General User Interface (C Implementation) Programmer's Guide

by

George N. Reeke, Jr.
The Rockefeller University
New York, N. Y. 10021

April 22, 2016

THE ROCKS ROUTINES

PROGRAMMER'S GUIDE TO THE C IMPLEMENTATION

INTRODUCTION

This document constitutes the programmer's guide for the "ROCKS" user interface routines. A separate file, PLOTTING.TXT, defines the plotting interface used in conjunction with the ROCKS routines. Both sets of routines were developed as part of the ROCKS Crystallographic Computing System, but are now used more generally in the Laboratory of Biological Modelling. Accordingly, the information contained here has been kept application-independent. A separate User's Guide provides a description of the operation of the routines as seen by the user.

This document describes the facilities available in the C language implementation of the library. Because these routines were originally developed in a FORTRAN environment, some of the calls more closely resemble what might be expected by a FORTRAN programmer rather than a C programmer. In particular, an entire set of format editing routines is available that uses control strings resembling FORTRAN format codes. Versions of the standard C library output formatting routines **printf**, **sprintf**, etc. are also available that provide the expected '%' codes plus all the extended formatting facilities of the FORTRAN-like set, including array output, printing with page headers and subtitles, and many others. In the lower-level routines where efficiency is important, codes are defined as the sums of appropriate constants indicating the services requested. These constants are given names defined with preprocessor directives or enums in header files to make the calls more readable.

Some of the C routines operate on character strings delimited by the standard end-of-string character ('\0') while others require the string length as an argument. The two cases are distinguished in the descriptions of the individual functions.

PROGRAMMER'S GUIDE TO THE ROCKS ROUTINES

GENERAL PROGRAMMING STYLE

The following remarks are rather obvious in nature, but they bear repeating. Programs for general use must be as foolproof, and, nowadays, as secure, as possible. For every possible combination of input parameters, the program must either respond correctly or give an error message indicating that the input is illegal. The ROCKS library has numerous features to facilitate error checking (particularly for punctuation and numeric ranges) and generation of error messages. Input data should be screened carefully, especially for errors that would cause memory allocations to be exceeded. Such checking is very costly (or impossible) when performed by compiler-generated code at every array reference, but can be done easily by a combination of programmed checks of user input combined with dynamic allocations. Dynamic allocations should be used where possible to minimize any restrictions on the size of calculation that can be performed.

Since the development of the ROCKS system, interactive computing has become the norm. Nonetheless, complex applications generally read input from a control file rather than the keyboard to facilitate repeated runs. Applications should provide for both cases. The ROCKS library contains many routines oriented towards interpretation of control files. For historical reasons, lines in control files are referred to as “cards”. When an input error is found in interactive mode, the user should be prompted to reenter the erroneous data. When an error is found in offline mode, the remainder of the control file should be scanned for further errors before execution is terminated (See ERROR PROCEDURES).

As much as possible, applications should not depend on the exact order of the cards in the input file. The general idea is to read each card without format conversion, identify the card, retrieve individual fields with the scan routines, and finally convert numerical fields to binary with the conversion routines. Higher-level routines are provided to combine as many of these functions as possible in the more common cases.

ASSUMPTIONS

Familiarity with the material on input cards in the User's Guide is presumed.

It is assumed throughout that the type 'int' defines an integer variable with at least 16 bits and that 'long' defines one with at least 32 bits. The types 'byte' and 'schr' are declared in sysdef.h (our system-definition header, see below) to correspond to 'unsigned char' and 'signed char', respectively, as the built-in type 'char' is ambiguous. Types 'si16', 'ui16', 'si32', 'ui32', 'si64', and 'ui64' are defined in sysdef.h to declare, respectively, signed and unsigned 16-bit integers, signed and unsigned 32-bit integers, and signed and unsigned 64-bit integers. Types 'smed' and 'umed' define "medium-sized" integers that have 16 bits in 32-bit systems and 32 bits in 64-bit systems. Types 'sbig' and 'ubig' have 32 bits in 32-bit systems and 64 bits in 64-bit systems (this is usually the situation with 'long' but not always). The type 'wseed' defines a 64-bit random number seed that is compatible with 32-bit and 64-bit ROCKS random number generators and the type 'xyf' defines an x,y pair of floating-point coordinates.

Routines are provided to facilitate exchange of binary data between different processor types. These routines convert all data in interprocessor messages to/from "little-endian" order, but all data in binary files to/from "big-endian" order, and convert all floating point data to/from IEEE format if necessary. The programmer must replace all instances of type 'int' and 'long' with types from the family 'si16', 'ui16', ... 'ui64' as appropriate in multiprocessor applications.

The names of all the constants defined in rocks.h begin with "RK_". The user should avoid defining names starting with these letters.

In accord with standard C practice, routine names are written in lower case, except each routine name is given in upper case once at the point where it is described in full, to facilitate searches of the manual. Routine names are set in **boldface** in this manual to distinguish them from ordinary text.

HEADER FILES

Several header files are provided for use with the ROCKS routines. Some routines are prototyped in more than one header file. The header files are:

bapkg.h	Prototypes for bit and byte array functions.
itercirc.h	Prototypes and definitions for circle iteration.
itercyl.h	Prototypes and definitions for cylinder iteration.
iterpgfg.h	Prototypes and definitions for iterating over figures containing arbitrary collections of polygons, possibly concave, possibly overlapping, possibly with holes.
iterepgf.h	Extended version of routines in iterpfgf.h that can allow a border of fixed width around the figure boundary, inside or outside the polygons.
iterpoly.h	Prototypes and definitions for iterating over figures containing only nonoverlapping convex polygons.

iterrect.h	Prototypes and definitions for rectangle iteration, working out from the center of the figure.
iterroi.h	Prototypes and definitions for rectangle iteration, working in from the ULHC of the figure.
itershl.h	Prototypes and definitions for spherical shell iteration.
itershl2.h	Protptypes and definitions for iterating over two-dimension shells.
itersph.h	Prototypes and definitions for sphere iteration.
itertape.h	Prototypes and definitions for iterating over one or more rectangular "tapes," possibly slanted.
nelmeadd.h	Prototypes and definitions for performing Nelder-Mead optimization on double-precision simplexes.
nelmeadf.h	Prototypes and definitions for performing Nelder-Mead optimization on single-precision simplexes.
plots.h	Prototypes for plotting functions.
plotdefs.h	Defined constants used with plotting functions.
rfdef.h	Prototypes and structures for file I/O functions.
rkarith.h	Prototypes for basic arithmetic functions.
rkhash.h	Prototypes for hash-table management functions.
rkilst.h	Prototypes for iteration list functions.
rkprintf.h	Prototypes and definitions used internally by **printf**-emulating routines.
rksubs.h	Prototypes for subset routines that do not use the card and page interface routines.
rkxtra.h	Prototypes for low-level character I/O functions.
rocks.h	Prototypes and constants for general formatting and parsing functions.
rockv.h	Prototypes and structues used internally by other ROCKS routines (not directly accessed by user).
swap.h	Prototypes for routines to move data to or from standard binary files or interprocessor messages.
sysdef.h	System-dependent definitions.

File sysdef.h

The header file "sysdef.h" is the standard means of specifying compiler-, operating system-, or architecture-specific information in ROCKS programs. A single copy of sysdef.h contains information for all supported systems. The system used for a particular compilation is specified by including a -D (or equivalent) compile-time definition. The system names recognized by sysdef.h and the implied architectures, operating systems, and compilers are given in the following table:

System Name	Architecture	Operating System	Compiler
IBMMVS	IBM370	MVS	IBMCC
IBMVM	IBM370	VM	IBMCC
OSXGCC	PPCG4	OSX,UNIX	GCC
OSXMWCC	PPCG4	OSX,UNIX	MWCC
PCLINUX	INTEL	LINUX,UNIX,BSD	GCC
PCLUX64	INTEL	LINUX,UNIX,BSD	GCC
SUN4	SPARC	SUNOS	ACC
SUN5	SPARC	SOLARIS,UNIX,SVR4	SWSCC
XP8I	INMOS8	SUNOS	ICC

Architecture-dependent definitions in sysdef.h include such things as whether numbers are "big-endian" (BYTE_ORDRE = +1) or "little-endian" (BYTE_ORDRE = -1) ["ORDER" is misspelled deliberately to avoid a conflict with a variable defined in some IRIX header files], and alignment (ALIGN_TYPE is TRUE if address alignment on a multiple of the item size is required or advised for performance reasons, BYTE_ALIGN is the size of the optimal alignment unit, STRUCT_ALIGN is the size of the alignment unit imposed on structures by the compiler, if any). The largest signed and unsigned byte (SCHR_MAX and BYTE_MAX), signed and unsigned short integer (SHRT_MAX and UI16_MAX), signed and unsigned medium integer (SMED_MAX and UMED_MAX), signed integer (INT_MAX), signed and unsigned 32-bit integer (SI32_MAX and UI32_MAX), signed and unsigned big integer (SBIG_MAX and UBIG_MAX), and signed long integer (LONG_MAX) are given. Also given are definitions of sign bits for signed and unsigned 32-bit integers (SI32_SGN and UI32_SGN) signed longs (LONG_SGN), and signed 64-bit integers (SI64_SGN). The number of decimal digits required to represent the largest long integer (LONG_SIZE) and the largest 64-bit integer (WIDE_SIZE) are given. Floating point numbers are characterized by the largest decimal exponent representable in a single-precision (FLT_EXP_MAX) or double-precision (DBL_EXP_MAX) number, the number of decimal digits needed to represent a double-precision number to full accuracy (OUT_SIZE), and the number of decimal digits needed to represent the exponent of the largest double-precision number (EXP_SIZE). Operating-system-dependent definitions include the length of the longest possible file name (LFILNM) and the length of the longest possible input (CDSIZE) and output (LNSIZE) lines. In addition, a few often-used constants (YES, NO, TRUE, FALSE, ON, OFF with the expected definitions 1,0,1,0,1,0, respectively) and routines (**abexit**, **abexitm**, **abexitme**, **abexitmq**, **abexitq**, **ssprintf**) are prototyped in sysdef.h. Also included are a typedef for **byte** and standard macros for **max** and **min**. Other useful macros are:

SRA(n,s) shifts the integer 'n' through 's' bits to the right, propagating the sign of negative numbers whether or

```
            not the compiler performs this action by default for
            the '>>' operator.
ALIGN_UP(s) aligns a length or pointer 's' to the next larger
            multiple of the memory size defined by BYTE_ALIGN,
            which is defined as the smallest unit of memory which
            can be accessed with no performance penalty.
abs32(x), abs64(x)  takes the absolute value of 'x', where 'x' is
            an si32, respectively si64, variable.
labs(x), llabs(x)   takes the absolute value of 'x', where 'x' is
            a long, respectively long long, variable, providing an
            equivalent function on machines where this function is
            not in the standard C library.
```

The listing of sysdef.h should be consulted for further information.

STATIC GLOBAL VARIABLES

Some of the headers contain definitions of static data objects which must be instantiated only once in each application program. This is done by defining the preprocessor variable MAIN in the main program before including ROCKS headers, as shown here:

```
#define MAIN
#include "rocks.h"      /* Brings in rockv.h and filedef.h */
```

A single global structure RK defined in rocks.h contains the variables needed for communication between user applications and the interface routines. These variables are initialized at load time to appropriate values--no other initialization is currently required. The RK structure is defined as follows:

```
    struct {
        char * last;        /* Pointer to last card read */
        ui32  mcbits;       /* Bits returned by last mcodes */
        ui32  erscan;       /* Current error flags */
        short mckpm;        /* Kind of action of last mcodes */
        short highrc;       /* Highest return code */
        short iexit;        /* Cumulative error flags */
        short scancode;     /* Code returned by last scan */
        short plevel;       /* Parentheses nesting level */
        short length;       /* Length of last field minus 1 */
        short numcnv;       /* Number of conversions performed */
        short pglns;        /* Number of lines printed per page */
        short pgcls;        /* Columns per print line */
        short ttcls;        /* Columns per terminal line */
        short rmlns;        /* Remaining lines on current page */
        short rmcls;        /* Remaining cols. in current line */
        short pgno;         /* Current page number */
        byte  expwid;       /* Width of exponent in bcdout() */
```

```
        byte  bssel;        /* Binary scale selector */
        } RK;
```

Detailed instructions for the use of these variables are provided in the writeups of the relevant function calls in this manual. Initially, variable 'pglns' is set to 60 and 'pgcls' and 'ttcls' are set to the value of 'LNSIZE' from sysdef.h (normally 132). These values may be changed if desired before the first **cryout** call. Variables 'scancode', 'plevel', 'length', and 'rmcls' must not be changed by the user. 'rmlns' may be set to -1 to force a new page on the next **cryout** call. Note that 'length' is defined as one less than the length of the last field processed by **scan**, **bcdout**, **ibcdwt**, **ubcdwt**, or **wbcdwt**; this is compatible with the FORTRAN ROCKS definition and is also equal to the C subscript of the last character in the result field.

ERROR PROCEDURES

The ROCKS library was developed in an IBM MVS environment where two types of program exits with different systems of return codes were used: an “abend” exit indicated a terminal error, usually generated by a system call, while applications usually returned a “condition code” that could be queried at the command language level to determine whether additional job steps should be executed. In UNIX-like (e.g. Linux) systems, this distinction does not exist; a program can only return a single integer return code, historically restricted to the range 0 to 255.

The ROCKS library retains the capability to generate either of the two types of returns; in a non-MVS environment, both lead to the return of the same type of UNIX exit code and so generally only the abend codes are generated by library routines. Abend (here called “abexit”) codes are supposed to be unique across all programs in the lab and are assigned in blocks to different applications. A command-line program called “abend” is provided that allows a user to get information about any abexit code that may be encountered. Programmers must obtain unused abexit blocks from GNR and must provide documentation to GNR for insertion in the abend database when new abexit codes are created. ROCKS library routine abexit codes are kept below 255, while application codes may be larger. Codes larger than 255 generate appropriate error messages, but the parent shell program will only receive the code modulo 256.

The global variable RK.iexit is used to keep track of errors that the application wishes to hold pending prior to termination of execution. These are errors that do not require immediate termination; they would traditionally lead to generation of a “condition code”. The application may define various bits of

RK.iexit to have any desired meanings; the interface routines set the low order bit (the 1 bit) when a control card error is detected. The application may rezero RK.iexit whenever it wishes, for example, if an error is deemed noncritical. Typically, when all control cards have been interpreted, RK.iexit is checked and the application exits if it is nonzero. In this way, multiple input errors can be detected in a single pass through the input file.

The global variable RK.highrc is provided to record the highest return code from any routine or procedure that may wish to use this service; when the application determines that it can no longer proceed because of errors, the value of RK.highrc is returned to the operating system as the "condition code" (IBM) or return code (other operating systems) of the job step.

By convention, the following return codes are used:

0 Successful execution.
4 An error occurred which probably did not affect the integrity of any file.
8 An error occurred which prevented an output file from being prepared or updated as requested; input files are probably preserved.
12 Terminal error; files may have been destroyed.
16 FORTRAN or C library error.

For ROCKS library programmers, errors occurring during parsing of input files are handled by routine **ermark**. The errors indicated in each call to **ermark** are held pending and are printed and cleared the next time **cryout** is called. This procedure assures that only one copy of a given message is produced even for multiple occurrences of the same error on any one control card. In addition to registering errors, **ermark** generates a marking symbol under the current scan column in the printed output and sets the 1 bit in RK.iexit.

Generating an **abexit** code

The functions **abexit**, **abexitm**, **abexitme**, **abexitmq**, and **abexitq** terminate execution and provide a return code, or abexit code, to the operating system. All three are similar to the standard C library function **exit** except for printing an error message. **abexit** prints only the error code. It is usually used to handle unexpected programming anomalies where additional explanation would not be helpful to the user. **abexitm** additionally prints an explanatory message supplied by the caller. **abexitme** prints the caller's message plus the value of the system variable **errno** under operating systems that have this variable. **abexitq** and **abexitmq** are respectively the same as **abexit** and **abexitm** except for the added feature of returning to the caller if e64qtest() indicates this is a test run. The reason for separating this function from traditional abexit() is that abexit() is now declared to be non-returning and that attribute is useful for optimization of codes that can never be tests. Tests that use printf() instead of cryout() need to make their own version of abexitq() as well as of abexit().

