDIR     - list subdirectories
KDOT     - list dot files  (UNIX for "hidden files")
KLINK    - list symbolic files
KSOCK    - list socket files
KREAD    - file is readable by caller
KWRITE   - file is writable by caller
KEXEC    - file is executable by caller
KRECURSE - recursively list all subdirectories
```

The list modes may be or’ed together in order to specify the entries in the directory that should be included in the string array produced. For example, "KFILE | KDIR" will cause the resulting string array to have entries for will list only files and directories.

O.2.5. karray_filelist() — *create an array of strings reflecting the contents of a file*

Synopsis

```
char **karray_filelist(
   char *filename,
   char *directory,
```

```
    int sort_flag,
    size_t * num)
```

Input Arguments

`filename`
: The name of the file to have its contents translated into a string array.

`directory`
: The directory path to the filename

`sort_flag`
: Pass TRUE if the resulting array is to be sorted alphabetically, FALSE otherwise

Output Arguments

`num`
: The number of entries in the resulting string array

Returns

The array of strings representing the lines in the file on success, NULL on failure.

Description

Translates the contents of a file into an array of strings, where each string in the array is a line of the specified file. If desired, the resulting string array may also be sorted in alphabetical order.

The 'directory' parameter specifies the directory path to the file which will be used to create of the string array. This directory path may include toolbox names (for example, $BOOTSTRAP) or tildas (for example, ~fred). Passing NULL implies that the 'filename' parameter will be used to specify the directory as well as the filename.

The 'filename' parameter specifies the name of the file which is to have its contents translated into a string array. If the 'directory' parameter is used to specify the directory path, the 'filename' parameter should only contain the name of the file; if the 'directory' parameter is passed as NULL, then the 'filename' parameter must include the directory path to the file.

O.2.6. karray_free() — *free memory used by an array*

Synopsis

```
<type> *karray_free(

  <type> *array,
  int     type,
  size_t  num,
  void  (*routine) (<type>))
```

Input Arguments

`array`
The array to be freed

`type`
The data type of the array and the entry

`num`
The number of entries currently in the array

`routine`
The routine to destroy each element if a specialized free routine is desired; NULL implies that kfree_and_NULL() should be used.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Frees the memory associated with the use of an array.

The array utilities, including karray_free(), support arrays of a variety of data types. The data type of the array being used is specified with the 'type' parameter; supported data types include:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

If desired, you may provide a routine to destroy each element of the array. Generally, this feature is used only for arrays of type KSTRUCT, where the structure in question contains pointers to strings or other structures that must be freed individually. If used, the routine specified must be declared as follows:

```
int free_routine(void *)
```

It must return TRUE on success and FALSE on failure. It must take a single parameter with which to pass the pointer to the element to free, cast as a (void *). If the routine parameter is NULL, then

karray_free() simply uses kfree_and_NULL() to free the array.

O.2.7. karray_insert() — *insert an entry into an array*

Synopsis

```
<type> *karray_insert(

  <type> *array,
   int    type,
  <type>  entry,
   size_t num,
   int    position,
   int    duplicates)
```

Input Arguments

`array`
: The array to which to add the entry

`type`
: The data type of the array and the entry

`entry`
: The entry to be added to the array

`num`
: The number of entries currently in the array (before the new entry is inserted)

`position`
: Where in the array to add the entry; provide one of KLIST_HEAD or KLIST_TAIL.

`duplicates`
: TRUE if multiple occurances of an entry should be allowed in the array, FALSE if only one occurance of each entry should be allowed. If this routine is called with 'duplicates' set to FALSE and the entry provided already appears as in the array, the array will remain unchanged.

Returns

The head of the modified array

Description

Inserts an entry into the array at the specified position, and returns the modified array to the calling routine.

The array utilities, including karray_insert(), support arrays of a variety of data types. The data type of the array being used is specified with the 'type' parameter; supported data types include:

```
KBYTE    - array of characters
```

```
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

Note that the array and the entry passed in MUST both be of the type specified by the type parameter, or the results of this routine are unpredictable.

The position parameter indicates where in the array an entry should be inserted. Supported positions include:

```
KLIST_HEAD - insert entry at the head of the array
KLIST_TAIL - insert entry at the end of the array
```

O.2.8. karray_locate() — *locate an entry in an array*

Synopsis

```
ssize_t karray_locate(

  <type>  *array,
   int    type,
  <type>  entry,
   size_t num)
```

Input Arguments

`array`
: The array in which to search for the entry

`type`
: The data type of the array

`entry`
: The entry to be be located in the array

`num`
The number of entries currently in the array

Returns

Returns the index of the entry if it is found, -1 otherwise

Description

Locate an entry in the array. If the entry exists in the array, then the index to that entry is returned. If the entry doesn't exist in the array, then -1 is returned.

The array utilities, including karray_locate(), support arrays of a variety of data types. The data type of the array being used is specified with the 'type' parameter; supported data types include:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

O.2.9. karray_merge() — *merge two arrays into one*

Synopsis

```
<type> *karray_merge(

  <type> *array1,
  <type> *array2,
  int     type,
  size_t  num1,
  size_t  num2,
  int     copy_entries)
```

Input Arguments

`array1`
: First array of strings

`array2`
: Second array of strings

`type`
: The data type of the array

`num1`
: Number of entries in array1

`num2`
: Number of entries in array2

`copy_entries`
: For string arrays, specifies whether or not to copy the strings into the merged array, or simply to duplicate the pointers; ignored for arrays of all other data types.

Returns

A pointer to the merged array on success, NULL on failure.

Description

Merges two arrays into a single array by concatenating the second array onto the first.

The array utilities, including karray_merge(), support arrays of a variety of data types. The data type of the array being used is specified with the 'type' parameter; supported data types include:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

For string arrays ('type' == KSTRING), strings in the merged array are only copied if the 'copy_entries' parameter is passed as TRUE. Otherwise, each entry in the merged array will simply point to the old string.

Side Effects

Allocates memory for the merged array

O.2.10. karray_sort() — *sort an array*

Synopsis

```
<type> *karray_sort(

  <type> *array,
  int     type,
  size_t  num,
  int     duplicates)
```

Input Arguments

`array`
: The array to sort

`type`
: The data type of the array. Only KSTRING is supported on this routine.

`num`
: The number of entries in the array

`duplicates`
: TRUE if any duplicate entries should be left in the sorted array, FALSE to remove any duplicate entries from the sorted array

Returns

The pointer to the header of the sorted array; NULL on failure

Description

Sorts a string array into ascending order.

Other array utilities support arrays of a variety of data types. The karray_sort() routine, however, can only be used with string arrays.

Side Effects

The sorted array passed back is the same as the original array passed in. So original input array should not be sorted.

O.2.11. karray_to_list() — *convert an array into a linked list*

Synopsis

```
klist *karray_to_list(

  <type> *array,
  int     type,
  size_t  num)
```

Input Arguments

`array`
: The array from which the linked list will be created.

`type`
: The data type of the array

`num`
: The number of entries in the array.

Returns

The newly created linked list on success, NULL upon failure

Description

Creates a linked list from an array. The resulting list may be used with the VisiQuest linked list utilities, such as klist_add(), klist_insert(), klist_delete(), klist_copy(), and so on.

The array utilities, including karray_to_list(), support arrays of a variety of data types. The data type of the array being used is specified with the 'type' parameter; supported data types include:

```
KBYTE    - array of characters
KUBYTE   - array of unsigned characters
KSHORT   - array of short integers
KUSHORT  - array of unsigned short integers
KINT     - array of integers
KUINT    - array of unsigned integers
KLONG    - array of long integers
KULONG   - array of unsigned long integers
KFLOAT   - array of floating point numbers
KDOUBLE  - array of double precision numbers
KSTRING  - array of strings
KSTRUCT  - array of pointers to structures
KLOGICAL - array of TRUE/FALSE values
```

O.2.12. karray_to_string() — *convert a string array into a single big string*

Synopsis

```
char *karray_to_string(
   char **array,
   int type,
   size_t num,
   char *separator)
```

Input Arguments

`array`
: The string array from which the return string will be created.

`type`
: The data type of the array

`num`
: The number of entries in the array.

`separator`
: The separation string to be inserted between each pair of strings in the array.

Returns

The newly created string on success, NULL on failure

Description

Creates one big string from an array of strings.

While most of the array routines support a variety of data types, karray_to_string() requires a string array. Arrays of other data types are not supported as of yet.

A specified separation string is inserted between each pair of strings in the array. For example, if the input array had the contents:

array[0] = "ls"
array[1] = "-a"
array[2] = "/tmp"

and the 'separator' parameter was specified as " ", then the string returned will be:

"ls -a /tmp"

If the 'separator' parameter was passed in as NULL, then the resulting string would be:

"ls-a/tmp"

O.2.13. knumber() — *the number of items in an array*

Synopsis

```
size_t knumber(char **array)
```

Input Arguments

`array`

the array of items

Returns

returns the number of items

Description

This function returns the number of items in an array

P. Linked List Creation & Manipulation

P.1. Introduction to Linked List Utilities

A set of functions for creating and maintaining linked lists are provided by the *kutils (libku.a)* library. These utilities work with a *klist* structure, which is defined as follows:

```
typedef struct _klist
{
        kaddr identifier;
        kaddr client_data;
        unsigned char head;

        struct _klist *next, *prev;
} klist;
```

identifier

This may be any value which will uniquely identify an entry in the list.

client_data

This is a pointer to a data structure which you define, in which the data node for the list is stored. For example, if you need a list in which each entry contains a serial number, model number, size, and name, you would define a data structure containing each of these fields:

```
typedef struct _my_listdata
{
     int serial_num;
     int model_num;
     int size;
     char *name;
} my_listdata;
```

You would allocate your structure, initialize the fields and then pass a pointer to it into the *klist_add* as the client_data field. Later, the structure can be retrieved from the list entry using *klist_clientdata*.

head

This is for internal use.

next, prev

These are pointers to the next entry in the linked list, and to the previous entry in the linked list. You may use them directly or obtain them with the *klist_next*.

P.2. Definitions of Linked List Utilities

- *klist_add()* - add an entry into the linked list
- *klist_checkentry()* - check if the klist entry is currently on the link list
- *klist_checkhead()* - check if the current entry is the head of the list
- *klist_checkident()* - check if the identifier is currently on the link list
- *klist_checktail()* - check if the current entry is the tail of the list
- *klist_clientdata()* - return the client data associated with an entry on the list
- *klist_copy()* - copy a linked list into a new linked list
- *klist_delete()* - delete an entry from the linked list
- *klist_dirlist()* - create a linked list of file names
- *klist_filelist()* - create a linked list of strings from a file
- *klist_free()* - free the entire linked list
- *klist_head()* - locate the head of the linked list
- *klist_identifier()* - return the identifier associated with an entry on the list
- *klist_insert()* - insert an entry into the linked list
- *klist_locate()* - locate an entry in the linked list
- *klist_locate_clientdata()* - locate an entry in the linked list according to it's client data
- *klist_makecircular()* - changes a consecutive or linear link list into a circular link list
- *klist_makelinear()* - changes a circular link list into a consecutive or linear link list
- *klist_merge()* - merge two linked list into a single linked list
- *klist_next()* - return the next entry on the list
- *klist_prev()* - return the previous entry on the list
- *klist_size()* - compute the size or number of entries in the list
- *klist_sort()* - sort the linked list
- *klist_split()* - split a single linked list into two linked lists

• *klist_tail()* - locate the tail of the linked list
• *klist_to_array()* - convert the linked list into an array
• *kalias_list()* - returns a string array of aliases

P.2.1. klist_add() — *add an entry into the linked list*

Synopsis

```
klist *klist_add(
   klist * list,
   kaddr identifier,
   kaddr client_data)
```

Input Arguments

`list`
: The current list in which we will be adding the entry to (if NULL then return the newly malloc'ed head).

`identifier`
: The entry identifier to be added to the linked list

`client_data`
: client data to be associated with the identifier

Returns

The modified linked list.

Description

Adds an entry to the linked list. This is done by adding the entry to the end of the linked list (if the list currently exists). If the list is currently NULL then the new item is returned as the head of the list. If the list is not NULL then the original list is passed back to the calling routine.

The routine first scans the list to make sure the identifier is not already on the list, if so then we don't change original list.

P.2.2. klist_checkentry() — *check if the klist entry is currently on the link list*

Synopsis

```
int klist_checkentry(

    klist *list)
```

Input Arguments

`list`

the link list in which we will be checking the klist entry against entry - the klist entry to check for on the link list

Returns

return TRUE if the entry is the list, otherwise FALSE is returned

Description

This routine simply returns whether the klist entry currently exists on the link list. If the entry exists then TRUE is returned otherwise FALSE is returned.

P.2.3. klist_checkhead() — *check if the current entry is the head of the list*

Synopsis

```
int klist_checkhead(

    klist *list)
```

Input Arguments

`list`

the list in which we will be returning whether it is the head of the list or not

Returns

return TRUE if the entry is the head of the link list, otherwise FALSE is returned

Description

This routine simply returns whether the current entry is the head of the link list. If the entry is the head then TRUE is returned otherwise FALSE is returned. This is helpful since if a link list is circular you

cannot rely on the previous value being NULL.

P.2.4. klist_checkident() — *check if the identifier is currently on the link list*

Synopsis

```
int klist_checkident(

    klist *list)
```

Input Arguments

`list`

the link list in which we will be checking the identifier against identifier - the identifier to check for on the link list

Returns

return TRUE if the identifier is the list, otherwise FALSE is returned

Description

This routine simply returns whether the identifier currently exists on the link list. If the identifier exists then TRUE is returned otherwise FALSE is returned.

P.2.5. klist_checktail() — *check if the current entry is the tail of the list*

Synopsis

```
int klist_checktail(

    klist *list)
```

Input Arguments

`list`

the list in which we will be returning whether it is the tail of the list or not

Returns

return TRUE if the entry is the tail of the link list, otherwise FALSE is returned

Description

This routine simply returns whether the current entry is the tail of the link list. If the entry is the tail then TRUE is returned otherwise FALSE is returned. This is helpful since if a link list is circular you cannot rely on the next value being NULL.

P.2.6. klist_clientdata() — *return the client data associated with an entry on the list*

Synopsis

```
kaddr klist_clientdata(

    klist *list)
```

Input Arguments

`list`

the list in which we will be returning the client data

Returns

the client data associated with the klist entry on the list

Description

This routine simply returns the client data associated with the klist entry on the link list. If the current entry is NULL then NULL is returned. Otherwise, whatever is stored in the klist entry is returned.

P.2.7. klist_copy() — *copy a linked list into a new linked list*

Synopsis

```
klist *klist_copy(
    klist * list)
```

Input Arguments

`list`

The current linked list to be copied

Returns

The new linked list or NULL if we fail.

Description

Copies a linked list. This is done by adding the items from one list to a new list.

P.2.8. klist_delete() — *delete an entry from the linked list*

Synopsis

```
klist *klist_delete(
   klist * list,
   kaddr identifier)
```

Input Arguments

`list`

The current linked list in which the entry will be deleted.

`identifier`

the identifier which identifies the entry to be deleted.

Returns

The modified linked list.

Description

Delete an entry from the linked list. This is done by deleting the entry from the linked list. The new list is then passed back to the calling routine.

P.2.9. klist_dirlist() — *create a linked list of file names*

Synopsis

```
klist *klist_dirlist(
   char *basename,
   char *directory,
   char *filter,
   int list_mode,
   int format)
```

Input Arguments

`basename`

base name of the files to be listed.

`directory`

the directory path is used as a prefix to basename

`filter`

the filter is used in matching the directory entries. The syntax used is the same as the "kparse" utilities. If filter is NULL then all entries are accepted.

`list_mode`
a flag indicating what we are to list (ie. files, directories, dot files, symbolic links, etc.)

`format`
whether the files should be prepended with type of file. (ie. "@" for sym links, "/" for directories)

Returns

A link list of strings to filenames that match the basename

Description

This module is used to create a linked list of file names according to a user supplied basename and an initial global directory. The list mode is used to indicate what we are going to list in the directory. The possible defines are listed in $KHOROS/include/khoros/kdefines.h. The following symbols are the current list mode:

```
KPATH  - prepend the path to each file
KFILE  - list plain text files
KDIR   - list directories
KDOT   - list dot files
KLINK  - list symbolic files
KSOCK  - list socket files
KREAD  - file is readable by caller
KWRITE - file is writable by caller
KEXEC  - file is executable by caller
```

The selections are or'ed together in order to choose the set of attributes that are desired. (e.g KFILE | KDIR) will list only files and directories that match the basename.

P.2.10. klist_filelist() — *create a linked list of strings from a file*

Synopsis

```
klist *klist_filelist(
   char *filename,
   char *directory,
   int sort_flag)
```

Input Arguments

`filename`
base name of the files to be listed.

`directory`
the directory path to be used as a prefix to the filename

`sort_flag`
a flag indicating whether we are to sort the link list before returning

Returns

The head of the linked list to the names in the file upon successfull completion of this routine, otherwise NULL is returned.

Description

This module is used to create a linked list from the supplied file. It reads the lines out of the specified file, and returns them in a linked list of strings. The programmer also has the option of specifying if they want the resulting string sorted or not.

P.2.11. klist_free() — *free the entire linked list*

Synopsis

```
void klist_free(
   klist * list,
   void (*routine) PROTO((klist *)))
```

Input Arguments

`list`
the linked list to be destroyed

`routine`
the routine to destroy each entry

Description

Destroys the linked list by walking thru the list and freeing each entry. One of the parameters is an element routine pointer that will allow the user to specify a routine to be used to destroy each element of the array. This routine must have a return type of void. It must also take a single argument as a parameter; the pointer to the list element to reclaim, which is of the data type klist. If the routine parameter is NULL, the routine uses kfree_and_NULL on the list structure only, the client_data and identifier are not freed.

P.2.12. klist_head() — *locate the head of the linked list*

Synopsis

```
klist *klist_head(
```

```
klist * list)
```

Input Arguments

`list`

The current linked list to find the head for.

Returns

The item's head entry or NULL if the current list does not exist.

Description

Tries to locate the head of the linked list. This is done by taking to current position and racing up the previous links until the head is found. The head klist structure is returned, NULL if the list is currently empty.

P.2.13. klist_identifier() — *return the identifier associated with an entry on the list*

Synopsis

```
kaddr klist_identifier(

    klist *list)
```

Input Arguments

`list`

the list in which we will be returning the identifier for

Returns

the client data associated with the klist entry on the list

Description

This routine simply returns the identifier associated with the klist entry on the link list. If the current entry is NULL then NULL is returned. Otherwise, whatever is stored in the klist entry identifier is returned.

P.2.14. klist_insert() — *insert an entry into the linked list*

Synopsis

```
klist *klist_insert(
   klist * list,
   kaddr identifier,
   kaddr client_data,
   int position,
   int duplicate_entries)
```

Input Arguments

`list`
: The current list in which we will be adding the entry to

`identifier`
: The entry identifier to be added to the linked list

`client_data`
: client data to be associated with the identifier

`position`
: where in the list to add the entry

`duplicate_entries`
: whether to be allowed multiple occurences of an identifier on the linked list.

Returns

The head of the modified linked list.

Description

Inserts an entry into the linked list. This is done by adding the entry to the end of the linked list (if the list currently exists). The new list is then passed back to the calling routine. The routine first scans the list to make sure the identifier is not already on the list, if so then we don't change original list.

The position field is used to indicate where in the list an entry should be inserted:

```
KLIST_HEAD - insert at the head of the list
KLIST_PREV - insert previous to the current position
KLIST_NEXT - insert next from th current position
KLIST_TAIL - insert at the tail of the list
```

P.2.15. klist_locate() — *locate an entry in the linked list*

Synopsis

```
klist *klist_locate(
   klist * list,
   kaddr identifier)
```

Input Arguments

`list`
: The current linked list in which the entry will be located.

`identifier`
: the identifier which identifies the entry to be located.

Returns

The item's klist entry or NULL if the identifier is not found.

Description

Tries to locate an entry on the linked list. This is done by racing thru the linked list until the desired identifier is found or the end of the list is encountered. If the identifier exists then the klist structure is returned. If the identifier doesn't exist then NULL is returned.

P.2.16. klist_locate_clientdata() — *locate an entry in the linked list according to it's client data*

Synopsis

```
klist *klist_locate_clientdata(
   klist * list,
   kaddr client_data)
```

Input Arguments

`list`
: The current linked list in which the entry will be located.

`client_data`
: the client_data which identifies the entry to be located.

Returns

The item's klist entry or NULL if the identifier is not found.

Description

Tries to locate an entry on the linked list according to it's client data, rather than it's identifier. This is done by racing thru the linked list until the desired identifier is found or the end of the list is

encountered. If the identifier exists then the klist structure is returned. If the identifier doesn't exist then NULL is returned.

P.2.17. klist_makecircular() — *changes a consecutive or linear link list into a circular link list*

Synopsis

```
int klist_makecircular(

    klist *list)
```

Input Arguments

`list`
the link list in which we will be making circular

Returns

return the head of the circular link list or NULL upon failure.

Description

This routine is used to make a consecutive or linear doubly linked list into a circular linked list.

P.2.18. klist_makelinear() — *changes a circular link list into a consecutive or linear link list*

Synopsis

```
int klist_makelinear(

    klist *list)
```

Input Arguments

`list`
the link list in which we will be making linear link list.

Returns

return the head of the circular link list or NULL upon failure.

Description

This routine is used to make a circular link list into a consecutive or linear linked list.

P.2.19. klist_merge() — *merge two linked list into a single linked list*

Synopsis

```
klist *klist_merge(

   klist *list1,
   klist *list2)
```

Input Arguments

`list1`

The first linked list

`list2`

The second linked list

Returns

The newly merge linked list, which is really the head of the first list, or NULL upon failure.

Description

Merge two linked list into a single linked by tacking the second list onto the end of the first.

P.2.20. klist_next() — *return the next entry on the list*

Synopsis

```
klist *klist_next(

   klist *list)
```

Input Arguments

`list`

the list in which we will be returning the next entry from

Returns

the next entry on the list or NULL if at the end and the link list is not circular

Description

This routine simply returns the next klist entry on the link list. If the current entry is last on the link list and the list is not a circular link list then NULL is returned.

P.2.21. klist_prev() — *return the previous entry on the list*

Synopsis

```
klist *klist_prev(

    klist *list)
```

Input Arguments

`list`

the list in which we will be returning the previous entry from

Returns

the previous entry on the list or NULL if at the beginning and the link list is not circular

Description

This routine simply returns the previous klist entry on the link list. If the current entry is first on the link list and the list is not a circular link list then NULL is returned.

P.2.22. klist_size() — *compute the size or number of entries in the list*

Synopsis

```
size_t klist_size(
    klist * list)
```

Input Arguments

`list`

The current linked list to count the number entries for.

Returns

The number of entries or zero if a NULL list is passed in.

Description

Given a list will indicate how many entries are on the list. This is done by counting the number of entries until the tail entry is reached.

P.2.23. klist_sort() — *sort the linked list*

Synopsis

```
klist *klist_sort(

   klist *list,
   kfunc_int compare,
   int   duplicate_entries)
```

Input Arguments

`list`

The current linked list to be sorted.

`compare`

An integer function which performs the comparision of whether the first identifier is less than (-1), equal to (0), or greater than (1) the second identifier.

`duplicate_entries`

toggle where TRUE allows duplicate entries and FALSE removes duplicate entries.

Returns

The sorted linked list

Description

Sorts the linked list into ascending order. A merge sort algorithm is used. This algorithm takes the input list and treats it as a collection of small sorted lists. It makes log N passes along the list, and in each pass it combines each adjacent pair of small sorted lists into one larger sorted list. When a pass only needs to do this once, the whole output list must be sorted.

The provided compare routine is used to compare the identifiers of two list elements. The routine is expected to return whether the first identifier is less than, equal to, or greater than the second. This is done by returning -1, 0, 1 respectively.

P.2.24. klist_split() — *split a single linked list into two linked lists*

Synopsis

```
int klist_split(
```

```
    klist  *entry,
    klist **list1,
    klist **list2)
```

Input Arguments

`entry`
: the linked list entry in which we will be performing the split

Output Arguments

`list1`
: If not NULL, then the head of the first linked list is passed back.

`list2`
: If not NULL, then the head of the second linked list is passed back.

Returns

TRUE if we were able to split the linked list or FALSE upon failure.

Description

Split a single linked list into two linked lists. Given the entry in which we will break into the head of the new second list. Since this is the head of the second list and the head of the first list is really the klist_head(entry), before the klist_split() routine is called, it is not necessary to pass back the heads of the two lists. But to make life easier if list1 or list2 are not NULL the respective heads will be initialized there.

P.2.25. klist_tail() — *locate the tail of the linked list*

Synopsis

```
klist *klist_tail(
    klist * list)
```

Input Arguments

`list`
: The current linked list to find the tail for.

Returns

The item's tail entry or NULL if the current list does not exist.

Description

Tries to locate the tail of the linked list, or last entry in the linked list. This is done by taking to current position and racing down the next links until the tail is found. The last klist structure is returned, NULL if the list is currently empty.

P.2.26. klist_to_array() — *convert the linked list into an array*

Synopsis

```
char **klist_to_array(
   klist * list,
   size_t * num)
```

Input Arguments

`list`

The current linked list in which the array of identifiers will be created.

Output Arguments

`num`

The number of entries in the array.

Returns

The newly malloc'ed array or NULL upon failure

Description

Creates an array from a linked list. This is done by malloc'ing an array of pointers and then loading the assigning the identifiers into each array.

P.2.27. kalias_list() — *returns a string array of aliases*

Synopsis

```
char **kalias_list(
   size_t * num_entries)
```

Output Arguments

`num_entries`

If this paramater is a valid address, it will hold the number of aliases in the array. If the parameter is passed in as NULL, this routine will ignore it.

Returns

A malloc'ed string array of aliases upon success conversion of the alias list, otherwise NULL is returned. Note you should use kfree_and_NULL to free the array, NOT the karray_free, since the array contents are shared.

Description

This routine converts the global alias list for files specified by each toolbox's $TOOLBOX/repos/Alias

file, and converts the alias names to an array of alphabetically sorted strings. This can be used for browsers to map aliases to real paths on disk.

Q. Simple Database Management

Q.1. Introduction to Database Management Routines

• *kdbm_check()* - check where file descripter is a valid kdbm file
• *kdbm_checkkey()* - check to see if a key exists in the database
• *kdbm_close()* - close a previously opened kdbm file
• *kdbm_delete()* - Remove the key and its associated data from the database.
• *kdbm_fetch()* - Find a key and return the associated data.
• *kdbm_firstkey()* - get the first key in the database
• *kdbm_fdopen()* - open the dbm file and initialize data structures for use
• *kdbm_getmachtype()* - gets the machine type for the database
• *kdbm_lseek()* - move read/write pointer of the key pointer
• *kdbm_nextkey()* - get the next key in the database
• *kdbm_open()* - open the dbm file and initialize data structures for use
• *kdbm_read()* - Find a key and reads the associated data.
• *kdbm_store()* - Add a new key/data pair to the database.
• *kdbm_write()* - Simple database write routine
• *kdbm_tell()* - indicate position of the key pointer
• *khash_copy()* - copy the hash table and all associated memory
• *khash_create()* - creates a hash table
• *khash_currkey()* - return current entry (key) within the hash
• *khash_location()* - finds location of entry within the hash table
• *khash_reinit()* - reinitializes the hash table to be empty
• *khash_firstkey()* - return the first entry (key) in the hash table
• *khash_lastkey()* - return the last entry (key) in the hash table
• *khash_nextkey()* - return the next entry (key) in the hash table
• *khash_prevkey()* - return previous entry (key) in the hash table
• *khash_value()* - polynomial conversion
• *khash_init()* - initialized the hash routines
• *khash_free()* - frees the hash table and all associated memory
• *khash_delete()* - delete an entry from the hash table
• *khash_clientdata()* - returns the clientdata of a hash entry
• *khash_check()* - check to see if a hash entry exists
• *khash_add()* - adds an entry to the hash table

Q.2. Definitions of Database Management Routines

Q.2.1. kdbm_check() — *check where file descripter is a valid kdbm file*

Synopsis

```
int kdbm_check(
   int fid)
```

Input Arguments

`fid`
: file descriptor

Returns

TRUE if it is a kdbm file and FALSE if it is not.

Description

This function is used check if the file descriptor points to a valid kdbm file.

Q.2.2. kdbm_checkkey() — *check to see if a key exists in the database*

Synopsis

```
int kdbm_checkkey(
     kdbm *dbm,
     kdatum key)
```

Input Arguments

`dbm`
: the database file

`key`
: the key to check for

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Check to see if a key exists. The key passed in is checked against the list of database keys.

Q.2.3. kdbm_close() — *close a previously opened kdbm file*

Synopsis

```
void kdbm_close(
     kdbm *dbm)
```

Input Arguments

`dbm`

open database pointer to be closed

Description

Close the dbm file and free all memory associated with the database file.

Q.2.4. kdbm_delete() — *Remove the key and its associated data from the database.*

Synopsis

```
int kdbm_delete(
   kdbm *dbm,
   kdatum key)
```

Input Arguments

`dbm`

open database pointer

Returns

0 on success, -1 otherwise

Description

Remove the *key*ed item and the *key* from the database *dbm*. The file on disk is updated to reflect the structure of the new database before returning from this procedure.

Q.2.5. kdbm_fetch() — *Find a key and return the associated data.*

Synopsis

```
kdatum kdbm_fetch(
      kdbm *dbm,
      kdatum key)
```

Input Arguments

`dbm`
: open database pointer

`key`
: database key to search for

Returns

The data associated with the specified key. If the *key* is not a part of the database, the returned *data.dsize* and *data.dptr* will be 0 and NULL respectively

Description

Look up a given *key* and return the information associated with that *key*. The pointer in the structure that is returned is a pointer to dynamically allocated memory block.

Q.2.6. kdbm_firstkey() — *get the first key in the database*

Synopsis

```
kdatum kdbm_firstkey(
   kdbm *dbm)
```

Input Arguments

`dbm`
: the database pointer

Returns

The first key of the database. If dbm is NULL, or there are no keys in the database, the returned *key.dsize* and *key.dptr* will be 0 and NULL respectively.

Description

Find the first key in the database, and return it to the user.

Q.2.7. kdbm_fdopen() — *open the dbm file and initialize data structures for use*

Synopsis

```
kdbm *kdbm_fdopen(
      int  fid,
      int  flags,
      int  mode)
```

Input Arguments

`flags`
: open flags used during the kopen system call.

`mode`
: access modes to be used when creating a new database

Returns

a pointer to a open database structure

Description

Initialize dbm system. *fid* is a file descriptor to an open file. If the file has a size of zero bytes, a file initialization procedure is performed, setting up the initial structure in the file.

If *flags* is set to **KOPEN_RDONLY** the user wants to just read the database and any call to dbm_store or dbm_delete will fail. Many readers can access the database at the same time. If *flags* is set to **KOPEN_WRONLY**, the user wants both read and write access to the database and requires exclusive access. If *flags* is **KOPEN_WRONLY|KOPEN_CREAT**, the user wants both read and write access to the database and if the database does not exist, create a new one. If *flags* is **KOPEN_WRONLY|KOPEN_CREAT|KOPEN_TRUNC**, the user want a new database created, regardless of whether one existed, and wants read and write access to the new database. Any error detected will cause a return value of null and an approprate value will be in errno. If no errors occur, a pointer to the *kdbm file descriptor* will be returned.

Q.2.8. kdbm_getmachtype() — *gets the machine type for the database*

Synopsis

```
int kdbm_getmachtype(
   kdbm   *dbm)
```

Input Arguments

`dbm`
: the database file

Returns

returns the machine type or -1 on error

Description

This function is used to return the current machine type for the database.

Q.2.9. kdbm_lseek() — *move read/write pointer of the key pointer*

Synopsis

```
int kdbm_lseek(
    kdbm   *dbm,
    kdatum key,
    int    offset,
    int    whence)
```

Input Arguments

`dbm`
: the database file

`key`
: the key in which to seek for

`offset`
: the offset in which to seek

`whence`
: the control of how the offset will be applied

Returns

returns -1 or the new seeked position

Description

This function is used to seek to the position of the database relative to the given key.

Q.2.10. kdbm_nextkey() — *get the next key in the database*

Synopsis

```
kdatum kdbm_nextkey(
    kdbm *dbm)
```

Input Arguments

`dbm`
: the database pointer

Returns

The next key in the database. If *dbm* is NULL, or there are no more keys in the database, it will return *key.dsize* and *key.dptr* as 0 and NULL respectively.

Description

After an initial call to kdbm_firstkey, this routine can be used to traverse the list of keys in a database.

Q.2.11. kdbm_open() — *open the dbm file and initialize data structures for use*

Synopsis

```
kdbm *kdbm_open(
    char *filename,
    int  flags,
    int  mode)
```

Input Arguments

`filename`
: file name of database to open

`flags`
: open flags used during the kopen system call.

`mode`
: access modes to be used when creating a new database

Returns

a pointer to a open database structure

Description

Initialize dbm system. *Filename* is a pointer to the file name to be opened as a database file. If the file has a size of zero bytes, a file initialization procedure is performed, setting up the initial structure in the file.

If *Iflags* is set to **KOPEN_RDONLY** the user wants to just read the database and any call to dbm_store or dbm_delete will fail. Many readers can access the database at the same time. If *flags* is set to **KOPEN_WRONLY**, the user wants both read and write access to the database and requires exclusive access. If *flags* is **KOPEN_WRONLY|KOPEN_CREAT**, the user wants both read and write access to the database and if the database does not exist, create a new one. If *flags* is **KOPEN_WRONLY|KOPEN_CREAT|KOPEN_TRUNC**, the user want a new database created, regardless of whether one existed, and wants read and write access to the new database. Any error detected will cause a return value of null and an approprate value will be in errno. If no errors occur, a pointer to the *kdbm file descriptor* will be returned.

Q.2.12. kdbm_read() — *Find a key and reads the associated data.*

Synopsis

```
int kdbm_read(
   kdbm   *dbm,
   kdatum key,
   kaddr   data,
   int     num,
   int     type)
```

Input Arguments

`dbm`
: the database file

`key`
: the key in which to read from

`data`
: the data array to read into

`num`
: number of data points to read

`type`
: the type of data points to read from

Returns

the number of data points actually read, or -1 upon error

Description

Look up a given *key* and return the information associated with that *key*. The pointer in the structure that is returned is a pointer to dynamically allocated memory block. This is similar to the kdbm_fetch routine, except that kdbm_read supports incremental reads.

Q.2.13. kdbm_store() — *Add a new key/data pair to the database.*

Synopsis

```
int kdbm_store(
   kdbm *dbm,
   kdatum key,
   kdatum data,
   int flags)
```

Input Arguments

`dbm`
: open database pointer

`key`
: key to store the information under

`data`
: data to store in the database

`flags`
: data overwrite options

Returns

0 on success, -1 otherwise

Description

Add a new element to the database. *Data* is keyed by *key*. The file on disk is updated to reflect the structure of the new database before returning from this procedure. The *flags* define the action to take when the *key* is already in the database. The value **KDBM_REPLACE** asks that the old data be replaced by the new *data*. The value **KDBM_INSERT** asks that an error be returned and no action taken. A return value of 0 means no errors. A return value of -1 means that the item was not stored in the data base because the caller was not an official writer, or it means that the item was not stored because the argument *flags* was **KDBM_INSERT** and the *key* was already in the database.

Q.2.14. kdbm_write() — *Simple database write routine*

Synopsis

```
int kdbm_write(
   kdbm   *dbm,
   kdatum key,
   kaddr   data,
   int     num,
   int     type)
```

Input Arguments

`dbm`
: open database file pointer

`key`
: data base key to write to

`data`
: data to write to key

`num`
: number of data items to write

`type`
: type of data to write out

Returns

the number of data points actually written, or -1 upon error

Description

Add a new element to the database. *Data* is keyed by *key*. The file on disk is updated to reflect the structure of the new database before returning from this procedure. This routine is similar to kdbm_store except that it allows incremental writes to this key in the database.

Q.2.15. kdbm_tell() — *indicate position of the key pointer*

Synopsis

```
int kdbm_tell(
   kdbm   *dbm,
   kdatum key)
```

Input Arguments

`dbm`
: the database file

`key`
: the key in which to seek for offset - the offset in which to seek whence - the control of how the offset will be applied

Returns

returns -1 or the new seeked position

Description

This function is used to get the position of the database relative to the given key.

Q.2.16. khash_copy() — *copy the hash table and all associated memory*

Synopsis

```
int khash_copy(

  khash  *hash,
  kaddr (*routine)(kaddr) )
```

Input Arguments

`hash`
: Pointer to the hash table to be copied

`routine`
: A routine to copy the clientdata pointer for each hash array member. If routine is NULL, then it simply copies the address of the clientdata pointer.

Returns

TRUE (1) on success, FALSE (0) otherwise.

Description

This function copies the hash array and all associated memory.

Q.2.17. khash_create() — *creates a hash table*

Synopsis

```
khash *khash_create(
   int id_type,
   int clientdata_type,
   size_t size)
```

Input Arguments

`id_type`
: The data type for hash entry identifiers

`clientdata_type`
: The data type for the hash entry's clientdata

`size`
: The initial hash size; we recommend a prime number estimate of the expected hash size

Returns

A pointer to the newly created hash table on success; NULL on failure

Description

This routine creates a hash table. A hash table is an efficient alternative to an array, appropriate in situations where each piece of data to be stored can be assigned a unique identifier.

Hash entries consist of:

1) an identifier of the appropriate data type
2) an associated piece of data (clientdata).

After the hash table is created, entries may be added and deleted, the associated clientdata retrieved and checked.

When creating the hash table, you must specify an initial size for the hash table. Note that this is a "best guess" size; it will be automatically adjusted if it is not big enough to accomodate all the hash entries that are subsequently added. A prime number is recommended for optimal performance.

You must also specify the data type of the identifier which will be used to locate entries, and the data type of the clientdata. The data types specified for the identifier and the clientdata must be adhered to with all subsequent khash_xxx() calls, or results are not predictable.

Supported data types for both the hash table entry identifiers and the hash entries include:

```
KBYTE    - hash array of characters
KUBYTE   - hash array of unsigned characters
KSHORT   - hash array of short integers
KUSHORT  - hash array of unsigned short integers
KINT     - hash array of integers
KUINT    - hash array of unsigned integers
KLONG    - hash array of long integers
KULONG   - hash array of unsigned long integers
KFLOAT   - hash array of floating point numbers
KDOUBLE  - hash array of double precision numbers
KSTRING  - hash array of strings
KSTRUCT  - hash array of pointers to structures
KLOGICAL - hash array of TRUE/FALSE values
KDATUM   - hash array of pointers to kdatum structures
```

and any data type created via the kstruct_define call.

Note that khash_init() must be called once to initialize the hash routines before khash_create() is called to create a hash table.

Examples

For example, the following code creates a hash table of an initial size of 11. Stored in it will be user-

defined data structures that will be retrieved using a string identifier.

```
khash *hash_table;

khash_init();
hash_table = khash_create(KSTRING, KSTRUCT, 11);
```

Q.2.18. khash_currkey() — *return current entry (key) within the hash*

Synopsis

```
int khash_currkey(

  khash  *hash,
  <id_type> *identifier,
  <clientdata_type>  *clientdata)
```

Input Arguments

`hash`
: Pointer to the the hash table

Output Arguments

`identifier`
: Returns the identifier of the entry at this location

`clientdata`
: Returns the clientdata of the entry at this location

Returns

TRUE (1) on success, FALSE (0) if the hash table is empty

Description

This function returns the current entry in the hash table. The internal index is not changed. If the entry at the current is not valid then FALSE is returned.

Q.2.19. khash_location() — *finds location of entry within the hash table*

Synopsis

```
long khash_location(

  khash *hash,
  <id_type> identifier)
```

Input Arguments

`hash`
: pointer to the hash table

`identifier`
: The identifier of the desired hash entry, of data type consistent with what was specified for identifiers during hash table creation

Returns

The index of the entry if the entry was found; -1 if an entry with that identifier cannot be found or if the identifier specified is not valid.

Description

This function returns the location in the hash array of the entry with the specified identifier. This is sometimes useful when the hash array index is needed for temporary amount of time. This routine does *not* update internal key index used by the routines khash_nextkey(), khash_prevkey(), khash_currkey().

Q.2.20. khash_reinit() — *reinitializes the hash table to be empty*

Synopsis

```
int khash_reinit(

  khash  *hash,
  void  (*routine)(kaddr) )
```

Input Arguments

`hash`
: Pointer to the hash table to be reinitialized

`routine`
: If desired, you pass in a routine to free the clientdata pointer for each hash table entry. Note that this is recommended for hash tables using user-defined data structures as the clientdata, otherwise a memory

leak will occur. If routine is passed in as NULL, then no action is performed.

Returns

TRUE (1) on success, FALSE (0) otherwise.

Description

This function reinitializes the hash table to be empty. This is useful if you wish to reuse a hash table.

Q.2.21. khash_firstkey() — *return the first entry (key) in the hash table*

Synopsis

```
int khash_firstkey(

  khash  *hash,
  <id_type> *entry,
  <clientdata_type>  *clientdata)
```

Input Arguments

`hash`

Pointer to the hash table

Output Arguments

`clientdata`

returns the clientdata of the first hash entry

Returns

TRUE (1) on success, FALSE (0) if the hash table is empty

Description

This function returns the first entry (key) in the hash table.

It also sets the internal index to the first entry, so that it can be used in conjunction with khash_nextkey(), and khash_currkey().

For example, the following code can be used to iterate forwards through every entry in the hash table:

```
if (!khash_firstkey(hash_table, &identifier, &clientdata))
    return;
do {
```

```
    do something with identifier and/or clientdata
} while (khash_nextkey(hash_table, &identifier, &clientdata));
```

Q.2.22. khash_lastkey() — *return the last entry (key) in the hash table*

Synopsis

```
int khash_lastkey(

  khash  *hash,
  <id_type> *identifier,
  <clientdata_type>  *clientdata)
```

Input Arguments

`hash`
: Pointer to the the hash table

Output Arguments

`identifier`
: Returns the identifier of the last entry

`clientdata`
: Returns the clientdata of the last entry

Returns

TRUE (1) on success, FALSE (0) if the hash is empty

Description

This function returns the last entry (key) in the hash table.

It also sets the internal index to the last entry, so that it can be used in conjunction with khash_prevkey(), khash_currkey().

For example, the following code can be used to iterate backwards through every entry in the hash table:

```
if (!khash_lastkey(hash_table, &identifier, &clientdata))
    return;
do {
    do something with identifier and/or clientdata
} while (khash_prevkey(hash_table, &identifier, &clientdata));
```

Q.2.23. khash_nextkey() — *return the next entry (key) in the hash table*

Synopsis

```
int khash_nextkey(

  khash  *hash,
  <id_type> *identifier,
  <clientdata_type>  *clientdata)
```

Input Arguments

`hash`
: Pointer to the hash table

Output Arguments

`identifier`
: Returns the identifier of the entry at this location

`clientdata`
: Returns the clientdata of the entry at this location

Returns

TRUE (1) on success, FALSE (0) if the hash table is empty

Description

This function returns the next entry (key) in the hash table.

It also sets the internal key index to the next entry, so that it can be used in conjunction with khash_nextkey(), khash_prevkey(), khash_currkey().

For example, the following code can be used to iterate forwards through every entry in the hash table:

```
if (!khash_firstkey(hash_table, &identifier, &clientdata))
    return;
do {
    do something with identifier and/or clientdata
} while (khash_nextkey(hash_table, &identifier, &clientdata));
```

Q.2.24. khash_prevkey() — *return previous entry (key) in the hash table*

Synopsis

```
int khash_prevkey(

  khash  *hash,
  <id_type> *identifier,
  <clientdata_type>  *clientdata)
```

Input Arguments

`hash`
: Pointer to the hash table

Output Arguments

`identifier`
: returns the identifier of the entry at this location

`clientdata`
: returns the clientdata of the entry at this location

Returns

TRUE (1) on success, FALSE (0) if the hash table is empty

Description

This function returns the previous entry in the hash table.

It also sets the internal index to the previous entry, so that it can be used in conjunction with khash_nextkey(), khash_prevkey(), khash_currkey().

For example, the following code can be used to iterate backwards through every entry in the hash table:

```
if (!khash_lastkey(hash_table, &identifier, &clientdata))
    return;
do {
    do something with identifier and/or clientdata
} while (khash_prevkey(hash_table, &identifier, &clientdata));
```

Q.2.25. khash_value() — *polynomial conversion*

Synopsis

```
unsigned long int khash_value(

    char *data,
    ssize_t length)
```

Input Arguments

`data`
: the data in which to create the has for

`length`
: the length of the hash to be used. If -1, then the length is computed using kstrlen().

Returns

returns the associated hash for the given data.

Description

polynomial conversion ignoring overflows [this seems to work remarkably well, in fact better then the ndbm hash function. Replace at your own risk]

use: 65599 nice.
65587 even better.

Q.2.26. khash_init() — *initialized the hash routines*

Synopsis

```
void khash_init(void)
```

Description

This function initializes the hash routines, it must be called before any of the khash_xxx() routines can be used.

Q.2.27. khash_free() — *frees the hash table and all associated memory*

Synopsis

```
int khash_free(

  khash  *hash,
  void  (*routine)(kaddr) )
```

Input Arguments

`hash`
: the hash table to be freed

`routine`
: If desired, you pass in a routine to free the clientdata pointer for each hash table entry. Note that this is recommended for hash tables using user-defined data structures as the clientdata. If routine is passed as NULL, then kfree() will be called to free the clientdata.

Returns

TRUE (1) on success, FALSE (0) otherwise.

Description

This function frees the hash table and all associated memory.

Q.2.28. khash_delete() — *delete an entry from the hash table*

Synopsis

```
int khash_delete(

  khash *hash,
  <id_type>  identifier)
```

Input Arguments

`hash`
: pointer to the hash table

`identifier`
: The identifier of the hash entry to be deleted, of data type consistent with what was specified for identifiers during hash table creation

Returns

TRUE (1) on success, FALSE (0) otherwise.

Description

This function deletes the entry with the specified identifier from the hash table.

This routine does **not** update internal key index used by the routines khash_nextkey(), khash_prevkey(), khash_currkey(). If the entry being deleted has the key index then index is left unmodified, but khash_currkey() will return FALSE until the key index is updated to a valid index.

Q.2.29. khash_clientdata() — *returns the clientdata of a hash entry*

Synopsis

```
int khash_clientdata(

  khash   *hash,
  <id_type>   identifier,
  <clientdata_type> *clientdata)
```

Input Arguments

`hash`

pointer to the hash table

`identifier`

The identifier of the desired hash entry, of data type consistent with what was specified for identifiers during hash table creation

`clientdata`

pointer used to return the clientdata, of data type consistent with what was specified for clientdata during hash table creation Note: to avoid compile warnings on some architectures, this parameter should be cast to (kaddr).

Returns

TRUE (1) if a hash entry with the specified identifier was found, FALSE (0) otherwise.

Description

Given an identifier, this routine returns the clientdata stored for the specified hash entry.

Note that this routine does **not** update internal key index used by the routines khash_nextkey(), khash_prevkey(), khash_currkey().

Q.2.30. khash_check() — *check to see if a hash entry exists*

Synopsis

```
int khash_check(

  khash *hash,
  <type>  entry)
```

Input Arguments

`hash`

pointer to the hash table identifier - The identifier of the desired hash entry, of data type consistent with what was specified for identifiers during hash table creation

Returns

TRUE (1) if there is a hash table entry with that identifier, FALSE (0) if there was not.

Description

This function checks to see if a hash entry with a particular identifier currently exists in the hash table

This routine does *not* update internal key index used by the routines khash_nextkey(), khash_prevkey(), khash_currkey().

Q.2.31. khash_add() — *adds an entry to the hash table*

Synopsis

```
int khash_add(

  khash *hash,
  <id_type>  identifier,
  <clientdata_type>   clientdata)
```

Input Arguments

`hash`

Pointer to the hash table to which to add the new entry

`identifier`

The identifier of the desired hash entry, of data type consistent with what was specified for identifiers during hash table creation

`clientdata`

Pointer used to return the clientdata, of data type consistent with what was specified for clientdata

during hash table creation Note: to avoid compile warnings on some architectures, this parameter should be cast to (kaddr).

Returns

TRUE (1) on success, FALSE (0) otherwise.

Description

Inserts an entry into the hash table. Entries consist of:

1) an identifier of the appropriate data type
2) an associated piece of data (clientdata).

Note that the data types for the identifier and the clientdata MUST be consistent with whatever was specified for the hash table when it was created using khash_create().

R. Attribute Management

R.1. Introduction to Attribute Management Routines

- *kattrs_init()* - Initialize the kattrs data type.
- *kattrs_create()* - create a new attribute list
- *kattrs_destroy()* - destroy an attribute list
- *kattrs_add()* - add a new attribute to an attribute list
- *kattrs_vadd()* - add a new attribute to an attribute list
- *kattrs_delete()* - delete an attribute
- *kattrs_check()* - check to see if an attribute exists
- *kattrs_query()* - query information about an attribute
- *kattrs_set()* - set an attribute of an attribute list
- *kattrs_vset()* - set an attribute of an attribute list
- *kattrs_get()* - get an attribute from an attribute list.
- *kattrs_vget()* - get an attribute from an attribute list.
- *kattrs_print()* - print an attribute
- *kattrs_search()* - search for a list of attribute names matching some criteria
- *kattrs_dup()* - duplicate an attribute from one list to another
- *kattrs_first()* - return the first entry (atom) within the kattrs
- *kattrs_last()* - return the last entry (atom) within the kattrs
- *kattrs_next()* - return the next entry (atom) within the kattrs
- *kattrs_prev()* - return the previous entry (atom) within the kattrs
- *kattrs_curr()* - return the current entry (atom) within the kattrs
- *katom_new()* - Create a new attribute atom
- *katom_vnew()* - Create a new attribute atom
- *katom_delete()* - delete the attribute

• *katom_get()* - Get the data associated with an atom
• *katom_vget()* - Get the data associated with an atom
• *katom_set()* - Set the data of an atom
• *katom_vset()* - Set the data of an atom
• *katom_match()* - match an atom.
• *katom_dup()* - clone an atom
• *katom_copy()* - Copy an atom.
• *katom_query()* - Query an atom for information
• *katom_print()* - print the value of an attribute
• *katom_set_methods()* - set method functions for an attribute

R.2. Definitions of Attribute Management Routines

R.2.1. kattrs_init() — *Initialize the kattrs data type.*

Synopsis

```
void kattrs_init(void)
```

Description

This routine must be called before any kattrs routines can be used.

R.2.2. kattrs_create() — *create a new attribute list*

Synopsis

```
kattrs* kattrs_create(
   int   dupstructs,
   kaddr clientdata)
```

Input Arguments

`dupstructs`
Duplicate structures when setting attributes of with a type of structure.

`clientdata`
The clientdata associated with the list.

Returns

A kattrs pointer on success, NULL on error.

Description

This function is used to create an attribute list.

R.2.3. kattrs_destroy() — *destroy an attribute list*

Synopsis

```
int
kattrs_destroy(kattrs *attrs)
```

Input Arguments

`attrs`

The attribute list to destroy.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to destroy an attribute list.

R.2.4. kattrs_add() — *add a new attribute to an attribute list*

Synopsis

```
int
kattrs_add(
   kattrs        *attrs,
   const kstring attribute,
   size_t        argsize,
   size_t        numargs,
   int           datatype,
   int           permanent,
   int             (*get)  (katom *, kaddr, kva_list *),
   int             (*set)  (katom *, kaddr, kva_list *),
   int             (*match)(katom *, katom *, kaddr, kaddr),
   int             (*copy) (katom *, katom *, kaddr, kaddr),
   int             (*query)(katom *, kaddr, size_t *, size_t *, int *, int *),
   int             (*print)(katom *, kaddr, kfile *),
   kaddr        clientdata,
   kaddr        calldata,
   kvalist)
```

Input Arguments

`attrs`
: The attribute list to create attributes in.

`attribute`
: The attribute name to create.

`argsize`
: The size of each argument.

`numargs`
: The number of arguments.

`permanent`
: A flag indicating whether or not the attribute is permanent or temporary. TRUE indicates permanent.

`get`
: Routine to use on this attribute if kattrs_get is called on it.

`set`
: Routine to use on this attribute if kattrs_set is called on it.

`match`
: routine to use on this attribute if kattrs_match is called on it.

`copy`
: routine to use when copying the attribute data.

`query`
: routine to use on this attribute if kattrs_query is called on it.

`print`
: routine to use on this attribute if kattrs_print is called on it.

`clientdata`
: Client-specific data.

`calldata`
: Client-specific call data.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to create an attribute that will be added to the specified attribute list.

R.2.5. kattrs_vadd() — *add a new attribute to an attribute list*

Synopsis

```
int
kattrs_vadd(
   kattrs       *attrs,
   const kstring attribute,
   size_t        argsize,
   size_t        numargs,
   int           datatype,
   int           permanent,
   int         (*get)  (katom *, kaddr, kva_list *),
   int         (*set)  (katom *, kaddr, kva_list *),
   int         (*match)(katom *, katom *, kaddr, kaddr),
   int         (*copy) (katom *, katom *, kaddr, kaddr),
   int         (*query)(katom *, kaddr, size_t *, size_t *, int *, int *),
   int         (*print)(katom *, kaddr, kfile *),
   kaddr         clientdata,
   kaddr  calldata,
   kva_list     *valist)
```

Input Arguments

`attrs`
: The attribute list to create attributes in.

`attribute`
: The attribute name to create.

`argsize`
: The size of each argument.

`numargs`
: The number of arguments.

`permanent`
: A flag indicating whether or not the attribute is permanent or temporary. TRUE indicates permanent.

`get`
: Routine to use on this attribute if kattrs_get is called on it.

`set`
: Routine to use on this attribute if kattrs_set is called on it.

`match`
: routine to use on this attribute if kattrs_match is called on it.

`copy`
: routine to use when copying the attribute data.

`query`
: routine to use on this attribute if kattrs_query is called on it.

`print`
: routine to use on this attribute if kattrs_print is called on it.

`clientdata`
: Client-specific data.

`calldata`

Client-specific call data.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to create an attribute that will be added to the specified attribute list. This routine is is the same as kattrs_add with one exception. It accepts an already opened kva_list instead of a variable argument list.

R.2.6. kattrs_delete() — *delete an attribute*

Synopsis

```
int
kattrs_delete(
   kattrs        *attrs,
   const kstring attribute)
```

Input Arguments

`attrs`

The attribute list to delete attributes in.

`attribute`

The name of the attribute to delete.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to delete an attribute from a given attribute list.

R.2.7. kattrs_check() — *check to see if an attribute exists*

Synopsis

```
int
kattrs_check(
   kattrs        *attrs,
   const kstring  attribute)
```

Input Arguments

`attrs`
: The attribute list to check an attribute in.

`attribute`
: The name of the attribute to query.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used check if an attribute exists within an attribute list.

R.2.8. kattrs_query() — *query information about an attribute*

Synopsis

```
int
kattrs_query(
   kattrs        *attrs,
   const kstring  attribute,
   kaddr          calldata,
   size_t        *numargs,
   size_t        *argsize,
   int           *datatype,
   int           *permanent)
```

Input Arguments

`attrs`
: The attribute list to query an attribute in.

`attribute`
: The name of the attribute to query.

`calldata`
: Client-specific call data.

Output Arguments

`numargs`
A pointer to a variable in which the number of arguments can be returned.

`argsize`
A pointer to a variable in which the size of an argument can be returned.

`datatype`
A pointer to a variable in which the datatype of the attribute can be returned.

`permanent`
A pointer to a variable in which the permanence flag of the attribute can be returned.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used query if an attribute has been created.

R.2.9. kattrs_set() — *set an attribute of an attribute list*

Synopsis

```
int
kattrs_set(
   kattrs       *attrs,
   const kstring attribute,
   kaddr         calldata,
   kvalist)
```

Input Arguments

`attrs`
The attribute list to in which to set attributes.

`calldata`
Client-specific call data.

`kvalist`
A variable argument list, in the form:

ATTRIBUTE_NAME, &value,

The attribute list must be terminated with the symbol

NULL

to signify the end of the variable argument list.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to set one or more attributes of an attribute list.

R.2.10. kattrs_vset() — *set an attribute of an attribute list*

Synopsis

```
int
kattrs_vset(
   kattrs       *attrs,
   const kstring attribute,
   kaddr         calldata,
   kva_list     *valist)
```

Input Arguments

`attrs`

The attribute list to in which to set attributes.

`calldata`

Client-specific call data.

`valist`

An opened variable argument list, in the form:

ATTRIBUTE_NAME, &value,

The attribute list must be terminated with the symbol

NULL

to signify the end of the variable argument list.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to set an attribute of an attribute list. This routine is is the same as kattrs_set with one exception. It accepts an already opened kva_list instead of a variable argument list.

R.2.11. kattrs_get() — *get an attribute from an attribute list.*

Synopsis

```
int
kattrs_get(
   kattrs        *attrs,
   const kstring  attribute,
   kaddr          calldata,
   kvalist)
```

Input Arguments

`calldata`

Client-specific call data.

`kvalist`

A variable argument list, in the form:

ATTRIBUTE_NAME1, &value1,
ATTRIBUTE_NAME2, &value2,

The attribute list must be terminated with the symbol

NULL

to signify the end of the variable argument list.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to get one or more attributes from an attribute list.

Given any kattrs structure, the following code will determine the object's type, name, and path: Note: the argument list must be terminated with the symbol NULL.

R.2.12. kattrs_vget() — *get an attribute from an attribute list.*

Synopsis

```
int
kattrs_vget(
   kattrs        *attrs,
   const kstring  attribute,
   kaddr          calldata,
   kva_list      *valist)
```

Input Arguments

`calldata`

Client-specific call data.

`valist`

A variable argument list, in the form:

ATTRIBUTE_NAME1, &value1,
ATTRIBUTE_NAME2, &value2,

The attribute list must be terminated with the symbol

NULL

to signify the end of the variable argument list.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to get one or more attributes from an attribute list. This routine is is the same as kattrs_get with one exception. It accepts an already opened kva_list instead of a variable argument list.

R.2.13. kattrs_print() — *print an attribute*

Synopsis

```
int
kattrs_print(
   kattrs       *attrs,
   const kstring attribute,
   kaddr         calldata,
   kfile        *outfile)
```

Input Arguments

`attribute`
: the attribute to be printed

`calldata`
: client-specific call data.

`outfile`
: the output file

Description

dump a single attribute

R.2.14. kattrs_search() — *search for a list of attribute names matching some criteria*

Synopsis

```
kstring *kattrs_search(
   kattrs       *attrs,
   const kstring filter,
   int           type,
   int           perm_attr,
   int           temp_attr,
   size_t       *num)
```

Input Arguments

`attrs`
: The attribute list ot search.

`filter`
: The regular expression to use to as the search key for the attribute name. if it's NULL, all names are added

`type`
: The type of data the attribute is storing. If it is specified as KUNDEFINED, all data types are selected.

`perm_attr`

If TRUE, it will search the list for permanent attributes.

`temp_attr`

If TRUE, it will search the list for temporary attributes.

Returns

A klist of attributes where the identifier returned matches the search criteria.

Description

Search the attribute list for a list of attributes that match a specified criteria. Type, permanence, and regular expression matching of the name is allowed.

R.2.15. kattrs_dup() — *duplicate an attribute from one list to another*

Synopsis

```
int
kattrs_dup(
   kattrs       *src,
   const kstring attribute,
   kattrs       *dest)
```

Input Arguments

`src`

the src attribute list to be dup'ed from

`attribute`

the attribute to be dup'ed

`dest`

the destination attribute list to be dup'ed

Description

duplicates a single attribute from one list to another

R.2.16. kattrs_first() — *return the first entry (atom) within the kattrs*

Synopsis

```
int kattrs_first(
   kattrs *attrs,
   kstring *attribute,
   katom   **atom)
```

Input Arguments

`attrs`
: the kattrs to be searched

Output Arguments

`attribute`
: returns the entry name at this location

`atom`
: returns the atom at this location

Returns

TRUE (1) on success, FALSE (0) if the attrs is empty

Description

This function returns the first entry within the kattrs. It also sets the internal index to the first entry, so that it can be used in conjunction with kattrs_next(), kattrs_prev(), kattrs_curr().

R.2.17. kattrs_last() — *return the last entry (atom) within the kattrs*

Synopsis

```
int kattrs_last(
   kattrs *attrs,
   kstring *attribute,
   katom   **atom)
```

Input Arguments

`attrs`
: the kattrs to be searched

Output Arguments

`attribute`
: returns the entry name at this location

`atom`

returns the atom at this location

Returns

TRUE (1) on success, FALSE (0) if the attrs is empty

Description

This function returns the last entry within the kattrs. It also sets the internal index to the last entry, so that it can be used in conjunction with kattrs_next(), kattrs_prev(), kattrs_curr().

R.2.18. kattrs_next() — *return the next entry (atom) within the kattrs*

Synopsis

```
int kattrs_next(
   kattrs *attrs,
   kstring *attribute,
   katom   **atom)
```

Input Arguments

`attrs`

the kattrs to be searched

Output Arguments

`attribute`

returns the entry name at this location

`atom`

returns the atom at this location

Returns

TRUE (1) on success, FALSE (0) if no entry

Description

This function returns the last entry within the kattrs. It also sets the internal index to the last entry, so that it can be used in conjunction with kattrs_next(), kattrs_prev(), kattrs_curr().

R.2.19. kattrs_prev() — *return the previous entry (atom) within the kattrs*

Synopsis

```
int kattrs_prev(
   kattrs *attrs,
   kstring *attribute,
   katom   **atom)
```

Input Arguments

`attrs`
: the kattrs to be searched

Output Arguments

`attribute`
: returns the entry name at this location

`atom`
: returns the atom at this location

Returns

TRUE (1) on success, FALSE (0) if no entry

Description

This function returns the previous entry within the kattrs. It also sets the internal index to the last entry, so that it can be used in conjunction with kattrs_next(), kattrs_prev(), kattrs_curr().

R.2.20. kattrs_curr() — *return the current entry (atom) within the kattrs*

Synopsis

```
int kattrs_curr(
   kattrs *attrs,
   kstring *attribute,
   katom   **atom)
```

Input Arguments

`attrs`
: the kattrs to be searched

Output Arguments

`attribute`
: returns the entry name at this location

`atom`

returns the atom at this location

Returns

TRUE (1) on success, FALSE (0) if no entry

Description

This function returns the current entry within the kattrs. It also sets the internal index to the last entry, so that it can be used in conjunction with kattrs_next(), kattrs_prev(), kattrs_curr().

R.2.21. katom_new() — *Create a new attribute atom*

Synopsis

```
katom *
katom_new(
   char  *attribute,
   int    datatype,
   size_t argsize,
   size_t numargs,
   int     permanent,
   int    (*get)    (katom *, kaddr, kva_list *),
   int    (*set)    (katom *, kaddr, kva_list *),
   int    (*match)  (katom *, katom *, kaddr, kaddr),
   int    (*copy)   (katom *, katom *, kaddr, kaddr),
   int    (*query)  (katom *, kaddr, size_t *, size_t *, int *, int *),
   int    (*print)  (katom *, kaddr, kfile *),
   kaddr  clientdata,
   kaddr  calldata,
   kvalist)
```

Input Arguments

`attribute`
: attribute name

`datatype`
: data type of the attribute data

`argsize`
: size of each argument

`numargs`
: number of argument

`permanent`
: TRUE if attribute should be stored

`get`
: get routine to use instead of default, NULL to use the default.

`set`
: set routine to use instead of default, NULL to use the default.

`match`
match routine to use instead of default, NULL to use the default.
`copy`
copy routine to use instead of default, NULL to use the default.
`query`
query routine to use instead of default, NULL to use the default.
`print`
print routine to use instead of default, NULL to use the default.
`clientdata`
client data to pass in to the handler functions
`calldata`
call data associated with the atom

Returns

the new katom structure

Description

This routine creates a new attribute atom structure

R.2.22. katom_vnew() — *Create a new attribute atom*

Synopsis

```
katom *
katom_vnew(
   char  *attribute,
   int    datatype,
   size_t argsize,
   size_t numargs,
   int     permanent,
   int    (*get)    (katom *, kaddr, kva_list *),
   int    (*set)    (katom *, kaddr, kva_list *),
   int    (*match)  (katom *, katom *, kaddr, kaddr),
   int    (*copy)   (katom *, katom *, kaddr, kaddr),
   int    (*query)  (katom *, kaddr, size_t *, size_t *, int *, int *),
   int    (*print)  (katom *, kaddr, kfile *),
   kaddr  clientdata,
   kaddr  calldata,
   kva_list *list)
```

Input Arguments

`attribute`
attribute name
`datatype`

data type of the attribute data

`argsize`
size of each argument

`numargs`
number of argument

`permanent`
TRUE if attribute should be stored

`get`
get routine to use instead of default, NULL to use the default.

`set`
set routine to use instead of default, NULL to use the default.

`match`
match routine to use instead of default, NULL to use the default.

`copy`
copy routine to use instead of default, NULL to use the default.

`query`
query routine to use instead of default, NULL to use the default.

`print`
print routine to use instead of default, NULL to use the default.

`clientdata`
client data to pass in to the handler functions

`calldata`
call data associated with the atom

`list`
open variable argument list to default

Returns

the new katom structure

Description

This routine creates a new attribute atom structure

R.2.23. katom_delete() — *delete the attribute*

Synopsis

```
void
katom_delete(kaddr data)
```

Input Arguments

`data`
the atom to delete cast to a kaddr

Description

This function deletes the atom, and frees all memory associated with the atom.

R.2.24. katom_get() — *Get the data associated with an atom*

Synopsis

```
int
katom_get(
   katom *atom,
   kaddr  calldata,
   kvalist)
```

Input Arguments

`atom`

the atom to get data out of

`calldata`

call data associated with the atom

Returns

TRUE on success, FALSE otherwise

Description

It takes a variable argument list as the source of variables to use in returning the atoms client data values.

R.2.25. katom_vget() — *Get the data associated with an atom*

Synopsis

```
int
katom_vget(
   katom *atom,
   kaddr  calldata,
   kva_list *list)
```

Input Arguments

`atom`

the atom to get data out of

`calldata`

call data associated with the atom

Returns

TRUE on success, FALSE otherwise

Description

It takes an open variable argument list as the source of variables to use in returning the atoms client data values.

R.2.26. katom_set() — *Set the data of an atom*

Synopsis

```
int
katom_set(
   katom *atom,
   kaddr  calldata,
   kvalist)
```

Input Arguments

`atom`

the atom to set data for

`calldata`

call data associated with the atom

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

It takes an open variable argument list as the source of data to use in setting the atom clientdata values.

R.2.27. katom_vset() — *Set the data of an atom*

Synopsis

```
int
katom_vset(
   katom *atom,
   kaddr  calldata,
   kva_list *list)
```

Input Arguments

`atom`
: the atom to set data for

`calldata`
: call data associated with the atom

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

It takes an open variable argument list as the source of data to use in setting the atom clientdata values.

R.2.28. katom_match() — *match an atom.*

Synopsis

```
int
katom_match(
   katom *atom1,
   katom *atom2,
   kaddr  calldata1,
   kaddr  calldata2)
```

Input Arguments

`atom1`
: the first attribute to compoare

`atom2`
: the second attribute to compare

`calldata1`
: the calldata associated with atom1

`calldata2`
: the calldata associated with atom2

Returns

TRUE (1) if attributes match, FALSE (0) otherwise

Description

This function compares two katoms, to see if they match. Determining whether they match or not is defined by the match function passed in to katom new. By default, it does compares the data values on scaler type data, string compares on strings, and the structure compares on structures.

R.2.29. katom_dup() — *clone an atom*

Synopsis

```
katom *
katom_dup(
   katom *atom)
```

Input Arguments

`atom`
: the atom structure to duplicate

Description

This function creates a new atom structure, and then copies all the atom data and setting to it.

R.2.30. katom_copy() — *Copy an atom.*

Synopsis

```
int
katom_copy(
   katom *atom1,
   katom *atom2,
   kaddr  calldata1,
   kaddr  calldata2)
```

Input Arguments

`atom1`
: the atom from which to copy the first one

`calldata1`
: the calldata associated with atom1

Output Arguments

`atom2`
: the atom to copy into

`calldata2`
: the calldata associated with atom2

Returns

TRUE on success, FALSE otherwise

Description

Copy the data and settings from one atom to another

R.2.31. katom_query() — *Query an atom for information*

Synopsis

```
int
katom_query(
   katom *atom,
   kaddr  calldata,
   size_t *numargs,
   size_t *argsize,
   int *datatype,
   int *permanent)
```

Input Arguments

`atom`
: the atom to query

`calldata`
: the calldata associated with the atom

Output Arguments

`numargs`
: number of arguments

`argsize`
: size of each argument

`datatype`
: datatype of each argument

`permanent`
: whether the attribute is permanent or not

Returns

TRUE if attribute exists, FALSE otherwise

Description

This routine takes and atom, and returns various information about the data stored in the atom. Note, any of the output parameters are passed in as NULL, will be skipped.

R.2.32. katom_print() — *print the value of an attribute*

Synopsis

```
int
katom_print(
   katom *atom,
   kaddr calldata,
   kfile *outfile)
```

Input Arguments

`atom`
: the attribute to print

`calldata`
: the calldata associated with the atom

`outfile`
: the file to print to

Returns

TRUE on success, FALSE otherwise

Description

Print the value of an attribute. This routine uses the print method of the specified attribute, to print the data value of an attribute to the file specified by outfile. If the attribute does not have a print method associated with it, it will use the default print method.

R.2.33. katom_set_methods() — *set method functions for an attribute*

Synopsis

```
void
katom_set_methods(
   katom *atom,
   int   (*get)       (katom *, kaddr, kva_list *),
   int   (*set)       (katom *, kaddr, kva_list *),
   int   (*match)     (katom *, katom *, kaddr, kaddr),
   int   (*copy)      (katom *, katom *, kaddr, kaddr),
   int   (*query)     (katom *, kaddr, size_t *, size_t *, int *, int *),
   int   (*print)     (katom *, kaddr, kfile *),
   kaddr  clientdata)
```

Input Arguments

`atom`
: the atom to set methods on

`get`
: a pointer to the get function

`set`
: a pointer to the set function

`match`
: a pointer to the match function

`copy`
: a pointer to the copy function

`query`
: a pointer to the query function

`print`
: a pointer to the print function

`clientdata`
: a new clientdata to associate with the atom

Description

This functions sets the method functions for an attribute, If any of the following are NULL, the default method will be set.

S. Math Utilities

The following section details the only math utilities that are a part of the *klibc* library.

S.1. Introduction to the Math Utilities

The math utilities are:

- *kmax()* - return the greater of two values.
- *kmin()* - return the lessor of two values.
- *krange()* - return a ranged value

S.2. Definitions of the Math Utilities

S.2.1. kmax() — *return the greater of two values.*

Synopsis

```
kmax(x, y)
```

Input Arguments

`x`
: a variable of any base data type.

`y`
: a variable of any base data type.

Returns

the larger value of the two input arguments.

Description

The kmax function obtains the larger of the two input arguments. This is a macro, so any data type is supported.

S.2.2. kmin() — *return the lessor of two values.*

Synopsis

```
kmin(x, y)
```

Input Arguments

`x`
: a variable of any base data type.

`y`
: a variable of any base data type.

Returns

the smaller value of the two input arguments.

Description

The kmin function obtains the smaller of the two input arguments. This is a macro, so any data type is supported.

S.2.3. krange() — *return a ranged value*

Synopsis

```
krange(lower, value, upper)
```

Input Arguments

`lower`
: a variable of any base data type.

`value`
: a variable of any base data type.

`upper`
: a variable of any base data type.

Returns

the range value of the three input arguments.

Description

The krange function obtains the range value, given a value, and a lower and upper bound. The value returned will be lower >= value <= upper.

T. File Format Utilities

The following section details the file format utilities that are a part of the *kutils (libku.a)* library.

T.1. Introduction to the Ascii Format Utilities

The ASCII format utilities are:

- *ascii_create()* - creates a ascii structure and it's associated data
- *ascii_free()* - frees a ascii structure and it's associated data
- *ascii_readheader()* - reads a ascii header structure from the specified kfile id
- *ascii_read()* - read a ascii structure from the specified filename
- *ascii_fdread()* - read a ascii structure from the specified file descriptor
- *ascii_writeheader()* - writes a ascii header structure from the specified kfile id
- *ascii_write()* - write a ascii structure to the specified filename
- *ascii_fdwrite()* - write a ascii structure to the specified file descriptor

T.1.1. Definitions of the Ascii Format Utilities

T.1.2. ascii_create() — *creates a ascii structure and it's associated data*

Synopsis

```
ascii *ascii_create(void)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine creates an khoros ascii structure and it's associated data.

T.1.3. ascii_free() — *frees a ascii structure and it's associated data*

Synopsis

```
int ascii_free(
   ascii *image)
```

Input Arguments

`image`

a pointer to an khoros ascii structure that contains the image structure to be freed.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine frees an khoros ascii structure and it's associated data.

Side Effects

Once this routine is called no further reference the image should be made.

T.1.4. ascii_readheader() — *reads a ascii header structure from the specified kfile id*

Synopsis

```
ascii *ascii_readheader(
   int   fid)
```

Input Arguments

`fid`

the kfile id opened previously with kopen()

Returns

image - explanation

Description

This routines reads a ascii header structure and it's data into the specified filename. The routine uses the ktransport library which allows for remote reading of the ascii header.

T.1.5. ascii_read() — *read a ascii structure from the specified filename*

Synopsis

```
ascii *ascii_read(
   char  *filename)
```

Input Arguments

`filename`

filename in which we will be reading the ascii

Returns

returns the newly read ascii or NULL upon failure

Description

This routines reads a ascii structure and it's data from the specified filename. The routine uses the ktransport library which allows for remote reading of the data and it's header.

T.1.6. ascii_fdread() — *read a ascii structure from the specified file descriptor*

Synopsis

```
ascii *ascii_fdread(int fid)
```

Input Arguments

`fid`
: file descriptor in which we will be reading the ascii

Returns

returns the newly read ascii or NULL upon failure

Description

This routines reads a ascii structure and it's data from the specified file descriptor. The routine uses the ktransport library which allows for remote reading of the data and it's header.

T.1.7. ascii_writeheader() — *writes a ascii header structure from the specified kfile id*

Synopsis

```
int ascii_writeheader(
   int        fid,
   ascii *image)
```

Input Arguments

`fid`
: the kfile id opened previously with kopen()

`image`
: the image header to be written

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routines writes a ascii header structure and it's data into the specified filename. The routine uses the ktransport library which allows for remote writing of the ascii header.

T.1.8. ascii_write() — *write a ascii structure to the specified filename*

Synopsis

```
int ascii_write(
   char       *filename,
   ascii *image)
```

Input Arguments

`filename`
the filename in which we will be writing the ascii image and associated data

`image`
the ascii structure to be written

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routines writes a ascii structure and it's data into the specified filename. The routine uses the ktransport library which allows for remote writing of the data.

T.1.9. ascii_fdwrite() — *write a ascii structure to the specified file descriptor*

Synopsis

```
int ascii_fdwrite(
   int    fid,
   ascii *image)
```

Input Arguments

`fid`
the file descriptor in which we will be writing the ascii image and associated data

`image`
the ascii structure to be written

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routines writes a ascii structure and it's data into the specified file descriptor. The routine uses the ktransport library which allows for remote writing of the data.

T.2. Introduction to the Pixmap Format Utilities

The pixmap format utilities are:

• *xpm_create()* - creates a xpm structure and it's associated data
• *xpm_free()* - frees a xpm structure and it's associated data
• *xpm_readheader()* - reads a xpm header structure from the specified kfile id
• *xpm_read()* - read a xpm structure from the specified filename
• *xpm_fdread()* - read a xpm structure from the specified file descriptor
• *xpm_parse()* - parses a xpm string array and returns an xpm structure
• *xpm_writeheader()* - writes a xpm header structure from the specified kfile id
• *xpm_write()* - write a xpm structure to the specified filename
• *xpm_fdwrite()* - write a xpm structure to the specified file descriptor

T.2.1. Definitions of the Pixmap Format Utilities

T.2.2. xpm_create() — *creates a xpm structure and it's associated data*

Synopsis

```
xpm *xpm_create(
   void)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine creates an khoros xpm structure and it's associated data.

T.2.3. xpm_free() — *frees a xpm structure and it's associated data*

Synopsis

```
int xpm_free(
   xpm   *image)
```

Input Arguments

`image`

a pointer to an khoros xpm structure that contains the image structure to be freed.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine frees an khoros xpm structure and it's associated data.

Side Effects

Once this routine is called no further reference the image should be made.

T.2.4. xpm_readheader() — *reads a xpm header structure from the specified kfile id*

Synopsis

```
xpm *xpm_readheader(
   int   fid)
```

Input Arguments

`fid`

the kfile id opened previously with kopen()

Returns

image - explanation

Description

This routines reads a xpm header structure and it's data into the specified filename. The routine uses the ktransport library which allows for remote reading of the xpm header.

T.2.5. xpm_read() — *read a xpm structure from the specified filename*

Synopsis

```
xpm *xpm_read(
   char  *filename)
```

Input Arguments

`filename`

filename in which we will be reading the xpm

Returns

returns the newly read xpm or NULL upon failure

Description

This routines reads a xpm structure and it's data from the specified filename. The routine uses the ktransport library which allows for remote reading of the data and it's header.

T.2.6. xpm_fdread() — *read a xpm structure from the specified file descriptor*

Synopsis

```
xpm *xpm_fdread(int fid)
```

Input Arguments

`fid`

file descriptor in which we will be reading the xpm

Returns

returns the newly read xpm or NULL upon failure

Description

This routines reads a xpm structure and it's data from the specified file descriptor. The routine uses the ktransport library which allows for remote reading of the data and it's header.

T.2.7. xpm_parse() — *parses a xpm string array and returns an xpm structure*

Synopsis

```
xpm *xpm_parse(
   char **definition)
```

Input Arguments

`definition`

a pointer to an array of strings that defines a khoros xpm structure

Returns

returns the newly parsed xpm or NULL upon failure

Description

This routine parses an xpm string array and returns an xpm structure.

T.2.8. xpm_writeheader() — *writes a xpm header structure from the specified kfile id*

Synopsis

```
int xpm_writeheader(
   int        fid,
   xpm   *image)
```

Input Arguments

`fid`

the kfile id opened previously with kopen()

`image`

the image header to be written

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routines writes a xpm header structure and it's data into the specified filename. The routine uses the ktransport library which allows for remote writing of the xpm header.

T.2.9. xpm_write() — *write a xpm structure to the specified filename*

Synopsis

```
int xpm_write(
   char  *filename,
   xpm   *image)
```

Input Arguments

`filename`

the filename in which we will be writing the xpm image and associated data

`image`

the xpm structure to be written

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routines writes a xpm structure and it's data into the specified filename. The routine uses the ktransport library which allows for remote writing of the data.

T.2.10. xpm_fdwrite() — *write a xpm structure to the specified file descriptor*

Synopsis

```
int xpm_fdwrite(int fid, xpm *image)
```

Input Arguments

`fid`

the file descriptor in which we will be writing the xpm image and associated data

`image`

the xpm structure to be written

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routines writes a xpm structure and it's data into the specified file descriptor. The routine uses the ktransport library which allows for remote writing of the data.

U. Ini Parsing Utilities

The following section details the ini parsing utilities that are a part of the *kutils (libku.a)* library. These utilities allow developers to read and edit windows style ini files.

U.1. Introduction to the Ini Parsing Utilities

The ini parsing utilities are:

- *kini_parse()* - parse dot ini configuration file
- *kini_write()* - write an ini configuration file
- *kini_get_val()* - get value for an IniConf parameter
- *kini_set_val()* - set value for an IniConf parameter
- *kini_free()* - free memory associated with an IniConf structure

U.2. Definitions of the Ini Parsing Utilities

U.2.1. kini_parse() — *parse dot ini configuration file*

Synopsis

```
IniConf *
kini_parse(
   kstring ini_file,
   kstring default_section)
```

Input Arguments

`ini_file`
dot ini configuration file to parse

`default_section`
name of section to try if parameter is not found in named section (for get_value)

Returns

ptr to IniConf structure created on success, NULL on failure

Description

Parse the file specified by *ini_file*, and return a pointer to the information it contains.

U.2.2. kini_write() — *write an ini configuration file*

Synopsis

```
int
kini_write(
   kstring  ini_file,
   IniConf *cfg)
```

Input Arguments

`ini_file`
dot ini configuration file to write

`cfg`
ptr to IniConf structure to use

Returns

TRUE or FALSE

Description

Write the file specified by *ini_file*.

U.2.3. kini_get_val() — *get value for an IniConf parameter*

Synopsis

```
kstring
kini_get_val(
   IniConf *cfg,
   kstring  section_name,
   kstring  parameter_name)
```

Input Arguments

`cfg`
: IniConf configuration to use

`section_name`
: section to find parameter

`parameter_name`
: parameter to get value for

Returns

ptr to value string found on success, NULL on failure

Description

Get a value for an IniConf parameter. If the value is not found in the specified section, it searches the default section.

U.2.4. kini_set_val() — *set value for an IniConf parameter*

Synopsis

```
int
kini_set_val(
   IniConf *cfg,
   kstring  section_name,
   kstring  parameter_name,
   kstring  value)
```

Input Arguments

`cfg`
: IniConf configuration to use

`section_name`
: section to find parameter

`parameter_name`
: parameter to get value for

`value`
: new value for parameter

Returns

TRUE or FALSE

Description

Set a value for an IniConf parameter.

U.2.5. kini_free() — *free memory associated with an IniConf structure*

Synopsis

```
void
kini_free(
   IniConf *cfg)
```

Input Arguments

`cfg`
: ptr to IniConf structure to free

Description

Free all memory associated with an IniConf structure.

V. Structure Passing Utilities

The following section details the structure passing utilities that are a part of the *kutils (libku.a)* library. These utilities allow developers to read, write, flatten, and unflatten data structures, in a machine independent way.

V.1. Introduction to the Structure Passing Utilities

The ini parsing utilities are:

• *kstruct_define()* - define a structure entry
• *kstruct_undefine()* - undefine a structure entry
• *kstruct_check()* - check to see if a datatype is defined
• *kstruct_free()* - frees a structure and any associated memory
• *kstruct_compare()* - compares two structures
• *kstruct_duplicate()* - duplicates a structure
• *kstruct_flatten()* - flattens a structure
• *kstruct_unflatten()* - unflattens data into a structure
• *kstruct_setinfo()* - override the info in a structure entry
• *kstruct_getinfo()* - retrieve the info in a structure entry

V.2. Definitions of the Structure Passing Utilities

V.2.1. kstruct_define() — *define a structure entry*

Synopsis

```
int
kstruct_define(
   const char *name,
   size_t size,
   int  (*read_struct)(int, kaddr, int),
   int  (*write_struct)(int, kaddr, int),
   int  (*parse_struct)(int, kaddr, int),
   int  (*print_struct)(int, kaddr, int),
   int  (*compare_struct)(kaddr, kaddr, int),
   void (*free_struct)(kaddr, int))
```

Input Arguments

`name`
the structure name
`size`
the size of the structure ’sizeof(struct)’
`read_struct`

the read routine to call for reading

`write_struct`
the write routine to call for writing

`parse_struct`
the ascii write routine to call for parsing

`print_struct`
the ascii read routine to call for printing

`compare_struct`
the compare routine to call for comparing

`free_struct`
the free routine to call for freeing memory

Returns

returns the structure data type on success, KUNDEFINED otherwise

Description

This module is used to define a structure entry, which can then be used by the kread_generic()/kwrite_generic() routines to read and write arbitary structures.

V.2.2. kstruct_undefine() — *undefine a structure entry*

Synopsis

```
int
kstruct_undefine(
   int type)
```

Input Arguments

`type`
the structure type in which to undefine

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to undefine a structure entry, which is used by the kread_generic()/kwrite_generic() routines to read and write arbitary structures.

V.2.3. kstruct_check() — *check to see if a datatype is defined*

Synopsis

```
int
kstruct_check(
   char *name)
```

Input Arguments

`name`
: the structure name

Returns

returns the structure data type on success, KUNDEFINED otherwise

Description

This module is used to check to see if a datatype structure is defined. It takes the name of the structure originally used to define the entry. If the structure exists kstruct_check() returns the structure data type define, which can then be used by the kread_generic()/ kwrite_generic() routines to read and write arbitary structures.

V.2.4. kstruct_free() — *frees a structure and any associated memory*

Synopsis

```
void kstruct_free(

 kaddr data,
 int   type,
 int   freeself)
```

Input Arguments

`data`
: the structure to be freed

`type`
: the structure type in which to free free_self - a flag indicating whether or not to free itself. This parameter is passed to the free routine associated with the type.

Description

This module is used to call the free handler associated with a structure data definition. The structure free handler is called to free the assocaited data with the structure.

V.2.5. kstruct_compare() — *compares two structures*

Synopsis

```
int
kstruct_compare(
   kaddr data1,
   kaddr data2,
   int type)
```

Input Arguments

`data1`
: the first structure to be compared

`data2`
: the second structure to be compared

`type`
: the structure type in which to be compared

Returns

returns TRUE on success, FALSE otherwise otherwise

Description

This module is used to compare two structures to see if they are the same. This is done by calling the associated compare handler associated with a structure data definition. The structure compare handler is called to compare the assocaited data with the two structures.

V.2.6. kstruct_duplicate() — *duplicates a structure*

Synopsis

```
kaddr
kstruct_duplicate(
   kaddr data,
   int type,
   kaddr duplicate)
```

Input Arguments

`data`
: the structure to be duplicated

`type`
: the structure type in which to duplicate

Returns

returns the duplicated structure on success, NULL otherwise otherwise

Description

This module is used to duplicate structure according to the supplied data type define.

V.2.7. kstruct_flatten() — *flattens a structure*

Synopsis

```
kaddr
kstruct_flatten(
   kaddr data,
   int type,
   int ascii,
   size_t * size)
```

Input Arguments

`data`
: the structure to be flattened

`type`
: the structure type in which to flatten

`ascii`
: flatten the structure using the ascii methods

Output Arguments

`size`
: if not NULL, then returns the size of the flattened array

Returns

returns the flattened structure on success, NULL otherwise otherwise

Description

This module is used to flatten a structure according to the supplied data type define.

V.2.8. kstruct_unflatten() — *unflattens data into a structure*

Synopsis

```
kaddr
kstruct_unflatten(
   kaddr data,
   int type,
   int ascii,
   size_t size)
```

Input Arguments

`data`
: the flattened data to be unflattened into a structure

`type`
: the structure type in which to unflatten

`ascii`
: flatten the structure using the ascii methods

`size`
: the size of the flatten data

Returns

returns the unflattened structure on success, NULL otherwise otherwise

Description

This module is used to unflatten data into a structure according to the supplied data type define.

V.2.9. kstruct_setinfo() — *override the info in a structure entry*

Synopsis

```
int
kstruct_setinfo(
   int type,
   const char *name,
   ssize_t size,
   int (*read_struct)(int, kaddr, int),
   int (*write_struct)(int, kaddr, int),
   int (*parse_struct)(int, kaddr, int),
   int (*print_struct)(int, kaddr, int),
   int (*compare_struct)(kaddr, kaddr, int),
   void (*free_struct)(kaddr, int))
```

Input Arguments

`type`
: the structure type in which to replace certain info

`name`
: the structure name (if not NULL)

`size`
: the size of the structure (if not -1)

`read_struct`
: the read routine (if not NULL)

`write_struct`
: the write routine (if not NULL)

`parse_struct`
: the parse routine (if not NULL)

`print_struct`
: the print routine (if not NULL)

`compare_struct`
: the compare routine (if not NULL)

`free_struct`
: the free routine (if not NULL)

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to override the information currently stored within a structure entry.

The name, size, read and write structure can be overridden. If a non-NULL name is passed in then the current structure name is replaced. If the size is not -1 then the structure size is replaced. If the read or write structure routines are not NULL then the respective routines are replaced.

V.2.10. kstruct_getinfo() — *retrieve the info in a structure entry*

Synopsis

```
int
kstruct_getinfo(
   int type,
   char **name,
   size_t *size,
   int (**read_struct)(int, kaddr, int),
   int (**write_struct)(int, kaddr, int),
   int (**parse_struct)(int, kaddr, int),
   int (**print_struct)(int, kaddr, int),
   int (**compare_struct)(kaddr, kaddr, int),
   void (**free_struct)(kaddr, int))
```

Input Arguments

`type`
: the structure type in which to replace certain info

Output Arguments

`name`
: the structure name (if not NULL)

`size`
: the size of the structure (if not NULL)

`read_struct`
: the read routine (if not NULL)

`write_struct`
: the write routine (if not NULL)

`parse_struct`
: the parse routine (if not NULL)

`print_struct`
: the print routine (if not NULL)

`compare_struct`
: the compare routine (if not NULL)

`free_struct`
: the free routine (if not NULL)

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to retrieve the information currently stored within a structure entry.