USE OF ROCKS ROUTINES OUTSIDE THE ROCKS ENVIRONMENT

It is expected that some applications will want to use some of the ROCKS mathematical or string manipulation routines but not the card and page I/O interface, and others, for example, MATLAB "mex" routines, will not be able to use the interface routines for environmental reasons. Other applications may require special clean-up procedures when an error exit occurs. These applications can provide their own versions of **abexit[q][m][e]** in order to meet their special requirements. To facilitate this usage, ROCKS routines that are not themselves part of the card and page interface package always use **abexit[m]** or **abexitme** for error printing, avoiding direct use of **cryout**. However, the default **abexit[m][e]** routines supplied with the library themselves call **cryout** to print error messages, and user-written **abexit[q][m][e]** routines may need to change this behavior. They also must use **malloc** etc. and not **mallocv**, etc. to allocate memory, because the "verbose" routines themselves call **abexitm** on failure. On the other hand, all the other routines use **mallocv**, etc. in order to provide a point where memory allocation calls can be intercepted and replaced with user versions. When this is done, **abexit[m][e]** must also call the user-supplied memory allocation routines, if any.

The routines that can be used without linking any of the ROCKS input/output routines are prototyped in rksubs.h and listed at the end of this document. These include **ssprintf**, which may be used to generate output, including error messages for **abexitm[e]**, using a small subset of **sprintf** features. Similarly, **sibcdin** may be used to input integers using a small subset of **scan** and **ibcdin** features.

FILE HANDLING

The routines described in this manual should be used to process all control card input, printed output, and SPOUT (“supplementary printed output”, described later). These data streams correspond to 'stdin', 'stdout', and 'stderr', respectively. These data streams do not require explicit opening or closing by the user.

For processing other files, the ROCKS library provides a standard file definition structure (rfdef) and a set of routines (**rfallo**, **rfopen**, **rfdups**, **rfprintf**, **rfqsame**, **rfread**, **rfgets**, **rfseek**, **rftell**, **rfwrite**, **rfflush**, and **rfclose**) that allow greater control over processing options and a greater degree of operating system independence than the standard C I/O library functions **fopen**, etc. These routines can transparently deal with UNIX or IBM MVS file systems and provide an easy interface to internet sockets. In addition, these routines provide transparent local buffering on systems where this improves performance. Routines **rfri2**, **rfri4**, **rfri8**, **rfru8**, **rfrr4**, **rfrr8**, **rfwi2**, **rfwi4**, **rfwi8**, **rfwu8**, **rfwr4**, and **rfwr8** read or write single short, 32-bit, signed or unsigned 64-bit, float, or double data items from or to their standard forms in binary data files. The corresponding routines for interprocessor messaging may be found in the 'NSITOOLS' library of functions for parallel computing. Data conversion without I/O is performed by routines **bemfmi2**, **bemfmi4**, **bemfmi8**, **bemfmu8**, **bemfmr4**, **bemfmr8**, **bemtoi2**, **bemtoi4**, **bemtoi8**, **bemtou8**, **bemtor4**, **bemtor8**, **lemfmi2**, **lemfmi4**, **lemfmi8**, **lemfmu8**, **lemfmr4**, **lemfmr8**, **lemtoi2**, **lemtoi4**, **lemtoi8**, **lemtou8**, **lemtor4**, and **lemtor8**. Some of these are implemented as macros on most systems.

CONTROL CARD INPUT

Input cards are either fixed format or scanned. At the highest level, fixed format cards are processed by **inform**. Scanned cards are processed by **inform** for positional variables and **kwscan** for keyword variables. Iteration lists are interpreted by **ilstread** and checked by **ilstchk**. Keywords are matched by **match** or **smatch** and option codes by **mcodes**. When more detailed control of the input process is necessary, separate lower-level routines for card input (**cryin**), printing (**cdprnt**, **cdprt1**), scanning (**cdscan**, **scan**), keyword matching (**ssmatch**), and numerical conversion (**bcdin**, **wbcdin**, and the obsolete routines **ibcdin** and **ubcdin**) may be used. Cards can be "pushed back" for later processing by calling **rdagn** and single fields can be pushed back with **scanagn**. An error can be generated if a card has too many data fields by calling **thatsall** and excess fields can be skipped over by calling **skip2end**. The input file is changed with **cdunit**. The interactive prompt is changed with **sprmpt**. Whitespace cards can be detected by calling **qwhite** and accepted as comments by calling **accwac**. The application can determine whether input is from an online terminal by calling **qonlin**. Input cards should always be read by **cryin** to assure that EXECUTE statements (see User’s Guide) and other special control cards, comments, and continuation cards are handled properly.

PRINTED OUTPUT

Formatted output can be prepared either using the FORTRAN-format-like facilities of **convrt** or the C-like facilities of the extended version of **printf.** This routine has the same name as the standard libc version of **printf** so that output from library functions will be routed through the **cryout** mechanism. In order to allow the programmer to choose whether the standard **printf** or the ROCKS version is used, **printf** is placed in a separate library (librkpf.a) from the rest of the ROCKS library. This library also contains ROCKS-compatible implementations of standard C library functions **fprintf, puts**, **snprintf**, and **sprintf,** along with the added routine **rfprintf** that allows **printf**-formatted output to be written to an rfdef-defined file.

The interface functions **printf** and **convrt** provide the standard ways to prepare formatted output for printing. They in turn call the lower-level functions **bcdout** and **wbcdwt** for numerical conversions and **cryout** for printing character strings. Binary codes can be formatted for printing with **mcodprt.** Function **cryout**, originally written for use with offline printers, generates a title at the top of each page containing the name of the program or calculation, user information from a TITLE card, the date, the CPU time elapsed since the start of the job, and the page number. Under the title is a subtitle which can contain up to 408 characters in any number of lines. The title can be changed by calling **settit**; the subtitle is changed by special calls to **printf, convrt**, or **cryout.** Both titles are automatically inserted when any of these routines is used for printing. Function **tlines** is used to override the automatic line counting in **convrt.** Selected output to time sharing terminals is generated by calling **spout.** Pagination is suspended by calling **nopage.**

Each line of output is assigned a printing priority by the programmer (see printf, convrt, and **cryout** function descriptions). The user selects the priority level that he/she wishes to appear in the output (OUTPRTY card); lower priority output is discarded.

The following conventions apply to printed output:

1) Normal printed output goes to 'stdout' and SPOUT output goes to 'stderr'.

2) Control cards are listed with single (**cdprt1**) or double (**cdprnt**) spacing and three blanks at the beginning of each line.

3) Continuation cards are single spaced and preceded by three blanks, giving an indented effect when the leading whitespace is considered (**cryin**).

4) Comment cards are listed with double spacing before the first comment in a group and single spacing thereafter. There are seven blanks before the text of each comment card (**cryin**).

5) Error messages are double spaced and preceded by "***". Warnings are double-spaced and preceded by "-->" (must be done by application program). Errors and warnings are automatically "spouted".

FUNCTION DESCRIPTIONS

The remainder of this document describes the individual ROCKS interface functions. (If any description here differs from that given in the C code for a given routine, the documentation in the source code should be considered authoritative.) The descriptions are grouped according to similarities of function, with the simpler ones first. For most purposes, the "higher level" routines such as **inform** and **convrt** will be found most convenient.

All of the input/output subroutines work through the standard C object-time I/O system and calls to the ROCKS I/O routines can sometimes be intermixed with ordinary C calls. However, the ROCKS routines may use **read**/**write** on some systems and **fread**/**fwrite** on others. In addition, they perform certain functions that are not performed by the standard library routines. The following points should be given particular attention when considering the use of standard library calls for I/O functions:

(1) Subroutine **cryin** should always be used to read control cards in order that online prompts, comment cards, TITLE, SPOUT, OUTPRTY, ERROR DUMP REQUESTED, EXECUTE, END, and QUIT cards, and continuation cards may be properly processed.

(2) Printed output with **cryout** or **convrt** or the ROCKS version of **printf** provides automatic pagination and printing of subtitles. The ROCKS routines also provide automatic use of exponential format when output fields overflow. SPOUT output can only be generated with **cryout** or **convrt**.

(3) **cryout** may hold output pending which would appear at the wrong point if **cryout** and libc **printf** etc. calls are intermixed. To avoid this problem, use the ROCKS **printf** or use the RK_FLUSH code with the last **cryout** call before libc **printf** is used. Use **lines** to provide pagination and subtitles when output is generated with C library functions.

(4) Calls to **rfread**, etc. and **fread**, etc. should not be intermixed, as the ROCKS routines may provide internal buffering.

(5) For automatic portability of binary data between unlike processors, routines **rfwi2**, etc. should be used to write the data and the corresponding routine **rfri2**, etc. should be used to read each item back in. These routines provide standard external representations for binary numbers that are portable across different machine architectures.

MEMORY MANAGEMENT EXTENSIONS

Functions **mallocv**, **callocv**, **reallocv**, and **freev** are similar to the standard library routines **malloc**, **calloc**, **realloc**, and **free**, except that they provide for generating an error message and terminating execution (via **abexitm**) when the request cannot be satisfied. A further set of routines is provided on parallel computers to allocate memory in shared memory pools. These are located in the separate "nsitools" library. On systems where shared allocation is not supported, these functions are equated to the corresponding standard functions by macros.

Function **MALLOCV**

Function **mallocv** allocates a memory block. Execution is terminated with an error message if allocation fails.

Usage: void ***mallocv**(size_t length, char *msg)

Prototyped in: rocks.h, rksubs.h

Arguments: 'length' is the length of the memory to be allocated, in bytes.

'msg' is a word or phrase of up to 48 characters describing the nature of the memory to be allocated. If allocation fails, this text is appended to a generic error message of the form "Memory alloc failed for: ". The abexit code for this error is 32.

Value returned: Pointer to the allocated storage. If allocation fails, **mallocv** does not return.

Function **CALLOCV**

Function **callocv** allocates an array of memory blocks and clears all of the allocated memory to 0's. Execution is terminated with an error message if allocation fails.

Usage: void ***callocv**(size_t n, size_t size, char *msg)

Prototyped in: rocks.h, rksubs.h

Arguments: 'n' is the number of items to be allocated.

'size' is the size of each item, in bytes.

'msg' is a word or phrase of up to 48 characters describing the nature of the memory to be allocated. If allocation fails, this text is appended to a generic error message of the form "Memory alloc failed for: ". The abexit code for this error is 32.

Value returned: Pointer to the allocated storage. If allocation fails, **callocv** does not return.

Function **REALLOCV**

Function **reallocv** changes the size of an allocated memory block. If the block is moved, the old contents are copied to the new location. Execution is terminated with an error message if reallocation fails.

Usage: void *__reallocv__(void *ptr, size_t length, char *msg)

Prototyped in: rocks.h, rksubs.h

Arguments: 'ptr' is a pointer to the existing memory block, which must have been previously allocated with **malloc**, **calloc**, etc.

'length' is the new length of the memory block in bytes.

'msg' is a word or phrase of up to 48 characters describing the nature of the memory to be allocated. If reallocation fails, this text is appended to a generic error message of the form "Memory realloc failed for: ". The abexit code for this error is 32.

Value returned: Pointer to the allocated storage. If reallocation fails, **reallocv** does not return.

Warning: There is no way that **reallocv** can update pointers inside or outside the allocated block that may point to information contained within the block. Avoid constructing pointers that point to locations within a memory block that may be subject to reallocation, other than the one returned by the routine.

Subroutine **FREEV**

Subroutine **freev** releases a block of memory previously allocated with **malloc**, **calloc**, **realloc**, **mallocd**, **mallocv**, etc.

Usage: void **freev**(void *freeme, char *msg)

Prototyped in: rocks.h, rksubs.h

Arguments: 'freeme' is a pointer to a previously allocated memory block.

'msg' is a word or phrase of up to 48 characters describing the nature of the memory block to be freed. If deallocation fails, this text is appended to a generic error message of the form "Attempt to free unallocated memory for: <msg>". The abexit code for this error is 33.

BYTE AND BIT-STRING MANIPULATION ROUTINES

These routines permit the user to set, clear, 'or', or test individual bits in bit arrays of any length, regardless of word or byte boundaries. Logical operations (copy, and, or, exclusive or) are provided for byte arrays of general length. In addition, function **strnlen** is provided as an extension of the standard library function **strlen**, to determine the length of a text string that may not be terminated with a null character; and **strncpy0** copies a string up to a given maximum length, but then appends a terminating '\0' every time (possibly modifyin n+1 characters).

Bit strings of any length not longer than a 32-bit word can be stored into (**bitpack**) and retrieved from (**bitunpk**) any offset in a bit array. The location and size of the bit array along with certain working information are kept in a **BITPKDEF** struct, which must be initialized by a call to **setbpack** or **setbunpk**, respectively. It is undefined to the user in what order the bit strings are packed (in fact, it is high order first) and the user should not access information in the BITPKDEF structure.

In the case of the byte-oriented routines (those whose names begin with 'ba' or 'byt'), the offsets and lengths are in bytes on machines which have a byte structure, otherwise in units of characters or eight bits, whichever is greater. In the following descriptions, the term 'byte' should be understood as referring to these units.

Subroutine **BITCLR**

Subroutine **bitclr** clears a single bit at a specified position in an array of bits.

Usage: void **bitclr**(unsigned char *array, long bit)

Prototyped in: bapkg.h

Arguments: 'array' is a pointer to the bit array in which the bit to be cleared is located.

'bit' is the number of the bit to be cleared (leftmost = 1).

Subroutine **BITCMP**

Subroutine **bitcmp** complements (inverts) a single bit at a specified position in an array of bits.

Usage: void **bitcmp**(unsigned char *array, long bit)

Prototyped in: bapkg.h

Arguments: 'array' is a pointer to the bit array in which the bit to be complemented is located.

'bit' is the number of the bit to be complemented (leftmost = 1).

Subroutine **BITIOR**

Subroutine **bitior** computes the logical 'OR' function of two bit strings aligned arbitrarily in memory. Bits shifted to the right always enter the byte at the next higher address, regardless of whether the machine is big-endian or little-endian. Bits outside the specified length are not affected, even if in the same byte as part of the result string.

Usage: void bitior(byte *t1, int jt, byte *s1, int js, int len)

Prototyped in: bapkg.h

Arguments: 't1' is the address of the target (first operand and result) bit array.

'jt' is the offset of the first bit in the target from the leftmost bit of the byte at t1, counting from 0. May exceed the size of one byte.

's1' is the address of the source (second operand) bit array.

'js' is the offset of the first bit in the source from the leftmost bit of the byte at s1, counting from 0. May exceed the size of one byte.

'len' is the length of the source and target bit arrays in bits.

Caution: Bit array offsets in **bitior** are counted from 0, but are counted from 1 in the other routines in this family, e.g. bitset(), etc.

Subroutine **SETBPACK**

Subroutine **setbpack** prepares a **BITPKDEF** data structure for use by the **bitpack** subroutine. Storage for the **BITPKDEF** must be provided by the user and left untouched until the last following call to **bitpack**.

Usage: void setbpack(struct BITPKDEF *pbpd, void *pbits, size_t npbpd, long io1)

Prototyped in: bapkg.h

Arguments: 'pbpd' is a pointer to a BITPKDEF structure where state information can be stored between **bitpack** calls.

'pbits' is a pointer to an array where bits can be stored.

'npbpd' is the number of bytes available in the pbits array.

'io1' is the bit offset in the pbits array where storage should start, counting from 0 at the start of the array.

Error procedures: If the starting position is beyond the end of the specified array, abnormal termination occurs with abend code 89.

Caution: Bit array offsets in **BITPKDEF**s are counted from 0, but are counted from 1 in the other routines in this family, e.g. bitset(), etc.

Function **BITPACK**

Function **bitpack** stores an arbitrary number of bits up to the length of a 32-bit word in a previously prepared data string. The initial bit offset in the string is given by a preliminary call to **setbpack.** Subsequent calls to **bitpack** result in storing additional data following the last data item already stored. Unused bits in the rightmost byte of stored data will be set to zero, but no data will be stored beyond the length of the bit array specified in the **setbpack** call.

Usage: size_t bitpack(struct BITPKDEF *pbpd, long item, int nbits)

Prototyped in: bapkg.h

Arguments: 'pbpd' is a pointer to a BITPKDEF structure that has been initialized by a previous call to **setbpack.**

'item' is a long word containing the data to be stored in its low-order bits.

'nbits' is the number of bits to be stored.

Value returned: Number of bits that were not stored because the item extended beyond the specified length of the bit array after part of the data were stored. On a normal call, this value is zero.

Error procedures: If the starting position is beyond the end of the specified array on entry, abnormal termination occurs with abend code 88.

Subroutine **SETBUNPK**

Subroutine **setbunpk** prepares a **BITPKDEF** data structure for use by the **bitunpk** subroutine. Storage for the **BITPKDEF** must be provided by the user and left untouched until the last following call to **bitunpk.**

Usage: void setbunpk(struct BITPKDEF *pbpd, void *pbits, size_t npbpd, long io1)

Prototyped in: bapkg.h

Arguments: 'pbpd' is a pointer to a BITPKDEF structure where state information can be stored between **bitunpk** calls.

'pbits' is a pointer to an array where bits are stored.

'npbpd' is the number of bytes available in the pbits array.

'io1' is the bit offset in the pbits array where storage starts, counting from 0 at the start of the array.

Error procedures: If the starting position is beyond the end of the specified array, abnormal termination occurs with abend code 89.

Caution: Bit array offsets in **BITPKDEF**s are counted from 0, but are counted from 1 in the other routines in this family, e.g. bitset(), etc.

Subroutine **BITUNPK**

Function **bitunpk** retrieves an arbitrary number of bits up to the length of a long word (usually 32 bits) from data stored in a previously prepared data string. The initial bit offset in the string is given by a preliminary call to **setbunpk.** Subsequent calls to **bitunpk** result in retrieving additional data following the last data item already retrieved.

Usage: long bitunpk(struct BITPKDEF *pbpd, int nbits)

Prototyped in: bapkg.h

Arguments: 'pbpd' is a pointer to a BITPKDEF structure that has been initialized by a previous call to **setbunpk.**

'nbits' is the number of bits to be stored.

Value returned: The requested data bits, packed into the low-order bits of a long word. The high-order bits are zeros.

Error procedures: If the requested item would require fetching bits from beyond the end of the specified bit array, abnormal termination occurs with abend code 88.

Subroutine **BITSET**

Subroutine **bitset** sets a single bit at a specified position in an array of bits.

Usage: void **bitset**(unsigned char *array, long bit)

Prototyped in: bapkg.h

Arguments: 'array' is a pointer to the bit array in which the bit to be set is located.

'bit' is the number of the bit to be set (leftmost = 1).

Function **BITTST**

Function **bittst** may be used to test a single bit at a specified position in a bit array.

Usage: int **bittst**(unsigned char *array, long bit)

Prototyped in: bapkg.h

Arguments: 'array' is a pointer to the array in which the bit to be tested is located.

'bit' is the number of the bit to be tested (leftmost = 1).

Return values: 0 if bit is 0, 1 if bit is 1.

Function **BITCNT**

Function **bitcnt** may be used to count the number of bits set to 1 in a byte array.

Usage: long **bitcnt**(unsigned char *array, long bytlen)

Prototyped in: bapkg.h

Arguments: 'array' is a pointer to the array in which the set bits are to be counted.

'bytlen' is the length of 'array' in bytes.

Return value: Number of bits set to 1 in the specified array.

Subroutine **BYTMOV**

Subroutine **bytmov** may be used to copy a string of bytes from a source array to a target array. It is similar to subroutine **bamove** except that it does not have displacement arguments.

Usage: void **bytmov**(char *array1, long bytlen, char *array2)

Prototyped in: bapkg.h

Arguments: 'array1' is the location of the target array.

'bytlen' is the number of bytes to be copied.

'array2' is the location of the source array.

Subroutine **BYTIOR**

Subroutine **bytior** may be used to perform the logical inclusive 'or' operation on two byte strings of arbitrary length.

Usage: void **bytior**(char *array1, long bytlen, char *array2)

Prototyped in: bapkg.h

Arguments: 'array1' is a pointer to the first operand. This string is replaced by the result.

'bytlen' is the number of bytes in each operand.

'array2' is a pointer to the second operand. This string is left unchanged by the operation.

Subroutine **BYTAND**

Subroutine **bytand** may be used to perform the logical 'and' operation on two byte strings of arbitrary length.

Usage: void **bytand**(char *array1, long bytlen, char *array2)

Prototyped in: bapkg.h

Arguments: 'array1' is a pointer to the first operand. This string is replaced by the result.

'bytlen' is the number of bytes in each operand.

'array2' is a pointer to the second operand. This string is left unchanged by the operation.

Subroutine **BYTNXR**

Subroutine **bytnxr** may be used to perform the logical 'not exclusive or' operation on two byte strings of arbitrary length (returns the complement of the exclusive or function).

Usage: void **bytnxr**(char *array1, long bytlen, char *array2)

Prototyped in: bapkg.h

Arguments: 'array1' is a pointer to the first operand. This string is replaced by the result.

'bytlen' is the number of bytes in each operand.

'array2' is a pointer to the second operand. This string is left unchanged by the operation.

Subroutine **BYTXOR**

Subroutine **bytxor** may be used to perform the logical exclusive 'or' operation on two byte strings of arbitrary length.

Usage: void **bytxor**(char *array1, long bytlen, char *array2)

Prototyped in: bapkg.h

Arguments: 'array1' is a pointer to the first operand. This string is replaced by the result.

'bytlen' is the number of bytes in each operand.

'array2' is a pointer to the second operand. This string is left unchanged by the operation.

Function **STRNLEN**

Function **strnlen** returns the length of a character string, subject to a specified maximum length. This function should be used instead of **strlen** when it is known that the argument string has a fixed maximum length and is not terminated by a NULL character when that length is reached. (**strnlen** bears the same relationship to **strlen** as **strncpy** bears to **strcpy**.)

Usage: size_t **strnlen**(const char *s, size_t mxl)

Prototyped in: rocks.h, rksubs.h

Arguments: 's' is the string whose length is to be determined.

'mxl' is the maximum length that string 's' may have.

Value returned: Length of string 's'. Number of non-null characters encountered before the first null character in the first 'mxl' characters beginning at 's'. If no null characters are found, 'mxl' is returned.

Note: This function exists in some C libraries but not others. The ROCKS definition is the same, so no special provision should be required in makefiles to deal with one or the other.

Function **STRNCPY0**

Function **strncpy0** copies a character string up a given maximum length, then appends a string-ending null character to the result, which **strncpy** does not do when the max length is reached. The intended use is for cases where a receiving array of size one greater than the max length is known to exist, but the source may be longer. Use of the standard **strncpy** in this case can produce strings with garbage on the end.

Usage: void **strncpy0**(char *d, const char *s, size_t mxl)

Prototyped in: rocks.h, rksubs.h

Arguments: 'd' is the destination string. It should have space for 'mxl'+1 characters.

's' is the source string.

'mxl' is the maximum length that string 's' may have.

BINARY DATA BUFFERING ROUTINES

The routines are used to move binary data items to and from memory buffers. Items are stored in a standard form that allows exchange between unlike processor types. Storage is independent of any memory alignment requirements for variables of a particular type in a particular processor. With these routines, the user can control whether storage is big-endian or little-endian for those situations in which the ROCKS standards (little-endian for interprocessor messages, big-endian for binary data files) for some reason cannot be followed. Generally, the I/O routines **rfri2**, etc. should be used in preference to these routines, as they work directly with I/O system buffers, avoiding the need for extra intermediate buffering of data items.

Subroutine **BEMFMI2**

This function stores a short integer into a big-endian buffer. Two eight-bit bytes are stored, regardless of the native length of a short integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **bemfmi2**(char *m, short i2)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'i2' is the short data item to be stored. It may be an expression.

Value returned: None.

Subroutine **BEMFMI4**

This function stores a 32-bit signed or unsigned integer into a big-endian buffer. Four eight-bit bytes are stored, regardless of the native length of a long integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **bemfmi4**(char *m, si32 i4)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'i4' is the 32-bit data item to be stored. It may be an expression.

Value returned: None.

Subroutine **BEMFMI8**

This function stores a 64-bit signed integer into a big-endian buffer. Eight eight-bit bytes are stored, regardless of the native length of a long or long long integer on the system where it is executed.

Usage: void **bemfmi8**(char *m, si64 i8)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored.

'i8' is the signed 64-bit data item to be stored. It may be an expression.

Value returned: None.

Note: Because 64-bit integers are implemented as structures on some systems, it is necessary to have separate routines for signed and unsigned 64-bit integers--see **bemfmu8** below.

Subroutine **BEMFMU8**

This function stores a 64-bit unsigned integer into a big-endian buffer. Eight eight-bit bytes are stored, regardless of the native length of a long or long long integer on the system where it is executed.

Usage: void **bemfmu8**(char *m, ui64 u8)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored.

'u8' is the ui64 data item to be stored. It may be an expression.

Value returned: None.

Subroutine **BEMFMR4**

This function stores a single-precision floating point value into a big-endian buffer. Four eight-bit bytes in the format of an IEEE-standard floating point number are stored, regardless of the native format and length of a float on the system where it is executed. This function may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **bemfmr4**(char *m, float r4)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'r4' is the float data item to be stored. It may be an expression.

Value returned: None.

Subroutine **BEMFMR8**

This function stores a double-precision floating point value into a big-endian buffer. Eight eight-bit bytes in the format of an IEEE-standard double-precision floating point number are stored, regardless of the native format and length of a double on the system where it is executed. This function may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **bemfmr8**(char *m, float r8)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'r8' is the double data item to be stored. It may be an expression.

Value returned: None.

Function **BEMTOI2**

This function retrieves a short integer from a big-endian buffer. Two eight-bit bytes are consumed, regardless of the native length of a short integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: short **bemtoi2**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The short data item at 'm', converted to the native format of the system where the program is running.

BINARY DATA BUFFERING ROUTINES

Function **BEMTOI4**

This function retrieves a 32-bit integer from a big-endian buffer. Four eight-bit bytes are consumed, regardless of the native length of a long integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: si32 **bemtoi4**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The 32-bit data item at 'm', converted to the native format of the system where the program is running.

Function **BEMTOI8**

This function retrieves a signed 64-bit integer from a big-endian buffer. Eight eight-bit bytes are consumed, regardless of the native length of a long or long long integer on the system where it is executed.

Usage: si64 **bemtoi8**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored.

Value returned: The si64 data item at 'm', converted to the native format of the system where the program is running.

Note: Because 64-bit integers are implemented as structures on some systems, it is necessary to have separate routines for signed and unsigned 64-bit data--see **bemtou8** below.

Function **BEMTOU8**

This function retrieves an unsigned 64-bit integer from a big-endian buffer. Eight eight-bit bytes are consumed, regardless of the native length of a long or a long long integer on the system where it is executed.

Usage: ui64 **bemtou8**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored.

Value returned: The ui64 data item at 'm', converted to the native format of the system where the program is running.

Function **BEMTOR4**

This function retrieves a single-precision floating-point variable from a big-endian buffer. Four eight-bit bytes are consumed, regardless of the native length of a float on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: float **bemtor4**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The single-precision IEEE floating-point data item at 'm', converted to the native format of the system where the program is running.

Function **BEMTOR8**

This function retrieves a double-precision floating-point variable from a big-endian buffer. Eight eight-bit bytes are consumed, regardless of the native length of a double on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: double **bemtor8**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The double-precision IEEE floating-point data item at 'm', converted to the native format of the system where the program is running.

Subroutine **LEMFMI2**

This function stores a short integer into a little-endian buffer. Two eight-bit bytes are stored, regardless of the native length of a short integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **lemfmi2**(char *m, short i2)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'i2' is the short data item to be stored. It may be an expression.

Value returned: None.

Subroutine **LEMFMI4**

This function stores a signed or unsigned 32-bit integer into a little-endian buffer. Four eight-bit bytes are stored, regardless of the native length of a long integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **lemfmi4**(char *m, si32 i4)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'i4' is the 32-bit data item to be stored. It may be an expression.

Value returned: None.

Subroutine **LEMFMI8**

This function stores a signed 64-bit integer into a little-endian buffer. Eight eight-bit bytes are stored, regardless of the native length of a long or a long long integer on the system where it is executed.

Usage: void **lemfmi8**(char *m, si64 i8)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored.

'i8' is the signed 64-bit data item to be stored. It may be an expression.

Value returned: None.

Subroutine **LEMFMU8**

This function stores an unsigned 64-bit integer into a little-endian buffer. Eight eight-bit bytes are stored, regardless of the native length of a long or a long long integer on the system where it is executed.

Usage: void **lemfmu8**(char *m, ui64 u8)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored.

'u8' is the ui64 data item to be stored. It may be an expression.

Value returned: None.

Subroutine **LEMFMR4**

This function stores a single-precision floating point value into a little-endian buffer. Four eight-bit bytes in the format of an IEEE-standard floating point number are stored, regardless of the native format and length of a float on the system where it is executed. This function may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **lemfmr4**(char *m, float r4)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'r4' is the float data item to be stored. It may be an expression.

Value returned: None.

Subroutine **LEMFMR8**

This function stores a double-precision floating point value into a little-endian buffer. Eight eight-bit bytes in the format of an IEEE-standard double-precision floating point number are stored, regardless of the native format and length of a double on the system where it is executed. This function may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: void **lemfmr8**(char *m, float r8)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data should be stored. It should be a simple variable, not an expression.

'r8' is the double data item to be stored. It may be an expression.

Value returned: None.

Function **LEMTOI2**

This function retrieves a short integer from a little-endian buffer. Two eight-bit bytes are consumed, regardless of the native length of a short integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: short **lemtoi2**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The short data item at 'm', converted to the native format of the system where the program is running.

Function **LEMTOI4**

This function retrieves a 32-bit integer from a little-endian buffer. Four eight-bit bytes are consumed, regardless of the native length of a long integer on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: si32 **lemtoi4**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The si32 data item at 'm', converted to the native format of the system where the program is running.

Function **LEMTOI8**

This function retrieves a signed 64-bit integer from a little-endian buffer. Eight eight-bit bytes are consumed, regardless of the native length of a long or a long long integer on the system where it is executed.

Usage: si64 **lemtoi8**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored.

Value returned: The signed 64-bit data item at 'm', converted to the native format of the system where the program is running.

Function **LEMTOU8**

This function retrieves an unsigned 64-bit integer from a little-endian buffer. Eight eight-bit bytes are consumed, regardless of the native length of a long or a long long integer on the system where it is executed.

Usage: ui64 **lemtou8**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored.

Value returned: The unsigned 64-bit data item at 'm', converted to the native format of the system where the program is running.

Function **LEMTOR4**

This function retrieves a single-precision floating-point variable from a little-endian buffer. Four eight-bit bytes are consumed, regardless of the native length of a float on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: float **lemtor4**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The single-precision IEEE floating-point data item at 'm', converted to the native format of the system where the program is running.

Function **LEMTOR8**

This function retrieves a double-precision floating-point variable from a little-endian buffer. Eight eight-bit bytes are consumed, regardless of the native length of a double on the system where it is executed. It may be implemented as a macro, in which case there is no type checking on the arguments.