The name, size, read and write structure can be retrieved. Only non-NULL entries will be returned. So only pointers to the desired information need to be passed in.

Table of Contents

Chapter 3

Mathematical Services

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 3 - Mathematical Services

A. Introduction

Mathematical Services is a collection of mathematical functions and utilities. It provides a portability layer on top of the standard C math library, *libm.a*, and a collection of routines that are not ordinarily part of the C math library, but are commonly-used in scientific data processing.

Eventually, Mathematical Services will encompass all of the functionality of the standard math library. In addition, it will contain many functions not provided by *libm*.

You are encouraged to make use of functions provided in Mathematical Services rather than using the standard *libm* counterparts, as the Mathematical Services variations are designed to be highly portable and efficient. Where possible, the VisiQuest versions of standard math functions actually use the *libm* versions for efficiency. However, when a math function is omitted from *libm* or when a math function is included in *libm* (but defective), an equivelent version implemented in *Mathematical Services* is used instead.

Mathematical Services augments the standard math utilities by offering functions for manipulating single- and double-precision complex data, generating random numbers and sequences with a variety of distributions, doing operations on matrices, generating periodic sequences, performing interpolation, converting data types and performing scaling and normalization. In addition, Mathematical Services provides C implementations of some FORTRAN 77 math functions that are not typically included in *libm*.

All Mathematical Services routines are located in the *klibm* library of the *bootstrap* toolbox. The #include files that are necessary to use this library are automatically included as part of bootstrap.h into every software object.

B. Complex Arithmetic

The following section details the math routines for complex arithmetic that are provided by VisiQuest. The complex data definitions (or typedefs), kcomplex and kdcomplex, are located in $BOOTSTRAP/include/machine/cdefs.h. Their prototypes are shown below.

```
typedef struct {
     float r,
     float i;
     }   kcomplex;

typedef struct {
     double r,
     double i;
     }   kdcomplex;
```

B.1. Introduction to Complex Arithmetic Utilities

The *klibm* library provides complex versions of the standard math library functions as well as basic *complex arithmetic operations*. These functions are intended to be used exclusively when operating on complex data.

The following complex library functions are available:

- *kcadd()* - add two complex numbers.
- *kcang()* - compute the radian angle of a complex number.
- *kccomp()* - construct a complex number from two real numbers.
- *kcconj()* - compute the conjugate of a complex number.
- *kcdiv()* - divide one complex number by another.
- *kcexp()* - complex exponential function
- *kcimag()* - return the imaginary component of a complex number.
- *kclog()* - complex natural logarithm
- *kclogmag()* - compute the log magnitude of a complex number.
- *kclogmagp1()* - compute the log magnitude of a complex number plus one.
- *kclogmagsq()* - compute the log magnitude squared of a complex number.
- *kclogmagsqp1()* - compute the log magnitude squared of a complex number plus one.
- *kcmag()* - compute the magnitude of a complex number.
- *kcmagsq()* - calculate the squared magnitude of a kcomplex number.
- *kcmult()* - multiply two complex numbers.
- *kcomp2dcomp()* - convert a kcomplex number to a kdcomplex number.
- *kcp2r()* - convert complex from polar coordinates to rectangular coordinates
- *kcpow()* - compute the value of a complex number raised to a complex power
- *kcr2p()* - convert complex from rectangular coordinates to polar coordinates
- *kcreal()* - return the real component of a complex number.
- *kcsqrt()* - calculate the complex square root of a complex argument.
- *kcsub()* - subtract one kcomplex number from another.

B.2. Definitions of Complex Arithmetic Utilities

B.2.1. kcadd() — *add two complex numbers.*

Synopsis

```
kcomplex kcadd(
   kcomplex a,
   kcomplex b)
```

Input Arguments

`a`
first operand of the complex addition

`b`
second operand of the complex addition

Returns

The result of the complex addition.

Description

kcadd() adds two complex numbers and returns the result. The two inputs are of type kcomplex. The typedef for kcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

B.2.2. kcang() — *compute the radian angle of a complex number.*

Synopsis

```
float kcang(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The angle in radians of the complex input argument.

Description

kcang() returns the radian angle of the input complex number.

B.2.3. kccomp() — *construct a complex number from two real numbers.*

Synopsis

```
kcomplex kccomp(
   double a,
   double b)
```

Input Arguments

`a`

the real component of the complex number

`b`

the imaginary component of the complex number

Returns

A complex number composed of the two inputs where the first input is the real component and the second input is the imaginary component.

Description

kccomp() constructs a complex number from two real numbers. The first argument becomes the real component and the second argument becomes the imaginary component of the resulting complex number.

Restrictions

This function returns an aggregate. If your compiler cannot deal with this, you are in trouble.

B.2.4. kcconj() — *compute the conjugate of a complex number.*

Synopsis

```
kcomplex kcconj(
   kcomplex a)
```

Input Arguments

`a`

complex number to be conjugated.

Returns

The complex conjugate of the input argument.

Description

kcconj() conjugates the complex input argument.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

B.2.5. kcdiv() — *divide one complex number by another.*

Synopsis

```
kcomplex kcdiv(
   kcomplex a,
   kcomplex b)
```

Input Arguments

`a`
: first operand of the complex division

`b`
: second operand of the complex division

Returns

The result of the complex division.

Description

kcdiv() divides the second complex number into the first and returns the result. The two inputs are of type kcomplex. The typedef for kcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Side Effects

This routine will issue warning messages via kinfo if b has a magnitude of 0. The result will not be accurate.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

B.2.6. kcexp() — *complex exponential function*

Synopsis

```
kcomplex kcexp(
   kcomplex x)
```

Input Arguments

`x`
: complex argument

Returns

a complex exponential of the input argument

Description

This function returns the complex exponential of the input argument.

B.2.7. kcimag() — *return the imaginary component of a complex number.*

Synopsis

```
float kcimag(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The value of the imaginary component of the input argument.

Description

kcimag() returns the value of the imaginary component of the complex input argument.

B.2.8. kclog() — *complex natural logarithm*

Synopsis

```
kcomplex kclog(
   kcomplex x)
```

Input Arguments

`x`

complex argument

Returns

a complex logarithm of the input argument.

Description

This function returns the complex natural logarithm of the input argument.

B.2.9. kclogmag() — *compute the log magnitude of a complex number.*

Synopsis

```
float kclogmag(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The magnitude of the complex input argument.

Description

kclogmag() returns the log magnitude of the input complex number.

B.2.10. kclogmagp1() — *compute the log magnitude of a complex number plus one.*

Synopsis

```
float kclogmagp1(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The magnitude of the complex input argument.

Description

kclogmag() returns the log magnitude of the input complex number plus one.

B.2.11. kclogmagsq() — *compute the log magnitude squared of a complex number.*

Synopsis

```
float kclogmagsq(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The magnitude of the complex input argument.

Description

kclogmag() returns the log magnitude of the input complex number.

B.2.12. kclogmagsqp1() — *compute the log magnitude squared of a complex number plus one.*

Synopsis

```
float kclogmagsqp1(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The magnitude of the complex input argument.

Description

kclogmag() returns the log magnitude of the input complex number plus one.

B.2.13. kcmag() — *compute the magnitude of a complex number.*

Synopsis

```
float kcmag(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The magnitude of the complex input argument.

Description

kcmag() returns the magnitude of the input complex number.

B.2.14. kcmagsq() — *calculate the squared magnitude of a kcomplex number.*

Synopsis

```
float kcmagsq(
    kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The magnitude squared of the input argument.

Description

kcmagsq() returns the squarred magnitude of the input argument.

B.2.15. kcmult() — *multiply two complex numbers.*

Synopsis

```
kcomplex kcmult(
   kcomplex a,
   kcomplex b)
```

Input Arguments

`a`
: first operand of the complex multiply

`b`
: second operand of the complex multiply

Returns

The result of the complex multiplication.

Description

kcsub() multiplies two complex numbers and returns the result. The two inputs are of type kcomplex. The typedef for kcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

B.2.16. kcomp2dcomp() — *convert a kcomplex number to a kdcomplex number.*

Synopsis

```
kdcomplex kcomp2dcomp(
   kcomplex a)
```

Input Arguments

`a`
: kcomplex number

Returns

the kdcomplex conversion of the input argument.

Description

kcomp2dcomp converts a single precision complex number in a typedef'd "kcomplex" structure into a double precision complex number in a typedef'd "kdcomplex" structure.

B.2.17. kcp2r() — *convert complex from polar coordinates to rectangular coordinates*

Synopsis

```
kcomplex kcp2r(
   kcomplex p)
```

Input Arguments

`p`

polar coordinate (kcomplex)

Returns

The rectangular coordinate complex value

Description

kdcr2p() convert a complex from polar coordinates to rectangular coordinates. Stored as the (real,imaginary) pair in a kcomplex data type. The polar coordinate input is also encoded in a kcomplex datatype, but the interpretation is a (radius,angle) pair.

B.2.18. kcpow() — *compute the value of a complex number raised to a complex power*

Synopsis

```
kcomplex kcpow(
   kcomplex x,
   kcomplex y)
```

Input Arguments

`x`

kcomplex number

`y`

kcomplex power

Returns

The kcomplex value of a complex number raised to a complex power.

Description

kcpow() returns the value of a complex number raised to a complex power

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using * an older C compiler that is not fully ANSI compliant, you may have * problems.

B.2.19. kcr2p() — *convert complex from rectangular coordinates to polar coordinates*

Synopsis

```
kcomplex kcr2p(
   kcomplex r)
```

Input Arguments

`r`

rectangular coordinate (kcomplex)

Returns

The polar coordinate complex value

Description

kdcr2p() convert a complex from rectangular coordinates to polar coordinates. The radius is stored as the real, and the angle as the imaginary.

B.2.20. kcreal() — *return the real component of a complex number.*

Synopsis

```
float kcreal(
   kcomplex a)
```

Input Arguments

`a`

complex number

Returns

The real component of the complex input argument.

Description

kcreal() returns the real component of the complex input argument.

B.2.21. kcsqrt() — *calculate the complex square root of a complex argument.*

Synopsis

```
kcomplex kcsqrt(
   kcomplex a)
```

Input Arguments

`a`

complex aargument

Returns

The complex square root of the input argument.

Description

kcsqrt() returns the complex square root of the complex argument passed as input.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

B.2.22. kcsub() — *subtract one kcomplex number from another.*

Synopsis

```
kcomplex kcsub(
   kcomplex a,
   kcomplex b)
```

Input Arguments

`a`

first operand of the complex subtraction

`b`

second operand of the complex subtraction

Returns

The result of the complex subtraction.

Description

kcsub() subtracts the second complex number from the first and returns the result. The two inputs are of type kcomplex. The typedef for kcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C. Double Complex Arithmetic

The following section details the math routines for *double complex arithmetic*. The double complex definition, `kdcomplex` can be found in $BOOTSTRAP/include/machine/cdefs.h.

C.1. Introduction to Double Complex Arithmetic Utilities

The *klibm* library provides double complex versions of the standard math library functions as well as basic double complex arithmetic operations. These functions are to be used exclusively when operating on double complex data. Typedef kdcomplex (double real, imaginary) is defined for double complex data.

The following routines are available:

• *kdcadd()* - add two double precision complex numbers.
• *kdcang()* - compute the radian angle of a double precision complex number.
• *kdccomp()* - construct a double precision complex number from two real numbers.
• *kdcconj()* - compute the conjugate of a double precision complex number.
• *kdccos()* - double complex cosine
• *kdccosh()* - double complex hyperbolic cosine
• *kdcdiv()* - divide one double precision complex number by another.
• *kdcexp()* - double complex exponential function
• *kdcimag()* - return the imaginary component of a double precision complex number.
• *kdclog()* - double complex natural logarithm
• *kdclogmag()* - compute the log magnitude of a double precision complex number.
• *kdclogmagp1()* - compute the log magnitude of a double precision complex number plus one.
• *kdclogmagsq()* - compute the log magnitude squared of a double precision complex number.
• *kdclogmagsqp1()* - compute the log magnitude squared of a double precision complex number plus one.
• *kdcmag()* - compute the magnitude of a double precision complex number.
• *kdcmagsq()* - calculate the squared magnitude of a double precision complex number.
• *kdcmult()* - multiply two double precision complex numbers.
• *kdcomp2comp()* - convert a kdcomplex number to a kcomplex number.
• *kdcp2r()* - convert double complex from polar coordinates to rectangular coordinates
• *kdcpow()* - compute the double complex value of a double complex number raised to a double complex power.

• *kdcr2p()* - convert double complex from rectangular coordinates to polar coordinates
• *kdcreal()* - return the real component of a double precision complex number.
• *kdcsin()* - double complex sine
• *kdcsinh()* - double complex hyperbolic sine
• *kdcsqrt()* - calculate the double precision complex square root of a double precision complex number.
• *kdcsub()* - subtract one double precision complex number from another.
• *kdctan()* - double complex tangent
• *kdctanh()* - double complex hyperbolic tangent
• *kdcomplex_to_arrays()* - separate array of double complex into real and imaginary arrays

C.2. Definitions of Double Complex Arithmetic Utilities

C.2.1. kdcadd() — *add two double precision complex numbers.*

Synopsis

```
kdcomplex kdcadd(
    kdcomplex a,
    kdcomplex b)
```

Input Arguments

`a`
first operand of the dcomplex addition

`b`
second operand of the dcomplex addition

Returns

The result of the kdcomplex addition.

Description

kdcadd() adds two double precision complex numbers and returns the result. The two inputs are of type kdcomplex. The typedef for kdcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.2. kdcang() — *compute the radian angle of a double precision complex number.*

Synopsis

```
double kdcang(
   kdcomplex a)
```

Input Arguments

`a`

double precision complex number

Returns

The angle in radians of the complex input argument. The angle will lie in the range of -Pi to +Pi.

Description

kdcang() returns the radian angle of the input double precision complex number.

C.2.3. kdccomp() — *construct a double precision complex number from two real numbers.*

Synopsis

```
kdcomplex kdccomp(
   double a,
   double b)
```

Input Arguments

`a`

the real component of the kdcomplex number

`b`

the imaginary component of the kdcomplex number

Returns

A kdcomplex number composed of the two inputs where the first input is the real component and the second input is the imaginary component.

Description

kdccomp() constructs a double precision complex number from two real numbers. The first argument becomes the real component and the second argument becomes the imaginary component of the resulting double precision complex number.

Restrictions

This function returns an aggregate. If your compiler cannot deal with this, you are in trouble.

C.2.4. kdcconj() — *compute the conjugate of a double precision complex number.*

Synopsis

```
kdcomplex kdcconj(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number to be conjugated.

Returns

The kdcomplex conjugate of the input argument.

Description

kdcconj() conjugates the double precision complex input argument.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.5. kdccos() — *double complex cosine*

Synopsis

```
kdcomplex kdccos(
   kdcomplex a)
```

Input Arguments

`a`

double complex argument

Returns

the double precision complex cosine of the argument.

Description

kdccos computes the double precision complex cosine of the input argument

C.2.6. kdccosh() — *double complex hyperbolic cosine*

Synopsis

```
kdcomplex kdccosh(
   kdcomplex a)
```

Input Arguments

`a`

double complex argument

Returns

the double precision complex hyperbolic cosine of the argument.

Description

kdccosh computes the double precision complex hyperbolic cosine of the input argument

C.2.7. kdcdiv() — *divide one double precision complex number by another.*

Synopsis

```
kdcomplex kdcdiv(
   kdcomplex a,
   kdcomplex b)
```

Input Arguments

`a`

first operand of the kdcomplex division

`b`

second operand of the kdcomplex division

Returns

The result of the kdcomplex division.

Description

kdcdiv() divides the second double precision complex number into the first and returns the result. The two inputs are of type kdcomplex. The typedef for kdcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Side Effects

This routine will issue warning messages via kinfo if b has a magnitude of 0. The result will not be accurate.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.8. kdcexp() — *double complex exponential function*

Synopsis

```
kdcomplex kdcexp(
    kdcomplex x)
```

Input Arguments

`x`

double complex argument

Returns

a double complex exponential of the input argument

Description

This function returns the double complex exponential of the input argument.

C.2.9. kdcimag() — *return the imaginary component of a double precision complex number.*

Synopsis

```
double kdcimag(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The value of the imaginary component of the input argument.

Description

kdcimag() returns the value of the imaginary component of the double precision complex input argument.

C.2.10. kdclog() — *double complex natural logarithm*

Synopsis

```
kdcomplex kdclog(
   kdcomplex x)
```

Input Arguments

`x`

double complex argument

Returns

a double complex natural logarithm of the input argument.

Description

This function returns the double complex natural logarithm of the input argument.

C.2.11. kdclogmag() — *compute the log magnitude of a double precision complex number.*

Synopsis

```
double kdclogmag(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The magnitude of the kdcomplex input argument.

Description

kdclogmag() returns the log magnitude of the input double precision complex number.

C.2.12. kdclogmagp1() — *compute the log magnitude of a double precision complex number plus one.*

Synopsis

```
double kdclogmagp1(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The magnitude of the kdcomplex input argument.

Description

kdclogmag() returns the log magnitude of the input double precision complex number.

C.2.13. kdclogmagsq() — *compute the log magnitude squared of a double precision complex number.*

Synopsis

```
double kdclogmagsq(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The magnitude of the kdcomplex input argument.

Description

kdclogmag() returns the log magnitude of the input double precision complex number.

C.2.14. kdclogmagsqp1() — *compute the log magnitude squared of a double precision complex number plus one.*

Synopsis

```
double kdclogmagsqp1(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The magnitude of the kdcomplex input argument.

Description

kdclogmag() returns the log magnitude of the input double precision complex number.

C.2.15. kdcmag() — *compute the magnitude of a double precision complex number.*

Synopsis

```
double kdcmag(
    kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The magnitude of the kdcomplex input argument.

Description

kdcmag() returns the magnitude of the input double precision complex number.

C.2.16. kdcmagsq() — *calculate the squared magnitude of a double precision complex number.*

Synopsis

```
double kdcmagsq(
    kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The magnitude squared of the input argument.

Description

kdcmagsq() returns the squarred magnitude of the input argument.

C.2.17. kdcmult() — *multiply two double precision complex numbers.*

Synopsis

```
kdcomplex kdcmult(
   kdcomplex a,
   kdcomplex b)
```

Input Arguments

`a`

first operand of the kdcomplex multiply

`b`

second operand of the dcomplex multiply

Returns

The result of the kdcomplex multiplication.

Description

kdcsub() multiplies two double precision complex numbers and returns the result. The two inputs are of type kdcomplex. The typedef for kdcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.18. kdcomp2comp() — *convert a kdcomplex number to a kcomplex number.*

Synopsis

```
kcomplex kdcomp2comp(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

the complex conversion of the input argument.

Description

kdcomp2comp converts a double precision complex number in a typedef'd "kdcomplex" structure into a single precision complex number in a typedef'd "kcomplex" structure.

C.2.19. kdcp2r() — *convert double complex from polar coordinates to rectangular coordinates*

Synopsis

```
kdcomplex kdcp2r(
   kdcomplex p)
```

Input Arguments

`p`

kdcomplex number

Returns

The rectangular coordinate double complex value

Description

kdcr2p() convert a double complex from polar coordinates to rectangular coordinates. Stored as the (real,imaginary) pair. Where the polar coordinate input is the (radius,angle) pair. This pair is stored in the same structure used for dcomplex data, except that what was formerly the real part is now the magnitude, and what was the imaginary component will now be the phase in radians.

C.2.20. kdcpow() — *compute the double complex value of a double complex number raised to a double complex power.*

Synopsis

```
kdcomplex kdcpow(
   kdcomplex x,
   kdcomplex y)
```

Input Arguments

`x`

kdcomplex number

`y`

kdcomplex power

Returns

The kdcomplex value of a double complex number raised to a double complex power.

Description

kdcpow() returns the double complex value of a double complex number raised to a double complex power.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.21. kdcr2p() — *convert double complex from rectangular coordinates to polar coordinates*

Synopsis

```
kdcomplex kdcr2p(
   kdcomplex r)
```

Input Arguments

`r`

kdcomplex number

Returns

The polar coordinate double complex value

Description

kdcr2p() convert a double complex from rectangular coordinates to polar coordinates. The radius is stored as the real, and the angle as the imaginary.

C.2.22. kdcreal() — *return the real component of a double precision complex number.*

Synopsis

```
double kdcreal(
   kdcomplex a)
```

Input Arguments

`a`

kdcomplex number

Returns

The real component of the kdcomplex input argument.

Description

kdcreal() returns the real component of the double precision complex input argument.

C.2.23. kdcsin() — *double complex sine*

Synopsis

```
kdcomplex kdcsin(
   kdcomplex a)
```

Input Arguments

`a`

double complex argument

Returns

the double precision complex sine of the argument.

Description

kdcsin computes the double precision complex sine of the input argument

C.2.24. kdcsinh() — *double complex hyperbolic sine*

Synopsis

```
kdcomplex kdcsinh(
   kdcomplex a)
```

Input Arguments

`a`

double complex argument

Returns

the double precision complex hyperbolic sine of the argument.

Description

kdcsinh computes the double precision complex hyperbolic sine of the input argument

C.2.25. kdcsqrt() — *calculate the double precision complex square root of a double precision complex number.*

Synopsis

```
kdcomplex kdcsqrt(
    kdcomplex a)
```

Input Arguments

`a`

kdcomplex argument

Returns

The kdcomplex square root of the input argument.

Description

kdcsqrt() returns the double precision complex square root of the kdcomplex argument passed as input.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.26. kdcsub() — *subtract one double precision complex number from another.*

Synopsis

```
kdcomplex kdcsub(
    kdcomplex a,
    kdcomplex b)
```

Input Arguments

`a`

first operand of the kdcomplex subtraction

`b`

second operand of the kdcomplex subtraction

Returns

The result of the kdcomplex subtraction.

Description

kdcsub() subtracts the second double precision complex number from the first and returns the result. The two inputs are of type kdcomplex. The typedef for kdcomplex can be found in $BOOTSTRAP/machine/cdefs.h.

Restrictions

This function returns a structure, not a pointer to a structure. This is allowed in ANSI C. However, if you are using an older C compiler that is not fully ANSI compliant, you may have problems.

C.2.27. kdctan() — *double complex tangent*

Synopsis

```
kdcomplex kdctan(
    kdcomplex a)
```

Input Arguments

`a`

double complex argument

Returns

the double precision complex tangent of the argument.

Description

kdctan computes the double precision complex tangent of the input argument

C.2.28. kdctanh() — *double complex hyperbolic tangent*

Synopsis

```
kdcomplex kdctanh(
   kdcomplex a)
```

Input Arguments

`a`

double complex argument

Returns

the double precision complex hyperbolic tangent of the argument.

Description

kdctanh computes the double precision complex hyperbolic tangent of the input argument

C.2.29. kdcomplex_to_arrays() — *separate array of double complex into real and imaginary arrays*

Synopsis

```
int kdcomplex_to_arrays(
   kdcomplex *c,
   int num,
   double **r,
   double **i)
```

Input Arguments

`c`

a dcomplex array to be split into real & imag parts

Output Arguments

`r`

an array of real numbers (double *) derived from the real component of the input array of double complex pairs

`i`

an array of real numbers (double *) derived from the imaginary component of the input array of double complex pairs.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kdcomplex_to_arrays returns the real and imaginary portion of a kdcomplex array in two double arrays.

D. Matrix Arithmetic

The following section details the matrix routines for floating-point and double-precision arithmetic.

D.1. Introduction to Matrix Arithmetic Routines

A small collection of matrix operators has been implemented for the purpose of providing basic *matrix computation* capabilities that are very efficient. A few simple matrix functions converted from the LINPACK (a library of advanced linear algebra functions originally written in FORTRAN at Oak Ridge National Labs) and BLAS (a library of basic linear algebra functions originally written in FORTRAN at Oak Ridge National Labs) are also available.

The following floating-point matrix functions are available:

• *kfmatrix_clear()* - zeros a matrix
• *kfmatrix_identity()* - set matrix to identity
• *kfmatrix_inner_prod()* - compute the inner product of two vectors.
• *kfmatrix_inverse()* - inverts a matrix.
• *kfmatrix_multiply()* - multiply two matrices
• *kfmatrix_princ_axis()* - obtain the principle axis of a covariance matrix.
• *kfmatrix_vector_prod()* - compute the matrix-vector product.

The following double-precision matrix functions are available:

• *kdmatrix_clear()* - zeros a matrix
• *kdmatrix_identity()* - set matrix to identity
• *kdmatrix_inner_prod()* - compute the inner product of two vectors.
• *kdmatrix_inverse()* - inverts a matrix.
• *kdmatrix_multiply()* - multiply two matrices
• *kdmatrix_princ_axis()* - obtain the principle axis of a covariance matrix.
• *kdmatrix_vector_prod()* - compute the matrix-vector product.

The following routines are from LINPACK and BLAS. Floating-point functions are indicated by ‘s’ and double precision functions are indicated by ‘d.’

• *klin_sgefa()* - factors a float matrix by gaussian elimination.
• *klin_sgedi()* - computes the determinate and inverse of a matrix
• *kblas_sscal()* - scale a float vector

• *kblas_saxpy()* - add two float vectors while scaling one
• *kblas_sswap()* - swap two float vectors
• *klin_dgefa()* - factors a double matrix by gaussian elimination.
• *klin_dgedi()* - computes the determinate and inverse of a matrix
• *kblas_dscal()* - scale a double vector
• *kblas_daxpy()* - add two double vectors while scaling one
• *kblas_dswap()* - swap two double vectors

D.2. Definitions of Matrix Arithmetic Routines

D.2.1. kfmatrix_clear() — *zeros a matrix*

Synopsis

```
int kfmatrix_clear(
   int    rows,
   int    cols,
   float *matrix)
```

Input Arguments

`rows`
number of rows in the matrix.

`cols`
number of columns in the matrix.

Output Arguments

`matrix`
the cleared matrix.

Returns

TRUE (1) on success, otherwise, it returns FALSE (0) and sets kerrno to KINVALID_PARAMETER if the input matrix dimensions are not positive.

Description

kfmatrix_clear() clears a matrix by setting all of its values to 0.0.

D.2.2. kfmatrix_identity() — *set matrix to identity*

Synopsis

```
int kfmatrix_identity(
   int    rows,
   int    cols,
   float *matrix)
```

Input Arguments

`rows`
the number of rows in the matrix.

`cols`
the number of columns in the matrix.

Output Arguments

`matrix`
output matrix stored in 1D array of floats.

Returns

TRUE (1) on success, otherwise it returns FALSE (0) and sets kerrno to KINTERNAL if the internal call to kfmatrix_clear fails.

Description

kfmatrix_identity() sets an input matrix to the identity matrix.

D.2.3. kfmatrix_inner_prod() — *compute the inner product of two vectors.*

Synopsis

```
float kfmatrix_inner_prod(
   float *x,
   float *y,
   int    n)
```

Input Arguments

`x`
first vector.

`y`
second vector.

`n`

number of components in each vector.

Returns

The float precision inner product of the two vectors.

Description

kfmatrix_inner_prod() computes the inner product of two vectors.

D.2.4. kfmatrix_inverse() — *inverts a matrix.*

Synopsis

```
int kfmatrix_inverse(
   float *matrix,
   int    order,
   float *outmatrix)
```

Input Arguments

`matrix`

input matrix stored in 1D array of floats

`order`

order of the matrix.

Output Arguments

`outmatrix`

the inverted matrix stored in 1D array

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function returns the inverse of the input matrix.

D.2.5. kfmatrix_multiply() — *multiply two matrices*

Synopsis

```
int kfmatrix_multiply(
   float *matrix1,
   int    rows1,
   int    cols1,
   float *matrix2,
   int    rows2,
   int    cols2,
   float *outmat)
```

Input Arguments

`matrix1`
: input matrix stored in 1D array of floats

`rows1`
: number of rows in the first matrix.

`cols1`
: number of columns in the first matrix.

`matrix2`
: input matrix stored in 1D array of floats

`rows2`
: number of rows in the second matrix.

`cols2`
: number of columns in the second matrix.

Output Arguments

`outmat`
: the output matrix. Its dimension will be rows1 * cols2.

Returns

TRUE (1) on success, otherwise it will return FALSE (0) and kerrno will be set to KLIMITATION if the input matrix dimensions are not 3x3 or 4x4.

Description

kfmatrix_multiply() multiplies two arbitrary matrices. The input matrices are expected to be organized in a 1 dimensional array as consecutive rows. The result is stored in the same format and has the dimensions rows1 * cols2. The result is returned in outmat.

Restrictions

This function really only works on when the two matrices are either 3x3 or 4x4.

D.2.6. kfmatrix_princ_axis() — *obtain the principle axis of a covariance matrix.*

Synopsis

```
int kfmatrix_princ_axis(
   float *a,
   int    n,
   float *y)
```

Input Arguments

`a`

input matrix stored in 1D array of floats

`n`

size of matrix (assumed square, n x n).

Output Arguments

`y`

the output vector (axis)

Returns

TRUE (1) on success, otherwise it returns FALSE (0) and an error is output indicating insufficient memory.

Description

kfmatrix_princ_axis() obtains the principal axis (the eigenvector associated with the largest eigenvalue) for the covariance matrix "a" of size n by n. Put the principal eigenvector in the place pointed to by "y".

The power iteration method to obtain the dominant eigenvalue and its associated eigenvector as described in Gollub and VanLoan, MATRIX COMPUTATIONS, pp 209.

Restrictions

Since a is a covariance matrix it is symmetric, real, and very likely to be positive definate. It is positive semidefinate for sure, and may also be diagonal dominant.

Other possible difficulties: The major hitch in this technique is that the convergence is proportional to lambda(2)/lambda(1) where lambda(1) is the dominant eigenvector. If these eigenvalues are closely spaced then we won't get a decent eigenvector (it will have an incorrect direction).

Fortunately, when using the principal axis to split a cluster and there are two very strongly dominant axes with the same ellipticity, then we can split on any combination of those axis and reduce the cluster variances greatly.

The 10 iterations used in the code have been found to be satisfactory for all of the data so far encountered unless it is an ugly special case.

D.2.7. kfmatrix_vector_prod() — *compute the matrix-vector product.*

Synopsis

```
int kfmatrix_vector_prod(
   float *a,
   float *x,
   int    rows,
   int    cols,
   float *y)
```

Input Arguments

`a`
: input matrix stored in 1D array of floats

`x`
: input vector

`rows`
: number of rows in matrix a as well as the number of elements in output vector y.

`cols`
: number of columns in matrix a as well as the number of elements in input vector x.

Output Arguments

`y`
: The output vector containing the matrix-vector product.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kfmatrix_vector_prod() computes a matrix-vector product.

D.2.8. kdmatrix_clear() — *zeros a matrix*

Synopsis

```
int kdmatrix_clear(
   int    rows,
   int    cols,
   double *matrix)
```

Input Arguments

`rows`
number of rows in the matrix.

`cols`
number oc columns in the matrix.

Output Arguments

`matrix`
the cleared matrix.

Returns

TRUE (1) on success, otherwise it returns FALSE (0) and kerrno is set to KINVALID_PARAMETER if the dimensions of the input matrix are not positive and non-zero.

Description

kdmatrix_clear() clears a matrix by setting all of its values to 0.0.

D.2.9. kdmatrix_identity() — *set matrix to identity*

Synopsis

```
int kdmatrix_identity(
   int    rows,
   int    cols,
   double *matrix)
```

Input Arguments

`rows`
the number of rows in the matrix.

`cols`
the number of columns in the matrix.

Output Arguments

`matrix`

output matrix stored in 1D array of doubles

Returns

TRUE (1) on success, otherwise this function returns FALSE (0) and kerrno is set to KINTERNAL if the internal call to kdmatrix_clear failed.

Description

kdmatrix_identity() sets an input matrix to the identity matrix.

D.2.10. kdmatrix_inner_prod() — *compute the inner product of two vectors.*

Synopsis

```
double kdmatrix_inner_prod(
   double *x,
   double *y,
   int    n)
```

Input Arguments

`x`

first vector.

`y`

second vector.

`n`

number of components in each vector.

Returns

The double precision inner product of the two vectors.

Description

kdmatrix_inner_prod() computes the inner product of two vectors.

D.2.11. kdmatrix_inverse() — *inverts a matrix.*

Synopsis

```
int kdmatrix_inverse(
   double *matrix,
   int    order,
   double *outmatrix)
```

Input Arguments

`matrix`
: input matrix stored in 1D array of doubles

`order`
: order of the matrix.

Output Arguments

`outmatrix`
: the inverted matrix stored in 1D array of doubles.

Returns

TRUE (1) on success, otherwise it returns FALSE (0) and kerrno is set to KNUMERIC if the input matrix is singular and cannot be inverted.

Description

kdmatrix_inverse() - returns the inverse of the input matrix.

D.2.12. kdmatrix_multiply() — *multiply two matrices*

Synopsis

```
int kdmatrix_multiply(
   double *matrix1,
   int    rows1,
   int    cols1,
   double *matrix2,
   int    rows2,
   int    cols2,
   double *outmat)
```

Input Arguments

`matrix1`

input matrix stored in 1D array of doubles

`rows1`
number of rows in the first matrix.

`cols1`
number of columns in the first matrix.

`matrix2`
input matrix stored in 1D array of doubles

`rows2`
number of rows in the second matrix.

`cols2`
number of columns in the second matrix.

Output Arguments

`outmat`
the output matrix. Its dimension will be rows1 * cols2.

Returns

TRUE (1) on success, otherwise it returns FALSE (0) and kerrno is set to KLIMITATION if the matrices are not 3x3 or 4x4.

Description

kdmatrix_multiply() multiplies two arbitrary matrices. The input matrices are expected to be organized in a 1 dimensional array as consecutive rows. The result is stored in the same format and has the dimensions rows1 * cols2. The result is returned in outmat.

Restrictions

This function really only works on when the two matrices are either 3x3 or 4x4.

D.2.13. kdmatrix_princ_axis() — *obtain the principle axis of a covariance matrix.*

Synopsis

```
int kdmatrix_princ_axis(
   double *a,
   int    n,
   double *y)
```

Input Arguments

`a`
input matrix stored in 1D array of doubles

`n`
size of matrix (assumed square. n x n).

Output Arguments

`y`

the output vector (axis)

Returns

TRUE (1) on success, otherwise it returns FALSE (0) if this function is unable to allocate sufficient memory for the operation.

Description

kdmatrix_princ_axis() obtains the principal axis (the eigenvector associated with the largest eigenvalue) for the covariance matrix "a" of size n by n. Put the principal eigenvector in the place pointed to by "y".

The power iteration method to obtain the dominant eigenvalue and its associated eigenvector as described in Gollub and VanLoan, MATRIX COMPUTATIONS, pp 209.

Restrictions

Since a is a covariance matrix it is symmetric, real, and very likely to be positive definate. It is positive semidefinate for sure, and may also be diagonal dominant.

Other possible difficulties: The major hitch in this technique is that the convergence is proportional to lambda(2)/lambda(1) where lambda(1) is the dominant eigenvector. If these eigenvalues are closely spaced then we won’t get a decent eigenvector (it will have an incorrect direction).

Fortunately, when using the principal axis to split a cluster and there are two very strongly dominant axes with the same ellipticity, then we can split on any combination of those axis and reduce the cluster variances greatly.

The 10 iterations used in the code have been found to be satisfactory for all of the data so far encountered unless it is an ugly special case.

D.2.14. kdmatrix_vector_prod() — *compute the matrix-vector product.*

Synopsis

```
int kdmatrix_vector_prod(
   double *a,
   double *x,
   int    rows,
   int    cols,
   double *y)
```

Input Arguments

`a`
: an input matrix stored in 1D array of doubles

`x`
: input vector of elements

`rows`
: number of rows in matrix a as well as the number of elements in output vector y.

`cols`
: number of columns in matrix a as well as the number of elements in input vector x.

Output Arguments

`y`
: the output vector of containing the matrix-vector product

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kdmatrix_vector_prod() computes a matrix-vector product.

D.2.15. klin_sgefa() — *factors a float matrix by gaussian elimination.*

Synopsis

```
int klin_sgefa(

    float *matrix,
    int    rows,
    int    cols,
    int   *pivot)
```

Input Arguments

`matrix`

input matrix stored in a 1D array floats in row-major order such that

m[i][j] = a[i * cols+j]

`rows`

the number of rows in matrix.

`cols`

the number of cols in matrix.

Output Arguments

`pivot`

the pivot vector

Returns

0 implies a normal value; a non-zero value of k means that U(k,k) == 0. This is not an error condition for this routine, but it does indicate that klin_sgesl or klin_sgedi will divide by zero if called. Use the argument rcond in klin_segco for a reliable indication of singularity.

Description

This routine is usually called by klin_sgeco, but it can be called directly with a saving in time if the rcond is not needed.

D.2.16. klin_sgedi() — *computes the determinate and inverse of a matrix*

Synopsis

```
int klin_sgedi(
   float *matrix,
   int    rows,
   int    cols,
   int   *pivot,
   int    job,
   float *work,
   float *det)
```

Input Arguments

`matrix`
input matrix stored in a 1D array of doubles in row-major order such that

m[i][j] = a[i * cols+j]

`rows`
the number of rows in matrix.

`cols`
the number of columns in matrix.

`pivot`
the pivot vector from klin_sgefa

`job`
defines what to compute: the inverse, the determinate, or both of the matrix. KLIN_INVERSE to compute inverse KLIN_DETERMINATE to compute determinate KLIN_INVERSE | KLIN_DETERMINATE to compute both

Output Arguments

`matrix`
inverse of the original matrix if requested, othewise unchanged.

`det`
determinant of original matrix if requested, otherwise not referenced. determinant = det(1)*10.0**det(2) with 1.0 < kabs(det(1)) < 10.0 or det(1) == 0.0

Returns

TRUE (1) all the time

Description

This routine will compute the determinant and inverse of an NxN matrix using the factors computed by

klin_sgeco or klin_sgefa.

The determinant functionality is not well tested.

Restrictions

A division by zero will occur if the input factor contains a zero on the diagonal and the inverse is requested. It will not occur if the subroutines are called correctly and if klin_sgeco has set rcond > 0.0 or klin_sgefa has set info < 0.

D.2.17. kblas_sscal() — *scale a float vector*

Synopsis

```
void kblas_sscal(

     float *sx,
     float  sa,
     int    n,
     int    incx)
```

Input Arguments

`sx`
: linear array to scale

`sa`
: scalar multiplier for sx array

`n`
: number of bins in array to scale

`incx`
: increment to use on sx array

Output Arguments

`sx`
: computation is done in place

Description

This routine multiplies a vector by a constant. It uses unrolled loops to improve performance for increments == 1.

D.2.18. kblas_saxpy() — *add two float vectors while scaling one*

Synopsis

```
void kblas_sscal(

    float *sx,
    float *sy,
    float  sa,
    int    n,
    int    incx,
    int    incy)
```

Description

This routine multiplies a vector by a constant and then adds it to another vector. The calculation will be : SY = SY + sa*SX It uses unrolled loops to improve performance for increments == 1. The computation is done in place and returned in sy.

D.2.19. kblas_sswap() — *swap two float vectors*

Synopsis

```
void kblas_sswap(

    float *sx,
    float *sy,
    int    n,
    int    incx,
    int    incy)
```

Input Arguments

`sx`
: linear array to swap

`sy`
: linear array to swap

`n`
: number of bins in array to swap

`incx`
: increment to use on sx

`incy`
: increment to use on sy

Description

This routine interchanges two vectors. It uses unrolled loops for increments == 1.

D.2.20. klin_dgefa() — *factors a double matrix by gaussian elimination.*

Synopsis

```
int klin_dgefa(

    double *matrix,
    int    rows,
    int    cols,
    int   *pivot)
```

Input Arguments

`matrix`

input matrix stored in a 1D array of doubles in row-major order such that

m[i][j] = a[i * cols+j]

`rows`

the number of rows in matrix.

`cols`

the number of cols in matrix.

Output Arguments

`pivot`

the pivot vector

Returns

0 implies a normal value; a non-zero value of k means that U(k,k) == 0 This is not an error condition for this routine, but it does indicate that klin_sgesl or klin_dgedi will divide by zero if called. Use the argument rcond in klin_segco for a reliable indication of singularity.

Description

This routine is usually called by klin_sgeco, but it can be called directly with a saving in time if the rcond is not needed.

D.2.21. klin_dgedi() — *computes the determinate and inverse of a matrix*

Synopsis

```
int klin_dgedi(
   double *matrix,
   int    rows,
   int    cols,
   int   *pivot,
   int    job,
   double *work,
   double *det)
```

Input Arguments

`matrix`
input matrix stored in a 1D array of doubles in row-major order such that

m[i][j] = a[i * cols+j]

`rows`
the number of rows in matrix.

`cols`
the number of columns in matrix.

`pivot`
the pivot vector from klin_dgefa

`job`
defines what to compute: the inverse, the determinate, or both of the matrix. KLIN_INVERSE to compute inverse KLIN_DETERMINATE to compute determinate KLIN_INVERSE | KLIN_DETERMINATE to compute both

Output Arguments

`matrix`
inverse of the original matrix if requested, othewise unchanged.

`det`
determinant of original matrix if requested, otherwise not referenced. determinat = det(1)*10.0**det(2) with 1.0 < kabs(det(1)) < 10.0 or det(1) == 0.0

Returns

TRUE (1) all the time

Description

This routine will compute the determinant and inverse of an NxN matrix using the factors computed by

klin_sgeco or klin_dgefa.

The determinant functionality is not well tested.

Restrictions

A division by zero will occur if the input factor contains a zero on the diagonal and the inverse is requested. It will not occur if the subroutines are called correctly and if klin_sgeco has set rcond > 0.0 or dgefa has set info < 0.

D.2.22. kblas_dscal() — *scale a double vector*

Synopsis

```
void kblas_dscal(

     double *sx,
     double  sa,
     int    n,
     int    incx)
```

Input Arguments

`sx`
: linear array to scale

`sa`
: scalar multiplier for sx array

`n`
: number of bins in array to scale

`incx`
: increment to use on sx array

Output Arguments

`sx`
: computation is done in place

Description

This routine multiplies a vector by a constant. It uses unrolled loops for increments == 1.

D.2.23. kblas_daxpy() — *add two double vectors while scaling one*

Synopsis

```
void kblas_dscal(

    double *sx,
    double *sy,
    double  sa,
    int    n,
    int    incx,
    int    incy)
```

Description

This routine multiplies a vector by a constant and then adds it to another vector. The calculation will be : SY = SY + sa*SX It uses unrolled loops for increments == 1.

D.2.24. kblas_dswap() — *swap two double vectors*

Synopsis

```
void kblas_dswap(

    double *sx,
    double *sy,
    int    n,
    int    incx,
    int    incy)
```

Input Arguments

`sx`
: linear array to swap :

`sy`
: linear array to swap

`n`
: number of bins in array to swap

`incx`
: increment to use on sx incx - increment to use on sy

Description

This routine interchanges two vectors. It uses unrolled loops for increments == 1.

E. Sequence Generation

The following section details the *sequence generation functions*.

E.1. Introduction to Sequence Generation Functions

The *klibm* library includes a series of functions that are employed to generate random sequences of data with various distributions as well as certain periodic sequences. The currently supported random distributions include uniform, exponential, Gaussian, Poisson, and Rayleigh. The periodic sequence generators are employed to generate real and complex sinusoidal sequences and piece-wise linear sequences.

The following functions are available:

- *kgen_expon()* - generate a vector of exponential random numbers.
- *kgen_gauss()* - generate a vector of gaussian random numbers.
- *kgen_poisson()* - generate a vector of Poisson random numbers.
- *kgen_rayleigh()* - generate a vector of Rayleigh random numbers.
- *kgen_unif()* - generate a vector of uniform random numbers.
- *kgen_linear()* - generate a piecewise linear data set.
- *kgen_sine()* - generates a sinusoid data set
- *kgen_sinec()* - generate a sinc data set.

E.2. Definitions of Sequence Generation Functions

E.2.1. kgen_expon() — *generate a vector of exponential random numbers.*

Synopsis

```
int kgen_expon(
   int    num,
   double variance,
   double *vect)
```

Input Arguments

`num`
: number of elements in data set.

`variance`
: variance of the data set

Output Arguments

`vect`
: a vector containing the generated set of exponential random numbers.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kgen_expon() generates a 1d exponential random noise data set

E.2.2. kgen_gauss() — *generate a vector of gaussian random numbers.*

Synopsis