Usage: double **lemtor8**(char *m)

Prototyped in: swap.h

Arguments: 'm' is a pointer to the memory location where the data are stored. It should be a simple variable, not an expression.

Value returned: The double-precision IEEE floating-point data item at 'm', converted to the native format of the system where the program is running.

FILE I/O ROUTINES

These routines all have names beginning with 'rf' for "ROCKS file". All use a file descriptor structure named 'RFdef', which is defined in header file rfdef.h. In addition, preprocessor-defined names are provided in rfdef.h for many of the arguments to these functions. The defined values of these names are given in parentheses following the description of each argument.

Function **RFALLO**

Function **rfallo** is used to allocate and initialize an RFdef structure, which holds various parameters needed to control file processing. It does NOT actually open the file. In the case of an internet socket, **rfallo** obtains the socket, and, if configured as a listener, initiates listening on the connection. In this case, **rfopen** should be called each time the program is ready to accept a connection.

Some of the parameters do not currently have any effect, but meaningful values should be provided for maximum compatibility with future enhancements to the file-handling system. The definition of the RFdef structure and the prototype for this function are in RFdef.h.

Usage: struct RFdef ***rfallo**(char *fname, int inout, int fmt, int accmeth, int append, int look, int norew, int retbuf, size_t blksize, size_t lrecl, long nrp, int ierr)

Prototyped in: rfdef.h

Arguments: Arguments are the same as those for the **rfopen** function described in the next section. Values can be changed between **rfallo** and subsequent **rfopen** calls on the same RFdef block, except the 'accmeth' parameter cannot be changed. Values are simply stored by **rfallo** for later use, except when 'accmeth' is LISTENER, in which case listening is initiated on the specified port and the port number then cannot be changed in subsequent **rfopen** calls. Zero values, if not replaced later, are interpreted as requests for defaults. The defaults are equivalent to the corresponding -1 arguments unless stated otherwise.

'ierr' controls the action taken when allocation fails. A value of NO_ABORT (-1) indicates that a NULL pointer should be returned and no error message should be issued. ABORT (0) indicates that an error message should be issued and the run terminated. NOMSG_ABORT (+1) indicates that the caller is attempting to write to the printed output, therefore the run should be terminated without attempting to issue an error message. The abexit code for this error is 98.

Value returned: A pointer to the RFdef structure created by the **rfallo** routine is returned. This structure is typedef'd to 'rkfd' in rfdef.h. If allocation failed and 'ierr' was NO_ABORT, a NULL pointer is returned.

Note: If an RFdef structure is allocated statically (without calling **rfallo**), be sure to zero the entire block before calling **rfopen**.

Function **RFOPEN**

Function **rfopen** should be used to open all files except stdin, stdout, and stderr, which are handled internally. An RFdef block should be allocated for each file using **rfallo** before **rfopen** is called. Function **rfopen** accepts certain parameters that currently have no effect, but which may be used in future enhancements. Meaningful values for these parameters should be supplied for maximum compatibility with future versions. The definition of the RFdef structure and of the RF value returned are in RFdef.h.

Usage: int **rfopen**(struct RFdef *fd, char *fname, int inout, int fmt, int accmeth, int append, int look, int norew, int retbuf, size_t blksize, size_t lrecl, long nrp, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to the RFdef block that defines the file to be opened. This block should have been created by a previous call to **rfallo**. The remaining parameters may be used to override values already loaded into the RFdef structure. A value of SAME (0) for any parameter indicates that the current value in *fd is to be left unchanged. If the previous value is itself zero, a default is used. The default is the same as the -1 value unless stated otherwise.

'fname' is the name of the file to be opened. Its format is operating-system dependent. A valid file name must be supplied in either the **rfallo** or the **rfopen** call. If 'accmeth' is INITIATOR, then 'fname' is instead the host name or fully qualified domain name (FQDN) of the host to be contacted

'inout' is READ (-1) if the file is only to be read, WRITE (+1) if it is only to be written, or READWRITE (+2) if it may be both read and written. If a file is opened for WRITE under UNIX, any existing file of the same name is truncated (written over). If a socket is to be both read and written, do not use READWRITE. Instead, call **rfdups** to generate a duplicate RFdef block for the socket and open one copy for reading and the other for writing. This permits separate buffering in both directions. The default is READ.

'fmt' is BINARY (-1) for binary files (imples fixed-length records in the IBM implementation) and TEXT (+1) for text files (corresponding to FORTRAN formatted files; imples variable-length records in the IBM implementation). The default is BINARY. (Undocumented values of this parameter are used internally to deal with stdin, stdout, and stderr. The value is generally irrelevant in UNIX implementations.)

'accmeth' is SEQUENTIAL (0) for ordinary sequential access, DIRECT (+1) for random (direct) access, INITIATOR (+2) to initiate a socket connection, and LISTENER (+4) to initiate listening on a socket connection. This parameter cannot be changed one specified in the initial **rfallo** call. The default is SEQUENTIAL.

'append' is TOP (-1) if the file is to be positioned at the beginning after it is opened and BOTTOM (+1) if it is to be positioned at the end. This parameter is only meaningful if 'inout' = WRITE and the file is old. The default is TOP.

'look' is NO_LOOKAHEAD (-1) if look-ahead reading is not be be done and LOOKAHEAD (+1) if it is to be done. In general, NO_LOOKAHEAD should be chosen if the file will be read mostly in random order, and LOOKAHEAD if the file will be read mostly sequentially. If reading from a socket, NO_LOOKAHEAD is forced. The default is LOOKAHEAD.

'norew' controls file positioning during a subsequent close (**rfclose**, rewind, or system-initiated close). Values are REWIND (-1) to cause a rewind to occur and NO_REWIND (+1) to prevent a rewind. The default is REWIND. Caution: 'norew' may not be implemented on all systems. It is mainly intended for use with magnetic tape to prevent rewinding when another file on the same tape is to be read later.

'retbuf' controls retention of buffers and sockets during a subsequent close. Values are RELEASE_BUFF (-1) to allow buffers and sockets to be released on close and RETAIN_BUFF (+1) to force retention of buffers and sockets. RETAIN_BUFF should be used by a server that will continue listening on a socket after processing one request. The default is RELEASE_BUFF. When it is known that a file will be reopened, buffers should be retained to minimize memory fragmentation. This parameter is considered advisory and may not be implemented on all systems.

'blksize' provides a suggested blocksize for new output files and buffer size for all files. On IBM MVS, it specifies the actual blocksize to be used. A value of IGNORE (0) indicates that a system-wide default (currently 1024) should be used. This parameter is considered advisory and may not be implemented on all systems.

'lrecl' is the logical record length and is required for the IBM implementation of fixed-length direct-access files. In situations where it is not meaningful or not needed, it will be ignored. A value of IGNORE (0) indicates that a value will be provided on a DD or FILEDEF statement.

'nrp' ("number of records or port") has a different meaning depending on 'accmeth'. If 'accmeth' is DIRECT and a new file is being created, 'nrp' is the number of records expected to be written. This parameter is considered advisory; it may be used to optimize disk allocations on some systems and may be ignored on others. A value of IGNORE (0) indicates that a system-wide default should be used. If 'accmeth' is INITIATOR or LISTENER, 'nrp' is the number of the internet port to be bound to the socket. This value is mandatory at **rfallo** time for LISTENER and at **rfopen** time for INITIATOR and cannot be changed in subsequent **rfopen** calls on the same RFdef. In all other cases, 'nrp' is ignored.

'ierr' controls the action taken when a file cannot be opened. The possible values are the same as for **rfallo**.

Value returned: If successful, a system-dependent nonzero file descriptor is returned. If an error has occurred and 'ierr' was NO_ABORT, zero is returned and the system error code ('errno') is returned in fd->rferrno. Otherwise, abnormal termination occurs with abexit code 99.

Function **RFDUPS**

Function **rfdups** is used to duplicate an RFdef block for a socket so that independent buffering can be performed for reading and writing accesses.

Usage: struct RFdef ***rfdups**(struct RFdef *fd, size_t blksize, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef block that has already been allocated and opened for access to an internet socket.

'blksize' is the block size for buffering. It may differ for read and write access to the same socket.

'ierr' is the same as for **rfallo**.

Value returned: A pointer to a new RFdef that is bound to the same socket connection as the original 'fd' block. If the 'inout' parameter of 'fd' is READ, the new block is set up for WRITE and vice-versa. Each block should be used exclusively for reading or writing, respectively and closed normally when finished.

Errors: Abexit 53 if there was not enough memory available to allocate the requested block or buffer or if 'fd' does not point to an open socket interface. If 'ierr' is NO_ABORT, a NULL pointer is returned and fd->rferrno is set to 53.

Function **RFQSAME**

Function **rfqsame** is used to determine whether two RFdef structures access the same file. It is used by **cryout** to initialize the SPOUT state according to whether stdout and stderr are the same, but it may also have uses in application programs.

Usage: int **rfqsame**(struct RFdef *fd1, struct RFdef *fd2)

Prototyped in: rfdef.h

Arguments: 'fd1' and 'fd2' are pointers to the two RFdef blocks to be compared. Both must have been opened by a previous call to **rfopen**.

Value returned: TRUE (=1) if the two files are the same, otherwise FALSE (=0). If either argument points to an invalid or closed RFdef block, an error occurs with abexit code 52.

Function **RFREAD**

Function **rfread** is used to transfer a specified number of bytes from a file to a user storage area. It implements as many as possible of the advisory **rfopen** parameters. It is implemented on each kind of system by I/O system calls that have been found by test to be efficient on that machine. It provides local buffering where that has been found to be faster than that provided by the normal C library calls. Be sure to close files when no longer needed in order to free up this buffer space.

Usage: long **rfread**(struct RFdef *fd, char *item,
    size_t length, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to a RFdef block that has been opened for reading by calling **rfopen**.

'item' is the address of the user data area into which data are to be placed.

'length' is the number of bytes to be transferred.

'ierr' is the same as for **rfallo**. The abexit code if an error occurs is 3.

Value returned: If read is successful, the number of bytes read is returned. On end-of-file, zero is returned. If a read error occurred and 'ierr' was NO_ABORT, a negative value, which may further encode the nature of the error in a system-dependent manner, is returned and fd->rferrno is set to the system error code ('errno').

Note: **rfread** may or may not convert line-ending characters used in text files in a particular operating system to standard C newlines (it does so in all existing implementations), but **rfread** and **rfwrite** will always handle new lines in a consistent manner. Text files written by other programs (e.g. text editors) should probably be read with **rfgets**.

Function **RFGETS**

This routine reads from a ROCKS RFdef input file until either a new line is found or the given length is satisfied. All input characters are retained in the data except the OS-specific end-of-line marker (NL, CR-LF, etc.) is eaten. (IBM systems with no newline are currently not supported.) A NULL is appended to mark the end of the data returned.

This routine is similar in function to the standard C library routine fgets(), but the arguments, return value, and handling of the newline marker are different.

Usage: long **rfgets**(struct RFdef *fd, char *item, size_t length, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

'item' is a pointer to a memory area where data are to be returned.

'length' is the size of the 'item' array. The maximum number of characters that can be read is (length-1) because of the NULL that is appended to the end of the data.

'ierr' is the same as for **rfallo**. The abexit code if an error occurs is 3.

Value returned: Length of data read (including the NULL terminator so empty lines can be distinguished from eof). On end-of-file, zero is returned. If a read error occurred and 'ierr' was NO_ABORT, a negative value, which may further encode the nature of the error in a system-dependent manner, is returned and fd->rferrno is set to the system error code ('errno'). The data are returned in the buffer pointed to by the 'item' argument.

Function **RFSEEK**

This function sets the reading pointer to a specified absolute or relative location in an open file. It may be used to skip over unwanted input data. Do not use on a socket file.

Usage: long **rfseek**(struct RFdef *fd, size_t offset, int kseek, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

'offset' is the location in the file relative to 'kseek' where the next read should commence.

'kseek' indicates how the 'offset' parameter is to be interpreted: SEEKABS (0) indicates that 'offset' is relative to the beginning of the file; SEEKREL (1) indicates that 'offset' is relative to the current location in the file; and SEEKEND (2) indicates that 'offset' is relative to the end of the file (not currently implemented).

'ierr' is the same as for **rfallo**. The abexit code if an error occurs is 4.

Value returned: The current position in the file is returned if the seek was successful. If an error occurred and 'ierr' was NO_ABORT, a negative value, which may further encode the nature of the error in a system-dependent manner, is returned and fd->rferrno is set to the system error code ('errno').

Function **RFTELL**

This function returns the current reading location in an open file. It may be used to create a "bookmark"--if the value returned is used in a later call to **rfseek** in SEEKABS mode, the reading location will be returned to its current position.

Usage: size_t **rftell**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: The offset of the current reading position from the beginning of the file, in bytes.

Function **RFWRITE**

Function **rfwrite** is used to transfer a specified number of bytes from a user storage area to a file. It implements as many as possible of the advisory **rfopen** parameters. It is implemented on each kind of system by I/O system calls that have been found by test to be efficient on that machine. It provides local buffering where that has been found to be faster than that provided by the normal C library calls. Be sure to close files when no longer needed in order to free up this buffer space.

With socket files, note that the system **write** call effectively flushes the output after each write. **rfwrite** buffers the output, and **rfflush** must be called each time it is desired to flush the output to the socket interface.

Usage: long **rfwrite**(struct RFdef *fd, char *item,
    size_t length, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef block that has been opened for writing by calling **rfopen**.

'item' is the address of the user data area to be written.

'length' is the number of bytes to be transferred.

'ierr' is the same as for **rfallo**. The abexit code if an error occurs is 5.

Value returned: If successful, the number of bytes written is returned. On a write error, if 'ierr' was NO_ABORT, a short count or negative value which may further encode the nature of the error in a system-dependent manner, is returned and fd->rferrno is set to the system error code ('errno').

Note: **rfwrite** may or may not convert newline characters in data written to text files to whatever line-ending character is used in a particular operating system (it does so in all existing implementations), but **rfread** and **rfwrite** will always handle new lines in a consistent manner.

<u>Function **RFFLUSH**</u>

Function **rfflush** is used to transfer any data buffered for writing to the output file. Be sure to use **rfflush** rather then **fflush** when using **rfwrite** to write the data.

Usage: long **rfflush**(struct RFdef *fd, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef block that has been opened for writing by calling **rfopen**.

'ierr' is the same as for **rfallo**. The abexit code if an error occurs is 5.

Value returned: If successful, the number of bytes flushed (which may be 0) is returned. On a write error, if 'ierr' was NO_ABORT, a negative value, which may further encode the nature of the error in a system-dependent manner, is returned and fd->rferrno is set to the system error code ('errno').

Function **RFCLOSE**

Function **rfclose** is used to close a file previously opened by **rfopen**.

Usage: int **rfclose**(struct RFdef *fd, int norew, int retbuf, int ierr)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to the RFdef structure defining the file to be closed.

'norew' controls file positioning after the file is closed. Values are REWIND (-1) to cause a rewind to occur, IGNORE (0) to use the value currently stored in the RFdef block, and NO_REWIND (+1) to leave the file at its present position.

'retbuf' controls retention of buffers after the file is closed. Values are RELEASE_BUFF (-1) to allow buffers and sockets to be released, IGNORE (0) to use the value currently stored in the RFdef block, and RETAIN_BUFF (+1) to retain buffers and sockets for reopening later. A nonzero value is stored and will govern a following **rfopen** call.

'ierr' is the same as for **rfallo**. The abexit code if an error occurs is 97.

Value returned: 0 if closing was successful, otherwise, if ierr was NO_ABORT, a nonzero, system-dependent error code is returned and also stored in fd->rferrno.

Function **RFRI2**

This function reads a short integer from a big-endian binary data file. Two eight-bit bytes are read from the file, regardless of the length of a short integer on the system where the program is running. Execution is terminated if any kind of error occurs. The user can determine whether end-of-file has been reached by testing for fd->lbsr == ATEOF.

Usage: short **rfri2**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: A short integer composed from the data contained in the next two bytes read from file 'fd'.

Function **RFRI4**

This function reads a 32-bit integer from a big-endian binary data file. Four eight-bit bytes are read from the file, regardless of the length of a long integer on the system where the program is running. Execution is terminated if any kind of error occurs. The user can determine whether end-of-file has been reached by testing for fd->lbsr == ATEOF.

Usage: si32 **rfri4**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: An si32 integer composed from the data contained in the next four bytes read from file 'fd'.

Function **RFRI8**

This function reads a 64-bit signed integer from a big-endian binary data file. The result is formatted according to the mode of storage of the si64 type defined in sysdef.h for the running host, which may be a native hardware type or a structure. Execution is terminated if any kind of error occurs. The user can determine whether end-of-file has been reached by testing for fd->lbsr == ATEOF.

Usage: si64 **rfri8**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: An si64 integer composed from the data contained in the next eight bytes read from file 'fd'.

Function **RFRU8**

This function reads a 64-bit unsigned integer from a big-endian binary data file. The result is formatted according to the mode of storage of the ui64 type defined in sysdef.h for the running host, which may be a native hardware type or a structure. Execution is terminated if any kind of error occurs. The user can determine whether end-of-file has been reached by testing for fd->lbsr == ATEOF.

Usage: ui64 **rfru8**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: A ui64 integer composed from the data contained in the next eight bytes read from file 'fd'.

Function **RFRR4**

This function reads a single-precision floating-point item from a big-endian binary data file. Four eight-bit bytes in IEEE-standard single-precision floating-point format are read from the file and converted if necessary to the length and format of a float on the system where the program is running. Execution is terminated if any kind of error occurs. The user can determine whether end-of-file has been reached by testing for fd->lbsr == ATEOF.

Usage: float **rfrr4**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: A float composed from the data contained in the next four bytes read from file 'fd'.

Function **RFRR8**

This function reads a double-precision floating-point item from a big-endian binary data file. Eight eight-bit bytes in IEEE-standard double-precision floating-point format are read from the file and converted if necessary to the length and format of a double on the system where the program is running. Execution is terminated if any kind of error occurs. The user can determine whether end-of-file has been reached by testing for fd->lbsr == ATEOF.

Usage: float **rfrr8**(struct RFdef *fd)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for reading by calling **rfopen**.

Value returned: A double composed from the data contained in the next eight bytes read from file 'fd'.

Subroutine **RFWI2**

This function writes a short integer to a big-endian binary data file. Two eight-bit bytes are written to the file, regardless of the length of a short integer on the system where the program is running. Execution is terminated if any kind of error occurs.

Usage: void **rfwi2**(struct RFdef *fd, short i2)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for writing by calling **rfopen**.

'i2' is the short integer to be stored in the file.

Value returned: None.

Subroutine **RFWI4**

This function writes a 32-bit integer to a big-endian binary data file. Four eight-bit bytes are written to the file, regardless of the length of a long integer on the system where the program is running. Execution is terminated if any kind of error occurs.

Usage: void **rfwi4**(struct RFdef *fd, si32 i4)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for writing by calling **rfopen**.

'i4' is the 32-bit integer to be stored in the file.

Value returned: None.

Subroutine **RFWI8**

This function writes a 64-bit signed integer to a big-endian binary data file. Eight eight-bit bytes are written to the file, regardless of the length and format (native data item or structure) of an si64 integer on the system where the program is running. Execution is terminated if any kind of error occurs.

Usage: void **rfwi8**(struct RFdef *fd, si64 i8)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for writing by calling **rfopen**.

'i8' is the si64 (see typedef in sysdef.h) integer to be stored in the file.

Value returned: None.

Subroutine **RFWU8**

This function writes a 64-bit unsigned integer to a big-endian binary data file. Eight eight-bit bytes are written to the file, regardless of the length and format (native data item or structure) of a ui64 integer on the system where the program is running. Execution is terminated if any kind of error occurs.

Usage: void **rfwu8**(struct RFdef *fd, ui64 u8)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for writing by calling **rfopen**.

'u8' is the ui64 (see typedef in sysdef.h) integer to be stored in the file.

Value returned: None.

Subroutine **RFWR4**

This function writes a single-precision floating-point datum to a big-endian binary data file. Four eight-bit bytes in IEEE-standard single-precision floating-point format are written to the file, regardless of the length and format of a float on the system where the program is running. Execution is terminated if any kind of error occurs.

Usage: void **rfwr4**(struct RFdef *fd, float r4)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for writing by calling **rfopen**.

'r4' is the floating point item to be stored in the file.

Value returned: None.

Subroutine **RFWR8**

This function writes a double-precision floating-point datum to a big-endian binary data file. Eight eight-bit bytes in IEEE-standard double-precision floating-point format are written to the file, regardless of the length and format of a double on the system where the program is running. Execution is terminated if any kind of error occurs.

Usage: void **rfwr8**(struct RFdef *fd, double r8)

Prototyped in: rfdef.h

Arguments: 'fd' is a pointer to an RFdef that has been opened for writing by calling **rfopen**.

'r8' is the double-precision number to be stored in the file.

Value returned: None.

FILE I/O ROUTINES

LOW-LEVEL I/O ROUTINES

Function **CRYIN**

Function **cryin** is used to read the next control card from the current input file specified by the last call to **cdunit.** Default input is from stdin, corresponding to unit 1 in the FORTRAN version. Function **cryin** handles all comment cards (cards with '*' in column 1 and also cards containing only blanks and tabs if **accwac** has been called), END, QUIT, TITLE, OUTPRTY, PROMPT, SPOUT, ERROR DUMP REQUESTED, EXECUTE, and DEFAULTS cards, and checks for unexpected continuations (unless **accwad(1)** has been called). If **rdagn** has been called before entry to **cryin**, the previous card is reread. When **cryin** determines that stdin is from a character device, it issues a prompt before each line of input. The prompt string can be changed by a call to **sprmpt** or by the user's entering a PROMPT card.

Usage: char ***cryin**(void)

Prototyped in: rocks.h

Value returned: pointer to the control card image. This pointer should be considered valid only until the next call to **cryin**, **cdscan**, or **inform**. (The current pointer is always updated in RK.last). When the end of input or an explicit END or QUIT card is reached, **cryin** returns a NULL pointer on subsequent calls. However, if an END card, but not a QUIT card, is reached within a file being executed, control returns to the card after the EXECUTE card. (The IBM Assembler version did not return a pointer.)

Error procedures: Execution is terminated with abexit code 60 if an input line contains more than CDSIZE (80) characters, code 62 if unable to allocate an input buffer, code 63 if unable to allocate space for variable symbols, and code 64 if a syntax error is found on an EXECUTE or DEFAULTS control card.

Subroutine **RDAGN**

Subroutine **rdagn** is called to make **cryin** re-read the current card when it is next called. Subroutine **rdagn** should be called whenever a subprogram detects an unrecognized card that is to be processed by other code later in the application. The unidentified card should of course not be printed. CAUTION: **rdagn** will not cause re-reading of cards read by non-ROCKS library functions.

Usage: void **rdagn**(void)

Prototyped in: rocks.h

Subroutine **ACCWAC**

Subroutine **accwac** may be called to force **cryin** to accept and print as a comment any subsequent card that contains only blanks and tabs. Normally, such cards are treated as ROCKS data cards and are returned to the caller by **cryin**.

Usage: void **accwac**(void)

Prototyped in: rocks.h

Subroutine **ACCWAD**

Subroutine **accwad** may be called to force **cryin** to accept any input line beginning with whitespace as a normal data card. Normally, such cards are treated as continuations and will produce an error message if they do not follow a scanned card ending with a continuation signal. The intended use of this routine is to allow traditional fixed-format (non-scanned) input to be read.

Usage: void **accwad**(int kaccw)

Prototyped in: rocks.h

Argument: If 'kaccw' is nonzero (TRUE), subsequent input lines beginning with whitespace are treated as data. If 'kaccw' is zero (FALSE), subsequent input lines beginning with whitespace are treated as continuations. The default is to treat them as continuations.

Subroutines **CDPRNT** and **CDPRT1**

Subroutine **cdprnt** prints a control card with double spacing; **cdprt1** does the same with single spacing. In both cases, the indenting specified under "PRINTED OUTPUT" is performed. Both routines set an internal flag signalling that any subsequent continuation card is to be printed automatically.

Usage: void **cdprnt**(char *card)
       void **cdprt1**(char *card)

Prototyped in: rocks.h

Arguments: 'card' is a pointer to the card image to be printed.

FILE I/O ROUTINES

Subroutine **CDUNIT**

Subroutine **cdunit** changes the file from which control cards are read by **cryin**. It retains the status of all previous input files on a push-down stack. This stack is popped up whenever an end-of-file or END card is read by **cryin**. This facility permits a return to normal processing following invocation of stored control cards by an EXECUTE statement. (The maximum allowable number of pending input files depends on the implementation.)

Usage: void **cdunit**(char *fname)

Prototyped in: rocks.h

Argument: 'fname' is a string giving the name of the new input file. If fname==NULL, the stack is popped up, that is, card reading continues on the most recently used previous file. If at the top level, popping the stack has no effect, unlike the IBM FORTRAN version, where an end-of-file is generated. There is no checking for recursive use of input files.

Function **GTUNIT**

Function **gtunit** is used to retrieve the name of the current input file.

Usage: char ***gtunit**(void)

Prototyped in: rkxtra.h

Value returned: a pointer to the name of the current **cryin** input file. This pointer should be considered valid only until the next **cryin** or **cdunit** call. If input is from stdin, a NULL pointer is returned.

Function **QONLIN**

Function **qonlin** is used to determine whether input is from an online terminal.

Usage: int **qonlin**(void);

Prototyped in: rkxtra.h

Value returned: TRUE if input is online, FALSE if input is offline or cannot be determined.

Subroutine **SPRMPT**

Subroutine **sprmpt** is used to change the prompt printed by **cryin** when reading from an online terminal. Note: A different prompt, "...?", is printed by **cryin** when a continuation of a previous control card is expected. This prompt cannot be changed by the application.

Usage: void **sprmpt**(char *prompt)

Prototyped in: rocks.h

Argument: 'prompt' is a pointer to the desired prompt string (maximum 16 characters). The prompt should be constructed assuming the cursor is already positioned at the start of a new line when it is printed. The prompt string is copied to an internal buffer--the memory pointed to by 'prompt' does not need to remain valid after the call. If 'prompt' is a NULL pointer, printing of the prompt is disabled.

Subroutine **CRYOUT**

Subroutine **cryout** is used to prepare printed output with page titles and subtitles. Output is written to stdout and repeated to stderr if the appropriate **spout** call has been made. Titles are supplied partly by **cryout** and partly from the user's TITLE card; subtitles are generated by **cryout** calls. Each **cryout** call may specify any number of text strings to be combined into one or more ordinary output lines or up to three lines of subtitles. **cryout** does not perform conversions of numerical values to character strings--such conversions must be performed by subroutines **bcdout** or **wbcdwt** before calling **cryout**, or formatting and writing may be combined in calls to **convrt** or **printf**. By default, there must be an ANSI carriage control character at the beginning of each line of output-these characters (listed below) are internally converted to whatever control characters are needed in a particular implementation. Alternatively, if the RK_PF bit is set in the 'sprty' argument, UNIX-style text is expected, with LF in the data to start a new line and CR to overlay lines. In either case, the line count is given in the 'lcode' argument, allowing the user to make room on the page for a block of output that may be supplied in multiple **cryout** calls. Finally, note that '***' should be included at the beginning of all error messages and '-->' at the beginning of all warnings (after the carriage-control character if there is one). Messages beginning with '***' or '-->' are automatically repeated in the SPOUT output. In addition, messages beginning with '***' cause the current output buffer to be flushed when operating interactively. IBM fixed and variable blocked record formats are supported.

Usage: void **cryout**(int sprty, char *field, unsigned int lcode, ..., NULL)

Prototyped in: rocks.h

Arguments: 'sprty' is the sum of three values indicating the method of line control, the iexit error level (times 2^12), priority class (times 2^8), and SPOUT count (unshifted) of the output. The RK_PF bit (0x800) indicates that ASCII carriage controls are used. The named constants RK_P1, ..., RK_P5 can be used to indicate priorities 1 (highest) through 5 (lowest), respectively, and RK_E1, ..., RK_E8 can be used to indicate error messages with the indicated iexit value and priority 1. Only outputs having priority numerically not less than that specified by the user on the most recent OUTPRTY control card will be printed (default: OUTPRTY = 1). See "The ROCKS Routines: Control Card Rules" p. 7 for documentation of the intended uses of the 5 priority classes.

The SPOUT count is an integer indicating the number of **cryout** calls for which output is to be duplicated in the SPOUT output (default: 0). A nonzero value has the same effect as a prior call to **spout** with the same argument. (The SPOUT count accumulates across **cryout** and **spout** calls.)

The remaining arguments must come in field, lcode pairs. The end of the list <u>must</u> be indicated by a null field pointer.

'field' is a text string to be printed.

'lcode' is a code made up by adding values from the following table indicating the operations desired plus possibly the length of the field. If a field length of 1 to 254 characters is added to the code, the program will use the lesser of that value or the actual string length. A count of 255 (named constant RK_SKIP) will cause the field to be skipped. A count of zero will cause the actual length of the text string to be used. Accordingly, the normal practice in writing new code will be to omit the length field.

Code Name	Value	Operation
RK_CCL	0	Continue current line. The text string should <u>not</u> begin with a carriage control character.
RK_SKIP	255	Omit this field regardless of other codes.
RK_LNk	2048+256*k	Begin new line. 'k' gives the number of lines, beginning with the current line, that must be kept together on one page. These lines may be written by one or more **cryout** calls beginning with the current one; the carriage control character on

```
                        each record determines the number of lines
                        it will occupy.  The maximum value of 'k'
                        is 6; for larger line counts, use tlines.
    RK_NEWPG     3840   Begin new page unless in RK_PGNONE mode.
                        The text string should begin with a blank
                        or zero carriage control as it will follow
                        the internally supplied title line.
    RK_OMITLN    4096   Omit this line if in RK_PGNONE mode.
    RK_SUBTTL    8192   This field and all following fields form a
                        new subtitle.  Next cryout call begins a
                        new page unless in RK_PGNONE mode.  The
                        new subtitle is not printed unless an
                        ordinary line is printed before the next
                        RK_SUBTTL call.
    RK_NTSUBTTL 16384   Same as RK_SUBTTL exept printing of the
                        subtitle is not triggered by the next
                        cryout call nor is a new page forced.  The
                        natural use of this code is to clear an
                        existing subtitle to blanks without
                        printing anything extra.
    RK_NFSUBTTL 24576   Same as RK_SUBTTL except a new page is not
                        forced.
    RK_FLUSH    32768   Flush print and spout buffers at the end
                        of this field.  See discussion below.
```

The ANSI carriage controls codes are: '+' to overstrike the previous line, ' ' to single space, '0' to double space, and '-' to triple space. However, if RK_PGNONE is in effect, all overstrikes are changed to single lines.

Buffer Flushing: Normally, each output line is held pending until the next call that starts a new line (RK_LNk code). This permits single lines to be constructed from fields passed in multiple **cryout** calls (and is necessary for the implementation of ANSI carriage control characters). An option code, RK_FLUSH, is provided to force a completed line to be written at once rather than held. This code should be used with any line that needs to be written at once (e.g. an interactive prompt) and MUST be used with the last line of a run so that an incomplete line is not left in the buffer when execution terminates. To flush the buffer without writing anything additional, code

```
        cryout(RK_P1,"\0",RK_FLUSH+RK_LN0+1,NULL);
```

The buffer is flushed automatically every time **cryin** reaches the end of an input file and every time an error message is printed. The line following a flushed line cannot be an overstrike line.