```
int kgen_gauss(
   int    num,
   double mean,
   double variance,
   double *vect)
```

Input Arguments

`num`

number of elements in vector

`mean`

mean of the data set

`variance`

variance of the data set

Output Arguments

`vect`

a vector containing the generated set of gaussian random numbers.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kgen_gauss() generates a one dimensional gaussian random noise data set.

The formula used to derive the gaussian random random numbers is the Box-Mueller method and was taken from Numerical Recipes : The Art of Scientific Computing (Press, Flannery, Teukolsky, and Vetterling) 1986.

E.2.3. kgen_poisson() — *generate a vector of Poisson random numbers.*

Synopsis

```
int kgen_poisson(
   int    num,
   double variance,
   double atime,
   double *vect)
```

Input Arguments

`num`
number of elements in data set.

`variance`
variance of the data set

`atime`
amount of time

Output Arguments

`vect`
a vector containing the generated set of Poisson random numbers.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kgen_poisson() generates a one dimensional Poisson random noise data set.

E.2.4. kgen_rayleigh() — *generate a vector of Rayleigh random numbers.*

Synopsis

```
int kgen_rayleigh(
   int    num,
   double variance,
   double *vect)
```

Input Arguments

`num`
number of elements in input vector

`variance`
variance of the data set

Output Arguments

`vect`

a vector containing the generated set of Rayleigh random numbers.

Returns

TRUE (1) on success, otherwise it returns FALSE (0) if it is unable to allocate sufficient memory or the internal call to kgen_gauss fails.

Description

kgen_rayleigh() generates a one dimensional Rayleigh random noise data set.

E.2.5. kgen_unif() — *generate a vector of uniform random numbers.*

Synopsis

```
int kgen_unif(
   int    num,
   double minimum,
   double maximum,
   double *vect)
```

Input Arguments

`num`

number of points

`minimum`

minimum of the data set

`maximum`

maximum of the data set

Output Arguments

`vect`

a vector containing the generated set of uniform random numbers.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kgen_unif() generates a one dimensional uniform random noise data set.

E.2.6. kgen_linear() — *generate a piecewise linear data set.*

Synopsis

```
int kgen_linear(
   int    num,
   double sample,
   double minimum,
   double maximum,
   double period,
   double rise,
   double fall,
   double width,
   double *vect)
```

Input Arguments

`num`
: number of elements in vector

`sample`
: sampling frequency

`minimum`
: minimum value of data set.

`maximum`
: maximum value of the data set.

`period`
: period of function

`rise`
: rise time of function

`fall`
: fall time of function

`width`
: width (when high) of pulse

Output Arguments

`vect`
: a vector containing the generated set of piecewise linear numbers.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Generates a one dimensional "piecewise linear" data set. A piecewise linear data set is a data set composed of connected end-to-end lines. By properly setting arguments to this routine, you can generate impulse data (spikes), triangular waves, sawtooth waves, reverse sawtooth waves, and square waves.

E.2.7. kgen_sine() — *generates a sinusoid data set*

Synopsis

```
int kgen_sine(
   int    num,
   double sample,
   double amp,
   double freq,
   double phase,
   double *vect)
```

Input Arguments

`num`
: number of elements in vector

`sample`
: sampling frequency

`amp`
: amplitude of data set

`freq`
: frequency of data set

`phase`
: phase of data set

Output Arguments

`vect`
: a vector containing the generated sinusoidal data set.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kgen_sine() generates a one dimensional sinusoidal data set.

E.2.8. kgen_sinec() — *generate a sinc data set.*

Synopsis

```
int kgen_sinec(
   int    num,
   double sample,
   double amp,
   double freq,
   int    center,
   double *vect)
```

Input Arguments

`num`
: number of elements in vector

`sample`
: sampling frequency

`amp`
: amplitude of signal

`freq`
: frequency of signal

`center`
: centering option: 0 - center the sync on the data set (i.e. the zero point of the data is at the midpoint of vect) 1 - left justify the sync (the center of the data is at vect[0], and you only get the right hand side of the sinc)

Output Arguments

`vect`
: a vector containing the generated sinc data set.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

kgen_sinec() generates a one dimensional sinc data set. The sinc function is defined as sin(x)/x.

F. General Math Utilities

The following sections detail the general *math library functions*.

F.1. Introduction to General Math Utilities

The foundation of the *klibm* library is the architecture and operating system independent-implementations of the standard *libm.a* functions. The implementation of these functions employs the *libm.a* versions of the function when they are available. When these are not available, the versions are automatically replaced with local versions that are designed as drop-in replacements.

The functions are:

- *kabs()* - return the absolute value of a single argument.
- *kacos()* - compute the arc cosine of the argument.
- *kacosh()* - compute the arc hyperbolic cosine of the argument.
- *kasin()* - compute the arc sine of the argument.
- *kasinh()* - compute the arc hyperbolic sine of the argument.
- *katan()* - compute the arc tangent of the argument.
- *katan2()* - compute the arc tangent of y/x.
- *katanh()* - compute the arc hyperbolic tangent of the argument.
- *kcbrt()* - compute the cube root of the argument.
- *kceil()* - compute the ceiling of the argument.
- *kclear()* - return an value with all bits clear
- *kcos()* - compute the double precision cosine of the argument.
- *kcosh()* - compute the hyperbolic cosine of the argument.
- *kdata_minmax()* - find the min/max values for a region of data
- *kdegrees_radians()* - return the radians given an input of degrees
- *kerf()* - compute the error function of the argument.
- *kerfc()* - compute the complement error function of the argument.
- *kexp()* - compute the exponential of the argument.
- *kexp10()* - compute the base 10 exponential function of the argument.
- *kexp2()* - compute the base 2 exponential function of the argument.
- *kexpm1()* - compute the exponential function minus 1 of the argument.
- *kfabs()* - compute the absolute value of the argument.
- *kfloor()* - compute the floor of the argument.
- *kfmod()* - compute the floating point modulo of the arguments.
- *kfraction()* - returns the fractional part of x
- *kfrexp()* - compute significand and exponent of argument
- *kgamma()* - compute the gamma function of the argument.
- *khypot()* - compute the euclidean distance from the origin of the arguments.
- *kintercept()* - interpolate the intercept
- *kj0()* - compute the Bessel function j0 of the argument.
- *kj1()* - compute the Bessel function j1 of the argument.
- *kjn()* - compute the general Bessel function jn of the argument.
- *kldexp()* - computes x * 2**n
- *klog()* - compute the natural logarithm of the argument.
- *klog10()* - compute the base 10 logarithm of the argument.
- *klog1p()* - compute the logarithm of x+1 of the argument.

• *klog2()* - compute the base 2 logarithm of the argument.
• *klogical_not()* - logical not (invert) function
• *klogn()* - base log n of argument
• *kmax3()* - return the greater of three values.
• *kmax4()* - return the greater of four values.
• *kmin3()* - return the lessor of three values.
• *kmin4()* - return the lessor of four values.
• *kmodf()* - compute the fractional component of the argument.
• *kneg()* - negative function
• *knot()* - bitwise not (invert) function
• *kpow()* - compute x to the y power.
• *kradians_degrees()* - return the degrees given an input of radians
• *krandom()* - generate random number in range [0,2**31-1]
• *krecip()* - reciprocal function.
• *kset()* - return a value with all bit set
• *kset_seed()* - Set the seed to a randome number generator.
• *ksin()* - compute the double precision sine of the argument.
• *ksinh()* - compute the hyperbolic sine of the argument.
• *ksqrt()* - compute the square root of the argument.
• *ksrandom()* - seed the krandom random number generator.
• *ktan()* - compute the double precision tangent of the argument.
• *ktanh()* - compute the hyperbolic tangent of the argument.
• *ktrunc()* - truncate a number
• *ky0()* - compute the Bessel function y0 of the argument.
• *ky1()* - compute the Bessel function y1 of the argument.
• *kyn()* - compute the general Bessel function yn of the argument.
• *kfact()* - compute factorial of input.
• *kimpulse()* - evaluate impulse function.
• *kpowtwo()* - determine if number is an integer power of two.
• *ksign()* - evaluate sign function.
• *ksinc()* - sinc function which is "sin(x)/x"
• *ksqr()* - return the square of a single value.
• *kstep()* - evaluate step function.
• *kurng()* - generate a uniform random number in the range [0:1].

F.2. Definitions of General Math Utilities

F.2.1. kabs() — *return the absolute value of a single argument.*

Synopsis

```
kabs(x)
```

Input Arguments

`x`

a variable of any base data type.

Returns

the absolute value of the input argument.

Description

The kabs function obtains the absolute value of the input argument. This is a macro, so any data type is supported.

F.2.2. kacos() — *compute the arc cosine of the argument.*

Synopsis

```
double kacos(double x)
```

Input Arguments

`x`

input argument to take arc cosine of.

Returns

the double precision arc cosine of the double precision input argument.

Description

The kacos function computes the arc cosine of the double precision input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kacos is simply a macro to acos because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.3. kacosh() — *compute the arc hyperbolic cosine of the argument.*

Synopsis

```
double kacosh(double x)
```

Input Arguments

`x`

argument to kacosh

Returns

the double precision are hyperbolic cosine of the input argument or KMAXFLOAT if the input argument is less than one.

Description

The kacosh function computes the arc hyperbolic cosine of the double precision argument.

kacosh is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to acosh, otherwise kasinh is a macro to the VisiQuest portable version of acosh.

F.2.4. kasin() — *compute the arc sine of the argument.*

Synopsis

```
double kasin(double x)
```

Input Arguments

`x`

input argument to take arc sine of.

Returns

the double precision arc sine of the input argument.

Description

The kasin function computes arc sine of the double precision input argument, whose value lies between -1 and 1.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kasin is simply a macro to asin because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.5. kasinh() — *compute the arc hyperbolic sine of the argument.*

Synopsis

```
double kasinh(double x)
```

Input Arguments

`x`

argument to kasinh

Returns

the evaluation of asinh(x)

Description

The kasinh function computes the arc hyperbolic sine function

kasinh is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to asinh, otherwise kasinh is a macro to the VisiQuest portable version of asinh.

F.2.6. katan() — *compute the arc tangent of the argument.*

Synopsis

```
double katan(double x)
```

Input Arguments

`x`

input argument to take arc tangent of.

Returns

the double precision arc tangent of the input argument.

Description

The katan function computes arc tangent of the double precison input argument. The result is an angle with the value between -pi/2 and pi/2 radians.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently katan is simply a macro to atan because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.7. katan2() — *compute the arc tangent of y/x.*

Synopsis

```
double katan2(
double y,
double x)
```

Input Arguments

`y`

numerator component of the tangent given y/x.

`x`

the non-zero denominator component of the tangent given by y/x.

Returns

the double precision arc tangent of the input arguments.

Description

The katan2 function converts the rectangular coordinates (x,y) into polar coordinates and returns the phase component in the range (-pi,pi)

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently katan2 is simply a macro to atan2 because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

Restrictions

The second argument x must be non-zero.

F.2.8. katanh() — *compute the arc hyperbolic tangent of the argument.*

Synopsis

```
double katanh(double x)
```

Input Arguments

`x`

argument to katanh

Returns

the double precision arc hyperbolic tangent of the input argument or KMAXFLOAT if the input parameter is less than -1 or greater than 1.

Description

The kasinh function computes the arc hyperbolic tangent of the double precision input argument whose value lies between -1 and 1.

katanh is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to atanh, otherwise katanh is a macro to the VisiQuest portable version of atanh.

F.2.9. kcbrt() — *compute the cube root of the argument.*

Synopsis

```
double kcbrt(double x)
```

Input Arguments

`x`

value perform the cube root function on

Returns

the double precision cube root of the input argument.

Description

The kcbrt function computes the cube root of the double precision input argument.

F.2.10. kceil() — *compute the ceiling of the argument.*

Synopsis

```
double kceil(double x)
```

Input Arguments

`x`

number to calculate ceiling of.

Returns

the double precision ceiling of the input argument.

Description

The kceil function computes the least integral value (smallest integer value) greater than or equal to the input argument and returns it as a double.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kceil is simply a macro to ceil because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.11. kclear() — *return an value with all bits clear*

Synopsis

```
unsigned long kclear(
   unsigned long x)
```

Input Arguments

`x`

input argument, value doesn't matter. Supplied only for consistency with the rest of the klibm routines.

Returns

An unsigned long with all bits clear.

Description

Return an unsiged value with all bits clear, independent of the word length. This function is supplied only for symmetry with the kset() function.

F.2.12. kcos() — *compute the double precision cosine of the argument.*

Synopsis

```
double kcos(double x)
```

Input Arguments

`x`

radian input argument to compute the cosine on.

Returns

the double precision cosine of the input argument

Description

The kcos function computes the cosine of the double precision input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kcos is simply a macro to cos because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.13. kcosh() — *compute the hyperbolic cosine of the argument.*

Synopsis

```
double kcosh(double x)
```

Input Arguments

`x`

input argument to compute the double precision hyperbolic cosine on.

Returns

the double precision hyperbolic cosine of the input argument or KMAXFLOAT if the input argument is less than one.

Description

The kcosh function computes the hyperbolic cosine of the double precison input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kcosh is simply a macro to cosh because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.14. kdata_minmax() — *find the min/max values for a region of data*

Synopsis

```
int kdata_minmax(
   kaddr  data,
   size_t num,
   int    datatype,
   double *minval,
   double *maxval)
```

Input Arguments

`data`

point to the data to find the min/max value

`num`

number of points to be searched

`datatype`

data type of the data

Output Arguments

`minval`

returns the minimum value

`maxval`

returns the maximum value

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to find the min/max values associated with a region of memory. A datatype is used to determine how to search the data set. The data types are byte, unsigned byte, short, unsigned short, long, unsigned long, int, unsigned int, float, and double.

F.2.15. kdegrees_radians() — *return the radians given an input of degrees*

Synopsis

```
kdegrees_radians(degrees)
```

Input Arguments

`degrees`

a variable of any base data type.

Returns

the argument in terms of radians

Description

This function converts the input degrees and returns the argument in radians. This is a macro, so any data type is supported.

F.2.16. kerf() — *compute the error function of the argument.*

Synopsis

```
double kerf(double x)
```

Input Arguments

`x`

argument to erf

Returns

the error function of the input argument.

Description

The kerf function computes the error function, defined by:

kerf(x) = 2.0/sqrt(pi)*{integral from 0 to x of exp(-t*t) dt}

kerf is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to erf, otherwise kerf is a macro to the VisiQuest portable version of erf.

F.2.17. kerfc() — *compute the complement error function of the argument.*

Synopsis

```
double kerfc(double x)
```

Input Arguments

`x`

argument to kerfc

Returns

One minus the error function of the input argument.

Description

The kerfc function computes the complement error function, defined by:

kerfc(x) = 1.0 - kerf(x)

kerfc is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to erfc, otherwise kerfc is a macro to the VisiQuest portable version of erfc.

F.2.18. kexp() — *compute the exponential of the argument.*

Synopsis

```
double kexp(double x)
```

Input Arguments

`x`

variable whose exponential is to be computed.

Returns

the double precision result of the exponential of the input argument.

Description

The kexp function computes the value of the exponential of the double precision input argument (e**x).

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kexp is simply a macro to exp because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.19. kexp10() — *compute the base 10 exponential function of the argument.*

Synopsis

```
double kexp10(double x)
```

Input Arguments

`x`

variable whose base 10 exponential is to be computed.

Returns

the double precision base 10 exponential of the input argument.

Description

The kexp10 function computes the base 10 exponential function.

kexp10 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to exp10, otherwise kexp10 is a macro to the VisiQuest portable version of exp10.

F.2.20. kexp2() — *compute the base 2 exponential function of the argument.*

Synopsis

```
double kexp2(double x)
```

Input Arguments

`x`

variable whose base 2 exponential is to be computed.

Returns

the double precision base 2 exponential of the input argument.

Description

The kexp2 function computes the base 2 exponential of the double precision input argument.

kexp2 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to exp2, otherwise kexp2 is a macro to the VisiQuest portable version of exp2.

F.2.21. kexpm1() — *compute the exponential function minus 1 of the argument.*

Synopsis

```
double kexpm1(double x)
```

Input Arguments

`x`

variable whose exponential minus 1 is to be computed.

Returns

the double precision exponential minus 1 of the input argument.

Description

The kexpm1 function computes the exponential function of the double precision input argument, then subtracts 1.

kexpm1 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to expm1, otherwise kexpm1 is a macro to the VisiQuest portable version of expm1.

F.2.22. kfabs() — *compute the absolute value of the argument.*

Synopsis

```
double kfabs(double x)
```

Input Arguments

`x`

number to take the absolute value of.

Returns

the double precision absolute value of the input argument.

Description

The kfabs function computes the absolute value of the double precision input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kfabs is simply a macro to fabs because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.23. kfloor() — *compute the floor of the argument.*

Synopsis

```
double kfloor(double x)
```

Input Arguments

`x`

number to calculate floor of.

Returns

the double precision floor of the input argument.

Description

The kfloor function computes the largest integral value (largest integer) less than or equal to the input argument and returns it as a double.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kfloor is simply a macro to floor because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.24. kfmod() — *compute the floating point modulo of the arguments.*

Synopsis

```
double kfmod(
double x,
double y)
```

Input Arguments

`x`
numerator of division operation.

`y`
denominator of division operation.

Returns

the double precision modulo remainder of x/y.

Description

The kfmod function computes the floating point remainder of x/y.

This is provided in all POSIX and X/OPEN compliant C math libraries. On most machines, kfmod is a macro to fmod. On certain machines that are not fully POSIX compliant, this is a macro to an internal implementation of fmod.

F.2.25. kfraction() — *returns the fractional part of x*

Synopsis

```
double kfraction(
   double x)
```

Input Arguments

`x`
argument to the kfraction function

Returns

The double precision fractional part of the input argument.

Description

The kfraction function evaluates the fractional value of the double precision input argument. The result is kmodf(x, *iptr) for a given x.

F.2.26. kfrexp() — *compute significand and exponent of argument*

Synopsis

```
double kfrexp(
double v,
int *e)
```

Input Arguments

`v`

value to compute on.

Output Arguments

`e`

exponent of the input argument.

Returns

the significand of the input argument, or zero if the input argument is zero.

Description

The kfrexp function computes the significand (or mantissa) of v as a double quantity between 0.5 and 1.0 and stores the exponent in the output argument e.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kfrexp is simply a macro to frexp because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.27. kgamma() — *compute the gamma function of the argument.*

Synopsis

```
double kgamma(double x)
```

Input Arguments

`x`

value to compute the log gamma of.

Returns

the double precision log gamma of the input argument.

Description

The kgamma function computes the log gamma of the double precision input argument.

Restrictions

If gamma does not exist on you system in libm, then a VisiQuest version of the program will be transparently installed in its place. This version of the function is not very good. The VisiQuest Group has plans to improve this function soon after the main release of VisiQuest 2.0.

F.2.28. khypot() — *compute the euclidean distance from the origin of the arguments.*

Synopsis

```
double khypot(
double x,
double y)
```

Input Arguments

`x`

value on x axis of vector to determine length of.

`y`

value on y axis of vector to determine length of.

Returns

returns the double precision euclidean distance from (0,0) to (x,y).

Description

The khypot function computes the Euclidean distance from (0,0) to (x,y). This function computes the square root of the sum of squares of the double precision input arguments x and y, giving the return value ‘ sqrt(x*x + y*y)

khypot is not a POSIX function, but is implemented in BSD math libraries. khypot is simply a macro to hypot because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.29. kintercept() — *interpolate the intercept*

Synopsis

```
kintercept(x0, x1, y0, y1, y2)
```

Input Arguments

`x0`

a variable of any base data type.

`x1`

a variable of any base data type.

`y0`

a variable of any base data type.

`y1`

a variable of any base data type.

`y2`

a variable of any base data type.

Returns

the intercept point

Description

The following macro is used to interpolate the intercept from a vector described by the two points (x0,y0) & (x1,y1) and a value along the vector, y2, the macro will return the corresponding x2 value. This is a macro, so any data type is supported.

F.2.30. kj0() — *compute the Bessel function j0 of the argument.*

Synopsis

```
double kj0(double x)
```

Input Arguments

`x`

argument to j0

Returns

the double precision value of the j0 Bessel function at x.

Description

The kj0 function computes an approximation of the j0 Bessel function. This function evaluates the Bessel function of the first kind of integer order at an input argument x.

kj0 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to j0, otherwise kj0 is a macro to the VisiQuest portable version of j0.

F.2.31. kj1() — *compute the Bessel function j1 of the argument.*

Synopsis

```
double kj1(double x)
```

Input Arguments

`x`

argument to j1

Returns

the double precision value of the j1 Bessel function at x.

Description

The kj1 function computes an approximation of the j1 Bessel function. This function evaluates the Bessel function of the first kind integer order of the input argument x.

kj1 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to j1, otherwise kj1 is a macro to the VisiQuest portable version of j1.

F.2.32. kjn() — *compute the general Bessel function jn of the argument.*

Synopsis

```
double kjn(int n, double x)
```

Input Arguments

`n`

integer order of the Bessel function.

`x`

argument to jn

Returns

the double precision value of the n-order Bessel function at x.

Description

The kjn function computes an approximation of the jn Bessel function. This function evaluates the Bessel function of the first kind of integer n order of the input argument x.

kjn is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to jn, otherwise kjn is a macro to the

VisiQuest portable version of jn.

F.2.33. kldexp() — *computes x * 2**n*

Synopsis

```
double kldexp(
double x,
int n)
```

Input Arguments

`x`

double argument to x * 2**n

`n`

integer argument to x * 2**n

Returns

the double precision value x * 2**n

Description

The kldexp function computes x * 2**n by performing exponent manipulation rather than multiplication.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kldexp is simply a macro to ldexp because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.34. klog() — *compute the natural logarithm of the argument.*

Synopsis

```
double klog(double x)
```

Input Arguments

`x`

argument to the natural log function.

Returns

the double precision logarithm of the input argument

Description

The klog function computes natural logarithm of the double precision input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently klog is simply a macro to log because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.35. klog10() — *compute the base 10 logarithm of the argument.*

Synopsis

```
double klog10(double x)
```

Input Arguments

`x`

argument to the base 10 log function.

Returns

the double precision base 10 logarithm of the input argument.

Description

The klog10 function computes the base 10 logarithm of the double precision input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently klog10 is simply a macro to log10 because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.36. klog1p() — *compute the logarithm of x+1 of the argument.*

Synopsis

```
double klog1p(double x)
```

Input Arguments

`x`

argument to klog1p

Returns

the double precision logarithm plus one of the input argument.

Description

The klog1p function computes the natural logarithm of the double precision argument plus 1.

klog1p is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to log1p, otherwise klog1p is a macro to the VisiQuest portable version of log1p.

F.2.37. klog2() — *compute the base 2 logarithm of the argument.*

Synopsis

```
double klog2(double x)
```

Input Arguments

`x`

argument to klog2

Returns

the double precision base 2 logarithm of the double precision input argument.

Description

The klog2 function computes the base 2 logarithm of the double precision input argument.

klog2 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to log2, otherwise klog2 is a macro to the VisiQuest portable version of log2.

F.2.38. klogical_not() — *logical not (invert) function*

Synopsis

```
long klogical_not(
   long x)
```

Input Arguments

`x`

argument to the knot function

Returns

The result of the knot function

Description

Changes the logic of the argument. TRUE becomes FALSE, FALSE becomes TRUE.

F.2.39. klogn() — *base log n of argument*

Synopsis

```
double klogn(
   double x,
   double n)
```

Input Arguments

`x`

value to take base-n log of.

`n`

base of logarithm.

Returns

the double precision base n logarithm of the double precision input argument x.

Description

The klogn function computes the base n log of the first double precision input argument, where n is the second double precision input argument.

F.2.40. kmax3() — *return the greater of three values.*

Synopsis

```
kmax3(x0, x1, x2)
```

Input Arguments

`x0`

a variable of any base data type.

`x1`

a variable of any base data type.

`x2`

a variable of any base data type.

Returns

the smaller value of the two input arguments.

Description

The kmax3 function obtains the greater of the three input arguments. This is a macro, so any data type is supported.

F.2.41. kmax4() — *return the greater of four values.*

Synopsis

```
kmax4(x0, x1, x2, x3)
```

Returns

the smaller value of the two input arguments.

Description

The kmax4 function obtains the greater of the four input arguments. This is a macro, so any data type is supported.

F.2.42. kmin3() — *return the lessor of three values.*

Synopsis

```
kmin3(x0, x1, x2)
```

Input Arguments

`x0`
: a variable of any base data type.

`x1`
: a variable of any base data type.

`x2`
: a variable of any base data type.

Returns

the smaller value of the three input arguments.

Description

The kmin3 function obtains the smaller of three input arguments. This is a macro, so any data type is supported.

F.2.43. kmin4() — *return the lessor of four values.*

Synopsis

```
kmin4(x0, x1, x2, x3)
```

Input Arguments

`x0`
: a variable of any base data type.

`x1`
: a variable of any base data type.

`x2`
: a variable of any base data type.

`x3`
: a variable of any base data type.

Returns

the smaller value of the four input arguments.

Description

The kmin4 function obtains the smaller of the four input arguments. This is a macro, so any data type is supported.

F.2.44. kmodf() — *compute the fractional component of the argument.*

Synopsis

```
double kmodf(
double x,
double *i)
```

Input Arguments

`x`
: input argument to separate into integral and fractional components.

Output Arguments

`i`
: the integral part of the input argument

Returns

the fractional component of the input argument.

Description

The kmodf function computes the fractional and integral part of the double precision input argument x. After decomposing the input argument it returns the fractional part and places the integral part in the argument i.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kmodf is simply a macro to modf because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.45. kneg() — *negative function*

Synopsis

```
double kneg(
   double x)
```

Input Arguments

`x`

input argument to find the negative value of.

Returns

The double precision negative value of the input argument.

Description

The kneg function evaluates the negative value of the double precisoin input argument. The result is -x for a given x.

F.2.46. knot() — *bitwise not (invert) function*

Synopsis

```
unsigned long knot(
   unsigned long x)
```

Input Arguments

`x`

input argument.

Returns

The unsigned long inverted bit value of the input argument.

Description

The knot function returns a value computed by inverting all bits in the input value.

F.2.47. kpow() — *compute x to the y power.*

Synopsis

```
double kpow(
double x,
double y)
```

Input Arguments

`x`

base of the exponentiation operation.

`y`

exponent to raise the base to.

Returns

the double precision result of the input argument x raised to the power of the input argument y.

Description

The kpow function computes x**y, which is the value of one double precision input argument x raised to the power of another double precision input argument y.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently kpow is simply a macro to pow because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

Restrictions

Neither argument can be 0.

F.2.48. kradians_degrees() — *return the degrees given an input of radians*

Synopsis

```
kradians_degrees(radians)
```

Input Arguments

`radians`

a variable of any base data type.

Returns

the argument in terms of degrees

Description

This function converts the input radians and returns the argument in degrees. This is a macro, so any data type is supported.

F.2.49. krandom() — *generate random number in range [0,2**31-1]*

Synopsis

```
long krandom(void)
```

Returns

the pseudo random number generated.

Description

The krandom function generates a pseudo random number with a value in the range [0,2**31-1]. The starting point of the pseudo random number sequence is set by calling ksrandom.

krandom is not a POSIX function, but is implemented in most BSD libraries. If it is implemented for the current architecture in libc.a, then this is simply a macro to random, otherwise krandom is a macro to the VisiQuest portable version of random.

F.2.50. krecip() — *reciprocal function.*

Synopsis

```
double krecip(
   double x)
```

Input Arguments

`x`

argument to get the reciprocal of.

Returns

The result of the krecip function evaluated as 1/x.

Description

The krecip function evaluates the reciprocal value of the double precision input argument. The result is 1/x for a given x.

F.2.51. kset() — *return a value with all bit set*

Synopsis

```
unsigned long kset(
   unsigned long x)
```

Input Arguments

`x`

input argument, value doesn't matter. Supplied only for consistency with the rest of the klibm routines.

Returns

An unsigned long with all bits set.

Description

Return an unsigned value with all bits set to 1 independent of the word length

F.2.52. kset_seed() — *Set the seed to a randome number generator.*

Synopsis

```
void kset_seed(int seed)
```

Description

The function sets the seed rather than using the time as a seed. It allows for the generation of a random pattern, while allowing duplication of the pattern.

This was due to a bug or request for the ability to add random gaussian noise to an image, but being able to reproduce the pattern. This modifies the static variable iseed used by kurng().

F.2.53. ksin() — *compute the double precision sine of the argument.*

Synopsis

```
double ksin(double x)
```

Input Arguments

`x`

radian input argument to compute the sine on.

Returns

the double precision sine of the input argument

Description

The ksin function computes the sine of the double precision input argument, which represents an angle in radians.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently ksin is simply a macro to sin because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.54. ksinh() — *compute the hyperbolic sine of the argument.*

Synopsis

```
double ksinh(double x)
```

Input Arguments

`x`

input argument to compute the hyperbolic sine on.

Returns

the double precision hyperbolic sine of the input argument

Description

The ksinh function computes the hyperbolic sine of the input argument

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently ksinh is simply a macro to sinh because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.55. ksqrt() — *compute the square root of the argument.*

Synopsis

```
double ksqrt(double x)
```

Input Arguments

`x`

number to take square root of.

Returns

the double precision square root of the input argument.

Description

The ksqrt function computes the square root of the double precision input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently ksqrt is simply a macro to sqrt because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.56. ksrandom() — *seed the krandom random number generator.*

Synopsis

```
void ksrandom(int seed)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

The ksrandom function sets the seed (starting value) of the random number generation algorithm used by krandom.

ksrandom is not a POSIX function, but is implemented in most BSD libraries. If it is implemented for the current architecture in libc.a, then this is simply a macro to srandom, otherwise ksrandom is a macro to the VisiQuest portable version of srandom.

F.2.57. ktan() — *compute the double precision tangent of the argument.*

Synopsis

```
double ktan(double x)
```

Input Arguments

`x`

radian input argument to compute the tangent on.

Returns

the double precision tangent of the input argument

Description

The ktan function computes the tangent of the input argument.

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently ktan is simply a macro to tan because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.58. ktanh() — *compute the hyperbolic tangent of the argument.*

Synopsis

```
double ktanh(double x)
```

Input Arguments

`x`

input argument to compute the hyperbolic tangent on.

Returns

the double precision hyperbolic tangent of the input argument

Description

The ktanh function computes the hyperbolic tangent of the double precision input argument

This is provided in all POSIX and X/OPEN compliant C math libraries. Currently ktanh is simply a macro to tanh because VisiQuest does not run on any machine whose math library does not include this function. If VisiQuest is ported to a machine without this function then a replacement will be written.

F.2.59. ktrunc() — *truncate a number*

Synopsis

```
double ktrunc(double x)
```

Input Arguments

x

value to truncate

Returns

returns the integral component of the iput argument

Description

The ktrunc function truncates a double precision value by removing its fractional component.

ktrunc is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to trunc, otherwise ktrunc is a macro to the VisiQuest portable version of trunc.

F.2.60. ky0() — *compute the Bessel function y0 of the argument.*

Synopsis

```
double ky0(double x)
```

Input Arguments

x

argument to y0

Returns

the double precision value of the y0 Bessel function at x.

Description

The ky0 function computes an approximation of the y0 Bessel function. This function evaluates the Bessel function of the second kind of integer order at input argument x. The input argument x must be positive.

ky0 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to y0, otherwise ky0 is a macro to the VisiQuest portable version of y0.

F.2.61. ky1() — *compute the Bessel function y1 of the argument.*

Synopsis

```
double ky1(double x)
```

Input Arguments

`x`

argument to y1

Returns

the double precision value of the y1 Bessel function at x.

Description

The ky1 function computes an approximation of the y1 Bessel function. This function evaluates the Bessel function of the second kind of integer order at input argument x. The input argument must be positive.

ky1 is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to y1, otherwise ky1 is a macro to the VisiQuest portable version of y1.

F.2.62. kyn() — *compute the general Bessel function yn of the argument.*

Synopsis

```
double kyn(int n, double x)
```

Input Arguments

`n`

integer order of the Bessel function

`x`

argument to yn

Returns

the double precision value of the n-order Bessel function at x.

Description

The kyn function computes an approximation of the yn Bessel function. This function evaluates the Bessel function of the second kind of integer order n at the input argument x. The input argument x must be positive.

kyn is not a POSIX function, but is implemented in a few BSD math libraries. If it is implemented for the current architecture in libm.a, then this is simply a macro to yn, otherwise kyn is a macro to the VisiQuest portable version of yn.

F.2.63. kfact() — *compute factorial of input.*

Synopsis

```
double kfact(
   double num)
```

Input Arguments

`num`

number to have factorial of computed.

Returns

The factorial of the input argument.

Description

kfact() computes the factorial of the double precision input argument.

F.2.64. kimpulse() — *evaluate impulse function.*

Synopsis

```
double kimpulse (
   double x)
```

Input Arguments

`x`

value to evaluate the impulse function at.

Returns

1.0 if x==0 and 0.0 otherwise

Description

The kimpulse function evaluates the impulse function at the specified double precision argument x.

Restrictions

If the value of x falls between -KEPSILON and +KEPSILON, then the value returned will be 1.0.

F.2.65. kpowtwo() — *determine if number is an integer power of two.*

Synopsis

```
int kpowtwo(
   int length)
```

Input Arguments

`length`

length of sequence to be checked.

Returns

TRUE (1) if the argument is an integer power of two, FALSE (0) otherwise

Description

The function kpowtwo checks to see if the integer input argument (a sequence length) is an integer power of 2.

F.2.66. ksign() — *evaluate sign function.*

Synopsis

```
double ksign (
   double x)
```

Input Arguments

`x`

input argument to evaluate

Returns

-1 if the input argument is negative, 0 if the input argument is 0, and 1 if the input argument is positive

Description

The ksign function evaluates the sign function at the value of the double precision input argument. The result is -1 if the argument is negative, 0 if the argument is 0, and 1 if the argument is positive.

F.2.67. ksinc() — *sinc function which is*

Synopsis

```
double ksinc(
   double x)
```

Input Arguments

`x`

input argument to the sinc function

Returns

The double precision result of the sinc function evaluated from sin(x)/x.

Description

The function ksinc evaluates the sinc function at the value of the input argument. The result is sin(x)/x for a given x.

F.2.68. ksqr() — *return the square of a single value.*

Synopsis

```
ksqr(x)
```

Input Arguments

`x`

a variable of any base data type.

Returns

the squared value of the input argument.

Description

The ksqr function obtains the square of the a single input argument. This is a macro, so any data type is supported.

F.2.69. kstep() — *evaluate step function.*

Synopsis

```
double kstep (
   double x)
```

Input Arguments

`x`

input argument to step

Returns

The double precision value of the step function at x.

Description

The function kstep evaluates the step function at the value of the specified double precision argument.

F.2.70. kurng() — *generate a uniform random number in the range [0:1].*

Synopsis

```
double kurng(void)
```

Returns

A double precision random number between 0 and 1.

Description

The function kurng() generates a double precision uniform random number between 0 and 1.

G. General Data Processing Help Routines

The following section details the general *data processing help routines*.

G.1. Introduction to Help Routines

The help routines create data, modify data, and return data types for processing data.

- *kdata_arith2_dcomplex()* - perform 2-input double complex arithmetic
- *kdata_arith2_double()* - perform 2-input double precision arithmetic
- *kdata_arith2_long()* - perform 2-input long arithmetic
- *kdata_arith2_ulong()* - perform 2-input unsigned long arithmetic
- *kdata_arith2_ubyte()* - perform 2-input unsigned byte arithmetic
- *kdata_fill()* - fill memory with value by data type

G.2. Definitions of Help Routines

G.2.1. kdata_arith2_dcomplex() — *perform 2-input double complex arithmetic*

Synopsis

```
void kdata_arith2_dcomplex(
   int           mode,
   size_t        numpts,
   kdcomplex     *data1,
   kdcomplex     *data2,
   kdcomplex     complex,
   unsigned char *mask1,
   unsigned char *mask2 )
```

Input Arguments

`mode`
: operation to be performed

`numpts`
: number of points in data1, mask1, data2 and mask2

`data1`
: first data array and first operand

`data2`
: second data array and second operand

`complex`
: second operand if data2 array is not valid

`mask1`
: mask associated with first data array

`mask2`
: mask associated with second data array

Output Arguments

`data1`
: data results are stored in data1

`mask1`
: if a mask exists, results are stored in mask1

Description

This function performs the operation defined by "mode" using data in array "data1" as the first operand, and data in array "data2", if it is valid, as the second operand. If "data2" is not valid, the operation will be performed using the value "complex" as the second operand. Valid operations are KADD, KSUB, KSUBFROM, KMULT, KDIV, KDIVINTO, and KPOW.

This function assumes that both input arrays, data1 and data2, are valid, and that if either mask1 or mask2 is passed in, then both masks have been passed in. If just data1 is valid, only mask1 is expected.

Potential division by zero is checked, and KMAXFLOAT is returned instead.

G.2.2. kdata_arith2_double() — *perform 2-input double precision arithmetic*

Synopsis

```
void kdata_arith2_double(
   int           mode,
   size_t        numpts,
   double        *data1,
   double        *data2,
   double        real,
   unsigned char *mask1,
   unsigned char *mask2 )
```

Input Arguments

`mode`
: operation to be performed

`numpts`
: number of points in data1, mask1, data2 and mask2

`data1`
: first data array and first operand

`data2`
: second data array and second operand

`real`
: second operand if data2 array is not valid

`mask1`
: mask associated with first data array

`mask2`
: mask associated with second data array

Output Arguments

`data1`
: data results are stored in data1

`mask1`
: if a mask exists, results are stored in mask1

Description

This function performs the operation defined by "mode" using data in array "data1" as the first operand, and data in array "data2", if it is valid, as the second operand. If "data2" is not valid, the operation will be performed using the value "real" as the second operand. Valid operations are KADD, KSUB, KSUBFROM, KMULT, KDIV, KDIVINTO, KABSDIFF, KMODULO, KMINIMUM, and KMAXIMUM.

This function assumes that both input arrays, data1 and data2, are valid, and that if either mask1 or mask2 is passed in, then both masks have been passed in. If just data1 is valid, only mask1 is expected.

Potential division by zero is checked, and KMAXFLOAT is returned instead.

G.2.3. kdata_arith2_long() — *perform 2-input long arithmetic*

Synopsis

```
void kdata_arith2_long(
   int           mode,
   size_t        numpts,
   long          *data1,
   long          *data2,
   long          real,
   unsigned char *mask1,
   unsigned char *mask2 )
```

Input Arguments

`mode`
: operation to be performed

`numpts`
: number of points in data1, mask1, data2 and mask2

`data1`
: first data array and first operand

`data2`
: second data array and second operand

`real`
: second operand if data2 array is not valid

`mask1`
: mask associated with first data array

`mask2`

mask associated with second data array

Output Arguments

`data1`
data results are stored in data1

`mask1`
if a mask exists, results are stored in mask1

Description

This function performs the operation defined by "mode" using data in array "data1" as the first operand, and data in array "data2", if it is valid, as the second operand. If "data2" is not valid, the operation will be performed using the value "real" as the second operand. Valid operations are KADD, KSUB, KSUBFROM, KMULT, KDIV, KDIVINTO, KABSDIFF, KMODULO, KMINIMUM, and KMAXIMUM.

This function assumes that both input arrays, data1 and data2, are valid, and that if either mask1 or mask2 is passed in, then both masks have been passed in. If just data1 is valid, only mask1 is expected.

Potential division by zero is checked, and KMAXLINT is returned instead.

G.2.4. kdata_arith2_ulong() — *perform 2-input unsigned long arithmetic*

Synopsis

```
void kdata_arith2_ulong(
   int           mode,
   size_t        numpts,
   unsigned long *data1,
   unsigned long *data2,
   unsigned char *mask1,
   unsigned char *mask2 )
```

Input Arguments

`mode`
operation to be performed

`numpts`
number of points in data1, mask1, data2 and mask2

`data1`
first data array and first operand

`data2`
second data array and second operand

`mask1`
mask associated with first data array

`mask2`

mask associated with second data array

Output Arguments

`data1`
data results are stored in data1

`mask1`
if a mask exists, results are stored in mask1

Description

This function performs the operation defined by "mode" using data in array "data1" as the first operand, and data in array "data2" as the second operand. Valid operations are KADD, KSUB, KSUB-FROM, KMULT, KDIV, KDIVINTO, KABSDIFF, KMODULO, KMINIMUM, and KMAXIMUM.

This function assumes that both input arrays, data1 and data2, are valid, and that if either mask1 or mask2 is passed in, then both masks have been passed in.

Potential division by zero is checked, and the maximum unsigned long value [2**(sizeof(unsigned long)) -1] is returned instead.

G.2.5. kdata_arith2_ubyte() — *perform 2-input unsigned byte arithmetic*

Synopsis

```
void kdata_arith2_ubyte(
   int           mode,
   size_t        numpts,
   unsigned char *data1,
   unsigned char *data2,
   unsigned char *mask1,
   unsigned char *mask2 )
```

Input Arguments

`mode`
operation to be performed

`numpts`
number of points in data1, mask1, data2 and mask2

`data1`
first data array and first operand

`data2`
second data array and second operand

`mask1`
mask associated with first data array

`mask2`
mask associated with second data array

Output Arguments

`data1`
: data results are stored in data1

`mask1`
: if a mask exists, results are stored in mask1

Description

This function performs the operation defined by "mode" using data in array "data1" as the first operand, and data in array "data2" as the second operand. Valid operations are KADD, KSUB, KSUB-FROM, KMULT, KDIV, KDIVINTO, KABSDIFF, KMODULO, KMINIMUM, and KMAXIMUM.

This function assumes that both input arrays, data1 and data2, are valid, and that if either mask1 or mask2 is passed in, then both masks have been passed in.

Potential division by zero is checked, and the maximum unsigned byte (255) returned instead.

G.2.6. kdata_fill() — *fill memory with value by data type*

Synopsis

```
int kdata_fill(
   kaddr  data,
   size_t num,
   int    datatype,
   double real,
   double imaginary)
```

Input Arguments

`data`
: point to the data to be filled

`num`
: number of points to be filled

`datatype`
: data type to fill with

`real`
: real part of data to fill with

`imaginary`
: imaginary part of data to fill with

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to fill a region of memory with a specified value. A datatype is used to determine

how to use interpret the real and imaginary values specified. The data types are byte, unsigned byte, short, unsigned short, long, unsigned long, int, unsigned int, float, double, complex, and double complex.

This page left intentionally blank

Table of Contents

Chapter 4

Expression Services

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 4 - Expression Services

A. Introduction

Expression Services provides a symbolic mathematical expression parser implemented using Lex and YACC. The Expression Services library contains routines to set variables, to parse mathematical expressions utilizing variables that have been previously set, and to evaluate these expressions. The expression parser supports a full variety of data types, including byte, unsigned byte, short, unsigned short, integer, unsigned integer, long, unsigned long, float, double, complex, double complex, and string.

The expression parser is used by cantata visual language to handle the functions used with control loops, and by the xprism plotting package to evaluate user-defined functions. It is also used by the *image* object (Please see Program Sevices, Volume III, GUI & Visualization Services) to allow the user to display functions of map columns such as red, green, and blue.

Any application that implements an interactive mathematical computation would find the functions provided by Math Services invaluable. In such an application, the user might want to define a variable "i" with some integer value, a variable "j" with another integer value, and then ask for the value of i*2+j. Math Services is designed specifically to solve equations like this. Variables may be defined, set, and reset. Expressions using those variables may be evaluated and a list of the current variables and their values may be obtained.