Examples: To print the message '***BAD INPUT.' at priority 1 with double spacing and set iexit bit 1, the **cryout** call could be:

```
cryout(RK_E1, "0***BAD INPUT.", RK_LN2, NULL);
                  -or-
cryout(RK_E1, "0***BAD INPUT.", RK_LN2+14, NULL);
```

(Note that 14 is the length of the string "0***BAD INPUT.". Constants RK_LN1 through RK_LN6 are defined in rocks.h for grouping 1-6 lines of output in a block on the same page. Note that it is possible to print several lines with the line count given separately for each line or together in the code for the first line. The difference is that, when the end of a page is reached, the entire cluster of lines would be placed on a new page in the second case, but not the first. To generate a cluster of more than 6 lines, use function **tlines**.)

The following example would create a subtitle to head a table:

```
cryout(RK_P2, "0 ARG  VALUE  STD-DEV",
    RK_SUBTTL+RK_LN2, NULL);
```

This title would be placed on the second line below the title (because of the "0" carriage control) on each page of output until cancelled by a subsequent RK_SUBTTL call. The standard method of turning off a subtitle is the call:

```
cryout(RK_P1, " ", RK_NTSUBTTL+RK_LN1+1, NULL);
```

RK variables: RK.pglns and RK.pgcls contain the number of lines and columns printed on a page, including all titles and margins. These variables may be changed by the application before any call to **cryout** is made, but should not be changed thereafter. RK.rmlns and RK.rmcls contain, respectively, the number of lines remaining on the current page and the number of columns remaining in the current line. RK.pgno contains the current page number. RK.rmlns can be set to -1 to force a new page; the other variables should never be changed explicitly by an application program.

64-bit compilations: The rule in C is that fixed-point arguments are promoted to int if smaller than int, otherwise kept at the larger size. cryout() expects ints for the lcodes. Therefore, an lcode that is a long or long long should never be passed.

Errors: If a line exceeds the logical record length for the output unit, it is continued automatically on another line. If a subtitle is longer than the buffer in the program (408 characters) the program is terminated with abexit code 50. If a subtitle attempts to start a new page, the program is terminated with abexit code 51. If one of the units stdout or stderr cannot be accessed, the program is terminated with abexit code 52.

Subroutine **SETTIT**

Subroutine **settit** is used to change the current output page title. It does not actually cause a title to be printed--it only stores the title until it is needed. Use a **cryout** call with code RK_SUBTTL or RK_NEWPG to start a new page. Normally, **settit** is called internally from **cryin** when a TITLE card is read, and the application typically calls **settit** only once at the beginning of execution to establish a default title to be used when the user does not enter a TITLE card.

Usage: void **settit**(char *title)

Prototyped in: rocks.h

Argument: 'title' is a pointer to a character array containing an image of a TITLE card. The title is set from the 7th through 66th characters in this array.

Function **GETTIT**

Function **gettit** is used to retrieve the current page title, e.g. for labelling a plot.

Usage: char ***gettit**(void)

Prototyped in: rocks.h

Value returned: The value returned is a pointer to the title, which may consist of up to 60 characters (if 60 characters, the string is <u>not</u> terminated by a null character). The pointer returned should be considered valid only until the next **cryin** or **settit** call.

Subroutine **SETPID**

Subroutine **setpid** is used to change the program identification (columns 1 to 32) in the current output page title. It does not actually cause a title to be printed--it only stores the title until it is needed. Normally, this space is used to display the name and version number of the application program.

Usage: void **setpid**(char *pid)

Prototyped in: rocks.h

Argument: 'pid' is a pointer to a character array containing the program identification information. The maximum length of 'pid' is 32 characters.

Function **GETDAT**

Function **getdat** is used to retrieve the current date, e.g. for labelling a plot.

Usage: char ***getdat**(void)

Prototyped in: rocks.h

The value returned is a pointer to the date on which the run began execution. The date is a printable string of 12 characters (not terminated by a null character) in the format 'bbddbmmmbyyb' where b = blank, dd = day of month, mmm = name of month, and yy = last 2 digits of year.

Subroutine **TSTAMP**

Subroutine **tstamp** returns a 12 character ASCII time stamp in the format yymmddhhmmss, where yy = last 2 digits of current year, mm = month number, dd = day of month, hh = hour, mm = minute, ss = second. The time returned is the current time, not the time when the run began execution.

Usage: void **tstamp**(char * string)

Prototyped in: rkxtra.h

Argument: 'string' is a pointer to an array of 12 characters where the result will be placed. The result is not terminated with a null character.

Note: 'tstamp' now contains its own binary to decimal code, making it useable in environments where wbcdwt() is not available.

Function **LINES**

Function **lines** is provided to inform **cryout** that lines are about to be written to stdout by some means other than calls to **cryout** or higher-level routines that call **cryout** (**convrt** or ROCKS **printf**). When **lines** determines that the bottom of a page would be exceeded by the specified number of lines, titles and subtitles are printed just as if **cryout** had been called. Following the **lines** call, a buffer-flush call should be made as described above.

Usage: int **lines**(int n)

Prototyped in: rkxtra.h

Argument: 'n' is the number of lines to be written on stdout by following I/O. If a group of related lines is to be printed on the same page, they should be combined in a single **lines** call.

Value returned: The number of lines remaining on the current page after 'n' has been subtracted.

Function **TLINES**

Function **tlines** ("test lines") is similar to **lines** but is intended for use when output is printed with **cryout** or **convrt**. **tlines** differs from **lines** in that it tests, but does not update, the number of lines remaining on the current page. Updating is done by normal **cryout** calls. When a new page is needed, **tlines** sets a trigger in **cryout** causing the new page to be started by the next **cryout** call.

Usage: int **tlines**(int n)

Prototyped in: rkxtra.h

Argument: 'n' is the number of lines written by following **cryout**, **convrt**, or ROCKS **printf** calls that are to be grouped on the same page. This call is entirely equivalent to use of the L format code of **convrt** or ROCKS **printf** or the 'k' parameter in a CRYOUT call. Function **tlines** is needed only when more than 6 lines (the maximum value permitted for 'k' in a **cryout** call) are to be printed together.

Value returned: the number of lines remaining on the current page after 'n' has been subtracted.

Subroutine **SPOUT**

Subroutine **spout** provides a way to duplicate important parts of the **cryout** output into the 'stderr' output. Typically, when an application is run from a video display terminal, the SPOUT output is directed to the terminal for immediate inspection, while the full output is directed to a file for printing or detailed inspection later if the run is satisfactory. The programmer should select only the most important output lines for SPOUT, keeping in mind that users generally do not want to see routine output while monitoring a run.

In the SPOUT output, all carriage controls are ignored and pagination is suppressed. Subtitles are printed only at their first occurrence, regardless of whether or not SPOUT was active when they were set up. All error messages and warnings (lines whose first three text characters are '***' or '-->') are automatically duplicated in the SPOUT output when SPOUT is active. Control cards are automatically printed when followed by error messages, so **spout** does not have to be called before **ermark** calls.

SPOUT is activated automatically by **cryout** when it detects that stdout and stderr are different files or devices. When SPOUT is turned on in this way, it can subsequently be turned off and on again by use of the SPOUT OFF and SPOUT ON control cards, which are processed by **cryin**, but when stdout and stderr are the same, SPOUT control cards are ignored. Application programs need to inform **cryout** of which lines to duplicate in the SPOUT output, either by calling **spout** or by including appropriate codes in calls to **cryout**, **convrt**, and ROCKS **printf**.

Usage: void **spout**(int n)

Prototyped in: rocks.h

Argument: If n == SPTM_GBLOFF, spouting is being globally turned off by a command-line parameter, or because stderr == stdout, and all later calls are ignored.

If n == SPTM_LCLOFF, spouting is being locally turned off by a cryin() SPOUT card or some application option. Later spout calls are ignored until SPTM_LCLON is received. (This is the default situation at start up.)

If n == SPTM_LCLON, SPTM_LCLOFF is converted to SPTM_NONE.

If n == SPTM_NONE, the LCLOFF|LCLON status is not changed, but the pending count is set to 0.

If n > 0, and the pending count is >= 0, then the print from the next 'n' cryout calls is sent to stderr in addition to stdout. All carriage controls are ignored. Subtitles are written only once, before the next output.

If n == SPTM_ALL, spouting is on for all subsequent cryout calls until SPTM_NONE or below is received.

Usage note: The **spout** function with n > 0 has now been incorporated in the **cryout** call. The **spout** function is mainly useful for the control functions.

Subroutine **NOPAGE**

Subroutine nopage is used to suppress pagination. It has two modes: RK_PGNONE causes **cryout** to stop counting lines, ignore RK_NEWPAG codes, omit fields with the RK_OMITLN bit set (typically underlines), and convert all RK_SUBTTL fields to RK_NFSUBTTL. RK_PGLONG mode only prevents line counting. **nopage**(RK_PGNONE) is called by **cryin** when an OUTPRTY card has the "NOPG" option. There is currently no way to reverse this action. The RK_PGLONG mode is useful for printing maps or matrices of various types that should not be interrupted by page titles. In this case, lines with RK_NEWPAG set can be used to start new pages exactly where needed.

Usage: void **nopage**(int kpage)

Prototyped in: rkxtra.h

Argument: If kpage = RK_PGNONE (=2), all pagination is disabled. If kpage = RK_PGLONG (=1), counting of lines on each page is disabled, but other pagination features are still available. If kpage = RK_PGNORM (=0), RK_PGLONG mode is turned off, but RK_PGNONE mode is unchanged.

Functions **CNTRLN** and **CNTRL**

It is sometimes necessary to determine whether a particular input card or data field contains numeric data or some other kind of control information. These functions determine whether a data field contains any characters not expected to be found in numbers (i.e. alphabetic characters other than 'e' or 'E' followed by one or more digits, or punctuation other than plus, minus, space, tab, or decimal point).

Usage: int **cntrl**(char *data)
int **cntrln**(char *data, int length)

Prototyped in: rkxtra.h, rksubs.h

Arguments: 'data' is the location of the character string to be tested. 'length' is the maximum length of the string. In **cntrl**(), the maximum length is taken to be the default control card length (usually 80 chars). All characters up to the maximum length or the first end-of-string (null) character, whichever comes first, are tested.

Value returned: These functions return NUMBRET (=0) if the string contains only characters valid in numeric fields, otherwise CNTLRET (=1). (The string may of course contain incorrectly coded numeric data.)

Function **QWHITE**

This routine is used to determine whether an input card contains only whitespace characters (blanks and tabs). Programs that do not expect to read numerical data cards can make **cryin** treat such inputs as comments by calling **accwac**. In any event, **cryin** accepts null lines as comments under UNIX. (Null lines do not exist under IBM systems with fixed-length records.)

Usage: int **qwhite**(char *card)

Prototyped in: rkxtra.h, rksubs.h

Argument: 'card' is the location of the character string to be tested. All characters up to the first end-of-string (null) character are tested.

Value returned: Function **qwhite** returns TRUE (=1) if the string contains only blanks or tabs, otherwise FALSE (=0).

Subroutine **CDSCAN**

Subroutine **cdscan** is used to initiate parsing of alphanumeric fields from control cards. Functions **scan** or **scanck** may then be called repeatedly to obtain the individual fields sequentially from left to right. Subroutine **cdscan** must be called once for each card processed to initialize the scanning routines.

Usage: void **cdscan**(char *card,int displ,int maxlen,int csflags)

Prototyped in: rkxtra.h

Arguments: 'card' is a string (any length) containing the card to be processed.

'displ' is interpreted differently depending on the value of 'csflags'. In the default case, which is provided mainly to permit compatible translation of FORTRAN programs, 'displ' is the displacement (column-1) of the first character to be inspected. 'displ' can be set to skip over fixed information at the beginning of the card. 'displ' is overridden as described in the User's Guide if a colon is found on the card before the normal starting column.

When the RK_WDSKIP flag is set, 'displ' is interpreted as the number of words to skip over at the beginning of the card before scanning for data fields. Word skipping permits users to abbreviate card identifiers without use of a colon. The colon is still needed, however, when abbreviation results in reducing the number of words in the card identifier.

Use of RK_WDSKIP is the recommended practice for use in new programs.

'maxlen' is the length in characters (not including the end-of-string character) of the longest field which the calling program is prepared to process, i.e. the length of the 'field' argument of the **scan** call. If any field exceeds this length, an error message is issued to the user and the value, truncated to 'maxlen' characters, is returned to the **scan** caller. (Note: Users are conditioned to expect 16 as the normal value of 'maxlen'. DFLT_MAXLEN is defined to this value in ROCKS.H)

'csflags' is the sum of various named constants denoting special options of the scanning routines. At present, the following flags are defined:

Flag Name	Value	Function
RK_AMPNULL	1	Treat ampersands as nulls. (Used internally by **cryin** to read names of variable symbols.)
RK_NEST	2	Nested parentheses are permitted. The nesting level is returned in RK.plevel after each **scan** call.
RK_NOCONT	4	Do not attempt to read continuation lines. Use when a string rather than an input card is being scanned.
RK_PMDELIM	8	Treat plus and minus signs as delimiters while performing a word skip (see below).
RK_ASTDELIM	16	Treat asterisks as delimiters while performing a word skip (see below).
RK_SLDELIM	32	Treat slashes as delimiters while performing a word skip (see below).
RK_NOPMEQ	64	Normally, punctuation combinations '+=' and '-=' are treated as if entered as '=+' and '=-' (but see scan codes below). RK_NOPMEQ causes those combinations to be treated as separate characters, i.e. the '+' and '-' characters are treated as data or punctuation according to whether or not RK_PMDELIM has been set.
RK_WDSKIP	128	Interpret 'displ' argument as count of words to skip rather than columns to skip at the beginning of the control card.

Error procedures: **cdscan** terminates with abexit code 22 if the specified starting column is beyond the end of the card, and with code 32 if unable to allocate a buffer for **scanagn.**

Function **SCAN**

Function **scan** is used to parse one field according to the rules described in the User's Guide. Returned fields are terminated with the normal C end-of-string character and punctuation characters are removed. The nature of the punctuation is indicated by a return code. Continuation cards are read by calling **cryin** when needed. Continuation cards are automatically printed with single spacing when a previous call to **cdprnt** or **cdprt1** has been made for the card passed to **cdscan.** Further processing of fields parsed by **scan** may include identification and numerical conversion of fields.

Usage: int **scan**(char *field, int scflags)

Prototyped in: rkxtra.h

Arguments: 'field' is an array large enough to hold 'maxlen' characters (plus an end-of-string character) into which the next input field is placed. If 'field' is a NULL pointer, **scan** will skip over a field without returning anything.

'scflags' are option flags taken from the following table:

Flag name	Value	Function
RK_NEWKEY	1	If the field is preceded by a delimiter other than a comma, blank, or left paren, error RK_PUNCERR (incorrect punctuation) is generated.
RK_REQFLD	2	If no field is found (end of card has been reached), error RK_REQFERR (required field missing) is generated.
RK_FENCE	4	When a maximal-length field is found, a C end-of-string character is not added.
RK_PMDELIM	8	Treat plus and minus signs as field delimiters. In contrast to the normal rule, these characters are returned in the data field even when used as delimiters.
RK_ASTDELIM	16	Treat asterisks as field delimiters.
RK_SLDELIM	32	Treat slashes as field delimiters.
RK_SCANFNM	64	Parsing a file name. Underscores, '+', '-', and parentheses are unconditionally treated as data. Slashes are treated as data or delimiters according to the state of the RK_SLDELIM flag, allowing path name

		parts to be separated or not. The other punctuation characters retain their normal field-separating functions.
RK_WDSKIP	128	Used internally when called from cdscan() to perform initial word skip.
RK_PMEQPM	256	Indicates that '+=' (entered as two consecutive characters) entered in the previous field is to be treated as if entered as '=+' and '-=' is to be treated as '=-'. The RK_EQUALS code will have been returned on the previous call. The plus or minus sign is prefixed to the data for the present field. If this code is not entered, these combinations are treated as errors. (This is for the convenience of kwscan code 'K' input.)
RK_PMVADJ	264	(=RK_PMDELIM|RK_PMEQPM) Treats '+' and '-' as delimiters as for RK_PMDELIM, except that if one of these signs is followed by a digit, it is treated as data. (This allows numbers with exponents in data.)