The expression parser can evaluate a variety of the mathematical functions provided by Math Services. Therefore, mathematical functions such as "cos,""sin," and "tan" can all be evaluated by the expression parser. Note that the expression parser is only capable of supporting those functions available in Math Services that (1) take double-precision arguments, and (2) return a double-precision result. A table of the math functions supported by the expression parser is as follows:

Supported Functions		
Expression	**Brief description**	**Function Called**
`sin`	double precision sine	ksin
`cos`	double precision cosine	kcos
`tan`	double precision tangent	ktan
`sinh`	hyperbolic sine	ksinh
`cosh`	hyperbolic cosine	kcosh
`tanh`	hyperbolic tangent	ktanh
`asin`	arc sine	kasin
`acos`	arc cosine	kacos
`atan`	arc tangent	katan
`atan2`	two argument arc tangent	katan2
`sinc`	sinc function sin(x)/x	ksinc

Supported Functions		
Expression	**Brief description**	**Function Called**
asinh	arc hyperbolic sine	kasinh
acosh	arc hyperbolic cosine	kacosh
atanh	arc hyperbolic tangent	katanh
ln	natural logarithm	klog
log2	base 2 logarithm	klog2
log10	base 10 logarithm	klog10
log1p	logarithm of x+1	klog1p
exp	exponential function	kexp
exp2	base 2 exponential	kexp2
exp10	base 10 exponential	kexp10
expm1	exponential - 1	kexpm1
pow	power function	kpow
sqrt	square root	ksqrt
cbrt	cube root	kcbrt
abs	absolute value	kfabs
trunc	truncate a number	ktrunc
fmod	floating point modulo	kfmod
ldexp	computes x * 2**n	kldexp
floor	floor	kfloor
ceil	ceiling	kceil
fact	factorial	kfact
impulse	impulse function	kimpulse
step	step function	kstep
sign	sign function	ksign
erf	error function	kerf
erfc	complement error function	kerfc
gamma	gamma function	kgamma
j0	Bessel function j0	kj0
j1	Bessel function j1	kj1
y0	Bessel function y0	ky0
y1	Bessel function y1	ky1
hypot	euclidean distance	khypot
frexp	compute significand and exponent of argument	kfreexp
urng	uniform random number generator	kurng
make_upper	Convert a string to upper case	kstring_upper
make_lower	Convert a string to lower case	kstring_lower
dirname	Get the directory portion of a path	kdirname
basename	Get the filename portion of a path	kbasename
extension	Get the file extension of a filename or path	*internal function*
getenv	Get the value of an environment variable	kgetenv

Supported Functions		
Expression	**Brief description**	**Function Called**
putenv	Set an environment variable	kputenv

The expression parser defines the scope of variables with an *identification number,* or *ID*. This ID is the first parameter to all Expression Services routines and must be declared as type *long*. It is defined by the calling routine to be any number that is deemed suitable. The value of the ID itself is not important, it *is* that it be used consistently.

The ID number, when used properly by the calling routine, defines the domain of variables set and employed. For instance, if all variables set by the user are to be considered global, the application program would use the same ID for all calls to Expression Services. However, if a set of variable definitions and expression evaluations were to be local to a particular module, a different ID must be used for calls to the expression parser in that module. For instance, the same variables with different values might be employed by the user in three different areas of an application, provided that the application used a different ID in each of the three areas.

All Expression Services routines are located in the *kexpr* library (*libkexpr.a*) of the bootstrap toolbox.

ALL programs that utilize these routines MUST include the statement:

```
#include "bootstrap.h"
```

B. Evaluation Utilities

The following sections details the evaluation routines that are part of the Expression Services library, *kexpr*. A list of the evaluation routines then a description of each routine follow.

B.1. Simplified interface routines

The VisiQuest *kexpr* library contains two new APIs that make interfacing with the expression parser more convenient. They are:

```
KexprResult *kexpr_eval(long id,
                        char *expr,
                        KexprResult *result,
                        char *error);

void kexpr_delete_result(KexprResult *result);
```

These functions make use of a new data structure, KexprResult, that records the result of expression evaluation.

The first function, kexpr_eval, parses an input string in the context of the expression ID that is passed in the first parameter, evaluates the compiled expression, and returns the result in a KexprResult structure. The caller can pass in a pointer to a KexprResult structure, or NULL, in which case a new KexprResult structure will be allocated and returned. By checking the 'status' and 'type' data values in the returned structure, the caller can determine if the operation succeeded (status == TRUE) or failed (status == FALSE), and what the data type of

the returned value is (KDOUBLE or KSTRING).

When the return value is a string, kexpr_eval allocates the necessary space in the KexprResult structure to hold it. For convenience, the kexpr_delete_result function is provided to delete the data structure and its string buffer, if necessary.

The kexpr routines described in below, have been added for VisiQuest.

• *kexpr_eval()* - evaluate an expression
• *kexpr_delete_result()* - delete a KexprResult struct

B.2. Routine Descriptions

B.2.1. kexpr_eval() — *evaluate an expression*

Synopsis

```
KexprResult *kexpr_eval(
   long        id,
   char        *string,
   KexprResult *result,
   char        *error)
```

Input Arguments

`id`
the variable identifier. This specifies the variable context to use. Use KEXPR_GLOBAL_ID to use the global variable context.

`string`
the string to be evaluated.

Output Arguments

`result`
holds the result of the evaluation. If this argument is NULL, a new KexprResult structure will be allocated. The result can be one of two types, KSTRING or KDOUBLE. The result type is stored in result->type. The result value is stored in the string_value or double_value members of result. Use the function kexpr_delete_result to free a KexprResult structure. If the result is a string, this function will allocate or re-allocate space in the KexprResult structure to hold it, if necessary.

`error`
If an error occurs the error message is stored in the error string and FALSE is returned in result->status. The error string must be an array of at least 1024 characters. It is allocated by the calling routine. If the error string has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

Pointer to a KexprResult structure. If the result argument was not NULL, 'result' is returned. Otherwise a pointer to a newly allocated KexprResult structure is returned.

Description

This routine evaluates an input expression and stores the resulting value in a KexprResult structure. Currently, the result can be one of two types, KSTRING or KDOUBLE. The result type is stored in result->type. The result value is stored in result->string_value or result->double_value.

Examples

res = kexpr_eval(KEXPR_GLOBAL_ID, "x = 25", NULL, NULL);

res = kexpr_eval(KEXPR_GLOBAL_ID, "ln(x)", res, myerror);

Side Effects

Variable values may be changed by assignment statements.

B.2.2. kexpr_delete_result() — *delete a KexprResult struct*

Synopsis

```
void kexpr_delete_result(KexprResult *result)
```

Input Arguments

`result`

pointer to the structure to free

Returns

none

Description

This routine frees a KexprResult structure allocated by the kexpr_eval() function.

B.3. Introduction to Generic Evaluation Utilities

Below is a list of all the evaluation utilities:

• *kexpr_evaluate_generic()* - evaluate an expression and return result using the desired data type
• *kexpr_evaluate_byte()* - evaluate char/byte expression
• *kexpr_evaluate_ubyte()* - evaluate unsigned byte/char expression
• *kexpr_evaluate_short()* - evaluate short expression
• *kexpr_evaluate_ushort()* - evaluate unsigned short expression
• *kexpr_evaluate_int()* - evaluate integer expression
• *kexpr_evaluate_uint()* - evaluate unsigned integer expression
• *kexpr_evaluate_long()* - evaluate long expression
• *kexpr_evaluate_ulong()* - evaluate unsigned long expression
• *kexpr_evaluate_float()* - evaluate float expression
• *kexpr_evaluate_double()* - evaluate double expression
• *kexpr_evaluate_complex()* - evaluate complex expression
• *kexpr_evaluate_dcomplex()* - evaluate double complex expression
• *kexpr_evaluate_string()* - evaluate string expression
• *kexpr_substitute_exprs()* - substitute expressions within a string

B.4. Definition of Evaluation Utilities

B.4.1. kexpr_evaluate_generic() — *evaluate an expression and return result using the desired data type*

Synopsis

```
int kexpr_evaluate_generic(
   long  id,
   char  *string,
   int   type,
   kaddr value,
   char  *error)
```

Input Arguments

`id`
: the variable identifier.

`string`
: the string to be evaluated.

`type`
: the data type for the value pointer. Valid data types are: KBYTE, KUBYTE, KSHORT, KUSHORT, KINT, KUINT, KLONG, KULONG, KFLOAT, KDOUBLE, KCOMPLEX, KDCOMPLEX, KSTRING.

Output Arguments

`value`

if no error occurred then the value of the expression is stored and True returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine evaluates the input string and returns a generic value of the expression according to the desired data type. See the kexpr_evaluate_{data type} for specific information about evaluation of an expression for the desired data type.

Side Effects

Variable values may be changed by assignment statements.

B.4.2. kexpr_evaluate_byte() — *evaluate char/byte expression*

Synopsis

```
int kexpr_evaluate_byte(

   long id,
   char *string,
   char *value,
   char *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the byte value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets character or byte variables and evaluates character or byte expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_byte() is to perform, and the byte value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_byte(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_byte() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_byte() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.3. kexpr_evaluate_ubyte() — *evaluate unsigned byte/char expression*

Synopsis

```
int kexpr_evaluate_ubyte(

    long     id,
    char     *string,
    unsigned char *value,
    char     *error)
```

Input Arguments

`id`
: the variable identifier.

`string`
: the string to be evaluated.

Output Arguments

`value`
: if no error occurred then the unsigned byte/char value of the expression is stored and TRUE returned.

`error`
if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets unsigned byte variables and evaluates unsigned byte expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_ubyte() is to perform, and the ubyte value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_ubyte(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_ubyte() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_ubyte() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.4. kexpr_evaluate_short() — *evaluate short expression*

Synopsis

```
int kexpr_evaluate_short(

    long  id,
    char  *string,
    short *value,
    char  *error)
```

Input Arguments

`id`
the variable identifier.

`string`
the string to be evaluated.

Output Arguments

`value`

if no error occurred then the short value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets short variables and evaluates short expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_short() is to perform, and the short value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_short(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_short() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_short() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.5. kexpr_evaluate_ushort() — *evaluate unsigned short expression*

Synopsis

```
int kexpr_evaluate_ushort(

    long     id,
    char     *string,
    unsigned short *value,
    char     *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the unsigned short value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets unsigned short variables and evaluates unsigned short expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_ushort() is to perform, and the unsigned short value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_ushort(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_ushort() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_ushort() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.6. kexpr_evaluate_int() — *evaluate integer expression*

Synopsis

```
int kexpr_evaluate_int(

    long id,
    char *string,
    int  *value,
    char *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the integer value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets integer variables and evaluates integer expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_int() is to perform, and the integer value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_int(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_int() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_int() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.7. kexpr_evaluate_uint() — *evaluate unsigned integer expression*

Synopsis

```
int kexpr_evaluate_uint(

    long     id,
    char     *string,
    unsigned int *value,
    char     *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the unsigned integer value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets unsigned integer variables and evaluates unsigned integer expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_uint() is to perform, and the unsigned integer value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_uint(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_uint() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_uint() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.8. kexpr_evaluate_long() — *evaluate long expression*

Synopsis

```
int kexpr_evaluate_long(

    long id,
    char *string,
    long *value,
    char *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the long value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets long integer variables and evaluates long integer expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_long() is to perform, and the long integer value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_long(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_long() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_long() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.9. kexpr_evaluate_ulong() — *evaluate unsigned long expression*

Synopsis

```
int kexpr_evaluate_ulong(

    long     id,
    char     *string,
    unsigned long *value,
    char     *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the unsigned long value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets unsigned long variables and evaluates unsigned long expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_ulong() is to perform, and the unsigned long value returned will reflect this. Let us illustrate with an example. Suppose a string of "i = 10" is passed to kexpr_evaluate_ulong(). This indicates that the variable i is to be defined and assigned the value of 10; the value returned will be 10. Later, suppose kexpr_evaluate_ulong() is called again with the same id, with a string of "i*2+5". Now, kexpr_evaluate_ulong() will evaluate the expression, using the value of i defined by the previous call - [10*2+5]. The value returned in this case would be 25.

B.4.10. kexpr_evaluate_float() — *evaluate float expression*

Synopsis

```
int kexpr_evaluate_float(

    long  id,
    char  *string,
    float *value,
    char  *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the float value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets floating point variables and evaluates floating point expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_float() is to perform, and the float value returned will reflect this. Let us illustrate with an example. Suppose a string of "x = 0.1" is passed to kexpr_evaluate_float(). This indicates that the variable x is to be defined and assigned the value of 0.1; the value returned will be 0.1. Then, kexpr_evaluate_float() is called again with a *string* of "y = 0.9"; now the value returned is 0.9. Finally, suppose kexpr_evaluate_float() is with the with a string of "(x+y)/2". Now, kexpr_evaluate_float() will evaluate the expression, using the values of x and y defined by the previous call: [(0.1 + 0.9) / 2]. The value returned in this case would be 0.5.

B.4.11. kexpr_evaluate_double() — *evaluate double expression*

Synopsis

```
int kexpr_evaluate_double(

   long   id,
   char   *string,
   double *value,
   char   *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the double value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets double variables and evaluates double expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_double() is to perform, and the double value returned will reflect this. Let us illustrate with an example. Suppose a string of "x = 0.1" is passed to kexpr_evaluate_double(). This indicates that the variable x is to be defined and assigned the value of 0.1; the value returned will be 0.1. Then, kexpr_evaluate_double() is called again with a *string* of "y = 0.9"; now the value returned is 0.9. Finally, suppose kexpr_evaluate_double() is with the with a string of "(x+y)/2". Now, kexpr_evaluate_double() will evaluate the expression, using the values of x and y defined by the previous call: [(0.1 + 0.9) / 2]. The value returned in this case would be 0.5.

B.4.12. kexpr_evaluate_complex() — *evaluate complex expression*

Synopsis

```
int kexpr_evaluate_complex(

   long     id,
   char     *string,
   kcomplex *value,
   char     *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the complex value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets complex variables and evaluates complex expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_complex() is to perform, and the complex value returned will reflect this. Let us illustrate with an example. Suppose a string of "x = (0,1)" is passed to kexpr_evaluate_complex(). This indicates that the variable x is to be defined and assigned the complex value of (0i,1j) the value returned will be in kcomplex structure as floating point pairs of real = 0.0, and imaginary = 1.0. Then, kexpr_evaluate_complex() is called again with a *string* of "y = (0,9)"; now the value returned is (0i,9j). Finally, suppose kexpr_evaluate_complex() is with the with a string of "(x+y)/2". Now, kexpr_evaluate_complex() will evaluate the expression, using the values of x and y defined by the previous call: [(0.1 + 0.9) / 2]. The value returned in this case would be (0.5i,0.0j).

B.4.13. kexpr_evaluate_dcomplex() — *evaluate double complex expression*

Synopsis

```
int kexpr_evaluate_dcomplex(

    long      id,
    char      *string,
    kdcomplex *value,
    char      *error)
```

Input Arguments

`id`

the variable identifier.

`string`

the string to be evaluated.

Output Arguments

`value`

if no error occurred then the double complex value of the expression is stored and TRUE returned.

`error`

if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine performs two functions: it sets double complex variables and evaluates double complex expressions. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_dcomplex() is to perform, and the double complex value returned will reflect this. Let us illustrate with an example. Suppose a string of "x = (0,1)" is passed to kexpr_evaluate_dcomplex(). This indicates that the variable x is to be defined and assigned the double complex value of (0i,1j) the value returned will be in kcomplex structure as double pair of real = 0.0, and imaginary = 1.0. Then, kexpr_evaluate_dcomplex() is called again with a *string* of "y = (0,9)"; now the value returned is (0i,9j). Finally, suppose kexpr_evaluate_dcomplex() is with the with a string of "(x+y)/2". Now, kexpr_evaluate_complex() will evaluate the expression, using the values of x and y defined by the previous call: [(0.1 + 0.9) / 2]. The value returned in this case would be (0.5i,0.0j).

B.4.14. kexpr_evaluate_string() — *evaluate string expression*

Synopsis

```
int kexpr_evaluate_string(

    long id,
    char *string,
    char *value,
    char *error)
```

Input Arguments

`id`
: the variable identifier.

`string`
: the string to be evaluated.

Output Arguments

`value`
: if no error occurred then the integer value of the expression is stored and TRUE returned.

`error`
: if an error occurred the error message is stored in the error string array and False returned. The error string array must be at least a 1024 string array that is allocated by the calling routine. If the error string array has not been allocated by the calling routine (error is passed in as NULL) then the error message is output with the kerror facility.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine evaluates the string and returns a string value of the expression. It will return TRUE (1) on success, FALSE (0) on failure. If the routine fails for some reason, an error message will be passed back in the error string. If the error string is NULL then the error is output using the kerror facility.

The string passed in will indicate which function (variable or expression) kexpr_evaluate_string() is to perform, and the string value returned will reflect this. Let us illustrate with an example. Suppose a string of "x = 0.1" is passed to kexpr_evaluate_string(). This indicates that the variable x is to be defined and assigned the value of 0.1; the value returned will be "0.1". Then, kexpr_evaluate_string() is called again with a *string* of "y = 0.9"; now the value returned is "0.9". Finally, suppose kexpr_evaluate_string() is with the with a string of "(x+y)/2". Now, kexpr_evaluate_string() will evaluate the expression, using the values of x and y defined by the previous call: [(0.1 + 0.9) / 2]. The value returned in this case would be "0.5".

B.4.15. kexpr_substitute_exprs() — *substitute expressions within a string*

Synopsis

```
char *kexpr_substitute_exprs(char *str, long id)
```

Input Arguments

`str`
: string to process

`id`
: variable id. Use KEXPR_GLOBAL_ID to use the global variable context.

Returns

result of substituting. This string should be freed by the caller.

Description

kexpr_substitute_exprs evaluates all embedded expressions within a string. These are of the form, $name, ${name}, or ${name:fmt}.

Examples

result = kexpr_substitute_exprs("file.${i:04d}", KEXPR_GLOBAL_ID)

C. Miscellaneous Expression Utilities

Expression Services library, *kexpr*. A list of the miscellaneous routines then descriptions of each routine follow.

C.1. Introduction to Miscellaneous Expression Routines

Below is a list of the miscellaneous expression routines:

- *kexpr_variables_copy()* - copies variables from one id to another
- *kexpr_variables_list()* - list variables associated with an id
- *kexpr_parent_set()* - set the parent variable list associated with an id
- *kexpr_parent_unset()* - unset the parent variable list associated with an id

C.2. Definitions of Miscellaneous Expression Utilities

C.2.1. kexpr_variables_copy() — *copies variables from one id to another*

Synopsis

```
int kexpr_variables_copy(
   long from,
   long to)
```

Input Arguments

`from`

the id variable whose list of current viariables will be copied the id.

`to`

the id variable to recieve the copied list of variables.

Returns

returns TRUE (1) if the list of variables copied successfully, or FALSE (0) if the list of variables was not copied.

Description

This routine copies the current list of variables from one particular id to another.

C.2.2. kexpr_variables_list() — *list variables associated with an id*

Synopsis

```
char **kexpr_variables_list(
   long id,
   int  mode,
   int  *num)
```

Input Arguments

`id`

the id variable whose list of current variables will be extracted.

`mode`

list mode of what to be included in the list of variables. 1 means name and value, 0 means just value.

Output Arguments

`num`

number of variables in list or NULL if not needed

Returns

the list of variables or NULL upon failure

Description

This routines extracts the current list of variables for a particular id variable and returns a pointer to the list of variables.

C.2.3. kexpr_parent_set() — *set the parent variable list associated with an id*

Synopsis

```
int kexpr_parent_set(
   long id,
   long parent)
```

Input Arguments

`id`

the id variable (for the child) in which to set the parent variable list.

`parent`

the id variable for the parent.

Returns

TRUE (1) if the child's parent variable list was set to that of the parent, FALSE (0) otherwise

Description

This routine sets the child alternate variable list specified by the parent id. This variable list will be searched when a variable is not found in the current variable list. If the parent's variable list "parent" is set then the procedure is repeated until either the variable is found or no more variable lists are available to search. In short, this routine provides a method of expanding the scope of variables used.

C.2.4. kexpr_parent_unset() — *unset the parent variable list associated with an id*

Synopsis

```
int kexpr_parent_unset(
   long id)
```

Input Arguments

`id`

the id variable (for the child) in which to unset the parent variable list.

Returns

TRUE (1) if the child's parent list was successfully set to NULL, FALSE (0) otherwise.

Description

This routine unsets the parent variable list previously specified by kexpr_parent_set(). The result of a call to this routine is that if a variable is not found in the variable list associated with the kexpr identifier, then no other variable list will be checked. In short, this routine provides a method of forcing variables used to be "local".

This page left intentionally blank

Table of Contents

This page left intentionally blank

Chapter 5

Operating System Services

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 5 - Operating System Services

A. Introduction

Operating System Services isolates VisiQuest from the operating system and extends the capabilities of the operating system. All applications that use Operating System Services are able to transparently support distributed computing. Operating System Services provides you with a powerful Application Programming Interface (API) which hides the details of data transport, distributed computing, and process execution. The API is modeled after UNIX function calls in order for existing applications to be quickly and easily converted over for distribution.

The distributed computing mechanism implements both a process abstraction and a transport abstraction that unite a network of machines in order to render transparent the location of data and compute resources to VisiQuest applications. Operating System Services has the ability to treat the various data transports (such as shared memory, sockets or files) as single *transport objects*. Thus, all software that uses Operating System Services is isolated from specific, operating system-dependent transport mechanisms.

phantomd is the distributed computing daemon program utility. It is responsible for VisiQuest distributed interprocess communication and managing remote data transport.

Operators are executed locally or remotely to efficiently use a heterogeneous network of machines. cantata utilizes the phantomd to negotiate the remote data transport and spawn processes on remote machines. The visual programmer assigns operators to specific machines interactively to either optimize execution speed or utilize specific hardware. The remote machines do not need full VisiQuest installations, just a running phantomd.

The following sections provide information to customize your environment, and to use the VisiQuest software more efficiently.

B. Data Transports

Data Transport refers to the method used to transfer data between processes. The objective of providing a variety of data transports is to allow the user to use the most efficient method of transferring data from one process to another without encumbering the programmer with the vagaries of the different interfaces to the different data transport mechanisms.

Data Transports simplify the interface to the data transport method by making the interface the same as that used with UNIX files. This approach also provides portability by allowing applications written under VisiQuest to transparently support data transport mechanisms that may not be available on all machines.

Data Transports can be local or remote. Local transport mechanisms include files, shared memory, memory mapped files, pipes, streams, and UNIX domain sockets. The only remote transport mechanism supported by VisiQuest is TCP/IP sockets. With the use of the TCP/IP socket remote data transport, the ability to get

input from and output to remote machines is implemented, and distributed processing is made possible.

Data Transports may be permanent or non-permanent (persistent or non-persistent). Permanent data transports store data to disk, while non-permanent data transports keep the data in a transient state as it is moved from the source to the destination.

Control over which data transports are used is offered within the cantata visual language. The data transport used may also be controlled by individual command line executions of VisiQuest programs.

The transport mechanisms that are supported by VisiQuest are:

file:
Standard UNIX File (local transport / permanent storage)

shm:
Shared Memory (local transport / permanent storage)

mmap:
Memory Mapped Files (permanent storage)

pipe:
Standard Pipes (local transport / no permanent storage)

stream:
Standard Stream (FIFO) (local transport / no permanent storage)

socket:
UNIX domain socket transport (local transport / no permanent storage)

tcpip
TCP/IP socket transport (network transport / no permanent storage)

The default data transport type is "file." The default can be changed to another permanent storage type by setting the environment variable KHOROS_TEMPFILE to "file" "shm", or "mmap". You cannot set the default data transport type using KHOROS_TEMPFILE to pipe, stream, socket, or tcpip because these are non-permanent data transports. Set the variable to "file" to return to the original default.

```
%setenv KHOROS_TEMPFILE file
%setenv KHOROS_TEMPFILE shm
%setenv KHOROS_TEMPFILE mmap
```

Your computer architecture will dictate which, if any, of these transport mechanisms are available for your use. Note that some Data Transports may not be available on some computer architectures.

B.1. Transport Buffering

There are three types of transport buffering models in VisiQuest. The first is for file based transports, the second is for stream based transports, and the third is for memory based transports.

B.1.1. File Buffering

File buffering is used by the file transport. With the file transport, the data is persistent. The file data transport reads data from the file into the transport buffer. When writing, when the transport buffer gets full the buffer is written back to the file. This is fairly straightforward and is well modeled in UNIX via the FILE utilities provided by libc (ie. fopen()). However, in UNIX you cannot implicitly buffer data when reading and writing to a file. The appliction must explicitly flush the data when switching between read/write operations.

In VisiQuest the file buffering has been extended to deal with this intricacy. So the application is free to call read/write operations in any order. This is implemented within the transport buffer by having a validity region which indicates the validity of the data residing in the transport buffer. This enhancement allows us to to get better buffering speeds than available with Sun's FILE buffering (measured using quantify).

B.1.2. Stream Buffering

Stream buffering is used by the stream, tcpip, socket and pipe transports. It was developed specifically to address compilications involved with attempting to map stream based transports into file based transports. The first complication is due to the fact that stream based transports are not persistent; therefore, after data is read or written it is lost. The second complication is that streams will block when reading or writing too much data. This complicates transports buffering code attempting to generically interface to a transport. The notion of stream buffering was developed to formalize the interfacing to streams and to address these problems.

With stream buffering, as data is read or written the transport buffer will accomodate this by dynamically grow to fit all data read or written to date. Seeking on a stream is accomplished by reading or writing to the desired position. One limitation is that the transport buffer is not paged. For example, when streaming a 100 MB file the buffer will grow to 100 MB and then write the data on close.

To override this behavior, applications must open the transport using KOPEN_STREAM, which indicates that persistent is not desired. This causes stream based transports to write the buffer when filled and discard the buffer on read. However, if an application isn't re-reading the data, re-writing the data, or seeking to a previously position this is much more efficient. Both Polymorphic Data Services and Streaming Data Services set KOPEN_STREAM for writing. Streaming data services additionally sets KOPEN_STREAM for reading. For Polymorphic Data Services, reading will be stream buffered when using kpds_open_input_object() on a stream based transport.

B.1.3. Memory Buffering

Memory buffering is used for memory based transports including Shared Memory (shm) and Memory Mapped Files (mmap). For these transports, the file buffering mechanism also works. However, file buffering is not geared to taking full advantage of the fact that the entire content of the transport is already available in memory. Therefore, to take full advantage of memory based transports, the buffering model makes the transport buffer and the transport's memory one and the same. In this manner, memory buffering the allows the memory based transport to read and write directly into memory.

The memory based transports' read and write methods will only be called when trying to read past the end of the available memory. It is then up to the memory based transport to either indicate EOF has been reached or resize the memory segment making more memory to buffer data to. Since the operation of resizing the memory segment is very expensive, both shm and mmap resize the memory segment by multiplying the requested size by 25% and rounding up to the nearest page size. Stream buffering also resizes the transport buffer rounding up to the nearest page size.

To accomodate the ability to have transport buffers be bigger than the data actually buffered, the transport validity has been split into the actual buffer size vs. validity region within the buffer.

B.2. Data Transport Identifier Syntax

If no data transport mechanism is explicitly specified, the VisiQuest data transport routines will use default data transport (normally this is the file data transport, but it can be set with the KHOROS_TEMPFILE environment variable, see above). If you would like to dictate the data transport mechanism of a particular process individually, then you must follow the transport specification syntax. The following syntax used to specify a data transport mechanism:

```
identifier=token
```

The *identifier* is one of the data transport mechanisms listed earlier, such as "shm," "file," or "socket."

The *token* is an identifier for that transport. For a file, it is simply the filename. For shared memory, it is the shared memory key. For a pipe, it is the input & output file descriptors, as in "pipe=[3,4]." For a socket, it is the number of the socket, as in "socket=5430."

C. Distributed Processing

Distributed Processing means the ability to specify remote machines on which to execute individual VisiQuest programs. The capability to do distributed processing is implemented via employment of the remote data transport mechanisms. With distributed processing, one needs a method to execute jobs remotely, as well as a mechanism to transport data back and forth from the remote machine. A remote daemon, *phantomd*, is started on the remote machine, takes requests to execute a job, and transports data involved with that job using the remote data transport mechanisms.

The Distributed Processing capability is utilized either either from the cantata visual language, or from individual command line executions of VisiQuest programs.

C.1. Data Transport Identifier Syntax for Distributed Processing

There are two conventions which dictate what type of data transport mechanism is to be used. If no data transport mechanism is explicitly specified, the VisiQuest transport and distribution routines will negotiate the transport mechanism automatically. However, if you would like to dictate the data transport mechanism yourself, then you must follow the transport specification syntax. The following syntax used to specify a data transport mechanism:

```
identifier=token
```

The *identifier* is one of the data transport mechanisms listed earlier, such as "shm," "file," or "socket."

The *token* is an identifier for that transport. For a file, it is simply the filename. For shared memory, it is the shared memory key. For a pipe, it is the input & output file descriptors, as in "pipe=[3,4]." For a socket, it is the number of the socket, as in "socket=5430."

The second convention is used to specify a remote machine for distributed processing; it specifies where a token is located. For instance, if you want to retrieve a regular file from a remote machine, then the syntax is:

```
file=filename@machine
```

or

```
filename@machine
```

This causes the VisiQuest I/O routines to look thru the internal list of data transport mechanisms and select the first available remote transport mechanism. In the currently available list, this will be "socket". If a specific remote transport mechanism is desired, then the following should be used:

```
filename@socket=machine
```

Multiple machine routing is also allowed. This means that if you cannot directly access a file from your local machine, but you can access the file via the machine "gateway", then you may use the following syntax to retrieve the file:

```
filename@machine@gateway
```

Almost all combinations are allowed, but it is important to remember that *local transport mechanisms CANNOT be used with distributed processing*. For instance, the following specification will result in an error:

```
filename@stream=machine
```

D. Data Types and Casting

D.1. Introduction to Data Type and Casting Utilities

These general utilities are for determining data size and casting between datatypes. They are used the the transports to correctly generate and read files generated on a number of different machines. The functions here include:

- *kdata_size()* - return the size of a khoros data type
- *kdatatype_cast_process()* - cast type for processing
- *kdatatype_to_define()* - takes the string version of the data type and returns the #define value.
- *kdefine_to_datatype()* - takes the #define data type value and returns the string value
- *kdatatype_cast_output()* - recommend an appropriate common data type for processing

D.2. Definitions of Data Transport IPC Utilities

D.2.1. kdata_size() — *return the size of a khoros data type*

Synopsis

```
size_t
kdata_size(int datatype)
```

Input Arguments

`datatype`

the data type for which the

size is being requested.

Returns

The size of the data type specified by the datatype input argument. If the datatype is not known, this function returns 0.

Description

This function returns the size of an element of data of a certain data type on the current machine. For example, this routine will return sizeof(float) if kdata_size(KFLOAT) is called.

Invoking this call:

kmach_sizeof(kmach_type(NULL),datatype)

will result in exactly the same information.

Restrictions

For the bit case, which really should return 1/8, this function returns 1.

D.2.2. kdatatype_cast_process() — *cast type for processing*

Synopsis

```
int kdatatype_cast_process(
   int type1,
   int type2,
   int return_values)
```

Input Arguments

`type1`
first data type to be used in the determination.

`type2`
second data type to be used in the determination.

`return_values`
the OR'ed return values that are acceptable

Returns

The recommended data type.

Description

The routine is used to recommend an appropriate common data type to be used as when processing data. What this routine does is examine the input data types for precision, sign, range, etc. and recommends one of four data types that can be used with minimal risk of destroying data integrity if the variable return_values is KANYTYPE. The values are KLONG, KULONG, KDOUBLE, or KDCOMPLEX. If the variable return_values is not KANYTYPE this routine will return only one of the data types specified. To specify more than one posible return value you must OR the data types together:

```
proc_type  = kdatatype_cast_process(type1, type2,
                                    KUBYTE | KDOUBLE);
```

Any number of data types can be specified. This routine will always try to return the data type that will not destroy data integrity. However, it will never return a data type not specified in return_values. It is up to the programmer to ensure that the data type specified by the return value either preserves the data integrity or handles the loss in data integrity properly.

For example, you would lose data integrity if you called this routine in the following manner :

```
proc_type  = kdatatype_cast_process(KUBYTE, KSHORT,
                                    KUBYTE | KUSHORT);
```

Return_values = KUBYTE | KUSHORT, the returned value will be KUSHORT, but since KSHORT means that negative numbers are posible the data processing the data as KUSHORT will not not reflect this properly. The call should OR KINT or KLONG data type so that negative numbers are processed as negative numbers. However, as stated above it is up to the programmer to specify the return_values so that data integrity is preserved or handles this lose in there processing routine correctly.

If either of the types is KSTRING or KSTRUCT than KNONE is returned.

D.2.3. kdatatype_to_define() — *takes the string version of the data type and returns the #define value.*

Synopsis

```
int kdatatype_to_define(
   char *datatype)
```

Input Arguments

`datatype`
string giving name of data type

Returns

The integer value of the data type or KUNDEFINED if not a datatype

Description

This routine returns the integer value of the string data type passed in. Here is a table of string data types and abbreviations of string data types that can be used and the return values.

Data Type	Return Value
bit	KBIT
byte	KBYTE
unsigned byte	KUBYTE
ubyte	KUBYTE
short	KSHORT
unsigned short	KUSHORT
ushort	KUSHORT
int	KINT
integer	KINT
unsigned int	KUINT

unsigned integer	KUINT
uint	KUINT
long	KLONG
unsigned long	KULONG
ulong	KULONG
float	KFLOAT
double	KDOUBLE
complex	KCOMPLEX
double complex	KDCOMPLEX
dcomplex	KDCOMPLEX
string	KSTRING
struct	KSTRUCT
logical	KLOGICAL
datum	KDATUM

D.2.4. kdefine_to_datatype() — *takes the #define data type value and returns the string value*

Synopsis

```
char *kdefine_to_datatype(
   int type)
```

Input Arguments

`type`

#define of data type

Returns

string that represents data type

Description

This routine returns a string value corresponding to the #define data type passed in. Here is a table of types that can be used and the return values.

Data Type	Return Value
KBIT	bit
KBYTE	byte
KUBYTE	unsigned byte
KSHORT	short
KUSHORT	unsigned short
KINT	integer

KUINT	unsigned integer
KLONG	long
KULONG	unsigned long
KFLOAT	float
KDOUBLE	double
KCOMPLEX	complex
KDCOMPLEX	double complex
KSTRING	string
KSTRUCT	struct
KLOGICAL	logical
KDATUM	datum

D.2.5. kdatatype_cast_output() — *recommend an appropriate common data type for processing*

Synopsis

```
int kdatatype_cast_output(
   int type1,
   int type2)
```

Input Arguments

`type1`
: first data type to be used in the determination.

`type2`
: second data type to be used in the determination.

Returns

The recommended data type.

Description

The routine is used to recommend an appropriate common data type to be used as when outputing data that is the result of calculations involving data of the types indicated in the input arguments. What this routine does is examine the input data types for precision, sign, range, etc. and recommends one of the legal data types that can be used with minimal risk of destroying data integrity.

E. Reading to and Writing from a Data Transport

E.1. Introduction to Data Transport I/O Utilities

These general utilities are for reading and writing from a stream. Data types that are supported include byte, short, int, long, float, double, complex, complex double, unsigned byte, unsigned short, unsigned int, and unsigned long. These routines automatically convert from one machine dependency to another. The functions here include:

- *kread()* - read input from a transport descriptor
- *kread_bit()* - read an array of bits
- *kread_byte()* - read an array of signed bytes
- *kread_complex()* - read an array of complex
- *kread_dcomplex()* - read an array of double complex
- *kread_double()* - read an array of doubles
- *kread_float()* - read an array of floats
- *kread_generic()* - read in any data type.
- *kread_int()* - read an array of signed ints
- *kread_long()* - read an array of signed longs
- *kread_short()* - read an array of signed shorts
- *kread_string()* - read an array of strings
- *kread_ubyte()* - read an array of unsigned bytes
- *kread_uint()* - read an array of unsigned ints
- *kread_ulong()* - read an array of unsigned longs
- *kread_ushort()* - read an array of unsigned shorts
- *kread_array()* - read in a variable array
- *kread_pointer()* - read in a variable array
- *kread_struct()* - read in a single structure
- *kparse_bit()* - read an array of bits
- *kparse_byte()* - read an array of signed bytes
- *kparse_complex()* - read an array of complex
- *kparse_dcomplex()* - read an array of double complex
- *kparse_double()* - read an array of doubles
- *kparse_float()* - read an array of floats
- *kparse_generic()* - read in any data type.
- *kparse_int()* - read an array of signed ints
- *kparse_long()* - read an array of signed longs
- *kparse_short()* - read an array of signed shorts
- *kparse_string()* - read an array of strings
- *kparse_ubyte()* - read an array of unsigned bytes
- *kparse_uint()* - read an array of unsigned ints
- *kparse_ulong()* - read an array of unsigned longs
- *kparse_ushort()* - read an array of unsigned shorts
- *kparse_array()* - read in a variable array
- *kparse_pointer()* - read in a variable array
- *kparse_struct()* - read in a single structure
- *kwrite()* - write output to a transport descriptor
- *kwrite_bit()* - write an array of bits

• *kwrite_byte()* - write an array of signed bytes
• *kwrite_complex()* - write an array of complex
• *kwrite_dcomplex()* - write an array of double complex
• *kwrite_double()* - write an array of doubles
• *kwrite_float()* - write an array of floats
• *kwrite_generic()* - write an array in any data type.
• *kwrite_int()* - write an array of signed ints
• *kwrite_long()* - write an array of signed longs
• *kwrite_short()* - write an array of signed shorts
• *kwrite_string()* - write an array of strings
• *kwrite_ubyte()* - write an array of unsigned bytes
• *kwrite_uint()* - write an array of unsigned ints
• *kwrite_ulong()* - write an array of unsigned longs
• *kwrite_ushort()* - write an array of unsigned shorts
• *kwrite_array()* - write a variable array
• *kwrite_pointer()* - write a variable array
• *kwrite_struct()* - write a single structure
• *kprint_bit()* - write an array of bits
• *kprint_byte()* - write an array of signed bytes
• *kprint_complex()* - write an array of complex
• *kprint_dcomplex()* - write an array of double complex
• *kprint_double()* - write an array of doubles
• *kprint_float()* - write an array of floats
• *kprint_generic()* - write an array in any data type.
• *kprint_int()* - write an array of signed ints
• *kprint_long()* - write an array of signed longs
• *kprint_short()* - write an array of signed shorts
• *kprint_string()* - write an array of strings
• *kprint_ubyte()* - write an array of unsigned bytes
• *kprint_uint()* - write an array of unsigned ints
• *kprint_ulong()* - write an array of unsigned longs
• *kprint_ushort()* - write an array of unsigned shorts
• *kprint_array()* - write a variable array
• *kprint_pointer()* - write a variable array
• *kprint_struct()* - write a single structure

E.2. Definitions of Data Transport Read Utilities

E.2.1. kread() — *read input from a transport descriptor*

Synopsis

```
ssize_t kread(
   int id,
   kaddr buf,
   size_t nbytes)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`buf`

the buffer to read the data into

`nbytes`

the number of bytes to be read

Returns

the number of bytes read, 0 when end of file is encountered, or -1 when an error is encountered.

Description

This function is a replacement for the system "read" call. The only difference is that kread() supports more than just files, it supports other data transports as well, such as shared memory, pipes, files, etc.

The routine will read nbytes into the character array "buf". If not all nbytes can be read then the kread() routine returns the number of bytes actually read.

E.2.2. kread_bit() — *read an array of bits*

Synopsis

```
ssize_t kread_bit(

  int            id,
  unsigned char *data,
  size_t         num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of bits to be read into the data array.

Output Arguments

`data`

the char array in which the data will be stored.

Returns

the number of bits read

Description

This module is used to read an array of bits. The data will need to contain enough memory to store "num" bits.

(note: bits are machine independent and therefore require no conversion. This routine is included to complete the set of read utilities)

E.2.3. kread_byte() — *read an array of signed bytes*

Synopsis

```
ssize_t kread_byte(

  int     id,
  char   *data,
  size_t  num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of signed bytes to be read into the data array.

Output Arguments

`data`

the char array in which the data will be stored.

Returns

the number of signed bytes read

Description

This module is used to read an array of signed bytes. The data will need to contain enough memory to store "num" signed bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of read utilities)

E.2.4. kread_complex() — *read an array of complex*

Synopsis

```
ssize_t kread_complex(

  int id,
  kcomplex *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of complex to be read into the data array.

Output Arguments

`data`
: the kcomplex array in which the data will be stored.

Returns

the number of complex read

Description

This module is used to read an array of complex numbers. The data will need to contain enough memory to store "num" complex structures.

E.2.5. kread_dcomplex() — *read an array of double complex*

Synopsis

```
ssize_t kread_dcomplex(

  int id,
  kdcomplex *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of double complex to be read into the data array.

Output Arguments

`data`

the kdcomplex array in which the data will be stored.

Returns

the number of double complex read

Description

This module is used to read an array of double complex numbers. The data will need to contain enough memory to store "num" double complex structures.

E.2.6. kread_double() — *read an array of doubles*

Synopsis

```
ssize_t kread_double(

  int     id,
  double *data,
  size_t  num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of doubles to be read into the data array.

Output Arguments

`data`

the double array in which the data will be stored.

Returns

the number of doubles read

Description

This module is used to read an array of doubles. The data will need to contain enough memory to store "num" doubles.

E.2.7. kread_float() — *read an array of floats*

Synopsis

```
ssize_t kread_float(

  int    id,
  float *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of floats to be read into the data array.

Output Arguments

`data`
: the float array in which the data will be stored.

Returns

the number of floats read

Description

This module is used to read an array of floats. The data will need to contain enough memory to store "num" floats.

E.2.8. kread_generic() — *read in any data type.*

Synopsis

```
ssize_t
kread_generic(
   int id,
   kaddr data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of data points to be read into the data array.

`type`

data type.

Output Arguments

`data`

the kaddr array in which the data will be stored.

Returns

the number of data points read on success. -1 otherwise.

Description

This module is used to read an array of data from a transport. The data will need to contain enough memory to store "num" data points of the specified type.

E.2.9. kread_int() — *read an array of signed ints*

Synopsis

```
ssize_t kread_int(

  int     id,
  int    *data,
  size_t  num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of signed integers to be read into the data array.

Output Arguments

`data`

the integer array in which the data will be stored.

Returns

the number of integers read

Description

This module is used to read an array of signed ints. The data will need to contain enough memory to store "num" signed ints.

E.2.10. kread_long() — *read an array of signed longs*

Synopsis

```
ssize_t kread_long(

  int     id,
  long   *data,
  size_t  num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of signed longs to be read into the data array.

Output Arguments

`data`
: the long array in which the data will be stored.

Returns

the number of longs read

Description

This module is used to read an array of signed longs. The data will need to contain enough memory to store "num" signed longs.

E.2.11. kread_short() — *read an array of signed shorts*

Synopsis

```
ssize_t kread_short(

  int     id,
  short  *data,
  size_t  num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of signed shorts to be read into the data array.

Output Arguments

`data`

the short array in which the data will be stored.

Returns

the number of short integers read

Description

This module is used to read an array of signed shorts. The data will need to contain enough memory to store "num" signed shorts.

E.2.12. kread_string() — *read an array of strings*

Synopsis

```
ssize_t kread_string(

  int id,
  kstring *data,
  size_t num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of string to be read into the data array.

Output Arguments

`data`

the kstring array in which the data will be stored.

Returns

the number of strings read

Description

This module is used to read an array of strings. The data will need to contain enough memory to store "num" string pointers. The memory that the strings are stored in will be allocated by kread_string.

Where "kstring" is really a typedef for a character pointer "char *". So an array of kstring is really an array of character pointers.

Side Effects

the strings that are returned should be freed by the programmer

E.2.13. kread_ubyte() — *read an array of unsigned bytes*

Synopsis

```
ssize_t kread_ubyte(

  int id,
  unsigned char *data,
  size_t num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of unsigned bytes to be read into the data array.

Output Arguments

`data`

the unsigned char array in which the data will be stored.

Returns

the number of bytes read

Description

This module is used to read an array of unsigned bytes. The data will need to contain enough memory to store "num" unsigned bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of read utilities)

E.2.14. kread_uint() — *read an array of unsigned ints*

Synopsis

```
ssize_t kread_uint(

  int id,
  unsigned int *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of unsigned ints to be read into the data array.

Output Arguments

`data`
: the unsigned integer array in which the data will be stored.

Returns

the number of integers read

Description

This module is used to read an array of unsigned ints. The data will need to contain enough memory to store "num" unsigned ints.

E.2.15. kread_ulong() — *read an array of unsigned longs*

Synopsis

```
ssize_t kread_ulong(

  int id,
  unsigned long *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of unsigned longs to be read into the data array.

Output Arguments

`data`

the unsigned long array in which the data will be stored.

Returns

the number of longs read

Description

This module is used to read an array of unsigned longs. The data will need to contain enough memory to store "num" unsigned longs.

E.2.16. kread_ushort() — *read an array of unsigned shorts*

Synopsis

```
ssize_t kread_ushort(

  int id,
  unsigned short *data,
  size_t num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of unsigned shorts to be read into the data array.

Output Arguments

`data`

the unsigned short array in which the data will be stored.

Returns

the number of shorts read

Description

This module is used to read an array of unsigned shorts. The data will need to contain enough memory to store "num" unsigned shorts.

E.2.17. kread_array() — *read in a variable array*

Synopsis

```
ssize_t
kread_array(
   int id,
   kaddr *data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the maximum number of data points to be read into the data array.

`type`
: data / structure type.

Output Arguments

`data`
: the kaddr array in which the data will be stored.

Returns

the number of points read or -1 on error

Description

This module is used to read in a pointer to an variable sized array. The size of the array and the data is stored in the transport. The reader first reads the number of points, allocates enough space for the return data, and then calls kread_generic() to do the actual reading of the data. If NULL is stored then kread_array() will set the data pointer to NULL and return that 0 data points were read.

The "num" argument should be used to represent a maximum number of data points to be read. In the case that num is less than the number of data points stored, then kread_array() will advance to the end of the stored data. This allows for partial reads of the stored data.

E.2.18. kread_pointer() — *read in a variable array*

Synopsis

```
int
kread_pointer(
   int id,
   kaddr *data,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`type`
: data / structure type.

Output Arguments

`data`
: the kaddr array in which the data will be stored.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to read in a pointer to an variable sized array. The size of the array and the data is stored in the transport. The reader first reads the number of points, allocates enough space for the return data, and then calls kread_generic() to do the actual reading of the data. If NULL is stored then kread_pointer() will set the data pointer to NULL and return that 0 data points were read.

E.2.19. kread_struct() — *read in a single structure*

Synopsis

```
ssize_t
kread_struct(
   int id,
   kaddr *data,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`type`

structure type.

Output Arguments

`data`

a pointer in which the structures will be stored.

Returns

the number of structs read

Description

This module is used to read in a single structure from a transport. Unlike kread_generic() this routine will malloc the returned memory for a single structure.

E.2.20. kparse_bit() — *read an array of bits*

Synopsis

```
ssize_t kparse_bit(

  int            id,
  unsigned char *data,
  size_t         num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of bits to be read into the data array.

Output Arguments

`data`

the char array in which the data will be stored.

Returns

the number of bits read

Description

This module is used to read an array of bits. The data will need to contain enough memory to store "num" bits.

(note: bits are machine independent and therefore require no conversion. This routine is included to complete the set of read utilities)

E.2.21. kparse_byte() — *read an array of signed bytes*

Synopsis

```
ssize_t kparse_byte(

  int     id,
  char   *data,
  size_t  num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of signed bytes to be read into the data array.

Output Arguments

`data`

the char array in which the data will be stored.

Returns

the number of signed bytes read

Description

This module is used to read an array of signed bytes. The data will need to contain enough memory to store "num" signed bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of read utilities)

E.2.22. kparse_complex() — *read an array of complex*

Synopsis

```
ssize_t kparse_complex(

  int id,
  kcomplex *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of complex to be read into the data array.

Output Arguments

`data`
: the kcomplex array in which the data will be stored.

Returns

the number of complex read

Description

This module is used to read an array of complex numbers. The data will need to contain enough memory to store "num" complex structures.

E.2.23. kparse_dcomplex() — *read an array of double complex*

Synopsis

```
ssize_t kparse_dcomplex(

  int id,
  kdcomplex *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of double complex to be read into the data array.

Output Arguments

`data`

the kdcomplex array in which the data will be stored.

Returns

the number of double complex read

Description

This module is used to read an array of double complex numbers. The data will need to contain enough memory to store "num" double complex structures.

E.2.24. kparse_double() — *read an array of doubles*

Synopsis

```
ssize_t kparse_double(

  int     id,
  double *data,
  size_t  num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of doubles to be read into the data array.

Output Arguments

`data`

the double array in which the data will be stored.

Returns

the number of doubles read

Description

This module is used to read an array of doubles. The data will need to contain enough memory to store "num" doubles.

E.2.25. kparse_float() — *read an array of floats*

Synopsis

```
ssize_t kparse_float(

  int    id,
  float *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of floats to be read into the data array.

Output Arguments

`data`
: the float array in which the data will be stored.

Returns

the number of floats read

Description

This module is used to read an array of floats. The data will need to contain enough memory to store "num" floats.

E.2.26. kparse_generic() — *read in any data type.*

Synopsis

```
ssize_t
kparse_generic(
   int id,
   kaddr data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of data points to be read into the data array.

`type`

data type.

Output Arguments

`data`

the kaddr array in which the data will be stored.

Returns

the number of data points read

Description

This module is used to read an array of data from a transport. The data will need to contain enough memory to store "num" data points of the specified type.

E.2.27. kparse_int() — *read an array of signed ints*

Synopsis

```
ssize_t kparse_int(

  int     id,
  int    *data,
  size_t  num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of signed integers to be read into the data array.

Output Arguments

`data`

the integer array in which the data will be stored.

Returns

the number of integers read

Description

This module is used to read an array of signed ints. The data will need to contain enough memory to store "num" signed ints.

E.2.28. kparse_long() — *read an array of signed longs*

Synopsis

```
ssize_t kparse_long(

  int     id,
  long   *data,
  size_t  num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of signed longs to be read into the data array.

Output Arguments

`data`
: the long array in which the data will be stored.

Returns

the number of longs read

Description

This module is used to read an array of signed longs. The data will need to contain enough memory to store "num" signed longs.

E.2.29. kparse_short() — *read an array of signed shorts*

Synopsis

```
ssize_t kparse_short(

  int     id,
  short  *data,
  size_t  num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of signed shorts to be read into the data array.

Output Arguments

`data`

the short array in which the data will be stored.

Returns

the number of short integers read

Description

This module is used to read an array of signed shorts. The data will need to contain enough memory to store "num" signed shorts.

E.2.30. kparse_string() — *read an array of strings*

Synopsis

```
ssize_t kparse_string(

  int id,
  kstring *data,
  size_t num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of string to be read into the data array.

Output Arguments

`data`

the kstring array in which the data will be stored.

Returns

the number of strings read

Description

This module is used to read an array of strings. The data will need to contain enough memory to store "num" string pointers. The memory that the strings are stored in will be allocated by kparse_string.

Where "kstring" is really a typedef for a character pointer "char *". So an array of kstring is really an array of character pointers.

Side Effects

the strings that are returned should be freed by the programmer

E.2.31. kparse_ubyte() — *read an array of unsigned bytes*

Synopsis

```
ssize_t kparse_ubyte(

  int id,
  unsigned char *data,
  size_t num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of unsigned bytes to be read into the data array.

Output Arguments

`data`

the unsigned char array in which the data will be stored.

Returns

the number of bytes read

Description

This module is used to read an array of unsigned bytes. The data will need to contain enough memory to store "num" unsigned bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of read utilities)

E.2.32. kparse_uint() — *read an array of unsigned ints*

Synopsis

```
ssize_t kparse_uint(

  int id,
  unsigned int *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the number of unsigned ints to be read into the data array.

Output Arguments

`data`
: the unsigned integer array in which the data will be stored.

Returns

the number of integers read

Description

This module is used to read an array of unsigned ints. The data will need to contain enough memory to store "num" unsigned ints.

E.2.33. kparse_ulong() — *read an array of unsigned longs*

Synopsis

```
ssize_t kparse_ulong(

  int id,
  unsigned long *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`

the number of unsigned longs to be read into the data array.

Output Arguments

`data`

the unsigned long array in which the data will be stored.

Returns

the number of longs read

Description

This module is used to read an array of unsigned longs. The data will need to contain enough memory to store "num" unsigned longs.

E.2.34. kparse_ushort() — *read an array of unsigned shorts*

Synopsis

```
ssize_t kparse_ushort(

  int id,
  unsigned short *data,
  size_t num)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`num`

the number of unsigned shorts to be read into the data array.

Output Arguments

`data`

the unsigned short array in which the data will be stored.

Returns

the number of shorts read

Description

This module is used to read an array of unsigned shorts. The data will need to contain enough memory to store "num" unsigned shorts.

E.2.35. kparse_array() — *read in a variable array*

Synopsis

```
ssize_t
kparse_array(
   int id,
   kaddr *data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`num`
: the maximum number of data points to be read into the data array.

`type`
: data / structure type.

Output Arguments

`data`
: the kaddr array in which the data will be stored.

Returns

the number of points read or -1 on error

Description

This module is used to read in a pointer to an variable sized array. The size of the array and the data is stored in the transport. The reader first reads the number of points, allocates enough space for the return data, and then calls kparse_generic() to do the actual reading of the data. If NULL is stored then kparse_array() will set the data pointer to NULL and return that 0 data points were read.

The "num" argument should be used to represent a maximum number of data points to be read. In the case that num is less than the number of data points stored, then kparse_array() will advance to the end of the stored data. This allows for partial reads of the stored data.

E.2.36. kparse_pointer() — *read in a variable array*

Synopsis

```
int
kparse_pointer(
   int id,
   kaddr *data,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`type`
: data / structure type.

Output Arguments

`data`
: the kaddr array in which the data will be stored.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to read in a pointer to an variable sized array. The size of the array and the data is stored in the transport. The reader first reads the number of points, allocates enough space for the return data, and then calls kparse_generic() to do the actual reading of the data. If NULL is stored then kparse_pointer() will set the data pointer to NULL and return that 0 data points were read.

E.2.37. kparse_struct() — *read in a single structure*

Synopsis

```
ssize_t
kparse_struct(
   int id,
   kaddr *data,
   int type)
```

Input Arguments

`id`
: the file id to be read which was opened earlier with kopen().

`type`

structure type.

Output Arguments

`data`

a pointer in which the structures will be stored.

Returns

the number of structs read

Description

This module is used to read in a single structure from a transport. Unlike kparse_generic() this routine will malloc the returned memory for a single structure.

E.3. Definitions of Data Transport Write Utilities

E.3.1. kwrite() — *write output to a transport descriptor*

Synopsis

```
ssize_t kwrite(
   int id,
   kaddr buf,
   size_t nbytes)
```

Input Arguments

`id`

the file id to be read which was opened earlier with kopen().

`buf`

the buffer to write the data from

`nbytes`

the number of bytes to be written

Returns

The number of bytes written, or -1 when an error is encountered.

Description

This function is a replacement for the system "write" call. The only difference is that kwrite() supports more than just files, it supports other data transports as well, such as shared memory, pipes, files, etc.

The routine will write nbytes from the character array "buf" to the appropriate transport mechanism specified by id. If not all nbytes can be read then the kread() routine returns the number of bytes actually read.

E.3.2. kwrite_bit() — *write an array of bits*

Synopsis

```
ssize_t kwrite_bit(

  int id,
  unsigned char *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written which was opened earlier with kopen().

`data`
: the unsigned char array which holds the data.

`num`
: the number of bits to be written.

Returns

the number of bits written

Description

This module is used to write an array of bits.

(note: bits are machine independent and therefore require no conversion. This routine is included to complete the set of write utilities)

E.3.3. kwrite_byte() — *write an array of signed bytes*

Synopsis

```
ssize_t kwrite_byte(

  int id,
  char *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the char array which holds the data.

`num`
: the number of signed bytes to be written.

Returns

the number of bytes written

Description

This module is used to write an array of signed bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of write utilities)

E.3.4. kwrite_complex() — *write an array of complex*

Synopsis

```
ssize_t kwrite_complex(

  int id,
  kcomplex *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`

the kcomplex array which holds the data.

`num`

the number of complex pairs to be written.

Returns

the number of complexs written

Description

This module is used to write an array of complex numbers.

E.3.5. kwrite_dcomplex() — *write an array of double complex*

Synopsis

```
ssize_t kwrite_dcomplex(

  int id,
  kdcomplex *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the kdcomplex array which holds the data.

`num`

the number of double complex pairs to be written.

Returns

the number of double complexs written

Description

This module is used to write an array of double complex numbers.

E.3.6. kwrite_double() — *write an array of doubles*

Synopsis

```
ssize_t kwrite_double(

  int id,
  double *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the double array which holds the data.

`num`
: the number of doubles to be written.

Returns

the number of doubles written

Description

This module is used to write an array of doubles.

E.3.7. kwrite_float() — *write an array of floats*

Synopsis

```
ssize_t kwrite_float(

  int id,
  float *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the float array which holds the data.

`num`
: the number of floats to be written.

Returns

the number of floats written

Description

This module is used to write an array of floats.

E.3.8. kwrite_generic() — *write an array in any data type.*

Synopsis

```
ssize_t
kwrite_generic(
   int id,
   kaddr data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the kaddr array which holds the data.

`num`
: the number of data points to be written.

`type`
: data type.

Returns

the number of points written or -1 on error

Description

This module is used to write an array of data to a transport.

E.3.9. kwrite_int() — *write an array of signed ints*

Synopsis

```
ssize_t kwrite_int(

  int id,
  int *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the integer array which holds the data.

`num`
: the number of integers to be written.

Returns

the number of ints written

Description

This module is used to write an array of signed ints.

E.3.10. kwrite_long() — *write an array of signed longs*

Synopsis

```
ssize_t kwrite_long(

  int id,
  long *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the long array in which holds the data.

`num`
: the number of longs to be written.

Returns

the number of longs written

Description

This module is used to write an array of signed longs. The data will need to contain enough memory to store "num" signed longs.

E.3.11. kwrite_short() — *write an array of signed shorts*

Synopsis

```
ssize_t kwrite_short(

  int id,
  short *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the short array which holds the data.

`num`

the number of shorts to be written.

Returns

the number of shorts written

Description

This module is used to write an array of signed shorts.

E.3.12. kwrite_string() — *write an array of strings*

Synopsis

```
ssize_t kwrite_string(

  int id,
  kstring *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the string array in which the data will be stored.

`num`
: the number of strings to be written.

Returns

the number of strings written

Description

This module is used to write an array of strings.

E.3.13. kwrite_ubyte() — *write an array of unsigned bytes*

Synopsis

```
ssize_t kwrite_ubyte(

  int id,
  unsigned char *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the unsigned char array which holds the data.

`num`
: the number of unsigned bytes to be written.

Returns

the number of bytes written

Description

This module is used to write an array of unsigned bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of write utilities)

E.3.14. kwrite_uint() — *write an array of unsigned ints*

Synopsis

```
ssize_t kwrite_uint(

  int id,
  unsigned int *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the unsigned integer array which holds the data.

`num`

the number of unsigned integers to be written.

Returns

the number of ints written

Description

This module is used to write an array of unsigned ints.

E.3.15. kwrite_ulong() — *write an array of unsigned longs*

Synopsis

```
ssize_t kwrite_ulong(

  int id,
  unsigned long *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the unsigned long array which holds the data.

`num`
: the number of unsigned longs to be written.

Returns

the number of longs written

Description

This module is used to write an array of unsigned longs.

E.3.16. kwrite_ushort() — *write an array of unsigned shorts*

Synopsis

```
ssize_t kwrite_ushort(

  int id,
  unsigned short *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the unsigned short array which holds the data.

`num`
: the number of unsigned shorts to be written.

Returns

the number of shorts written

Description

This module is used to write an array of unsigned shorts.

E.3.17. kwrite_array() — *write a variable array*

Synopsis

```
ssize_t
kwrite_array(
   int id,
   kaddr data,
   size_t num,
   int type)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the array in which the data will be written from.

`num`

the number of data points to be written to the transport.

`type`

data / structure type.

Returns

the number of points written or -1 on error

Description

This module is used to write a pointer to an variable sized array. The size of the array and the data is stored in the transport. The writer first writes the number of points, and then calls kwrite_generic() to do the actual writing of the data. If NULL is passed in then kwrite_array() will write out 0 for the number of points, which will then be used by kwrite_array() to return NULL.

E.3.18. kwrite_pointer() — *write a variable array*

Synopsis

```
int
kwrite_pointer(
   int id,
   kaddr *data,
   int type)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the array in which the data will be written from. num - the number of data points to be written to the transport.

`type`
: data / structure type.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to write a pointer to an variable sized array. The size of the array and the data is stored in the transport. The writer first writes the number of points, and then calls kwrite_generic() to do the actual writing of the data. If NULL is passed in then kwrite_pointer() will write out 0 for the number of points, which will then be used by kread_pointer() to return NULL.

E.3.19. kwrite_struct() — *write a single structure*

Synopsis

```
ssize_t
kwrite_struct(
   int id,
   kaddr data,
   int type)
```

Input Arguments

`id`
: the file id to write to which was opened earlier with kopen().

`type`
: structure type.

Output Arguments

`data`

a pointer to a single structure to be written.

Returns

the number of structs written

Description

This module is used to write a single structure to a transport.

E.3.20. kprint_bit() — *write an array of bits*

Synopsis

```
ssize_t kprint_bit(

  int id,
  unsigned char *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written which was opened earlier with kopen().

`data`

the unsigned char array which holds the data.

`num`

the number of bits to be written.

Returns

the number of bits written

Description

This module is used to write an array of bits.

(note: bits are machine independent and therefore require no conversion. This routine is included to complete the set of write utilities)

E.3.21. kprint_byte() — *write an array of signed bytes*

Synopsis

```
ssize_t kprint_byte(

  int id,
  char *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the char array which holds the data.

`num`
: the number of signed bytes to be written.

Returns

the number of bytes written

Description

This module is used to write an array of signed bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of write utilities)

E.3.22. kprint_complex() — *write an array of complex*

Synopsis

```
ssize_t kprint_complex(

  int id,
  kcomplex *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`

the kcomplex array which holds the data.

`num`

the number of complex pairs to be written.

Returns

the number of complexs written

Description

This module is used to write an array of complex numbers.

E.3.23. kprint_dcomplex() — *write an array of double complex*

Synopsis

```
ssize_t kprint_dcomplex(

  int id,
  kdcomplex *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the kdcomplex array which holds the data.

`num`

the number of double complex pairs to be written.

Returns

the number of double complexs written

Description

This module is used to write an array of double complex numbers.

E.3.24. kprint_double() — *write an array of doubles*

Synopsis

```
ssize_t kprint_double(

  int id,
  double *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the double array which holds the data.

`num`
: the number of doubles to be written.

Returns

the number of doubles written

Description

This module is used to write an array of doubles.

E.3.25. kprint_float() — *write an array of floats*

Synopsis

```
ssize_t kprint_float(

  int id,
  float *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the float array which holds the data.

`num`
: the number of floats to be written.

Returns

the number of floats written

Description

This module is used to write an array of floats.

E.3.26. kprint_generic() — *write an array in any data type.*

Synopsis

```
ssize_t
kprint_generic(
   int id,
   kaddr data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the kaddr array which holds the data.

`num`
: the number of data points to be written.

`type`
: data type.

Returns

the number of points written or -1 on error

Description

This module is used to write an array of data to a transport.

E.3.27. kprint_int() — *write an array of signed ints*

Synopsis

```
ssize_t kprint_int(

  int id,
  int *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the integer array which holds the data.

`num`
: the number of integers to be written.

Returns

the number of ints written

Description

This module is used to write an array of signed ints.

E.3.28. kprint_long() — *write an array of signed longs*

Synopsis

```
ssize_t kprint_long(

  int id,
  long *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the long array in which holds the data.

`num`
: the number of longs to be written.

Returns

the number of longs written

Description

This module is used to write an array of signed longs. The data will need to contain enough memory to store "num" signed longs.

E.3.29. kprint_short() — *write an array of signed shorts*

Synopsis

```
ssize_t kprint_short(

  int id,
  short *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the short array which holds the data.

`num`

the number of shorts to be written.

Returns

the number of shorts written

Description

This module is used to write an array of signed shorts.

E.3.30. kprint_string() — *write an array of strings*

Synopsis

```
ssize_t kprint_string(

  int id,
  kstring *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the string array in which the data will be stored.

`num`
: the number of strings to be written.

Returns

the number of strings written

Description

This module is used to write an array of strings.

E.3.31. kprint_ubyte() — *write an array of unsigned bytes*

Synopsis

```
ssize_t kprint_ubyte(

  int id,
  unsigned char *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the unsigned char array which holds the data.

`num`
: the number of unsigned bytes to be written.

Returns

the number of bytes written

Description

This module is used to write an array of unsigned bytes.

(note: bytes are machine independent and therefore require no conversion. This routine is included to complete the set of write utilities)

E.3.32. kprint_uint() — *write an array of unsigned ints*

Synopsis

```
ssize_t kprint_uint(

  int id,
  unsigned int *data,
  size_t num)
```

Input Arguments

`id`

the file id to be written to which was opened earlier with kopen().

`data`

the unsigned integer array which holds the data.

`num`

the number of unsigned integers to be written.

Returns

the number of ints written

Description

This module is used to write an array of unsigned ints.

E.3.33. kprint_ulong() — *write an array of unsigned longs*

Synopsis

```
ssize_t kprint_ulong(

  int id,
  unsigned long *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the unsigned long array which holds the data.

`num`
: the number of unsigned longs to be written.

Returns

the number of longs written

Description

This module is used to write an array of unsigned longs.

E.3.34. kprint_ushort() — *write an array of unsigned shorts*

Synopsis

```
ssize_t kprint_ushort(

  int id,
  unsigned short *data,
  size_t num)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the unsigned short array which holds the data.

`num`
: the number of unsigned shorts to be written.

Returns

the number of shorts written

Description

This module is used to write an array of unsigned shorts.

E.3.35. kprint_array() — *write a variable array*

Synopsis

```
ssize_t
kprint_array(
   int id,
   kaddr data,
   size_t num,
   int type)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the array in which the data will be written from.

`num`
: the number of data points to be written to the transport.

`type`
: data / structure type.

Returns

the number of points written or -1 on error

Description

This module is used to write a pointer to an variable sized array. The size of the array and the data is stored in the transport. The writer first writes the number of points, and then calls kprint_generic() to do the actual writing of the data. If NULL is passed in then kprint_array() will write out 0 for the number of points, which will then be used by kprint_array() to return NULL.

E.3.36. kprint_pointer() — *write a variable array*

Synopsis

```
int
kprint_pointer(
   int id,
   kaddr *data,
   int type)
```

Input Arguments

`id`
: the file id to be written to which was opened earlier with kopen().

`data`
: the array in which the data will be written from. num - the number of data points to be written to the transport.

`type`
: data / structure type.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This module is used to write a pointer to an variable sized array. The size of the array and the data is stored in the transport. The writer first writes the number of points, and then calls kprint_generic() to do the actual writing of the data. If NULL is passed in then kprint_pointer() will write out 0 for the number of points, which will then be used by kread_pointer() to return NULL.

E.3.37. kprint_struct() — *write a single structure*

Synopsis

```
ssize_t
kprint_struct(
   int id,
   kaddr data,
   int type)
```

Input Arguments

`id`
: the file id to write to which was opened earlier with kopen().

`type`
: structure type.

Output Arguments

`data`

a pointer to a single structure to be written.

Returns

the number of structs written

Description

This module is used to write a single structure to a transport.

F. I/O Utilities

These utilities deal with I/O, and are modeled after the file I/O utilities provided by *libc*. The difference is that the VisiQuest utilities support the VisiQuest Data Transport capabilities, and *must* be used by any program that is to take advantage of Data Transport mechanisms, including distributed computing. Some utilities here are also geared directly for dealing with Data Transport mechanisms.

Data Transport refers to the method used to transfer data between processes. Data Transport mechanisms can be local or remote. Local transport mechanisms include shared memory, files, pipes and streams; remote transport mechanisms include sockets, tli and rpc. With the use of remote data transport, the ability to get input from and output to remote machines is implemented, and distributed processing is made possible.

Distributed Processing means the ability to specify remote machines on which to execute individual VisiQuest programs. The capability to do distributed processing is implemented via employment of the remote data transport mechanisms. With Distributed Processing, one needs a method to execute jobs remotely, as well as a mechanism to transport data back & forth from the remote machine. A remote daemon, phantomd, is started on the remote machine, takes requests to execute a job and transports data involved with that job using the remote Data Transport mechanisms.

When writing VisiQuest routines that are to support the VisiQuest Data Transports and Distributed Processing, it is crucial that they use the functions described in this section. Standard VisiQuest file I/O functions use the *kfile* pointer (as opposed to a FILE pointer). The kfile pointer is treated similarly to the *libc* FILE pointer. However, the kfile pointer may not, in fact, be a stream to a file. It may be any of the VisiQuest Data Transports, such as shared memory, pipes or streams. The low-level VisiQuest I/O functions use a *data transport ID (descriptor)* or *kfile ID (descriptor)*. The kfile ID is treated similarly to the *libc* file identifier, except that it may also refer to any of the supported VisiQuest Data Transports.

F.1. Introduction to Low-level I/O Functions

These functions cover the closing and locking of a file. Also included are functions to obtain various statistics of the file. These routines include:

- *kaccess()* - determine accessibility of file
- *kclearerr()* - clear the EOF/error flags of a data transport stream
- *kclose()* - close and delete a transport descriptor
- *kcreat()* - routine for creating a khoros transport
- *kdup()* - duplicate an existing khoros transport descriptor
- *kdup2()* - specifically duplicate an existing khoros transport descriptor
- *kexit()* - terminate a process
- *kfileno()* - return the transport descriptor
- *kgetbuffer()* - get the current data transport stream's input/output buffer and it's size
- *kgetc()* - get character from khoros transport
- *kgetdescriptors()* - get true UNIX file descriptors
- *kgethostname()* - get the current hostname
- *kgets()* - reads from kstdin until a newline or EOF
- *kinput()* - opens a file for reading using kopen()
- *klseek()* - move read/write pointer of a transport descriptor
- *kopen()* - open or create a file for reading and/or writing
- *koutput()* - opens/creates a file for writing using kopen()
- *kpclose()* - close a pipe (for I/O) from or to a process
- *kpopen()* - open a pipe (for I/O) from or to a process
- *kpinfo()* - gets the associated process id
- *kprintf()* - print formatted output to kstdout
- *kputc()* - put a character onto the khoros transport
- *krename()* - rename a khoros transport from path1 to path2
- *krewind()* - rewind a data transport stream to the beginning
- *kscanf()* - scan kstdin and format the input into one or more arguments of the type specified
- *ksetvbuf()* - set the I/O buffer of a data transport stream
- *ksprintf()* - print one or more arguments in the format specified to an output string.
- *kmsprintf()* - print one or more arguments in the format specified and return an allocated output string.
- *ksscanf()* - scan input string and format it into one or more arguments of the type specified.
- *ktell()* - report the position of the read/write pointer
- *ktouch()* - routine for touching a temporary transport
- *ktmpfile()* - create a temporary data transport stream
- *kungetc()* - push a character back onto the data transport stream
- *kunlink()* - remove a filename from a directory entry

F.2. Definitions of Low-level I/O Functions

F.2.1. kaccess() — *determine accessibility of file*

Synopsis

```
int kaccess(
   const char *path,
   int mode)
```

Input Arguments

`path`

is the string containing the path name to the desired file to be accessed. The path name ident- ifies the file/token to be tested, just like the regular UNIX access() command.

`mode`

the mode in which kaccess() should test for the file. The mode is comprised of an or'ing of the following flags:

R_OK - the file readable
W_OK - the file writeable
X_OK - the file executable
F_OK - the file and directory leading to the file accessible.

Returns

return -1 for failure otherwise 0 for success

Description

This function is a replacement for the system "access" call. The only difference is that kaccess() checks more than just files, it will also check other data transports as well, such as shared memory, pipes, files, etc.

The routine will check to see if the token specified by the transport is either readable (R_OK), writeable (W_OK), or executable (X_OK), or accessible (F_OK) any possible combination by or'ing the flags together.

F.2.2. kclearerr() — *clear the EOF/error flags of a data transport stream*

Synopsis

```
void kclearerr(
   kfile *file)
```

Input Arguments

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

Description

This function is a replacement for the system "clearerr" call; kclearerr() supports all khoros data transport mechanisms, not just Unix files.

The function resets the EOF and error flags for the data transport stream specified. Note that these flags are cleared originally when the data transport is opened, or when the data transport stream is re-wound using krewind(). The EOF flag is also cleared by calls to kfseek().

F.2.3. kclose() — *close and delete a transport descriptor*

Synopsis

```
int kclose(
   int id)
```

Input Arguments

`id`

the file to be close which was opened earlier with kopen().

Returns

-1 or 0 depending on whether the operation failed or succeeded.

Description

This function is a replacement for the system "close" call. The only difference is that kclose() supports more than just files, it supports other data transports as well, such as shared memory, pipes, files, etc.

F.2.4. kcreat() — *routine for creating a khoros transport*

Synopsis

```
int kcreat(
   const char *path,
   mode_t mode)
```

Input Arguments

`path`
: path to file to be created

`mode`
: the permissions or mode in which to create the file

Returns

open fid on success, -1 otherwise

Description

This function is used to create a khoros transport file. Depending on the different transport being used the behavior will be different. For a file the kcreat call simply create the file with the mode,

F.2.5. kdup() — *duplicate an existing khoros transport descriptor*

Synopsis

```
int kdup(
   int id)
```

Input Arguments

`id`
: the existing khoros transport descriptor to be dup'ed

Returns

the newly dup'ed descriptor on success, or -1 upon failure

Description

kdup() is used to duplicate an existing khoros transport descriptor. The "id" is integer index in the process's transport descriptor table. The new descriptor returned kdup() will be the lowest table entry id.

Side Effects

If too many descriptors are active then errno will be set to EMFILE and -1 will be returned. If id is not

a valid or active descriptor then errno will be set to EBADF and -1 will be returned.

F.2.6. kdup2() — *specifically duplicate an existing khoros transport descriptor*

Synopsis

```
int kdup2(
   int id1,
   int id2)
```

Input Arguments

`id1`
: the existing khoros transport descriptor to be dup'ed

`id2`
: the descriptor to be used for the newly dup'ed descriptor

Returns

the newly dup'ed descriptor on success, or -1 upon failure

Description

kdup2() is used to duplicate an existing khoros transport descriptor. The "id1" is integer index in the process's transport descriptor table. The new descriptor returned will the specific one specified by "id2". If id2 is active then it will be closed (kclose) before being dup'ed.

Side Effects

If too many descriptors are active then errno will be set to EMFILE and -1 will be returned. If id is not a valid or active descriptor then errno will be set to EBADF and -1 will be returned.

F.2.7. kexit() — *terminate a process*

Synopsis

```
void kexit(

   int status)
```

```
Oh what a trail of tears!  It turns out that Windows and UNIX handle exit-han-
dlers differently.  On UNIX, you get one set of handlers for a process.  I can
call atexit() from anywhere, and the handler is added to the single list.
```

They're all called when the process terminates. On Windows, EVERY .EXE and EVERY .DLL GETS ITS OWN SET OF EXIT HANDLERS! Thus, calling atexit() here puts a handler on the kutils.dll list. This would work okay, except for 2 things:

1) The exit handlers for each DLL are called as that DLL is unloaded from the process. The DLLs are unloaded in LIFO order. The unload order is WinSock2, then ktutils, then klibc. Since the exit handler has to clean up IPC and TCP/IP, it needs to be invoked BEFORE the WinSock2 DLL is unloaded. 2) The most bizarre thing is that the exit handler for the .EXE is only called when the main() routine exits, or if the exit() routine is called from somewhere inside the .EXE. If it's called from within a DLL (as happens when the CLUI processing objects to the command line), then the .EXE exit handler is NEVER CALLED AT ALL!

So... dear reader, here's what we've done.

1) The call to atexit() now takes place in the main() routine of each kroutine and codegen - it doesn't happen in exit.c. It still sets up the exit_handler() function (also in exit.c) as the actual exit handler, but it puts it on the .EXE's handler list. 2) The "kexit" function (which is #defined as "exit()" on UNIX), is defined as a call through a global function-pointer named pkexit that's defined in kprocess.c. That function-pointer is set in khoros_init().

Input Arguments

`status`

exit status of either KEXIT_SUCCESS or KEXIT_FAILURE

Description

closes any VisiQuest transports left open and exits the system with the status passed in.

F.2.8. kfileno() — *return the transport descriptor*

Synopsis

```
int kfileno(
   kfile *file)
```

Input Arguments

`file`

the kfile structure to be referenced.

Returns

The entry descriptor (or -1 if it doesn't exist)

Description

This function is used to retrieve the corresponding kfile entry descriptor, which can be used with the (int fid) routines like: kread(), kwrite().

F.2.9. kgetbuffer() — *get the current data transport stream's input/output buffer and it's size*

Synopsis

```
char *kgetbuffer(
   kfile *file,
   size_t * bufsiz)
```

Input Arguments

`file`

the kfile transport to be written which was opened earlier with kfopen().

Output Arguments

`bufsiz`

return the size of the new buffer if not NULL

Returns

The externally inialized buf or NULL if the the transport is non-buffered or if the buffer is the internal buffer.

Description

kgetbuffer(), the opposite from ksetvbuf(), can be used after a stream buffer has been initialized. It returns the initialized stream buffer. The character array buf whose size is returned as the bufsiz argument, unless NULL.

F.2.10. kgetc() — *get character from khoros transport*

Synopsis

```
int kgetc(

  kfile *file)
```

Input Arguments

`file`

the kfile transport to be read which was opened earlier with kfopen().

Returns

returns the character read or EOF upon error.

Description

This function is a replacement for the system getc() routine. The only difference is that kgetc() supports all available transport mechanisms. The routine will get a single character from the specified transport. If the character could not be read, EOF is returned; otherwise, the character that is read is returned.

F.2.11. kgetdescriptors() — *get true UNIX file descriptors*

Synopsis

```
int kgetdescriptors(
   kfile *file,
   int *inum,
   int *onum)
```

Input Arguments

`file`

the kfile structure to be referenced.

Output Arguments

`inum`

the input descriptor if exists (or if non exists minus 1)

`onum`

the output descriptor if exists (or if non exists minus 1)

Returns

TRUE on success, FALSE on failure

Description

This function is used to retrieve the input and output file descriptor associated with a transport. Note: this routine should not be used without extreme caution, since certain transports don't have file descriptors (such as shared memory).

F.2.12. kgethostname() — *get the current hostname*

Synopsis

```
#ifdef KOPSYS_WIN32
int kgethostname(
   char *hostname,
   int namelen)
```

Input Arguments

`namelen`
the size of the "hostname" buffer

Output Arguments

`hostname`
the current hostname string

Returns

0 on success, and -1 on failure

Description

This routine is the same as the system gethostname() call except that it will get the hostname depending on the machine name is specified.

F.2.13. kgets() — *reads from kstdin until a newline or EOF*

Synopsis

```
char *kgets(

  char *buffer)
```

Input Arguments

`buffer`
the buffer to read the data into.

Returns

returns the string read from kstdin or NULL upon error.

Description

This function is a replacement for the system gets() routine. The only difference is that kgets() uses the khoros transport mechanisms. The routine will get a single line from kstdin. It reads until a newline or end of file is encountered.

F.2.14. kinput() — *opens a file for reading using kopen()*

Synopsis

```
int kinput(

  char *filename)
```

Input Arguments

`filename`
: the filename to be opened for reading using the kopen().

Returns

returns the kfile id on success, -1 otherwise

Description

This function is just a simplified interface to kopen() that opens a file for reading. The macros calls kopen using the following syntax:

kopen(filename, KOPEN_RDONLY, 0666)

F.2.15. klseek() — *move read/write pointer of a transport descriptor*

Synopsis

```
off_t klseek(
   int id,
   off_t offset,
   int whence)
```

Input Arguments

`id`
: the id of the object to be seeked.

`offset`
: the offset in which to seek

`whence`
: the control of how the offset will be applied

Returns

-1 or the new seeked position within the transport

Description

This function is used to seek the transport. Depending on the different transport being used the behavior will be different. For a file it calls lseek() to lock the file, but for a shared memory segment it it simply resets the internal offset pointer.

F.2.16. kopen() — *open or create a file for reading and/or writing*

Synopsis

```
int kopen(

  const char *path,
  int         flags,
  mode_t      mode)
```

Input Arguments

`path`

is the string containing the path name to the desired file to be open. The path name identifies the file to be opened, just like the regular UNIX open() command.

`flags`

how the file is to be opened.

```
KOPEN_RDONLY  - Open file for only reading        NOTE!! this is 0x0000, not a
KOPEN_WRONLY  - Open file for only writing
KOPEN_RDWR    - Open file for both reading & writing

  KOPEN_NONBLOCK - Whether to block when reading and
                   writing to the file.
  KOPEN_APPEND   - Append to the file (writing).
  KOPEN_TRUNC    - Truncate the file for writing.
  KOPEN_EXCL     - Error if file exists
  KOPEN_CREAT    - the file exists, this flag has no
                    effect.
  Otherwise, the file is created, and the
  owner ID of the file is set to the effective
  user ID of the process.
```

`mode`

the permissions to be used when creating a new file. Note that this parameter is only needed when KOPEN_CREAT is specified in the flags.

Returns

the kfile id on success, -1 otherwise

Description

This function is a replacement for the system "open" call. The only difference is that kopen() supports more than just files, it supports other data transports as well. The path should be in the form of an identifier, followed by an "=" equal indicator, and then the transport token. ie) a Shared memory path would look like:

"shm=1032"

If a file was desired then either

"/usr/tmp/vadd1235"
"file=/usr/tmp/vadd1235"

will work.

F.2.17. koutput() — *opens/creates a file for writing using kopen()*

Synopsis

```
int koutput(

  char *filename)
```

Input Arguments

`filename`

the filename to be opened for writing using the kopen().

Returns

returns the kfile id on success, -1 otherwise

Description

This function is just a simplified interface to kopen() that creates and opens a file for writing. The file is truncated upon a successful kopen call. The macros calls kopen using the following syntax:

kopen(filename, KOPEN_WRONLY|KOPEN_CREAT|KOPEN_TRUNC, 0666)

F.2.18. kpclose() — *close a pipe (for I/O) from or to a process*

Synopsis

```
int kpclose(
   kfile *file)
```

Input Arguments

`file`
tranport pointer to close

Returns

0 on success, -1 on failure

Description

Close a pipe (for I/O) from or to a process

F.2.19. kpopen() — *open a pipe (for I/O) from or to a process*

Synopsis

```
kfile *kpopen(
   const char *command,
   const char *type)
```

Input Arguments

`command`
command to send pipe output to or from

`type`
open status either 'r' or 'w'

Returns

The open file transport pointer

Description

open a pipe (for I/O) from or to a process

F.2.20. kpinfo() — *gets the associated process id*

Synopsis

```
pid_t kpinfo(
   kfile *file)
```

Input Arguments

`file`
: tranport pointer to close

Returns

process id on success, -1 on failure

Description

Gets the pipe process id

KOPSYS_WIN32 Note: the "pid" returned by this function is actually a Windows Process HANDLE. You can use it to call _cwait.

F.2.21. kprintf() — *print formatted output to kstdout*

Synopsis

```
int kprintf(
   const char *format,
   kvalist)
```

Input Arguments

`format`
: the format in which to print the values

`kvalist`
: variable number of values to format and write to kstdout. The format string determines the data type of the value(s) to be provided.

Returns

The number of characters written to stdout

Description

This function is a replacement for the system "printf" call; however, kprintf() uses data transport mechanisms to print the text, in order to support distributed computing.

The kprintf() function converts, formats, and writes its value parameters as specified by the format parameter to kstdout. If there are insufficient values for the format, the behavior is undefined; if the format is exhausted while values remain, the excess values are ignored.

F.2.22. kputc() — *put a character onto the khoros transport*

Synopsis

```
int kputc(

  int   character,
  kfile *file)
```

Input Arguments

`character`
the character to be written

`file`
the kfile transport to be read which was opened earlier with kfopen().

Returns

returns the character written or EOF upon error.

Description

This function is a replacement for the system "fputc()" routine. The only difference is that kfputc() supports more than just files, it supports other data transports such as shared memory, pipes, files, etc.

The routine will put a single character onto the specified transport. If the character could not be written, EOF is returned; otherwise, the character that is written is returned.

F.2.23. krename() — *rename a khoros transport from path1 to path2*

Synopsis

```
int krename(
   const char *oldname,
   const char *newname)
```

Input Arguments

`oldname`
the old khoros transport name

`newname`
the new khoros transport name

Returns

0 on success, -1 on failure and sets errno to indicate the error

Description

krename() is used to move the contents from one file to another. If the two files are standard UNIX files then krename() simply use the system rename(), but if this fails then the khoros transport mechanisms are used to copy the data from the first file to the second, and then the first is unlinked using kunlink().

F.2.24. krewind() — *rewind a data transport stream to the beginning*

Synopsis

```
void krewind(
   kfile *file)
```

Input Arguments

`file`
the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

Description

This function is a replacement for the system "rewind" call; krewind() supports all khoros data transport mechanisms, not just Unix files.

This function is used to re-position the data transport pointer to the beginning of the stream. It is equivalent to:

(void) kfseek(file, 0L, 0);

The behavior of the function will differ depending on the data transport being used. For a file, it calls fseek() to reposition the file pointer; for shared memory segments, it simply resets the internal offset pointer.

F.2.25. kscanf() — *scan kstdin and format the input into one or more arguments of the type specified*

Synopsis

```
int kscanf(
   const char *format,
   kvalist)
```

Input Arguments

`format`

the format in which to interpret the input

Output Arguments

`kvalist`

variable number of arguments into which to format the input coming from kstdin. The format string determines the type of pointer(s) to be provided as the argument(s). Note that each argument must be a pointer, and that sufficient space to accomodate the output must be provided for strings.

Returns

The number of arguments successfully scanned into the input argument(s).

Description

This function is a replacement for the system "scanf" call; however, kscanf() uses data transport mechanisms to scan the input, in order to support distributed computing.

The kscanf() function reads characters from kstdin, interprets them according to the format specified, and stores the result(s) in the input argument(s) specified by a variable argument list. In addition, the kscanf() routine guarantees that the integer returned will be the number of arguments correctly scanned, and that scanning will never continue past the first error. If there are insufficient arguments for the format, the behavior is undefined; if the format is exhausted while arguments remain, the excess arguments are ignored.

F.2.26. ksetvbuf() — *set the I/O buffer of a data transport stream*

Synopsis

```
void ksetvbuf(
   kfile *file,
   char *buf,
   int mode,
   size_t bufsiz)
```

Input Arguments

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

`buf`

the new I/O buffer or NULL if no buffer is desired

`mode`

mode that the buffer runs in

`bufsiz`

the size of the new buffer

Description

ksetvbuf(), an alternate form of setvbuf(), can be used after a stream has been opened but before it is read or written. It uses the character array buf whose size is specified by the bufsiz argument instead of an automatically fixed size of BUFSIZ. If buf is NULL, then no buffer is is used.

F.2.27. ksprintf() — *print one or more arguments in the format specified to an output string.*

Synopsis

```
int ksprintf(
   char *string,
   const char *format,
   kvalist)
```

Input Arguments

`string`

the output string in which to format the values

`format`

the format in which to print the values

`kvalist`

variable number of values to format and write to the output file string. The format string determines the data type of the value(s) to be provided.

Returns

The number characters written to the output string.

Description

This function is a replacement for the system "sprintf" call; however, ksprint() uses data transport mechanisms to print the output string, in order to support distributed computing.

The ksprintf() function converts, formats, and writes its value parameters as specified by the format parameter to the output string given. If there are insufficient values for the format, the behavior is undefined; if the format is exhausted while values remain, the excess values are ignored.

F.2.28. kmsprintf() — *print one or more arguments in the format specified and return an allocated output string.*

Synopsis

```
char *kmsprintf(
   const char *format,
   kvalist)
```

Input Arguments

`format`
: the format in which to print the values

`kvalist`
: variable number of values to format and write to the output file string. The format string determines the data type of the value(s) to be provided.

Returns

The string on success, NULL on failure

Description

This function is similar to "ksprintf" call; however, kmsprint() returns an allocated string instead of expecting pre-allocated string.

The kmsprintf() function converts, formats, and writes its value parameters as specified by the format parameter to a dynamically allocated string. There is no internal limits to the size of string other than the amount of system memory allowed via malloc(). If there are insufficient values for the format, the behavior is undefined; if the format is exhausted while values remain, the excess values are ignored.

F.2.29. ksscanf() — *scan input string and format it into one or more arguments of the type specified.*

Synopsis

```
int ksscanf(
   const char *string,
   const char *format,
   kvalist)
```

Input Arguments

`string`
: the input string to format

`format`
: the format in which to interpret the input

Output Arguments

`kvalist`

variable number of arguments into which to format the input string. The format string determines the type of pointer(s) to be provided as the argument(s). Note that each argument must be a pointer, and that sufficient space to accomodate the output must be provided for strings.

Returns

The number of fields successfully parsed from the input argument(s).

Description

This function is a replacement for the system "sscanf" call; however, ksscanf() uses data transport mechanisms to scan the input, in order to support distributed computing.

The ksscanf() function reads character data input, interprets them according to the format specified, and stores the result in the input arguments(s) specified by the variable argument list. In addition, the ksscanf() routine guarantees that the integer returned will be the number of fields correctly scanned, and that scanning will never continue past the first error. If there are insufficient arguments for the format, the behavior is undefined; if the format is exhausted while arguments remain, the excess arguments are ignored.

F.2.30. ktell() — *report the position of the read/write pointer*

Synopsis