Value returned: a punctuation code which is the sum of the relevant values from the following table is returned and also stored in RK.scancode. The old practice of marking the last field with code 8 has been eliminated.

Return code	Name	Meaning
0	RK_BLANK	Field ended by blank or quote.
1	RK_PMINUS	Field ended by plus or minus sign.
2	RK_COMMA	Field ended by comma.
4	RK_EQUALS	Field ended by equals sign.
8	RK_ASTERISK	Field ended by asterisk, or by a colon when parsing a DOS filename.
16	RK_SLASH	Field ended by a slash.
32	RK_LFTPAREN	Field ended by a left parenthesis.
64	RK_INPARENS	Field is enclosed in parentheses.
128	RK_RTPAREN	Field ended by right parenthesis.
256	RK_COLON	Field ended by a colon.
512	RK_SEMICOLON	Semicolon comment found (CrkParse).
1024	RK_ENDCARD	No field found--end of card (this code is returned on the next call <u>after</u> the last field has already been returned).

The following combinations of punctuation characters are valid and produce the return codes indicated:

',	2	'+	1	'-	1	'*	8
'=	4	')	192	'),	194	),	194
')+	193	')-	193	)+	193	)-	193
')*	200	)*	200	')=	196	)=	196

In addition, the length of the returned field, minus one, is returned in RK.length.

Function **SCANCK**

Function **scanck** is the same as **scan** except it additionally performs the service of checking the return code generated by **scan**. An **ermark** punctuation error is generated when the return code is one of a "bad" set specified by the 'badpn' argument.

Usage: int **scanck**(char *field, int scflags, int badpn)

Prototyped in: rkxtra.h

Arguments: 'field' and 'scflags' are the same as specified above for function **scan**.

'badpn' is the logical 'OR' of all the RK.scancode values defined above which are to be considered illegal on this call to **scan**.

Value returned: Scan code as defined above for function **scan**.

Subroutine **SCANAGN**

Subroutine **scanagn** "pushes back" a field that has been scanned so that the same field and code will be returned again on the next call to **scan**. This feature simplifies the coding of "look-ahead" parsing routines.

Usage: void **scanagn**(void)

Prototyped in: rkxtra.h

Function **SCANLEN**

Function **scanlen** may be used to change the maxlen parameter at any time during a series of **scan** calls. This may be necessary, for example, if the length of the longest field is not known at the time **cdscan** is called.

Usage: int **scanlen**(int maxlen)

Prototyped in: rkxtra.h

Argument: 'maxlen' is the length of the longest field that is allowed to be returned by subsequent **scan** calls.

Value returned: **scanlen** returns the previously active value of 'maxlen'. This may be used later to restore the scanning environment, for example, when a parsing routine returns to a caller whose 'maxlen' was otherwise unknown.

Function **CURPLEV**

This function returns the parentheses nesting level at the current scan location.

Usage: int curplev(void)

Prototyped in rkxtra.h

Subroutine **SKIP2END**

Subroutine **skip2end** should be called when the parsing of a control card must be terminated for some reason before the last field on the card has been scanned. **skip2end** reads and prints continuation cards until a card is encountered that is not a continuation card. It then calls **rdagn** internally. This process eliminates unwanted warnings about unexpected continuation cards.

Usage: void **skip2end**(void)

Prototyped in: rkxtra.h

Subroutine **THATSALL**

Subroutine **thatsall** should be called when all the expected fields on a card have been processed. It generates an error message if any more unscanned fields exist, then calls **skip2end** to consume any remaining continuation cards.

Usage: void **thatsall**(void)

Prototyped in: rkxtra.h

Subroutine **ERMARK**

A series of numbered error messages is provided to report errors that arise in scanning control cards. These messages can be invoked by calling subroutine **ermark.** Messages are registered and printed at the next **cryout** call. If the card has not already been printed, it is printed. The location of each error is marked by a '$' printed below the control card at the point of the error (the current scan location), unless **okmark**(FALSE) has been called (or the error is by nature not localized). Multiple messages can be produced by separate calls or by adding codes in a single call. Subroutine **ermark** also sets the 1 bit of RK.iexit on and sets RK.highrc to 4 if its current value is less than 4.

Usage: void **ermark**(ui32 mcode)

Prototyped in: rocks.h

Argument: 'mcode' is constructed by summing message codes from the following table:

Code name	Value	Error message
RK_PUNCERR	0x00001	Punctuation is incorrect, e.g. equals sign missing.
RK_IDERR	0x00002	Field is not identifiable.
RK_LENGERR	0x00004	Field is longer than 'maxlen' characters.
RK_PNQTERR	0x00008	Parentheses or quotes not matched.
RK_REQFERR	0x00010	A required field was not present.
RK_CARDERR	0x00020	Entire card was not recognized.
RK_NUMBERR	0x00040	Invalid numerical value or decimal in integer field.
RK_CHARERR	0x00080	Non-numeric character in numeric field.
RK_ABBRERR	0x00100	Abbreviation is not unique.
RK_TOOMANY	0x00200	Too many fields (usually in parens).
RK_ECNFERR	0x00400	Expected continuation not found.
RK_BCNFERR	0x00800	Blank card where continuation expected.
RK_EXCLERR	0x01000	More than one of an exclusive set of options was specified.
RK_NESTERR	0x02000	Parentheses were nested where not allowed.
RK_PNRQERR	0x04000	Field must be enclosed in parentheses.
RK_EQRQERR	0x08000	An equals sign was required but not found.
RK_UNITERR	0x10000	Invalid units specifier.
RK_MULTERR	0x20000	Invalid units multiplier.
RK_SYMBERR	0x40000	Undefined variable symbol.
RK_VANSERR	0x80000	Variable adjustment requested not entered.
RK_WARNING	0x01000000	The user will provide a warning message. Print the last card if not yet printed, but do not set iexit.

```
RK_MARKDLM 0x02000000  The user will provide the message.
                      Mark a '$' under the field delimiter but
                      do not generate an error message.
RK_MARKFLD 0x04000000  The user will provide the message.
                      Mark a '$' under the last character in the
                      data field, but do not generate a message.
```

Notes: (1) Generation of errors RK_PUNCERR, RK_REQFERR, and RK_NESTERR by **scan** is under the control of 'scflags' in the **scan** call.

(2) The call **ermark**(0) is allowed and results in printing the most recent control card if it was not already printed and setting RK.iexit and RK.highrc, but no other action.

(3) To avoid printing duplicate error messages, **ermark** does not in fact print anything when called, but rather sets flags in RK.erscan causing the requested messages to be printed when **cryout** is next called. Accordingly, to assure that all error messages are printed when a run terminates, at least one **cryout** call must follow the last **ermark** call. Often, a message such as '***JOB TERMINATED' can be used for this purpose. A call to **abexit** will also suffice. RK.erscan can be examined at any time by the application to determine whether any errors are pending.

(4) RK.highrc can be used to determine whether errors have occurred in some block of code. At the beginning of the block, save the current value of RK.highrc in another variable and set it to 0. At the end of the block, a nonzero value indicates that a call to ermark(), or certain other documented errors, has occurred. After this test, reset RK.highrc to the larger of its present value or its saved value.

(5) **ermark** terminates execution with abexit code 32 if unable to allocate a buffer area for the error message.

Subroutine **OKMARK**

Subroutine **ermark** normally uses scan column information from **scan** to position a '$' under the item in error. This procedure becomes invalid if the user scans a string not related to the current control card. To indicate to ermark that marking should be omitted, call **okmark**(FALSE).

Usage: void **okmark**(int dmflag)

Argument: 'dmflag' is FALSE if subsequent calls to **ermark** should not generate '$' marks, TRUE to restore this function. Every call to cryin implicitly calls **okmark**(TRUE).

Subroutines **BCDOUT, IBCDWT, UBCDWT,** and **WBCDWT**
Functions **BCDIN, IBCDIN, UBCDIN,** and **WBCDIN**

These are the basic subroutines for performing conversions between external (ASCII or EBCDIC decimal character strings) and internal (binary) representations of numbers. One conversion is performed for each call--multiple conversions can be carried out by use of **convrt**, **sconvrt**, ROCKS **printf** and **sprintf**, **inform**, and **sinform**. Conversions are similar to those provided by C library functions, but additional options are provided such as fixed point numbers with fractions, automatic decimal point selection, automatic use of exponential format on overflows and underflows, and integer format conversion of floating point numbers.

Routines **ibcdwt**, **ubcdwt**, **ibcdin**, and **ubcdin** are obsolete and are retained for compatibility with old applications only. They have been replaced by **wbcdwt** and **wbcdin**, which handle variable types in a more consistent manner and add the capability of outputting and inputting 64-bit fixed-point values. Documentation for the obsolete routines has been removed here; if needed, the C source code can be consulted.

The functions of the four current routines are as follows:

bcdout: Floating point to decimal.
wbcdwt: Fixed point (signed or unsigned, 8, 16, 32, or 64-bit) to decimal.
bcdin: Decimal to floating point.
wbcdin: Decimal to fixed point (signed or unsigned, 8, 16, 32, or 64-bit).

Usage: void **bcdout**(ui32 ic, char *field, double arg)
void **wbcdwt**(void *pitm, char *field, ui32 ic)
double **bcdin**(ui32 ic, char *field)
void **wbcdin**(const char *field, void *pitm, ui32 ic)

Prototyped in: rkxtra.h

Arguments: 'arg' is a double-precision floating point argument to be converted. Single-precision arguments to **bcdout** will be automatically coerced to double by the compiled code.

'pitm' is a pointer to a fixed-point item to be converted for output (**wbcdwt**) or from input (**wbcdin**). The item may be a signed or unsigned fixed-point value of length 1, 2, 4, or 8 bytes as indicated by the 'ic' code.

'field' for **bcdout** or **wbcdwt** is a character array into which the result is placed, e.g. for printing by **cryout**. It must be large enough to contain at least the number of characters specified in the 'ic' code. An end-of-string character is not added. For **wbcdin**, 'field' is an array containing the string to be converted.

'ic' is an operation code which is the sum of four quantities:

ic = scale + dec + op + width - 1

(Note: Named constants are used to define the components of 'ic' as described below.)

'scale' is required only for noninteger fixed-point variables. Such variables have an assumed binary point somewhere within the word. The scale code is 's' times RK_BSCL (= 16777216 = 16**6) or 's' left-shifted by RK_BS (= 24), where 's' is the number of bits to the right of the binary point in '*pitm' (maximum, 63). For example, to convert a number with 8 binary fraction bits, add (8<<RK_BS) to 'ic'.

'dec' (output) specifies the location of the printed decimal point. 'dec' is equal to (d << RK_DS), where RK_DS is 16 and 'd' is one more than the number of places to the right of the decimal point (d <= 15). 'd' = 0 corresponds to integer format (no decimal inserted). If the RK_AUTO bit is set, 'dec' gives the maximum value to be assigned internally to 'd' as a result of automatic scaling.

'dec' (input) specifies the location of the assumed decimal point. If the RK_DSF bit (=0x00800000) is set, scaling is forced, i.e. the value entered by the user is always multipled by the scale. If RK_DSF is not set, scaling only occurs when no decimal point is found in the input. 'dec' is equal to (d << RK_DS & 0x007f0000), where RK_DS is 16 and the scale is 10**(-d), -63 <= d <= 63. Thus, positive values of 'd' correspond to the number of places to the right of the inserted decimal when no decimal is found in the input. Users normally expect 'd' to be 0.

'op' specifies the particular operation to be performed. It is made up by adding the desired codes from the following table. The argument types and format codes are independent. Thus, for example, E format can be used with an integer variable. However, binary scales are not allowed with **bcdout**. Note that some codes are different for **wbcdin** and **wbcdwt** than for **bcdin**, **bcdout**, and the older, obsolete fixed-point conversion routines. These have been given different names to avoid confusion.

Operation or type	'operation' value	'operation' code
1a) Variable Types (**bcdout** or **bcdin**)		
Double precision	0	RK_DBL
Single precision	256	RK_SNGL
(Forces rounding of input values to the shorter format. This does not affect the actual type of the value returned. On hexadecimal input or output, sets the number of characters to be converted to the number needed for the shorter format.)		
1b) Variable Types (**wbcdwt** or **wbcdin**)		
Unsigned	32768	RK_NUNS
Default type (int, 16 or 32 bit)	0	RK_NDFLT
Byte	0x0040	RK_NBYTE
Half-word fixed-point (16 bits)	0x0080	RK_NHALF
Integer (16 or 32 bit)	0x00c0	RK_NINT
Fullword fixed-point (32 bits)	0x0100	RK_NI32
Long fixed-point (32 or 64 bits)	0x0140	RK_NLONG
Double-word fixed-point (64 bits)	0x0180	RK_NI64
Reserved for future 128-bit fixed-point	0x01c0	RK_NI128
2) Formats		
E Format (output only)	0	RK_EFMT
(E format on input is recognized automatically--no code required.)		
Z (hexadecimal) format	512	RK_HEXF
O (octal) format	1024	RK_OCTF
I or F format	1536	RK_IORF
3a) Optional Features (output only)		
Pad with zeros	2048	RK_NPAD0
(Leading or trailing zeros are supplied instead of blanks.)		
Automatic decimal selection.	4096	RK_AUTO
(The decimal is adjusted to give the lesser of 'd'-1 or the largest number of significant figures that will fit in the output field following a single blank.)		
Hexadecimal prefix	4096	RK_NZ0X
(Hexadecimal output is prefixed with '0x' or '0X' according to whether or not the RK_NZLC bit also set. The RK_AUTO bit is redefined for this purpose when output format is hexadecimal.)		
Octal prefix	4096	RK_Oct0
(Prefix octal output with '0'. The RK_AUTO bit is redefined for this		

```
            purpose when output format is
            octal.)
        Underflow control.                        8192   RK_UFLW
            (Output is automatically converted
            to E format when the number would
            otherwise have no significant
            digits.)
        Output type width for hex output          8192   RK_NZTW
            (The number of digits written for
            hex output is determined by the type
            of the item rather than by its
            magnitude.  This may be less, but
            cannot be more, than the field
            width.)
        Left justification.                      16384   RK_LFTJ
            (Output is placed at the left of
            the specified field and remaining
            positions are filled with blanks
            (zeros if RK_NPAD0 set).)
        Insert plus sign.                      1048576   RK_PLUS
            (A plus sign is inserted before
            positive output.  The RK_LSPC bit is
            ignored (used to implement printf()
            '+' flag).)
        Left protective space.                 2097152   RK_LSPC
            (Protects unsigned output on the
            left with at least one blank (used
            to implement printf() ' ' flag).)
        Negative sign forced.                  4194304   RK_NEGV
            (Inserts a negative sign in the
            converted value of an unsigned
            argument.)
        Decimal scale forced.                  8388608   RK_DSF
            (Inserts a decimal in the output
            string without scaling the numeric
            value.)
        Lower-case hexadecimal output.         8388608   RK_NZLC
            (Generates hexadecimal output with
            lower case 'a' - 'f' numeric values
            and '0x' prefix.  The RK_DSF bit is
            redefined for this purpose when
            output format is hexadecimal.)

    3b) Optional Features (input only)
        Query Positive.                           4096   RK_QPOS
            (An error message is issued if the
            input string contains a minus sign
            and the result is set to zero.)
        Character test.                           8192   RK_CTST
```

(Tests for presence of characters other than numbers, signs, or decimal points. This bit is always set by **inform** and **kwscan** when reading numeric variables, but it is not the default because certain cards in legacy applications may legitimately have letters in numeric fields.)		
Zero test. (An error message is issued if the input value is zero and the result is set to a small positive value.)	16384	RK_ZTST

'width' is one less than the width of the input or output field in bytes (maximum 64-1 (**wbcdwt**), otherwise 32-1). The maximum number of digits that can be generated is limited to the number that can be represented by a double precision variable, however, the remaining width can contain the sign, decimal point, exponent, or blanks as required.