```
long int ktell(
   int id)
```

Input Arguments

`id`

the id of the object to be told.

Returns

-1 or the current offset depending on whether the operation failed or succeeded.

Description

This function is used to tell the current position within the transport. Depending on the different transport being used the behavior will be different. For a file it calls tell() to locate the offset within the file, but for a shared memory segment it it simply indicates the internal offset.

F.2.31. ktouch() — *routine for touching a temporary transport*

Synopsis

```
int ktouch(
   const char *path,
   mode_t mode)
```

Input Arguments

`path`
: path to file to be created

`mode`
: the permissions or mode in which to touch the file

Returns

0 on success, -1 otherwise

Description

This function is used to create a khoros transport file. Depending on the different transport being used the behavior will be different. For a file the ktouch call simply open the file and then close it. This is analgous to the "touch" system call.

F.2.32. ktmpfile() — *create a temporary data transport stream*

Synopsis

```
kfile *ktmpfile(void)
```

Returns

The kfile pointer representing the open stream to the temporary transport on success, NULL on failure

Description

This function is a replacement for the UNIX system call "ktmpfile"; ktmpfile() supports all khoros data transports, not just unix files. The data transport will automatically be deleted when closed.

F.2.33. kungetc() — *push a character back onto the data transport stream*

Synopsis

```
int kungetc(
   int character,
   kfile *file)
```

Input Arguments

`character`
: the character to be put back onto the stream

`file`
: the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen().

Returns

The character that was put back on success, EOF if an error is encountered.

Description

This function is a replacement for the system ungetc() routine; kungetc() supports all khoros data transport mechanisms, not just Unix files.

The routine will put a single character back onto the specified data transport stream, and moves the data transport pointer back one character. If the character could not be put back, EOF is returned; otherwise, the character that is put back is returned.

F.2.34. kunlink() — *remove a filename from a directory entry*

Synopsis

```
int kunlink(
   const char *path)
```

Input Arguments

`path`

the path to the object ot be unlinked.

Returns

-1 or 0 depending on whether the operation failed or succeeded.

Description

This function is used to unlink the kfile. Depending on the different transport being used the behavior will be different. For a file it unlinks the file, but for a shared memory segment it initializes it before using the shmctl() to destroy it.

F.3. Introduction to Variable Argument I/O Functions

These routines print and scan variable argument lists.

- *kvfprintf()* - print formatted kfile output of variable arguments
- *kvfscanf()* - scan formatted kfile input of variable arguments
- *kvprintf()* - print formatted kstdout output of variable arguments
- *kvscanf()* - scan formatted kstdin input of variable arguments
- *kvsprintf()* - print formatted string output of variable arguments list
- *kvsscanf()* - scan formatted string of a variable argument list

F.4. Definitions of Variable Argument I/O Functions

F.4.1. kvfprintf() — *print formatted kfile output of variable arguments*

Synopsis

```
int kvfprintf(
   kfile *file,
   const char *format,
   kva_list args)
```

Input Arguments

`file`
: kfile structure

`format`
: the format in which to the arguments will be

`args`
: an argument list which are used as the the inputs to the format command.

Returns

The number of characters written to kfile transport.

Description

This function is a replacement for the system "vfprintf" call. The only difference is that kvfprintf() uses the transport routines to print the text. The text will be printed to the kfile transport.

F.4.2. kvfscanf() — *scan formatted kfile input of variable arguments*

Synopsis

```
int kvfscanf(
   kfile *file,
   const char *format,
   kva_list args)
```

Input Arguments

`file`
: kfile structure

`format`
: the format in which to the arguments will be

Output Arguments

`args`

an argument list which are used as the the inputs to the format command.

Returns

number of matched arguments found in the kfile transport

Description

This function is a replacement for the system "vfscanf" call. The only difference is that kvfscanf() uses the transport routines to scan the text. The text will be scanned from the kfile transport.

F.4.3. kvprintf() — *print formatted kstdout output of variable arguments*

Synopsis

```
int kvprintf(
   const char *format,
   kva_list args)
```

Input Arguments

`format`

the format in which to the arguments will be

`args`

an argument list which are used as the the inputs to the format command.

Returns

The number of characters written to kstdout.

Description

This function is a replacement for the system "vprintf" call. The only difference is that kvprintf() uses the transport routines to print the string. The text will be printed to kstdout.

F.4.4. kvscanf() — *scan formatted kstdin input of variable arguments*

Synopsis

```
int kvscanf(
   const char *format,
   kva_list args)
```

Input Arguments

`format`
the format in which to the arguments will be

Output Arguments

`args`
an argument list which are used as the the inputs to the format command.

Returns

number of match arguments found from kstdin

Description

This function is a replacement for the system "vscanf" call. The only difference is that kvscanf() uses the transport routines to scan the string. The text will be scanned in from kstdin.

F.4.5. kvsprintf() — *print formatted string output of variable arguments list*

Synopsis

```
int kvsprintf(
   char *string,
   const char *format,
   kva_list args)
```

Input Arguments

`string`
the output string to format the arguments into
`format`
the format in which to the arguments will be
`args`
variable argument list which are used as the the inputs to the format command.

Returns

The number of characters written to the string

Description

This function is a replacement for the system "vsprintf" call. The only difference is that kvsprintf() uses the transport routines printing routines. The text will be printed to the supplied string.

F.4.6. kvsscanf() — *scan formatted string of a variable argument list*

Synopsis

```
int kvsscanf(
   const char *string,
   const char *format,
   kva_list args)
```

Input Arguments

`string`

the output string to format the arguments into

`format`

the format in which to the arguments will be

Output Arguments

`args`

variable argument list which are used as the the inputs to the format command.

Returns

number of match arguments found in the supplied string

Description

This function is a replacement for the system "vsscanf" call. The only difference is that kvsscanf() uses the transport routines scanning routines. The text will be scanned to the supplied string.

F.5. Introduction to Standard File I/O Functions

These functions cover the *buffered* opening, closing, and locking of a file. Also included are functions to obtain various statistics of the file. These routines include:

- *kfopen()* - open a data transport stream
- *kfclose()* - close a data transport stream
- *kfdopen()* - open an existing transport descriptor as a data transport stream
- *kfeof()* - check if a data transport stream is at EOF
- *kfflush()* - flush buffered output of a data transport stream
- *kflock()* - apply or remove an advisory lock on an open transport descriptor
- *kfreopen()* - re-open a data transport stream

• *kfseek()* - set position in a data transport stream
• *kftell()* - report current position in the data transport stream

F.6. Definitions of Standard File I/O Functions

F.6.1. kfopen() — *open a data transport stream*

Synopsis

```
kfile *kfopen(
   const char *path,
   const char *type)
```

Input Arguments

`path`
the path specifying the data transport to be opened; see above.

`type`
how the data transport is to be opened.

"r" - Open file for only reading
"w" - Open file for only writing
"a" - Append file for only writing
"A" - Append file for writing; non over-write
"r+" - Open file for both reading & writing
"w+" - Open file for both reading & writing
"a+" - Append file for only writing
"A+" - Append file for writing; non over-write

Returns

The kfile pointer representing the open stream to the data transport on success, NULL on failure

Description

This function is a replacement for the system "fopen" call; kfopen() supports the opening of any khoros data transport mechanism.

Identifiers for VisiQuest data transport mechanisms include the following:

'file' (standard Unix file; local transport / permanent storage)
'pipe' (standard pipes; local transport / no permanent storage)
'mmap' (virtual memory; local transport / permanent storage)
'shm' (shared memory; local transport / permanent storage)
'socket' (sockets; local or remote transport / no permanent storage)
'stream' (named streams/pipes; local transport / no permanent storage)

The token is the type of identifier for that transport. For a file, it is simply the filename. For shared memory, it is the shared memory key. For a pipe, it is the input & output file descriptors, as in "pipe=[3,4]". For a socket, it is the number of the socket, as in "socket=5430".

Together, the identifier and the token identify the path to the data transport to be opened. The path must be in the form of an identifier and a token, separated by an "=" equal indicator, as in:

"{identifier}={transport token}"

For example, the path parameter for opening shared memory might be:

"shm=1032"

The path parameter for opening a file might be:

"file=/usr/tmp/vadd1235"

When opening a file as the data transport mechanism, the identifier and the "=" sign may be omitted, as in:

"./foobar"

F.6.2. kfclose() — *close a data transport stream*

Synopsis

```
int kfclose(
   kfile *file)
```

Input Arguments

`file`

the kfile pointer representing the open stream to a khoros data transport mechanism that was opened earlier using kfopen().

Returns

0 on success, -1 on failure.

Description

This function is a replacement for the system "fclose" call; kfclose() supports all khoros data transport mechanisms, not just Unix files.

F.6.3. kfdopen() — *open an existing transport descriptor as a data transport stream*

Synopsis

```
kfile *kfdopen(
   int id,
   const char *type)
```

Input Arguments

`id`

the transport identifier opened earlier

`type`

how the data transport is to be opened.

"r" - Open file for only reading
"w" - Open file for only writing
"a" - Append file for only writing
"A" - Append file for writing; non over-write
"r+" - Open file for both reading & writing
"w+" - Open file for both reading & writing
"a+" - Append file for only writing
"A+" - Append file for writing; non over-write

Returns

The kfile pointer representing the open stream to the data transport on success, NULL on failure

Description

This function is a replacement for the system "fdopen" call; kfdopen() supports all data transport mechanisms, not just unix files.

The kfdopen() function associates a data transport stream with an existing transport descriptor previously obtained from a call to kopen(), kdup(), kdup2(), or kpipe(). These functions open data transports, but do not return pointers to kfile structures; many of the khoros I/O functions require pointers to the kfile structure. Note that the type of khoros data transport must agree with the mode of the open data transport.

F.6.4. kfeof() — *check if a data transport stream is at EOF*

Synopsis

```
int kfeof(
   kfile *file)
```

Input Arguments

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

Returns

A non-zero value if EOF has been reached on the specified data transport stream; otherwise, zero is returned.

Description

This function is a replacement for the system "feof" call; kfeof() supports all khoros data transport mechanisms, not just Unix files.

This routines test to see if the data transport pointer of a stream open for reading has reached the end of file.

F.6.5. kfflush() — *flush buffered output of a data transport stream*

Synopsis

```
int kfflush(
   kfile *file)
```

Input Arguments

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

Returns

0 on success, EOF if the file was not open for writing, or if a write error occurred.

Description

This function is a replacement for the system "fflush" call; kfflush() supports all khoros data transport mechanisms, not just Unix files.

This routine is used to flush any buffered output; any data in the buffer of the output stream is written to the data transport stream specified.

F.6.6. kflock() — *apply or remove an advisory lock on an open transport descriptor*

Synopsis

```
int kflock(
   int id,
   int operation)
```

Input Arguments

`id`

the id of the object to be flock.

`operation`

the mode in which kflock() should lock the file. The operation is comprised of an or'ing of the following flags:

KLOCK_SH - lock file shareable (when reading)
KLOCK_EX - lock file execlusive (when writing)
KLOCK_NB - lock file for no block (don't block)
KLOCK_UN - unlock the file (free previous lock)

Returns

0 on success, -1 on failure

Description

This function is used to lock the transport. Depending on the different transport being used the behavior will be different. For a file it calls flock() to lock the file, but for a shared memory segment it it calls shmctl() to lock & unlock the shared memory segment.

F.6.7. kfreopen() — *re-open a data transport stream*

Synopsis

```
kfile *kfreopen(
   const char *path,
   const char *type,
   kfile *file)
```

Input Arguments

`path`

the path specifying the data transport to be re-opened; see above.

`type`
how the data transport is to be re-opened.

"r" - Open file for only reading
"w" - Open file for only writing
"a" - Append file for only writing
"A" - Append file for writing; non over-write
"r+" - Open file for both reading & writing
"w+" - Open file for both reading & writing
"a+" - Append file for only writing
"A+" - Append file for writing; non over-write

`file`
the existing data transport stream to be reopened for the specified path.

Returns

The kfile pointer representing the open stream to the re-opened data transport on success, NULL on failure

Description

This function is a replacement for the UNIX system call "freopen"; kfreopen() supports all khoros data transport mechanisms, not just unix files.

kfreopen() will open a specified file on for an existing stream. The existings stream is closed before the new filename is opened. This function is typically used to open a specified file as one of the predefined streams; such as standard input, standard output, or standard error.

The kfreopen() function substitutes the specified data transport in place of the open data transport stream. The original stream is closed regardless of whether the kfreopen() function succeeds with the new data transport. The kfreopen() function is typically used to attach the preopened data transport streams associated with kstdin, kstdout, and kstderr to other data transport mechanisms.

Identifiers for VisiQuest data transport mechanisms include the following:

'file' (standard Unix file; local transport / permanent storage)
'pipe' (standard pipes; local transport / no permanent storage)
'mmap' (virtual memory; local transport / permanent storage)
'shm' (shared memory; local transport / permanent storage)
'socket' (sockets; local or remote transport / no permanent storage)
'stream' (named streams/pipes; local transport / no permanent storage)

The token is the type of identifier for that transport. For a file, it is simply the filename. For shared memory, it is the shared memory key. For a pipe, it is the input & output file descriptors, as in "pipe=[3,4]". For a socket, it is the number of the socket, as in "socket=5430".

Together, the identifier and the token identify the path to the data transport to be opened. The path must be in the form of an identifier and a token, separated by an "=" equal indicator, as in:

"{identifier}={transport token}"

For example, the path parameter for opening shared memory might be:

"shm=1032"

The path parameter for opening a file might be:

"file=/usr/tmp/vadd1235"

When opening a file as the data transport mechanism, the identifier and the "=" sign may be omitted, as in:

"./foobar"

F.6.8. kfseek() — *set position in a data transport stream*

Synopsis

```
int kfseek(
   kfile *file,
   long int offset,
   int whence)
```

Input Arguments

`file`
: the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

`offset`
: offset specifying new position of file pointer

`whence`
: controls how the offset will be applied:

  0 - sets the pointer to the absolute value of the offset parameter
  1 - sets the pointer to its current location plus the value of the offset parameter
  2 - sets the pointer to the size of the file plus the value of the offset parameter

Returns

File position on success, -1 on failure

Description

This function is a replacement for the system "fseek" call; kfseek() supports all khoros data transport mechanisms, not just Unix files.

This function sets the position of the next input or output operation on the data transport stream by changing the location of the data transport pointer.

F.6.9. kftell() — *report current position in the data transport stream*

Synopsis

```
long int kftell(
   kfile *file)
```

Input Arguments

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

Returns

0 on success, -1 on failure

Description

This function is a replacement for the system "ftell" call; kftell() supports all khoros data transport mechanisms, not just Unix files.

This function is used to check the current position of the data transport pointer within the data transport stream.

F.7. Introduction to File Read/Write Utilities

These functions allow various methods of reading from and writing to a VisiQuest Data Transport stream.

- *kfdup()* - duplicate an existing data transport stream
- *kfdup2()* - duplicate an existing data transport into a specific stream
- *kfgetc()* - get a character from the data transport stream
- *kfgets()* - get a string from a data transport stream
- *kfinput()* - opens a file for reading using kfopen()
- *kfoutput()* - opens/creates a file for writing using kfopen()
- *kfprintf()* - print one or more arguments in the format specified to an output file stream
- *kfputc()* - put a character onto the data transport stream
- *kfputs()* - put a string onto the data transport stream

• *kfread()* - read from a data transport stream
• *kfscanf()* - scan file input and format it into one or more arguments of the type specified
• *kfwrite()* - write to a data transport stream
• *kputs()* - writes a string to kstdout

F.8. Definitions of File Read/Write Utilities

F.8.1. kfdup() — *duplicate an existing data transport stream*

Synopsis

```
kfile *kfdup(
   kfile *file)
```

Input Arguments

`file`
the kfile pointer associated with the open stream to the data transport to be duplicated, opened earlier with kfopen()

Returns

The newly duplicated stream on success, NULL upon failure

Description

Duplicates an existing data transport stream. The "file" is entry in the process's transport descriptor table. The new entry returned kfdup() will be the lowest table entry.

F.8.2. kfdup2() — *duplicate an existing data transport into a specific stream*

Synopsis

```
kfile *kfdup2(
   kfile *file1,
   kfile *file2)
```

Input Arguments

`file1`
the existing khoros transport stream to be dup'ed
`file2`
the stream to be used for the newly dup'ed stream

Returns

the newly dup'ed stream on success, or NULL upon failure

Description

kfdup2() is used to duplicate an existing khoros transport stream. The "file1" is an entry in the process's transport descriptor table. The newly dupedd entry will be stored in the transport table entry specified by "file2". If file2 is open then it will be closed (kfclose) before the dup of file1 is put into file2.

F.8.3. kfgetc() — *get a character from the data transport stream*

Synopsis

```
int kfgetc(
   kfile *file)
```

Input Arguments

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen().

Returns

The character read on success, EOF if an error is encountered.

Description

This function is a replacement for the system "fgetc()" routine; kfgetc() supports all khoros data transport mechanisms, not just Unix files.

The routine gets a single character from the data transport stream, and moves the data transport pointer ahead by one character. If the character could not be read, EOF is returned; otherwise, the character that is read is returned.

F.8.4. kfgets() — *get a string from a data transport stream*

Synopsis

```
char *kfgets(
   char *buffer,
   int num,
   kfile *file)
```

Input Arguments

`num`

maximum number of characters to read (the size of the character array).

`file`

the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen()

Output Arguments

`buffer`

pointer to allocated character array into which to read the string.

Returns

Returns the string read on success, NULL if EOF is encountered before any characters have been read, or if an error occurs.

Description

This function is a replacement for the system "fgets" call; kfgets() supports all khoros data transport mechanisms, not just Unix files.

The routine reads a string of characters from the specified data transport stream, and moves the data transport pointer ahead by that number of characters. kfgets() reads characters from the stream into the array pointed to by 'buffer' until 'num'-1 characters are read, or until a '\n' (newline character) is read (and added to the string), or until EOF is encountered. The string is terminated with a NULL character. If the string could not be read, NULL is returned; otherwise, the string is returned.

F.8.5. kfinput() — *opens a file for reading using kfopen()*

Synopsis

```
kfile *kfinput(

  char *filename)
```

Input Arguments

`filename`

the filename to be opened for reading using the kfopen().

Returns

returns the kfile structure on success, NULL upon failure

Description

This function is just a simplified interface to kfopen() that opens a file for reading. The macros calls kfopen using the following syntax:

kfopen(filename, "r")

F.8.6. kfoutput() — *opens/creates a file for writing using kfopen()*

Synopsis

```
Declaration: kfile *kfoutput(

  char *filename)
```

Input Arguments

`filename`
the filename to be opened for writing using the kfopen().

Returns

returns the kfile structure on success, NULL upon failure

Description

This function is just a simplified interface to kfopen() that creates and opens a file for writing. The file is truncated upon a successful kfopen call. The macros calls kfopen using the following syntax:

kfopen(filename, "w")

F.8.7. kfprintf() — *print one or more arguments in the format specified to an output file stream*

Synopsis

```
int kfprintf(
   kfile *file,
   const char *format,
   kvalist)
```

Input Arguments

`file`
stream to open output file

`format`
the format in which to print the values

`kvalist`
variable number of values to format and write to the output file stream. The format string determines

the data type of the value(s) to be provided.

Returns

The number of characters written to the output file.

Description

This function is a replacement for the system "fprintf" call; however, kfprint() uses data transport mechanisms to print the output, in order to support distributed computing.

The kfprintf() function converts, formats, and writes its value parameters as specified by the format parameter to the output stream given. If there are insufficient values for the format, the behavior is undefined; if the format is exhausted while values remain, the excess values are ignored.

F.8.8. kfputc() — *put a character onto the data transport stream*

Synopsis

```
int kfputc(
   int character,
   kfile *file)
```

Input Arguments

`character`

the character to be written

`file`

the kfile pointer associated with the open stream to the data transport

Returns

The character written on success; EOF on failure

Description

This function is a replacement for the system "fputc" call; kfputc() supports all khoros data transport mechanisms, not just Unix files.

The routine will write a single character to the data transport stream. If the character could not be written, EOF is returned; otherwise, the character that is written is returned.

F.8.9. kfputs() — *put a string onto the data transport stream*

Synopsis

```
int kfputs(
   const char *buffer,
   kfile *file)
```

Input Arguments

`buffer`
: the string to be written

`file`
: the kfile pointer associated with the open stream to the data transport, opened earlier with kfopen().

Returns

The number of characters written on success, EOF if an error is encountered.

Description

This function is a replacement for the system "fputs" call; kfputs() supports all khoros data transport mechanisms, not just Unix files.

The routine will write a NULL-terminated string to the data transport output stream. If the string could not be written, EOF is returned; otherwise, the number of characters that were written are returned.

F.8.10. kfread() — *read from a data transport stream*

Synopsis

```
size_t kfread(
   kaddr ptr,
   size_t size,
   size_t nitems,
   kfile *file)
```

Input Arguments

`ptr`
: a pointer to allocated space into which to read the data

`size`
: the size, in bytes, of each item

`nitems`
: the number of items to be read

`file`

the data transport transport to be read from; must have been opened earlier using kfopen().

Returns

The number of items read on success; 0 when end-of-file is encountered or an error occurs.

Description

This function is a replacement for the system "fread" call; kfread() supports all khoros data transport mechanisms, not just Unix files.

The routine will read 'nitem' items, each of 'size' bytes from the data transport stream associated with 'file' into the memory location accessed by 'ptr'.

F.8.11. kfscanf() — *scan file input and format it into one or more arguments of the type specified*

Synopsis

```
int kfscanf(
   kfile *file,
   const char *format,
   kvalist)
```

Input Arguments

`file`
stream to open input file

`format`
the format in which to interpret the input

Output Arguments

`kvalist`
variable number of arguments into which to format the input from the file stream. The format string determines the type of pointer(s) to be provided as the argument(s). Note that each argument must be a pointer, and that sufficient space to accomodate the output must be provided for strings.

Returns

The number of arguments successfully scanned into the input argument(s).

Description

This function is a replacement for the system "fscanf" call; however, kfscanf() uses data transport mechanisms to scan the input, in order to support distributed computing.

The kfscanf() function reads characters from an input transport, interprets them according to the format specified, and stores the result in the input argument(s) specified by the variable argument list. In addition, the kfscanf() routine guarantees that the integer returned will be the number of fields correctly scanned, and that scanning will never continue past the first error. If there are insufficient arguments

for the format, the behavior is undefined; if the format is exhausted while arguments remain, the excess arguments are ignored.

F.8.12. kfwrite() — *write to a data transport stream*

Synopsis

```
size_t kfwrite(
   kaddr ptr,
   size_t size,
   size_t nitems,
   kfile *file)
```

Input Arguments

`ptr`
: a pointer to allocated space from which to write the data

`size`
: the size, in bytes, of each item

`nitems`
: the number of items to be written

`file`
: the data transport transport to be written to; must have been opened earlier using kfopen().

Returns

The number of items written on success, 0 if an error is encountered.

Description

This function is a replacement for the system "fwrite" call; kwrite() supports all khoros data transport mechanisms, not just Unix files.

The routine will write ’nitem’ items, each of ’size’ bytes to the data transport stream associated with ’file’ from the memory location accessed by ’ptr’.

F.8.13. kputs() — *writes a string to kstdout*

Synopsis

```
int kputs(
   const char *buffer)
```

Input Arguments

`buffer`
the buffer to write to kstdout.

Returns

returns the characters written; otherwise EOF is returned.

Description

This function is a replacement for the system puts() routine. The only difference is that kputs() uses the khoros transport mechanisms. The routine will print a single line to kstdout. It uses the kstrlen() to determine the number of characters to print.

F.9. Introduction to Data Transport Utilities

The previous utilities were generalized front ends that opened, closed, read from and wrote to the VisiQuest data transport currently being used. These utilities perform various functions with respect to the current VisiQuest Data Transport itself.

- *kfile_clrstate()* - remove a flag from an open transport id state
- *kfile_comparedata()* - compare the contents of a transport id to another transport id
- *kfile_copydata()* - copy the contents of a transport id to another transport id
- *kfile_filename()* - return the filename associated with khoros transport id
- *kfile_flags()* - return the flags associated with transport id
- *kfile_getmachtype()* - gets the machine architecture type for a file transport id
- *kfile_getpermanence()* - returns whether a file has data permanence or not
- *kfile_isdup()* - khoros transport has been has been/is a dupped transport
- *kfile_iseof()* - khoros transport is at end of file (eof)
- *kfile_ismybuf()* - khoros transport buffer was set by the application
- *kfile_isopen()* - khoros transport has been properly opened
- *kfile_islock()* - khoros transport has been/is a locked transport
- *kfile_isreacquire()* - khoros transport will be reacquired
- *kfile_ispermanent()* - khoros transport has permanence
- *kfile_isbufferd()* - khoros transport is buffered or not
- *kfile_isstreambuf()* - khoros transport is stream buffered
- *kfile_islinebuf()* - khoros transport is line buffered
- *kfile_isfullbuf()* - khoros transport is full buffered
- *kfile_ismembuf()* - khoros transport is memory buffered

• *kfile_isread()* - khoros transport is readable
• *kfile_isrdwr()* - khoros transport is both readable and writeable
• *kfile_istemp()* - khoros transport is temporary
• *kfile_iswrite()* - khoros transport is writeable
• *kfile_mode()* - return the mode associated with transport id
• *kfile_readdata()* - read the contents of a khoros transport into a data array
• *kfile_remotehost()* - returns whether a host is remote
• *kfile_reopen()* - re-open a stream khoros transport
• *kfile_seddata()* - string edit from one transport to another
• *kfile_setmachtype()* - sets the machine architecture type for a file transport id
• *kfile_setstate()* - adds a flag to the open transport id state
• *kfile_getstate()* - return the current internal stream transport state
• *kfile_type()* - return the type flag field used when opening the transport with kfopen()
• *kfile_writedata()* - write the contents of the data to a khoros transport
• *kfidfile()* - return the kfile structure associated with a fid
• *ktmpfid()* - create a temporary transport descriptor
• *ktransport_add()* - add a new transport to the list of transports
• *ktransport_delete()* - delete a transport from the list of all transports
• *ktransport_list()* - get the list of supported transports

F.10. Definitions of Data Transport Utilities

F.10.1. kfile_clrstate() — *remove a flag from an open transport id state*

Synopsis

```
unsigned long kfile_clrstate(
   kfile *file,
   unsigned long flag)
```

Input Arguments

`file`
: the transport id from which to clear the flag for

`flag`
: the flag to be clear (such as KFILE_TEMP)

Returns

The updated flags associated with the id or -1 upon failure

Description

This routine is used to clear a flag within the transport id state field.

F.10.2. kfile_comparedata() — *compare the contents of a transport id to another transport id*

Synopsis

```
int kfile_comparedata(
   int id1,
   int id2,
   size_t num,
   size_t * num_compared)
```

Input Arguments

`id1`
: the first khoros transport descriptor

`id2`
: the second khoros transport descriptor

`num`
: if not 0 the number of bytes to be compared.

Output Arguments

`num_compared`
: the number of bytes actually compared.

Returns

returns an integer less than, equal to, or greater than 0, according as id1 is lexicographically less than, equal to, or greater than id2.

Description

This routine compare the contents of a input transport descriptor to an output transport descriptor. These descriptors are specified by the VisiQuest transport mechanism. An optional "num_compare" can be used to specify the actual number of bytes to be compared. If "num_compare" is 0 then all the data from the two transports are compared.

Side Effects

data is read from id1 and id2, which means if the transports do not support data permanence seeking maynot work.

F.10.3. kfile_copydata() — *copy the contents of a transport id to another transport id*

Synopsis

```
ssize_t kfile_copydata(
   int id1,
   int id2,
   size_t num)
```

Input Arguments

`id1`
: the input khoros transport descriptor

`id2`
: the output khoros transport descriptor

`num`
: if not 0 the number of bytes to be copied.

Returns

the number of bytes copied or -1 upon failure

Description

This routine copies the contents of a input transport descriptor to an output transport descriptor. These descriptors are specified by the VisiQuest transport mechanism. An optional "num_copy" can be used to specify the actual number of bytes to be copied. If "num_copy" is 0 then all the data from input is written to the output.

Side Effects

the data written to the output descriptor will not be flushed until a kfflush() or kclose() is performed.

F.10.4. kfile_filename() — *return the filename associated with khoros transport id*

Synopsis

```
char * kfile_filename(
   int id)
```

Input Arguments

`id`
: the transport id from which the filename is returned

Returns

The filename associated with the id or NULL upon failure

Description

This routine returns the filename associated with a khoros transport descriptor id. This is the filename used in originally creating the transport.

Side Effects

the filename returned is the internal copy of the kfile transport. This means that you should not modify or free the string.

F.10.5. kfile_flags() — *return the flags associated with transport id*

Synopsis

```
unsigned long kfile_flags(
   int id)
```

Input Arguments

`id`

the transport id from which the flags is returned

Returns

The flags associated with the id or -1 upon failure

Description

This routine returns the open flags associated with a khoros transport descriptor id. This is the flags used in originally creating/opening of the transport.

F.10.6. kfile_getmachtype() — *gets the machine architecture type for a file transport id*

Synopsis

```
int kfile_getmachtype(
   int id)
```

Input Arguments

`id`

the file id to be retrieve the machtype, which was opened earlier with kopen().

Returns

the machine type on success, -1 otherwise

Description

This function returns the machine architecture type for a particular file id. The machine type is used in conjunction with the kread_XXXX & kwrite_XXXX routines which provide data conversion for data independence.

F.10.7. kfile_getpermanence() — *returns whether a file has data permanence or not*

Synopsis

```
int kfile_getpermanence(
   char *path)
```

Input Arguments

`path`

path of identifier to inquire about

Returns

returns TRUE or FALSE depending whether the transport is known to support data permanence or -1 upon error

Description

This function is used to return whether a particular file or transport identifier is a permanent data transport (like shared memory) or a connection oriented protocol like (stream or sockets).

F.10.8. kfile_isdup() — *khoros transport has been has been/is a dupped transport*

Synopsis

```
int kfile_isdup(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport has been dupped.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport has been or is a dupped transport. A transport can be dupped by the routines kdup(), kdup2(), kfdup(), kfdup2().

F.10.9. kfile_iseof() — *khoros transport is at end of file (eof)*

Synopsis

```
int kfile_iseof(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport is at end of file (eof).

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is currently at end of file. A transport end of file flag can be cleared using the kclearerr() routine.

F.10.10. kfile_ismybuf() — *khoros transport buffer was set by the application*

Synopsis

```
int kfile_ismybuf(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the buffer was set by the programmer.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport's read/write buffer was set by the programmer or was internally allocated by the open transport process.

F.10.11. kfile_isopen() — *khoros transport has been properly opened*

Synopsis

```
int kfile_isopen(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport has been opened.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport has been opened.

F.10.12. kfile_islock() — *khoros transport has been/is a locked transport*

Synopsis

```
int kfile_islock(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport has been opened lock.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport has been or is a locked transport. A transport can be locked using the KFILE_LOCK flag on open.

F.10.13. kfile_isreacquire() — *khoros transport will be reacquired*

Synopsis

```
int kfile_isreacquire(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport is allowed to be reacquired

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport will be reacquired. This is how applications like VisiQuest can get fast IPC between processes.

F.10.14. kfile_ispermanent() — *khoros transport has permanence*

Synopsis

```
int kfile_ispermanent(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport has data permanence.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport has data permanence. If a transport has permanence then the data can be accessed multiple times. In the case of files, shared memory, memory mapped files, these transports have permanence. Where as for pipes and sockets the data can only be accessed (read or written) to once.

F.10.15. kfile_isbufferd() — *khoros transport is buffered or not*

Synopsis

```
int kfile_isbuffered(

  kfile *file)
```

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is buffered or not

F.10.16. kfile_isstreambuf() — *khoros transport is stream buffered*

Synopsis

```
int kfile_isstreambuf(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport is stream buffered

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is stream buffered

F.10.17. kfile_islinebuf() — *khoros transport is line buffered*

Synopsis

```
int kfile_islinebuf(

  kfile *file)
```

Input Arguments

`file`
: the khoros transport to test whether the transport is line buffered

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is line buffered

F.10.18. kfile_isfullbuf() — *khoros transport is full buffered*

Synopsis

```
int kfile_isfullbuf(

  kfile *file)
```

Input Arguments

`file`
: the khoros transport to test whether the transport is full buffered

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is full buffered

F.10.19. kfile_ismembuf() — *khoros transport is memory buffered*

Synopsis

```
int kfile_ismembuf(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport is memory buffered

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is memory buffered

F.10.20. kfile_isread() — *khoros transport is readable*

Synopsis

```
int kfile_isread(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether readable

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is readable.

F.10.21. kfile_isrdwr() — *khoros transport is both readable and writeable*

Synopsis

```
int kfile_isrdwr(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether it's readable and writeable

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is readable and writeable.

F.10.22. kfile_istemp() — *khoros transport is temporary*

Synopsis

```
int kfile_istemp(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether the transport is a temporary

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport has been created as a temporary. A transport created via ktempnam() or kfile_tempnam() is flagged as a temporary transport.

F.10.23. kfile_iswrite() — *khoros transport is writeable*

Synopsis

```
int kfile_iswrite(

  kfile *file)
```

Input Arguments

`file`

the khoros transport to test whether writeable

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

Whether a khoros transport is writeable.

F.10.24. kfile_mode() — *return the mode associated with transport id*

Synopsis

```
mode_t kfile_mode(
   int id)
```

Input Arguments

`id`

the transport id from which the flags is returned

Returns

The mode associated with the id or KMODE_INVALID upon failure

Description

This routine returns the mode associated with a khoros transport descriptor id. This is the mode used in originally creating/opening of the transport.

F.10.25. kfile_readdata() — *read the contents of a khoros transport into a data array*

Synopsis

```
kaddr kfile_readdata(
   int id,
   kaddr ptr,
   size_t * num)
```

Input Arguments

`id`
: the khoros transport descriptor

`ptr`
: data to be used if not NULL

Output Arguments

`num`
: if not NULL the number of bytes read is returned

Returns

a pointer to the contents of the file or NULL upon failure

Description

This routine reads the contents of a file, using the khoros transport mechanisms, into an array. If the data array is NULL then a data array of suitable size is allocated and passed back. An optional "num_read" can be used to find out the actual number of bytes read. If "num_read" is NULL then the parameter is ignored.

Side Effects

the data returned is allocated and needs to freed using kfree_and_NULL().

F.10.26. kfile_remotehost() — *returns whether a host is remote*

Synopsis

```
int kfile_remotehost(
   char *machine)
```

Input Arguments

`machine`
: the machine name we wish to find out about

Returns

TRUE (1) if the machine name is a remote name, FALSE (0) otherwise

Description

This function determines whether or not the machine name passed in is the hostname or an alias of the current machine, or if it is the name of another machine on the network.

F.10.27. kfile_reopen() — *re-open a stream khoros transport*

Synopsis

```
kfile *kfile_reopen(
   char *path,
   char *type,
   kfile *file)
```

Input Arguments

`path`

is the string containing the path name to the desired file to be open. The path name identifies the file to be opened, just like the regular UNIX freopen() command.

`type`

how the file is to be re-opened.

"r" - Open file for only reading
"w" - Open file for only writing
"a" - append file for only writing
"A" - append file for writing; non overwrite
"r+" - Open file for both reading & writing
"w+" - Open file for both reading & writing
"a+" - append file for only writing
"A+" - append file for writing; non overwrite

`file`

the existing khoros stream transport to be reopened for the specified filename.

Returns

returns the kfile structure or NULL upon failure

Description

This function is similar to the khoros kfreopen() call. The only difference is that kfile_reopen() re-open the a khoros stream transport as well close and re-open all of transports that have been kdup'ed from the existing transport.

kfile_reopen() will open a specified file on for an existing stream. The existings stream is closed before the new filename is opened. This function is typically used to open a specified file as one of the predefined streams; such as standard input, standard output, or standard error.

F.10.28. kfile_seddata() — *string edit from one transport to another*

Synopsis

```
int kfile_seddata(

   int id1,
   int id2,
   kvalist)
```

Input Arguments

`id1`
: source transport id to copy data from

`id2`
: destination transport id to copy data to

`kvalist`
: list of search patterns and replacement patterns to be used to modify the data as it is being copied.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine takes two transport id's, and copies the data from the source file to the destination file. It also uses a variable argument list of strings pairs, where each pair indication a search pattern and a replacement pattern. This list will be used to replace strings in pattern strings in the input transport with the corresponding replacement pattern. Note, the variable argument list MUST be terminated with a NULL in one of the search pattern slots

For example, the following is a correct termination:

```
kfile_seddata(id1, id2, "t1", "t2", FALSE, NULL);
```

And this is an incorrect termination:

```
kfile_seddata(id1, id2, "t1", NULL);
```

And this is an incorrect termination:

```
kfile_seddata(id1, id2, "t1", NULL, FALSE);
```

The reason the NULL must be on the search pattern, is so that NULL can be sent in as the replacement string. The replacement mechanism is kstring_replace, and its documentation explains restrictions on search strings and replace strings.

The regex parameter is used to indicate whether the search and replacement parameters are specified in regular expression form, or are just simply string replacement patterns. For more information about regular expressions please see the khoros string parsing section of the VisiQuest Programmers Manual.

kfile_seddata does not support the "mode" operation as used by ksedfile(). The mode operation is used to how the data should be updated, but it relies on the ability to read as well as write to the destination id (id2).

F.10.29. kfile_setmachtype() — *sets the machine architecture type for a file transport id*

Synopsis

```
int kfile_setmachtype(
   int id,
   int machtype)
```

Input Arguments

`id`
: the file id on which to set the machtype, which was opened earlier with kopen().

`machtype`
: the machine architecture type to set the file id to.

Returns

0 on success, -1 otherwise

Description

This function sets the machine architecture type for a particular file id. The machine type is used in conjunction with the kread_XXXX & kwrite_XXXX routines which provide data conversion for data independence.

F.10.30. kfile_setstate() — *adds a flag to the open transport id state*

Synopsis

```
unsigned long kfile_setstate(
   kfile *file,
   unsigned long flag)
```

Input Arguments

`file`
the transport id from which to set the flag for

`flag`
the flag to be set (such as KFILE_TEMP)

Returns

returns the updated flags associated with the id or -1 upon failure

Description

This routine is used to set a flag within the transport id state field.

F.10.31. kfile_getstate() — *return the current internal stream transport state*

Synopsis

```
unsigned long int kfile_getstate(
   kfile *file)
```

Input Arguments

`file`
the kfile transport to be returned

Returns

The type associated with the kfile transport on success, or NULL upon failure

Description

This routine returns the internal state of a khoros stream transport. The state is mask of the following defines:

KFILE_READ - whether transport is readable
KFILE_WRITE - whether transport is writeable
KFILE_RDWR - whether transport is readable &

writeable

KFILE_MYBUF - whether transport uses user defined

stream buffer

KFILE_EOF - whether transport is at end of file
KFILE_ERR - whether transport is in an error
KFILE_PERM - whether transport has data permanence
KFILE_OPEN - whether transport is currently opened
KFILE_DUP - whether transport is currently dupped
KFILE_TEMP - whether transport is a temporary
KFILE_LOCK - whether transport enforces locking

Side Effects

the type field returned is the internal copy of the kfile transport. This means that you should not modify or free the string.

F.10.32. kfile_type() — *return the type flag field used when opening the transport with kfopen()*

Synopsis

```
char *kfile_type(
   kfile *file)
```

Input Arguments

`file`

the kfile transport returned earlier by kfopen()

Returns

The type associated with the kfile transport on success, or NULL upon failure

Description

This routine returns the type field associated with a khoros transport. This is the type fields used originally when opening the transport with kfopen().

Side Effects

the type field returned is the internal copy of the kfile transport. This means that you should not modify or free the string.

F.10.33. kfile_writedata() — *write the contents of the data to a khoros transport*

Synopsis

```
ssize_t kfile_writedata(
    int id,
    kaddr data,
    size_t num)
```

Input Arguments

`id`
: the khoros transport descriptor

`data`
: the data array (or string) to be written

`num`
: if not 0 the number of bytes to be written.

Returns

The number of bytes written or -1 upon failure

Description

This routine writes the contents of a data array to the specified khoros transport mechanism. An optional "num_write" can be used to specify the actual number of bytes to be written. If "num_write" is 0 then the number of bytes to be written is computed from the data by using kstrlen().

F.10.34. kfidfile() — *return the kfile structure associated with a fid*

Synopsis

```
kfile *kfidfile(
    int id)
```

Input Arguments

`id`
: the transport identifier opened earlier

Returns

The kfile pointer representing the open stream to the data transport on success, NULL on failure

Description

This function is used to retrieve the corresponding kfile structure given a khoros transport descriptor (fid).

F.10.35. ktmpfid() — *create a temporary transport descriptor*

Synopsis

```
int ktmpfid(void)
```

Returns

The fid descriptor representing the temporary transport on success, -1 on failure

Description

This function is similar to ktmpfile(), except that it returns a transport descriptor instead of a transport structure. The data transport will automatically be deleted when closed.

F.10.36. ktransport_add() — *add a new transport to the list of transports*

Synopsis

```
int ktransport_add(
   TransportInformation * transport)
```

Input Arguments

`transport`
the transport to be added

Returns

TRUE if the transport was successfully added, FALSE otherwise

Description

This function is used to add a new transport to the compiled list of transports. The transport is dynamically added to during runtime, rather than during compile time.

F.10.37. ktransport_delete() — *delete a transport from the list of all transports*

Synopsis

```
int ktransport_delete(
   TransportInformation * transport)
```

Input Arguments

`transport`
: the transport to be deleted

Returns

TRUE if the transport was successfully deleted, FALSE otherwise

Description

This function is used to delete a new transport from the compiled list of transports. The transport is dynamically deleted from the list, rather than during compile time.

F.10.38. ktransport_list() — *get the list of supported transports*

Synopsis

```
char **ktransport_list(
   int local,
   int permanence,
   int stream,
   size_t * num)
```

Input Arguments

`local`
: a boolean indicating that local transports should be listed

`permanence`
: a boolean indicating that data permanent transports should be included in the list

`stream`
: a boolean indicating that stream transports should be included in the list

Output Arguments

`num`
: the number of entries returned

Returns

A pointer to an array of the available transports, or NULL on error.

Description

This function creates an array of available transports. Boolean inputs allow the calling routine to selectively mask in or out local, stream, and permanent transports.

F.11. Introduction to General File Utilities

The functions cover general file utilities to compare the contents of two files and to copy the contents from one file to another. Also included are functions to read, write, and string-edit (sed) a file.

- *kcomparefile()* - compare the contents of one filename to another
- *kcopyfile()* - copy the contents of one filename to another
- *keditfile()* - start up an edit program to edit an input file
- *kprintfile()* - print a file to a printer
- *kreadfile()* - read the contents of a file into a data array
- *ksedfile()* - khoros string edit a file
- *kwritefile()* - write the contents of the data to a file

F.12. Definitions of General File Utilities

F.12.1. kcomparefile() — *compare the contents of one filename to another*

Synopsis

```
int kcomparefile(
   const char *filename1,
   const char *filename2,
   size_t num,
   size_t * num_compared)
```

Input Arguments

`filename1`
the first file to be copied

`filename2`
the output file to be copied

`num`
the number of bytes to be compared, if 0 then all are compared

Output Arguments

`num_compared`
if not NULL then returns the number of bytes actually compared.

Returns

An integer less than, equal to, or greater than 0, according as is lexicographically less than, equal to,

or greater than id2.

Description

This routine compares the contents of the first filename to that of second filename. The contents of the files are compared using memcmp().

F.12.2. kcopyfile() — *copy the contents of one filename to another*

Synopsis

```
ssize_t kcopyfile(
   const char *ifilename,
   const char *ofilename)
```

Input Arguments

`ifilename`
the input file to be copied

`ofilename`
the output file to be copied

Returns

the number of actual bytes written or -1 upon failure.

Description

This routine copies the contents of a input filename to an output filename. The contents of the files are copied using the VisiQuest transport mechanism.

F.12.3. keditfile() — *start up an edit program to edit an input file*

Synopsis

```
int keditfile(

   char *filename,
   int  spawn,
   kvalist)
```

Input Arguments

`filename`

the filename to be edited

`spawn`

to spawn the process into background or not

`kvalist`

the NULL terminated list of editfile options

Description

This routine executes a system call to the system editor. It uses the environment variable KHOROS_EDITOR to determine which system editor to use. If KEDITOR is not set then EDITOR will be checked. If neither of these environment variables are set then the routine will use 'vi' as the default.

Also if the DISPLAY environment variable is set then the editor will be executed within an xterm. The xterm and editor will be started in background and keditfile will return immediately. If not then the editor will be executed locally and will not return until the process finishes executing.

Also, options can be specified to keditfile. The following are a list of options:

KEDITOR_GEOMETRY - the geometry of the editor
KEDITOR_TITLE - the editor title
KEDITOR_FOREGROUND - the editor foreground color
KEDITOR_BACKGROUND - the editor background color
KEDITOR_ICON - the editor icon file
KEDITOR_PID - the pid of the editor
KEDITOR_CHDIR - change to file's directory

F.12.4. kprintfile() — *print a file to a printer*

Synopsis

```
int kprintfile(

    char *filename,
    kvalist)
```

Input Arguments

`filename`

the filename to be printed

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine executes a system call to the system printer. It uses the environment variable PRINTER to determine which printer to use. This can be overriden using the KPRINTER_NAME option below. If neither of these are specified then the routine will use 'default' printer by calling print command with no print option.

Also, options can be specified to kprintfile. The following are a list of options:

KPRINTER_NAME - the printer name to print the file to

F.12.5. kreadfile() — *read the contents of a file into a data array*

Synopsis

```
kaddr kreadfile(
   const char *filename,
   size_t * num_read)
```

Input Arguments

`filename`

the filename which contains the data to be read

Output Arguments

`num_read`

if not NULL the number of bytes read is returned

Returns

returns a pointer to the contents of the file or NULL upon failure

Description

This routine opens the specified and using the khoros transport mechanisms reads the contents of the file into an array. An optional "num_read" can be used to find out the actual number of bytes read. If "num_read" is NULL then the parameter is ignored.

Side Effects

the data returned is allocated and needs to freed using kfree_and_NULL().

F.12.6. ksedfile() — *khoros string edit a file*

Synopsis

```
int ksedfile(
   const char *ifilename,
   const char *ofilename,
   int mode,
   int *status,
   kvalist)
```

Input Arguments

`ifilename`
: source filename to copy data from

`ofilename`
: destination filename to copy data to

`mode`
: the mode in which to update the destination file

`kvalist`
: list of search patterns and replacement patterns to be used to modify the data as it is being copied.

Output Arguments

`status`
: the status of whether the destination file was or could be changed, set according to the specified mode.

Returns

TRUE (1) on success, FALSE (0) otherwise

Description

This routine takes two filenames, and copies the data from the source file to the destination file. It also uses a variable argument list consisting of pairs of strings and a logical value. The strings are used as a search pattern and a replacement pattern, respectively. The logical value is used to tell ksedfile whether or not to use kstring_replace or kregex_replace. A value of TRUE tells it to use kregex_replace, FALSE tells it to use kstring_replace. kregex_replace allows sed like substitutions, but is much slower than a straight string replace. This offers the user speed when patterns are simple, and versatility when necessary. Note, the variable argument list MUST be terminated with a NULL in one of the search pattern slots

For example, the following is a correct termination:

```
ksedfile(ifilename, ofilename, KFILE_UPDATE, NULL, "t1", "t2", FALSE, NULL);
```

And this is an incorrect termination:

ksedfile(ifilename, ofilename, KFILE_UPDATE, NULL, "t1", NULL);

The reason the NULL must be on the search pattern, is so that NULL can be sent in as the replacement string.

The mode parameter dictates how ksedfile should perform the string editing. The first mode is to query what would happen if string edition would take place. Whether any difference exists in which to update the destination. The second is to update the destination, only if a difference exists. The final mode is to overwrite the destination irregardless if a difference exists or not. The mode parameter can be one of:

KFILE_QUERY - returns whether an update would take place
KFILE_UPDATE - updates the destination only if different
KFILE_OVERWRITE - updates the destination regardless

if KFILE_QUERY is specified then the "status" parameter will retain whether or not the output file would be updated. If KFILE_UPDATE is used then the "status" parameter is will retain whether to output file was actually changed or not. If KFILE_OVERWRITE is specified then the "status" parameter will always be TRUE, unless an error is encountered and ksedfile() returns FALSE.

F.12.7. kwritefile() — *write the contents of the data to a file*

Synopsis

```
ssize_t kwritefile(
   const char *filename,
   kaddr data,
   size_t num_write)
```

Input Arguments

`filename`
: the filename which the data will be written to

`data`
: the data array (or string) to be written

`num_write`
: if not 0 the number of bytes to be written.

Returns

returns the number of actual bytes written or -1 upon failure.

Description

This routine opens the specified and using the khoros transport mechanisms writes the contents of the

data array to the specified file. An optional "num_write" can be used to specify the actual number of bytes to be written. If "num_write" is 0 then the number of bytes to be written is computed from the data by using kstrlen().

F.13. Miscellaneous Utilities

• *kflags_to_type()* - convert kopen() flags to kfopen() type parameter
• *ktype_to_flags()* - convert kfopen() type to kopen() flags

F.13.1. kflags_to_type() — *convert kopen() flags to kfopen() type parameter*

Synopsis

```
char *kflags_to_type(
   int flags,
   char *type)
```

Input Arguments

`flags`
kopen flags

Output Arguments

`type`
a string to old the kfopen type field. If NULL then space is malloc'ed

Returns

type is returned if it is not NULL. Otherwise the malloc'ed string is returned. NULL is returned on an error.

Description

The kflags_to_type() routine is used to convert the flags used in the kopen() call into their corresponding equivalent in the type field of the kfopen() call. The "flags" parameter is converted and stored into the "type" parameter. If the "type" parameter is NULL then space is malloc'ed and returned. The following is an example of how to use kflags_to_type():

```
type = kflags_to_type(KOPEN_WRONLY | KOPEN_CREAT | KOPEN_TRUNC, NULL);
```

the input flags are converted and result returned to type will be "w".

Here is the table as it translates:

```
a+  = KOPEN_RDWR   | KOPEN_APPEND | KOPEN_CREAT
```

w+ = KOPEN_RDWR | KOPEN_CREAT | KOPEN_TRUNC
r+ = KOPEN_RDWR
a = KOPEN_WRONLY | KOPEN_APPEND | KOPEN_CREAT
w = KOPEN_WRONLY | KOPEN_CREAT | KOPEN_TRUNC
r = KOPEN_RDONLY

F.13.2. ktype_to_flags() — *convert kfopen() type to kopen() flags*

Synopsis

```
int ktype_to_flags(
   const char *type)
```

Input Arguments

`type`
the kfopen string field.

Returns

the kopen flags on success. If type is NULL or an invalid type, a -1 will be returned.

Description

The ktype_to_flags() routine is used to convert the type field used in the kfopen() call into their corresponding equivalent in the flags field of the kopen() call. The "type" parameter is converted and returned in kopen flag form. The following is an example of how to use ktype_to_flags():

flags = ktype_to_flags("w");

the input flags are converted and result returned to type will be KOPEN_WRONLY|KOPEN_CREAT|KOPEN_TRUNC.

Here is the table as it translates:

a+ = KOPEN_RDWR | KOPEN_APPEND | KOPEN_CREAT
w+ = KOPEN_RDWR | KOPEN_CREAT | KOPEN_TRUNC
r+ = KOPEN_RDWR
a = KOPEN_WRONLY | KOPEN_APPEND | KOPEN_CREAT
w = KOPEN_WRONLY | KOPEN_CREAT | KOPEN_TRUNC
r = KOPEN_RDONLY

G. Process Execution

G.1. Introduction to Process Execution Utilities

These routines allow the execution of processes.

- *kexecvp()* - execute a command
- *kexit_handler()* - adds an kexit handler
- *ksignal()* - khoros signal handler
- *ksignal_format()* - format the actual signal error
- *kspawn()* - spawn a command
- *ksystem()* - issue a shell command

G.2. Definitions of Process Execution Utilities

G.2.1. kexecvp() — *execute a command*

Synopsis

```
int kexecvp(
   const char *arg0,
   char *const args[])
```

Input Arguments

`arg0`
the routine to be executed

`args`
the arguments, include arg0, in which execute the command

Returns

Note: this call overlays the calling process with the named file, then transfers control to the new core image. If the command is successfully executed then there will be no return from the new process. If the core cannot be loaded or found then we return with an error of -1.

Description

This function is a replacement for the system "execvp" call. The only difference is that kexecvp() supports executing processes on remote machines.

The routine will execute another process (specified by arg0) and replace the current core image with the specified core. For example, the command:

"vfileinfo -i1 file"

would be specified as:

arg0 = "vfileinfo"

args[0] = "vfileinfo"
args[1] = "-i1"
args[2] = "file"
args[3] = NULL

If the command is to be executed on a different machine such as "borris" then the same command would look like:

"vfileinfo@borris -i1 file"

would be specified as:

arg0 = "vfileinfo@borris"

args[0] = "vfileinfo@borris"
args[1] = "-i1"
args[2] = "file"
args[3] = NULL

If the command is to be executed on a different machine such as "borris", and the file to be read is on "natasha" then the command would look like this:

"vfileinfo@borris -i1 file@natasha"

would be specified as:

arg0 = "vfileinfo@borris"

args[0] = "vfileinfo@borris"
args[1] = "-i1"
args[2] = "file@natasha"
args[3] = NULL

G.2.2. kexit_handler() — *adds an kexit handler*

Synopsis

```
int kexit_handler(
   void (*handler) (int, kaddr),
   kaddr client_data)
```

Input Arguments

`handler`
the handler to be called

`client_data`
the data to passed when calling the routine

Returns

TRUE on success, FALSE on failure

Description

kexit_handler adds an handler so that upon call of kexit(), the handler is called. kexit_handler() can be called several times, in which case the termination handlers will be called in reverse order of their instantiation. The handlers should be declared as follows:

```
void handler(
  int status,
  kaddr client_data)
```

G.2.3. ksignal() — *khoros signal handler*

Synopsis

```
void (*ksignal(

   int signo,
   void (*sigfunc)(int)))(int)
```

Input Arguments

`signo`
the signal number that you wish to install the handler for.

`sigfunc`
the signal function to be called when the signal is encountered.

Returns

the previously installed signal handler on success, SIG_ERR on failure

Description

This routine is a replacement for the UNIX signal() handler. The problem with UNIX signal() is that under SVR4 signal() provedes the older, unreliable signal semantics.

The signal handling function takes a single integer which indicates the single to handle.

G.2.4. ksignal_format() — *format the actual signal error*

Synopsis

```
char *ksignal_format(
   int signo,
   char *string)
```

Input Arguments

`signo`

the signal to print.

Output Arguments

`string`

the string

Description

This routine actually prints to a string the signal that occured.

G.2.5. kspawn() — *spawn a command*

Synopsis

```
#ifndef KOPSYS_WIN32
pid_t kspawn(
   const char *command)
#else
pid_t kspawn(
   const char *command,
   HANDLE *phProcess)
#endif
```

Input Arguments

`command`

the command to be spawn phProcess (KOPSYS_WIN32 only) - pointer to HANDLE for created process

Returns

the sub-process id on success, -1 on failure

Description

This function is used to spawn a command. Similar to "ksystem" call, the kspawn() supports more than just files, it supports other data transports in order to allow remote execution of a job.

G.2.6. ksystem() — *issue a shell command*

Synopsis

```
int ksystem(
   const char *command)
```

Input Arguments

`command`

the command to be systemed

Returns

The job status

Description

This function is a replacement for the "system" library call. The only difference is that ksystem() supports more than just files, it supports other data transports in order to allow remote execution of a job.

H. InterProcess Communication

H.1. Introduction to Data Transport IPC Utilities

These general utilities are for passing file descriptors between processes in client/server programs. The functions here include:

- *kipc_install_handler()* - install IPC handler
- *kipc_check_handler()* - check to see if an IPC handler is installed

• *kipc_remove_handler()* - remove IPC handler
• *kipc_stop()* - stop or discontinue an IPC.
• *kipc_enable()* - enable IPC before exec'ing a new process
• *kipc_process()* - get and dispatch a message
• *kipc_dispatch()* - dispatch a message
• *kipc_get_data()* - get raw data from a process
• *kipc_send_data()* - send raw data to a process
• *kipc_get_message()* - get a message
• *kipc_send_message()* - send a message
• *kipc_response_message()* - send a response message
• *kipc_pending()* - waits for a message
• *kipc_check()* - check that the IPC is enabled
• *kipc_verify()* - verify an IPC handler is installed
• *kipc_debug()* - Turn on debugging messages
• *kipc_debug_file()* - specify a file to send debug data

H.2. Definitions of Data Transport IPC Utilities

H.2.1. kipc_install_handler() — *install IPC handler*

Synopsis

```
int kipc_install_handler(
   const char *type,
   void (*handler) PROTO((KipcMsg *)))
```

Input Arguments

`type`
: the type of IPC to check for

`handler`
: the IPC callback handler

Returns

TRUE on success, and FALSE on failure

Description

This routine installs a handler for the given IPC type.

H.2.2. kipc_check_handler() — *check to see if an IPC handler is installed*

Synopsis

```
int kipc_check_handler(
   const char *type)
```

Input Arguments

`type`
the type of IPC to check for

Returns

TRUE on success, and FALSE on failure

Description

This routine checks to see if a handler is installed for the given IPC type.

H.2.3. kipc_remove_handler() — *remove IPC handler*

Synopsis

```
int kipc_remove_handler(
   const char *type,
   void (*handler) PROTO((KipcMsg *)))
```

Input Arguments

`type`
the type of IPC to check for
`handler`
the IPC callback handler

Returns

TRUE on success, and FALSE on failure

Description

This routine removes a handler for the given IPC type.

H.2.4. kipc_stop() — *stop or discontinue an IPC.*

Synopsis

```
int kipc_stop(void)
```

Returns

TRUE on success, FALSE upon failure

Description

This routine is used to stop or discontinue an IPC so that processes will no longer communicate with each other. This is done by actually closing the IPC transports and removing it from the user's environment variables.

H.2.5. kipc_enable() — *enable IPC before exec'ing a new process*

Synopsis

```
void kipc_enable(
   int enable)
```

Description

This routine is used to enable or disable the client IPC so that processes exec'ed will communicate with the server. This is done by actually adding or removing the KIPC_IDENTIFIER from the environment.

H.2.6. kipc_process() — *get and dispatch a message*

Synopsis

```
int kipc_process(
   pid_t pid,
   const char *type)
```

Input Arguments

`pid`
: the process to send the message to

`type`
: the type of IPC message to processed, or NULL for any

Returns

TRUE on success, and FALSE on failure

Description

This routine gets the next IPC message and dispatches it. The routine blocks until a message arrives. The type of message to processed is specified by the "type" parameter, or NULL be passed in order to process any IPC message type.

H.2.7. kipc_dispatch() — *dispatch a message*

Synopsis

```
int kipc_dispatch(
   KipcMsg * msg)
```

Returns

TRUE on success, and FALSE on failure

Description

This routine dispatches a message retrieved by kipc_get_message().

H.2.8. kipc_get_data() — *get raw data from a process*

Synopsis

```
int kipc_get_data(
   pid_t pid,
   const char *type,
   int timeout,
   KipcMsg *msg)
```

Input Arguments

`pid`
: the process to send the message to

`type`
: the type of IPC message to processed, or NULL for any

`timeout`
: the amount of time (in msec) to wait

Returns

TRUE on success, and FALSE on failure

Description

This routine gets raw data to an IPC receiver. The data is stored in the message field of the KipcMsg structure and the size updated to represent the amount of data received.

H.2.9. kipc_send_data() — *send raw data to a process*

Synopsis

```
int kipc_send_data(
   pid_t pid,
   const char *type,
   const char *data,
   size_t size,
   KipcMsg *response)
```

Input Arguments

`pid`
: the process to send the message to

`type`
: the type of IPC message to processed, or NULL for any

`data`
: the data to be sent

`size`
: the size of the data to be sent

Returns

TRUE on success, and FALSE on failure

Description

This routine sends raw data to an IPC receiver. The data is stored in the message

H.2.10. kipc_get_message() — *get a message*

Synopsis

```
int kipc_get_message(
   pid_t pid,
   const char *type,
   int timeout,
   KipcMsg *msg)
```

Input Arguments

`pid`
: the process to send the message to

`type`
: the type of IPC message to processed, or NULL for any

`timeout`

the amount of time (in msec) to wait

Returns

TRUE on success, and FALSE on failure

Description

This routine gets a message

H.2.11. kipc_send_message() — *send a message*

Synopsis

```
int kipc_send_message(
   pid_t pid,
   const char *type,
   const char *message,
   char *response)
```

Input Arguments

`pid`
: the process to send the message to

`type`
: the type of IPC message to processed, or NULL for any

`message`
: the message to be sent

Output Arguments

`response`
: the response to the message

Returns

TRUE on success, and FALSE on failure

Description

This routine sends a message

H.2.12. kipc_response_message() — *send a response message*

Synopsis

```
int kipc_response_message(
   pid_t pid,
   int msgid,
   const char *format,
   kvalist)
```

Input Arguments

`format`
: the format in which to the arguments will be

`kvalist`
: variable number of values to format and write to the output file stream. The format string determines the data type of the value(s) to be provided.

Returns

TRUE on success, and FALSE on failure

Description

This routine sends a response message

H.2.13. kipc_pending() — *waits for a message*

Synopsis

```
pid_t kipc_pending(
   int timeout)
```

Input Arguments

`timeout`
: the amount of time (in msec) to block

Returns

returns the process id on success or -1 on failure

Description

This routine listens to the active clients for an incoming message.

H.2.14. kipc_check() — *check that the IPC is enabled*

Synopsis

```
int kipc_check(void)
```

Description

This routine verifies that the IPC is enabled

H.2.15. kipc_verify() — *verify an IPC handler is installed*

Synopsis

```
int kipc_verify(
   pid_t pid,
   const char *type)
```

Description

This routine verifies that a handler is installed for the specified IPC process.

H.2.16. kipc_debug() — *Turn on debugging messages*

Synopsis

```
void kipc_debug(
   int debug)
```

Description

This routine is used to enable or disable the client/server ipc debugging messages.

H.2.17. kipc_debug_file() — *specify a file to send debug data*

Synopsis

```
void kipc_debug_file(const char *filename)
```

Input Arguments

`filename`
file name

Description

Specify the file to write debugging stmts to

I. Distributed Computing

The following section details the distributed computing utilities that is a part of the *klibc* library. This routines help developers use the built in distributed computing transport available in the VisiQuest system.

I.1. Introduction to the Distributed Computing Utilities

The distributed computing utilities are:

- *kremote_exec()* - creates a khoros command string which can exec'ed on a remote host
- *kremote_file()* - creates a khoros file string which can be accessed on a remote machine
- *kremote_location()* - return the remote location
- *kremote_check()* - check to see if we can access host
- *kremote_start()* - starts daemon on specified host
- *kremote_stop()* - stops daemon on the specified host
- *kremote_running()* - Check to see if host is currently running daemon
- *kremote_list()* - List the remote hosts with daemons started

I.2. Definitions of the Distributed Computing Utilities

I.2.1. kremote_exec() — *creates a khoros command string which can exec'ed on a remote host*

Synopsis

```
char *
kremote_exec(
    const char *machine,
    const char *command,
    char *remotexec)
```

Input Arguments

`command`
the command to be executed

Output Arguments

`remotexec`
if NULL malloc the return string

Description

This function make a proper command that the transport mechanisms will interpet as rsh://host/command. If a command of "ls -al" is given and they wish to execute that command on the host "borris". Then a new string will that looks like "rsh://borris/ls -al" will be returned.

I.2.2. kremote_file() — *creates a khoros file string which can be accessed on a remote machine*

Synopsis

```
char *
kremote_file(
    const char *machine,
    const char *filename,
    char *remotefile)
```

Input Arguments

`filename`
the file to be accessed

Output Arguments

`remotefile`
if NULL malloc the return string

Description

This function make a proper file that the transport mechanisms will interpet as rsh://host/file. If the file ".login" is given and they wish to access that file on the host "borris". Then a new string will that looks like "rsh://borris/.login" will be returned.

I.2.3. kremote_location() — *return the remote location*

Synopsis

```
char *
kremote_location(
   const char *name,
   char *location)
```

Input Arguments

`name`

the file or command to be parsed

Output Arguments

`location`

if NULL malloc the return string

Description

This function is used to find the location or machine in which the user wants to re-direct to via some transport. The location returned is the form of "machine" or "machine:username" or NULL if the location is local.

I.2.4. kremote_check() — *check to see if we can access host*

Synopsis

```
int
kremote_check(
   const char *host)
```

Input Arguments

`host`

the host to be checked

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to see if we can successfully access the host in which to distributed processes. Hostname must be specified in the form "machine:user:cmd", for example, cibola:sandy:rsh

I.2.5. kremote_start() — *starts daemon on specified host*

Synopsis

```
int
kremote_start(
    const char *host)
```

Input Arguments

`host`

the host to be started

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to start the khoros process daemon on the specified host. Hostname must be specified in the form "machine:user:cmd", for example, sandia:george:rsh

I.2.6. kremote_stop() — *stops daemon on the specified host*

Synopsis

```
int
kremote_stop(
    const char *host)
```

Input Arguments

`host`

the host to be stopped

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

Stops the khoros process daemon on the specified host. Hostname must be specified in the form

"machine:user:cmd", for example, acoma:debbie:rsh

I.2.7. kremote_running() — *Check to see if host is currently running daemon*

Synopsis

```
int
kremote_running(
    const char *host)
```

Input Arguments

`host`
the host to check

Returns

return TRUE (1) if the host is running, FALSE (0) otherwise

Description

Returns TRUE if the host is currently running a khoros process daemon, FALSE otherwise. Hostname must be specified in the form "machine:user:cmd", for example, nambe:arthur:rsh

I.2.8. kremote_list() — *List the remote hosts with daemons started*

Synopsis

```
char **
kremote_list(
    size_t * num)
```

Returns

An array of strings containing the remote hosts with active daemons on success, NULL on failure

Description

This function returns an allocated list of the remote hosts that have khoros process daemons started. Hostnames will be specified in the form "machine:user:cmd", for example, fantase:blair:rsh

I.3. Introduction to the Host Utilities

• *khost_list()* - get the list of machines that processes can be distributed to
• *khost_add()* - add a host to the list of machines that processes can be distributed to
• *khost_delete()* - delete a host from the list of machines that processes can be distributed to
• *khost_save()* - saves the current list of machines back into the khoros host file
• *khost_location()* - get the location part of the host spec
• *khost_username()* - get the username part of the host spec
• *khost_remotecmd()* - get the username part of the host spec
• *khost_entry()* - get the entry the best matches the partial host

I.4. Definitions of the Host Utilities

I.4.1. khost_list() — *get the list of machines that processes can be distributed to*

Synopsis

```
char **khost_list(
   size_t * num)
```

Output Arguments

`num`
the number of entries returned

Returns

a pointer to an array of strings with the list of machine names. It will return NULL on error.

Description

This function is used to return the list of initial machines to try using. They are gotten via the user specified file and also by broadcasting to see if any khoros daemons are available (Not Available as of yet).

I.4.2. khost_add() — *add a host to the list of machines that processes can be distributed to*

Synopsis

```
int khost_add(
   const char *host)
```

Input Arguments

`host`

the host to be added

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to add a host to the list of initial machines to try using.

I.4.3. khost_delete() — *delete a host from the list of machines that processes can be distributed to*

Synopsis

```
int khost_delete(
   const char *host)
```

Input Arguments

`host`

the host to be added

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to delete a host to the list of initial machines to try using.

I.4.4. khost_save() — *saves the current list of machines back into the khoros host file*

Synopsis

```
int khost_save(
   const char *hostfile)
```

Input Arguments

`hostfile`

the filename to save the hosts to, if NULL then save it to the default khoros file.

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to save the current list of machines back into the khoros host file. If the hosts are to be saved into the default khoros file then NULL should be passed for the filename. Otherwise the hosts will be saved into the specified filename.

I.4.5. khost_location() — *get the location part of the host spec*

Synopsis

```
char *khost_location(
   const char *host,
   char *location)
```

Input Arguments

`host`

the host to get the location for

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to get the location part of the host spec.

I.4.6. khost_username() — *get the username part of the host spec*

Synopsis

```
char *khost_username(
   const char *host,
   char *username)
```

Input Arguments

`host`

the host to get the username for

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to get the username part of the host spec.

I.4.7. khost_remotecmd() — *get the username part of the host spec*

Synopsis

```
char *khost_remotecmd(
   const char *host,
   char *remotecmd)
```

Input Arguments

`host`

the host to get the username for

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to get the username part of the host spec.

I.4.8. khost_entry() — *get the entry the best matches the partial host*

Synopsis

```
char *khost_entry(
   const char *host,
   char *entry)
```

Input Arguments

`host`

the host to get the username for

Returns

return TRUE (1) on success, FALSE (0) otherwise

Description

This function is used to get the best entry that matches a host. It looks thru the list and fully qualifies the machine:user:method. This is typically used by the transports in order to build remote commands and files.

This page left intentionally blank

Table of Contents

Chapter 6

Exception Handling

Copyright (c) AccuSoft Corporation, 2004. All rights reserved.

Chapter 6 - Exception Handling

A. Exception Handling In VisiQuest

A.1. Background

Previous versions of VisiQuest have used a cascading failure mechanism in dealing with errors. Each routine checks the return status on every function call it makes, and, in the event of a failure of any function, the routine itself fails. This mechanism was augmented with the use of the system errno.

The use of system errno allows VisiQuest to provide some info at the source of the error without having to print an error message. An integer assignment of the errno with an error value would let you know what triggered the error. For example, a data services call might fail because of a memory allocation failure.

VisiQuest extended the system errno by allowing libraries to add their own error numbers. To avoid collisions in error numbers, the VisiQuest errno's are actually externed integers which are dynamically assigned at runtime. The errno also has a string associated with it which is used in the *kerror()* function to indicate the category of the error. An entire set of errnos can be initialized at once using the function kerrno_init_errors(). There is generally one initialization done per library.

The VisiQuest unspoken convention on when to actually print an error message has been to "print the error when you have the information." Ideally, nothing actually prints an error message unless it is a top-level application function. However errors within libraries whose sole purpose is to provide application-level functionality (such as the datamanip or kcms libraries) often print errors. General purpose libraries such as data services never print an error unless it is fatal or very serious.

Checking the return status of every function call is good for robustness, but bad for code readability. The KCALL macro was developed to standardize the practice of checking the return statuses, printing an error message, and returning FALSE within the datamanip routines.

The unpopularity of the KCALL macro, coupled with the lack of a definitive standard for dealing with error conditions prompted a redesign of our error handling system. The new system needed to address the following goals for error handling:

- □ find alternative to unpopular KCALL
- □ model after C++ exceptions if possible
- □ provide more information about errors
- □ lightweight approach : minimize resource spending if no error has occurred

A.2. What is an exception?

An error condition occurs when something is amiss deep inside a function. Maybe a parameter is off or maybe a a resource can't be accessed. All the function knows is that something is happening which "may be bad." But it generally isn't bad enough to warrant killing the program. The solution is to let the calling function or functions know that something has happened so *they* can deal with it.

The previous solution for this had been to set an errno, and then exit with some type of failing return status.

An exception allows us to retain this functionality, but also allows the calling routine to optionally *intercept* the error condition as it is occurring. In this way, it is up to the calling function to determine the severity of the error.

A.3. How to raise an exception?

The function kthrow should be used to throw an exception. The exception thrown is indicated by the errno. The same old errnos used in the old cascading error system can be thrown.

Since valuable information about the cause of the error is generally available when the errno is set, the errno is augmented with an optional error message. A printf style syntax allows a detailed error message to be specified. The line number and file name are also associated with the error using the *__LINE__* and *__FILE__* compiler directives.

An example error condition set with a kthrow is illustrated below.

```
int function1(void)
{
   kaddr data;
   int   nbytes = 1000000000;

   if ( (data = kmalloc((size_t) nbytes)) == NULL)
   {
      kthrow(KMEMORY_ALLOCATION, "Unable to allocate %d bytes", nbytes);
      return FALSE;
   }
   else
      return TRUE;
}
```

In reality, this error is thrown *within* the kmalloc call.

Note that the routine still returns FALSE to indicate failure. There is never a guarantee that a calling function is going to intercept any errors, so code should still depend on the old cascading error return way of handling failure conditions.

A.4. What about the caller?

The function kcatch should be used to intercept exceptions.

The functions ktry_begin and ktry_end should be used to define a scope in which errors can be intercepted.

These functions have behavior that is analogous to the C++ try and catch function, although the syntax and ordering are different.

The basic use of the ktry and kcatch calls is as follows :

```
lkarith2()
{
   ktry_begin("kdatamanip", "lkarith2");
   /*-- catch the following errors --*/
   if (kcatch(KANY_ERROR,         khandle_warn,
              KMEMORY_ALLOCATION, khandle_error,
              KCMS_ERRORS,        khandle_error,
              NULL))
   {
      /*-- if we caught an error, we will end up here --*/
      kprintf("routine exiting unsuccessfully.\n");
      return FALSE;
   }
   /*-- call any functions we want, we don't have to check return status --*/
   function1();
   function2();
   function3();
   function4();
   function5();
   ktry_end();
   kprintf("routine exiting successfully!\n");
   return TRUE;
}
```

If an error is caught and the catch handler for that error indicates that it is a fatal error, the program will return back to the block below the kcatch call. At this point the routine must exit, but it may choose to do a return FALSE, exit(1), or whatever is appropriate for that function to indicate a failed return status.

Since you have no guarantee that your caller is catching any errors, unless you are catching exceptions yourself *within your function*, you should program as you usually do and check the return status of each function call you make.

A.5. Important Nuances

The ktry scopes can be nested. A thrown error will propagate up until an appropriate error handler has been found. If no error handler is found, program flow continues after the kthrow call. If a handler is found and it dictates that the error is fatal, the program flow will continue after the kcatch call *in which that handler was installed*.

One nuance of this functionality is that a ktry block must be opened and closed *within the same bracketed C scope*. It is not be possible to put the ktry_begin within a different call from the ktry_end. For example, we could not put a ktry_begin and kcatch call in khoros_initialize and install a ktry_end in the exit handler. The ktry_begin, kcatch, and ktry_end *must* occur in the same corresponding bracketed C scope. (This at least is no worse than C++ exception handling, in which the try and catch functions must occur in the same scope).

Note also that while the ktry_begin and kcatch calls are minimally expensive, they are not free. Since the cost of these calls is incurred *even if there are no errors*, they should NOT be used within every function of every library.

A.6. Handling Exceptions

An error handler is used to interpret the exception. On an exception, this handler is called with the error that had been thrown along with the library and routine for the try block in which the error handler was installed.

The return value of the handler indicates if the program should continue past the kthrow or return back to the kcatch call. A handler could also rethrow the error (or throw a new error) simply by calling kthrow itself.

- **TRUE** - return back to the kthrow of this error
- **FALSE** - return back to the kcatch of this catcher

Three standard handlers are already available : khandle_warn, khandle_error, and khandle_fatal.

You can, of course, write your own handlers. A simple handler for printing warnings would be as follows :

```
int khandle_warn(
   int   error,
   char *message,
   char *library,
   char *routine,)
{
   kwarn(library, routine, "things are *not* looking good");
   return TRUE;
}
```

A.7. Error Classes

Since it would be tedious to list every possible errno you might want to catch in the kcatch function, a basic extension to the kerrno_init_errors function was made to also define a class (or category) identifier for the errors defined. The class errno can then be used within the kcatch function to catch any exceptions of a particular class.

How to use this new functionality? The return value of kerrno_init_errors should be assigned to an externed integer representing the error class.

Currently, the only classes available are KSYSTEM_ERRORS for all the UNIX system errors, KUIS_ERRORS for user interface errors, KENV_ERRORS for environment variable processing errors, KMATH_ERRORS to catch mathematical processing errors, KPARSE_ERRORS for errors returned by the parsing routines, KDBM_ERRORS for database errors, and KANY_ERROR for any error. Other libraries

use this functionality to define other groups of errors.

A.8. Error Traceback

As errors are thrown, they are maintained in a list. With this functionality, we can print a traceback of the error.

This functionality is not yet being used by any handler functions, although you can generate a traceback yourself for debugging purposes using the call in your error handler:

```
kprintf("%s\n", ktraceback_report());
```

A.9. Introduction for Exceptions Handling routines

- *ktry_begin()* - open a try block for exception handling
- *ktry_end()* - close a try block for exception handling
- *kcatch()* - catch errors
- *kthrow()* - raise an error exception
- *krethrow()* - propagate an error exception
- *khandle_warn()* - handler exception as an warning
- *khandle_ignore()* - handler exception as an ignore
- *khandle_error()* - handler exception as an error
- *khandle_fatal()* - handler exception as a fatal error

A.10. Definitions of Exception Handling routines

A.10.1. ktry_begin() — *open a try block for exception handling*

Synopsis

```
void ktry_begin(
   char *library,
   char *routine)
```

Input Arguments

`library`
: library name for this try block

`routine`
: routine name for this try block

Description

This function opens a try block for exception handling. Any errors thrown by functions called within the try block can be intercepted with a kcatch call. The try block is closed with a call to the function

ktry_end.

The use of these exception handling functions allows a programmer to elegantly intercept error conditions without requiring the program to check the return status of every single function call.

The kcatch function should be used within this try block to indicate which specific errors or error classes should be caught. A handler is provided for each error or error class.

The kcatch call should be the first call after the ktry_begin. There should only be a single call to kcatch for every call to ktry_begin.

The kcatch call must occur within an if statement. If a fatal error is caught, the program will jump back to the conditional block after the if statement. All try blocks up to and including the current try block will be closed automatically. The routine must exit at this point.

The kcatch call should appear approximately as follows:

```
// catch the following errors
if (kcatch(KANY_ERROR,          khandle_warn,
           KMEMORY_ALLOCATION, khandle_error,
           KCMS_ERRORS,         khandle_error,
           NULL))
{
   // if we caught an error, we will end up here
   kprintf("routine exiting unsuccessfully.");
   return FALSE;
}
```

Try blocks can be nested. Each function call may contain a try block and with its own kcatch call. Each subsequent ktry_begin opens up a new level.

A thrown error will propagate up each try block until an appropriate error handler has been found. An error handler is a simple function which interprets the error condition. It may print a warning, or set some internal status variables. The exit status of the handler indicates the severity of the error.

For convenience, three standard error handlers are provided.

```
khandle_warn  - print a warning and continue past kthrow
khandle_error - print an error and fail
khandle_fatal - print an error and abort the program
```

You can, of course, write your own handlers and install them as well. An exit handler which prints a

simple warning message is illustrated below.

```
int khandle_warn(
   int   error,
   char *message,
   char *library,
   char *routine)
{
   kwarn(library, routine, "things are not looking so good");
   return TRUE;
}
```

The error handler may choose to interpret the error as a warning and return TRUE, indicating that the program flow should continue back after the kthrow call in which the error was thrown.

If the handler chooses to interpret the error as a fatal error, it may return FALSE. This will divert the program flow to continue after the kcatch call in which that handler was installed. This will allow you to do any necessary cleanup and exit the routine.

If no error handler is found, program flow continues after the kthrow call. In this event, the kthrow call serves only to set the system errno. Note that if no try blocks are ever opened, the only function of the kthrow will be to set the system errno.

Examples

A simple example of the use of a try block and a kcatch function is provided below.

```
lkarith2()
{
   // open a try block
   ktry_begin("kdatamanip", "lkarith2");

   // catch the following errors
   if (kcatch(KANY_ERROR,         khandle_warn,
              KMEMORY_ALLOCATION, khandle_error,
              KCMS_ERRORS,        khandle_error,
              NULL))
   {
      // if we caught an error, we will end up here
      kprintf("routine exiting unsuccessfully.");
      return FALSE;
   }
```

```
    function1();
    function2();
    function3();
    function4();
    function5();

    // close the try block
    ktry_end();

    kprintf("routine exiting successfully!");

    return TRUE;
}
```

Note that it was not necessary to check the exit status of each function call made within the try block. Any errors that occur within the functions will be intercepted with a warning. Memory allocation errors and cms errors will be intercepted with an error message and will cause the routine to exit.

Restrictions

If the kcatch call is not placed within an if statement (or if the routine fails to return after the if statement), it will just re-execute the same sequence of functions which produced the fatal error in the first place. This probably will result in an infinite loop as the same fatal error is thrown over and over again.

A try block must be opened and closed in the same bracketed C scope. Both the ktry_begin and ktry_end must be in the same function. Failure to close the try block within the same C scope will result in undefined behavior when catching errors.

Only a relatively small, finite number of try blocks can be opened when nesting, so avoid opening a try block in a deeply recursive function.

A.10.2. ktry_end() — *close a try block for exception handling*

Synopsis

```
void ktry_end(void)
```

Description

This function closes a try block for exception handling. Any errors thrown by functions called within the try block can be intercepted with a kcatch call. The try block should have been opened with a call to ktry_begin. See ktry_begin for details.

Restrictions

A try block must be opened and closed in the same bracketed C scope. Both the ktry_begin and ktry_end must be in the same function.

A.10.3. kcatch() — *catch errors*

Synopsis

```
int kcatch(int error,

           int (*handler)(int error, char *, char *, char *),
           kvalist)
```

Input Arguments

`error`

error number to catch

`handler`

handler function to use when error is thrown. This function should be of the form :

```
int handler_error(int error,
                  char *message,
                  char *library,
                  char *routine)
```

`kvalist`

variable argument list of error numbers and associated handlers. This list should be NULL terminated.

Returns

FALSE when kcatch is called to install the errors, and TRUE on a return from an error condition. This function should be placed with an if statement to redirect program flow when it returns TRUE.

You must exit the routine in the event of a fatal error. When an error does occur, the current try block will have been closed for you.

Description

This function is used within a try block for intercepting thrown exceptions.

The use of these exception handling functions allows a programmer to elegantly intercept error conditions without requiring the program to check the return status of every single function call.

The kcatch function should be used within an if statement. If a fatal error is thrown, program flow will

divert back to this function at which point it will return TRUE. This will direct control into the conditional block following the if statement.

The kcatch function should be used within a try block to indicate which specific errors or error classes should be caught. A handler should be provided for each error or error class which you wish to intercept.

A try block is opened with a call to ktry_begin and closed with a call to ktry_end. Try blocks can be nested. Each function call may contain a try block and with its own kcatch call. Each subsequent ktry_begin opens up a new level.

A thrown error will propagate up each try block until an appropriate error handler has been found. An error handler is a simple function which interprets the error condition. It may print a warning, or set some internal status variable. The exit status of the handler indicates the severity of the error.

For convenience, three standard error handlers are provided.

```
khandle_warn  - print a warning and continue past kthrow
khandle_error - print an error and fail
khandle_fatal - print an error and abort the program
```

You can, of course, write your own handlers and install them as well. An exit handler which prints a simple warning message is illustrated below.

```
int khandle_warn(
   int   error,
   char *message,
   char *library,
   char *routine)
{
   kwarn(library, routine, "things are not looking so good");
   return TRUE;
}
```

The error handler may choose to interpret the error as a warning and return TRUE, indicating that the program flow should continue back after the kthrow call in which the error was thrown.

If the handler chooses to interpret the error as a fatal error, it may return FALSE. This will divert the program flow to continue after the kcatch call in which that handler was installed. This will allow you to do any necessary cleanup and exit the routine.

If no error handler is found, program flow continues after the kthrow call. In this event, the kthrow call

serves only to set the system errno. Note that if no try blocks are ever opened, the only function of the kthrow will be to set the system errno.

Examples

A simple example of the use of a try block and a kcatch function is provided below.

```
lkarith2()
{
   // open a try block
   ktry_begin("kdatamanip", "lkarith2");

   // catch the following errors
   if (kcatch(KANY_ERROR,         khandle_warn,
              KMEMORY_ALLOCATION, khandle_error,
              KCMS_ERRORS,        khandle_error,
              NULL))
   {
      // if we caught an error, we will end up here
      kprintf("routine exiting unsuccessfully.");
      return FALSE;
   }

   function1();
   function2();
   function3();
   function4();
   function5();

   // close the try block
   ktry_end();

   kprintf("routine exiting successfully!");

   return TRUE;
}
```

Note that it was not necessary to check the exit status of each function call made within the try block. Any errors that occur within the functions will be intercepted with a warning. Memory allocation errors and cms errors will be intercepted with an error message and will cause the routine to exit.

Side Effects

On return from an error condition, the try blocks up to and including the current try block will be closed. It is very important to exit the function in the event of an error condition.

Restrictions

If the kcatch call is not placed within an if statement (or if the routine fails to return after the if statement), it will just re-execute the same sequence of functions which produced the fatal error in the first place. This probably will result in an infinite loop as the same fatal error is thrown over and over again.

A.10.4. kthrow() — *raise an error exception*

Synopsis

```
kthrow(int error,

       char *format,
       kvalist)
```

Input Arguments

`error`
: errno being thrown

`format`
: a formatted error message

`kvalist`
: optional variables which should be printed within the formatted message

Description

This function is used to raise an error exception. An exception will consist an error number and a an optional error message which describes the error condition. A printf style syntax is used for specifying the error message in order to allow variable values to be included as part of the message.

It may be used as an alternative to setting the system errno. The thrown error will be assigned to the system errno as well as being placed into an error stack. This function will also look for an installed error handler to deal with the error.

An error handler should be installed within a higher level function inside of a try block using the kcatch function. A try block is opened and closed using the ktry_begin and ktry_end functions.

A thrown error will propagate up until an appropriate error handler has been found. An error handler is a simple function which interprets the error condition and is installed in one or more higher level calling functions. If the error handler indicates that the error is not serious, program flow will return to after the kthrow. If the error handler indicates that the error is serious, program flow will be diverted from the kthrow to the kcatch call in which the error handler was installed. If no error handler has been found, the program will continue past the kthrow.

The use of these exception handling functions allows a programmer to elegantly intercept error conditions without requiring the program to check the return status of every single function call.

Examples

A simple example of the use of the kthrow function to throw an error and associated message is included below.

```
kthrow(KMEMORY_ALLOCATION, "Unable to alloc %d bytes.", number);
```

A.10.5. krethrow() — *propagate an error exception*

Synopsis

```
void krethrow(void)
```

Description

This function is used to propagate an error exception. The original errno, message and associated information are preserved. This function will look for the next error handler to deal with the message.

The place to use this function is in the kcatch block after appropriate actions are taken in that subroutine and the error is to be propagated further up the call stack. This allows a subroutine to terminate, free resources in the kcatch block and then rethrow the error. Note that if a rethrown error is caught by a handler that tries to continue execution, the execution will continue at the point the error is rethrown.

A.10.6. khandle_warn() — *handler exception as an warning*

Synopsis

```
int khandle_warn(
   int error_number,
   char *message,
   char *library,
   char *routine)
```

Input Arguments

`message`

message associated with the exception being thrown

`library`

library name for the try block in which the error has been intercepted

`routine`
routine name for the try block in which the error has been intercepted

Returns

TRUE indicating that control should return to the kthrow call which generating the exception

Description

This function is a handler which may be used with the kcatch function for intercepting exceptions.

This handler considers exceptions to be non-fatal and will simply print a warning using the kwarn function. On return, it will return back to the kthrow call which generated the exception.

A.10.7. khandle_ignore() — *handler exception as an ignore*

Synopsis

```
int khandle_ignore(
   int error_number,
   char *message,
   char *library,
   char *routine)
```

Input Arguments

`message`
message associated with the exception being thrown. This is not used by the ignore handler.

`library`
library name for the try block in which the error has been intercepted

`routine`
routine name for the try block in which the error has been intercepted

Returns

TRUE indicating that control should return to the kthrow call which generating the exception

Description

This function is a handler which may be used with the kcatch function for intercepting exceptions.

This handler considers exceptions to be non-fatal and will simply ignore them. On return, it will return back to the kthrow call which generated the exception.

A.10.8. khandle_error() — *handler exception as an error*

Synopsis

```
int khandle_error(
   int error_number,
   char *message,
   char *library,
   char *routine)
```

Input Arguments

`message`
: message associated with the exception being thrown

`library`
: library name for the try block in which the error has been intercepted

`routine`
: routine name for the try block in which the error has been intercepted

Returns

FALSE indicating that control should return to the kcatch call in which this handler was installed

Description

This function is a handler which may be used with the kcatch function for intercepting exceptions.

This handler considers exceptions to be errors and will print an error message using the kerror function. This function will return back to the kcatch call in which the handler was installed.

A.10.9. khandle_fatal() — *handler exception as a fatal error*

Synopsis

```
int khandle_fatal(
   int error_number,
   char *message,
   char *library,
   char *routine)
```

Input Arguments

`message`
: message associated with the exception being thrown

`library`

library name for the try block in which the error has been intercepted

`routine`

routine name for the try block in which the error has been intercepted

Returns

FALSE

Description

This function is a handler which may be used with the kcatch function for intercepting exceptions.

This handler consideres exceptions to be fatal and will abort the program after printing an error message.

Table of Contents

This page left intentionally blank