Additionally, the global RK.expwid can be set by the user before a call to **bcdout**. 'expwid' is used only for E format output, and specifies the number of digits in the printed exponent (including its sign). If 'expwid' is zero or larger than EXP_SIZE, the number of digits needed to express the largest exponent that can be represented on the machine, 'expwid' is replaced by EXP_SIZE (defined in sysdef.h). If 'expwid' is too small to represent the result, a larger value is substituted. If E format is forced because a number overflows the space available, the exponent width is always the smallest needed to hold the exponent.

Value returned: Function **bcdin** returns a floating point result in double-precision format. When **bcdin** is used for single-precision input, the appropriate 'ic' code may be used to force rounding, which may not be performed automatically when the returned value is coerced to single precision.

Function **wbcdin** returns a fixed-point result of any supported length via the void pointer 'pitm' in the call. An error message is issued if the input value would overflow the specified variable length.

Accuracy: Because portability is the goal, accuracy is limited to what can be obtained using ordinary double-precision arithmetic (no more than one bit error for **wbcdin**).

Miscellaneous Notes:

1) In E format output, the letter 'E' is not generated. The mantissa is followed directly by the signed exponent. The mantissa always has one digit before the decimal point. For example 1.893+27 = 1.893 x 10**27. On input, a sign following one or more digits is taken as the start of an exponent field. The letter E is optional, but will always be taken as introducing an exponent, regardless of the RK_CTST flag.

2) If the input does not contain a decimal point, the decimal default will be applied, even if an exponent is present.

3) If the RK_CTST flag is on, one or more numeric digits must be present in the input field or an error message is generated. If RK_CTST is off, a field with no digits is legal, and is interpreted as the value zero.

4) Use of RK.length: On output calls, the number of non-blank characters in the output field, less one, is returned in RK.length. This information may be useful, for example, when constructing an output string from a concatenation of left-justified output fields (see J code in **convrt** call).

Error Procedures:

1) Output is automatically converted to E format when the number to be converted overflows the allotted field, provided there is room for at least one blank, one significant figure, and the exponent. The output field is never expanded beyond the specified width. The decimal point is placed to give the largest number of significant figures that will fit in the field (but not more than 'd' -1). If the width is too small, the field is filled with asterisks.

2) On input, no error is generated for an invalid character in a numeric field unless code RK_CTST is specified. This permits conversion of numbers from unparsed control card fields such as "DATA SET 5". Blanks are always treated as nulls on input.

3) An overflow can occur on fixed point input conversions if the number is larger than the maximum variable size specified in the 'ic' code. When this happens, a message is issued and the largest possible signed or unsigned value is returned. However, for signed or unsigned halfword or character types, if the result is exactly one larger than the largest value that can be represented, it is reset without issuing an error.

4) Abexit 40 is issued if an invalid combination of control parameters is supplied, for example, binary scaling with octal

format output. Abexit 41 is issued if hexadecimal output was requested to contain the number of digits implied by the variable type, but the field was not wide enough to contain this many characters. Abexit 199 is issued if certain internal checks fail. If this happens, please contact the author with details.

Restrictions (symbols defined in sysdef.h):

1) No more than OUT_SIZE (16) decimal digits (20 for **wbcdwt**) can be handled.

2) Exponents larger than EXP_SIZE can not be handled.

3) Program assumes 8 bits per byte.

4) Floating and integer variables must be stored with the same byte order.

5) The representations of the characters '0' through '9' and 'A' through 'F' must be consecutive codes.

Example:

To convert the floating-point variable THETA to a 12 character decimal field with four digits to the right of the decimal point and insert the answer in array PUTOUT, the calling parameters would be SCALE = 0, D = 5, DECIMAL = 327680, OPERATION = 1792, WIDTH = 12 and the function call would be:

```
bcdout(5*RK_D+RK_IORF+12,putout,theta);
          -or-
bcdout(329483,putout,theta);
```

Function **SIBCDIN**

This function provides a subset of the facilities available in the functions **wbcdin** and **scan** in a completely self-contained unit. It is intended mainly for use in environments (parallel computers, MATLAB, etc.) where the full ROCKS card/page formatting facilities may require too much memory or may be unwanted for other reasons. **sibcdin** supports scanning and fixed-width number conversion, but not floating-point or general fixed-point numbers.

Usage: long **sibcdin**(int ic, char *field)

Prototyped in: rkxtra.h, rksubs.h

Arguments: 'ic' is the sum of any of the following processig codes. (RK_SSCN is unique to this routine; the other codes have the same definitions and values as their counterparts used with **wbcdin**):

```
RK_HEXF  (0x0200)   to get hexadecimal conversion,
RK_OCTF  (0x0400)   to get octal conversion,
RK_IORF  (0x0600)   (or no code) to get decimal conv,
RK_SSCN  (0x0800)   to perform simple scan, i.e. terminate
                    number at first whitespace,
RK_QPOS  (0x1000)   to give an error if the input value is
                    negative,
RK_CTST  (0x2000)   to give an error if the input string
                    contains nonnumeric or nonwhitespace
                    characters,
RK_ZTST  (0x4000)   to give an error if the input value is
                    zero.
```

To these codes, add one less than the maximum width to be scanned (max 31).

'field' is the location of the input field to be scanned. If the RK_SSCN bit is set, input conversion terminates when a blank or tab or null character is found. Otherwise, the full width is scanned. The location of the next character following the terminating character is retained by sibcdin, and scanning resumes there on the next call if 'field' is a NULL pointer.

Globals: char *sibcdinf on return points to the next character following the field that was converted. If *sibcdinf is zero, the scan reached the end of the input string.

Errors: **sibcdin** terminates with abexit code 55 if the new and previous field locations are both NULL pointers, with 56 if a nonnumeric character is found and the RK_CTST flag was set, and with 57 if the numeric value is invalid.

<u>Subroutines **WSEEDIN** and **WSEEDOUT**</u>

These routines may be used, respectively, to read or format for output a 'wseed' (pair of 31- and 27-bit random number seeds). Formats recognized by **wseedin** are (1) 'nnnnn', a single integer, 0= < nnnnn < 2**31-1, sets the seed31 component of the result to nnnnn and the seed27 component to -1. (2) '(mm,nn)', two integers separated by a comma and enclosed in parens, with mm <= 2**27 and 0= < nn < 2**31-1, sets seed27 to mm and seed31 to nn. mm < 0 sets compatibility with old udev(), nn == 0 causes all wdevxx calls to return 0 for tests. **wseedin** assumes **cdscan** has been called and the input scanned such that the wseed to be read in is the next input field.

Usage: void **wseedin**(wseed *pwsd)
        void **wseedout**(wseed *pwsd, char *field, ui32 ic)

Prototyped: wseed typedef is in sysdef.h, routines in rkxtra.h

Arguments: 'pwsd' is a pointer to the wide seed to be read in or converted for output.

'field' is a pointer the string where output is to be placed.

'ic' is a wbcdwt-style control code. Only the width and left-justify codes are used.

HIGHER-LEVEL ROUTINES

Function **JFIND**

Function **jfind** is used to search for a particular character string on a control card independent of ROCKS syntax (except that it distinguishes user data from comments and ignores the latter). Continuation cards are not read or scanned. Because it does not obey the rules for control cards given in the User's Guide, **jfind** should be used with caution. It does, however, provide a convenient way to find a particular keyword without invoking the more ponderous **cdscan**/**scan** routines. Use of **jfind** is generally indicated in writeups by the phrase 'xxx may be punched anywhere on the card'.

Usage: int **jfind**(char *card, char *key, int idspl)

Prototyped in: rkxtra.h, rksubs.h

Arguments: 'card' is the card array to be scanned.

'key' is the literal string to be found.

'idspl' is the number of columns displacement (= column number - 1) from the beginning of the card at which the search is to begin.

Value returned: Function **jfind** returns the displacement of the search key if the key is found, and zero if the key is not found. (Ambiguity is possible if 'idspl' = 0, a case that should be avoided.)

Function **SSMATCH**

Function **ssmatch** ("single-string match") is used to determine whether a given string is an initial substring of another string. A match indicates that the first string (the "item") is a proper abbreviation of the second (the "keyword"). The number of characters that match is returned, permitting the calling program to determine whether the abbreviation is unique according to the rules given in the User's Guide. Keywords may contain a period, indicating a qualifying prefix that may be present or omitted in a matching item. Function **ssmatch** should be used to match all card identifiers and keywords in order that the rules for abbreviation may appear uniform to the user for all his or her input. Function **ssmatch** is used for this purpose by **match**, **smatch**, and **kwscan**.

Usage: int **ssmatch**(char *item, char *key, int mnc)

Prototyped in: rkxtra.h, rksubs.h

Arguments: 'item' is the item to be tested. It may be terminated by a standard C end-of-string character, by a blank if past the minimum columns required to match, or by a colon. (This definition permits **smatch** (which calls **ssmatch**) to be used directly to identify control cards.) If the key contains a period, the matching item may or may not contain the period. The comparison is not case-sensitive.

'key' is the full identifier to which the item is to be compared. It may be terminated by a standard C end-of-string character or by a percent sign (for use by **kwscan**) but never by a blank. The key may be given in any mixture of upper and lower case. The key may contain a period. If 'mnc' < 0, this indicates a qualifying prefix. A matching item may omit the prefix.

'mnc' is the minimum number of initial characters which must be identical in order for a match to be accepted. The normal value for this argument is 1. A value mnc < 0 indic-cates that a period in the key is to be treated as a qualify-ing prefix, otherwise it is an ordinary match character. The absolute value |mnc| is the minimum match.

Value returned: **ssmatch** returns 0 if 'item' is not an initial substring of 'key', otherwise it returns the number of initial characters that match (i.e. the length of 'item').

Functions **MATCH** and **SMATCH**

Function **match** is used to compare a field with a list of acceptable character string keys. It calls **scan** to parse the field, checks punctuation, then attempts to match the field to a given list of keys. It returns the ordinal position of the successful match in the list, or zero if there is no unique match. Function **smatch** is similar, but it takes the string to be matched as an argument and does not call **scan**. Both functions allow the keys to be of mixed lengths and cases. Both functions return only unambiguous matches as defined in the User's Guide. (Matches are tested by **ssmatch**. If an exact match is found, it is returned; otherwise the longest unique match is returned.)

```
Usage:  int match(int keq, int iscn, int mask, int ipnc,
            char *keys[], int nkeys)
        int smatch(int keq, char *item, char *keys[], int nkeys)
```

Prototyped in: rocks.h

Arguments: 'keq' controls error checking. It is the sum of any of the following codes that are desired: Add RK_EQCHK (value 1) or RK_BPMCK (value 4) to generate an RK_EQRQERR error if the most recently scanned field was not delimited by an equals sign (possibly with plus|minus). Add RK_NMERR (=2) to suppress generating an error if the item is not matched.

'iscn' controls field scanning from **match** as follows:

```
= 1                  obsolete--replace by smatch call
= RK_MSCAN  (=2)     match executes scan(field,0)
= RK_MNEWK  (=3)     match executes scan(field,RK_NEWKEY)
= RK_MDLIM  (=4)     match executes scan(field,RK_NEWKEY|
                     RK_PMDELIM|RK_ASTDELIM|RK_SLDELIM)
= RK_MREQF  (=5)     match executes scan(field,RK_REQFLD)
```

(where 'field' is an internal buffer containing space for up to 16 characters).

'mask' and 'ipnc' are codes used by **match** for punctuation checking. After scanning, if (RK.scancode & mask) != ipnc, an error occurs (see below).

'item' is a pointer to the character string to be matched by **smatch**.

'keys' is a pointer to an array of strings, each of which is a possible match for the item being tested. Each key is terminated by a standard C end-of-string (null) character. The last character may not be a blank.

'nkeys' is the number of keys in the 'keys' list.

Value returned: Both functions return the ordinal position of the item in the list, or zero if the string is not matched. Negative values are returned for other errors, as follows:

RK_MPERR (=-1) A punctuation test failed
RK_MENDC (=-2) **scan** was called and no field was found (the end of the input card was reached).

Error procedures: If no match is found and (keq & RK_NMERR == 0), an error message is issued and RK.iexit is set nonzero. If a punctuation test fails, an error message is issued, RK.iexit is set nonzero, and RK_MPERR (=-1) is returned. If **scan** is called and no field is found, RK_MENDC (=-2) is always returned, but an error message is issued and RK.iexit is set nonzero only if 'iscn' was RK_MREQF (=5). If 'iscn' is not one of the values defined above, execution is terminated with abexit code 26.

Note: The user must call **cdscan** prior to calling **match**. 'maxlen' must be no larger than DFLT_MAXLEN (normally 16).

Function **MCODES**

Function **mcodes** is used to initialize a flag word according to a string of one-letter option codes entered by a user. The 'keys' argument specifies a list of possible option codes. These codes map onto bits in the flag word in order from right to left. Each character in the 'data' string is compared with all of the characters in 'keys'. When the n'th code from the right of 'keys' is found in 'data', the n'th bit from the right of 'flagword' is selected. The use of the selected bits depends on the first character in the data as follows:

+	OR the result with any existing codes in 'flagword',
-	Clear bits in 'flagword' corresponding to given codes,
other	Replace 'flagword' with the selected bits.

If the data string is literally "0", then 'flagword' is set to 0 unless "0" is itself a recognized option code. If any character in the data is not matched, an error message is generated, RK.iexit is set nonzero, and mcodes returns an error code of 1.

To handle special situations, the result bits are left in RK.mcbits and a code indicating the selected action is left in RK.mckpm (see **mcodop** routines below). Values of this code are defined in rkxtra.h.

Usage: int **mcodes**(char *data, char *keys, ui32 *item)

Prototyped in: rkxtra.h

Arguments: 'data' is a character string containing the codes entered by the user, typically a scanned control-card field. No more than 32 characters can be processed.

'keys' is a string that specifies the codes recognized in the current data. Blanks may be used to skip over bits that are not to be set in 'item' by any code. The length of 'keys' may not exceed 32 characters. While any characters except '+' and '-' may occur in 'keys', in general only letters and numbers should be used. If any characters are used that might be interpreted by **scan** as delimiters, the 'data' string will have to be enclosed in quotes by the user.

'item' is the control word to be initialized by **mcodes**. A NULL pointer can be passed if the only desired action is to store code information for a subsequent call to an **mcodop** family routine.

Value returned: 0 if matching is successful, otherwise 1.

Error procedures: **mcodes** terminates with abexit code 25 if more than 32 keys or codes are specified.

Subroutines **MCODOPC, MCODOPH, MCODOPI,** and **MCODOPL**

Subroutines **mcodopc**, **mcodoph**, **mcodopi**, and **mcodopl** can be used to apply the transformation from the last **mcodes** call to additional flag words. This is useful when a control card must update selected options for a set of related objects or when the type of the flag word is not a ui32 integer. The routines are the same except for the type of the flag word to be modified.

Usage:
```
void mcodopc(unsigned char *cflag)
void mcodoph(unsigned short *hflag)
void mcodopi(unsigned int *iflag)
void mcodopl(ui32 *lflag)
```

Prototyped in: rkxtra.h

Arguments: 'cflag', 'hflag', 'iflag', or 'lflag' is a control word of the indicated type. It is initialized in exactly the same way as the 'item' argument to the most recent **mcodes** call.

Function **MCODPRT**

Function **mcodprt** is used to prepare an output string that can be used to list single-letter or digit codes corresponding to binary options that have been encoded as bits in a word, possibly by a call to **mcodes**.

Usage: char ***mcodprt**(ui32 item, char *keys, int olen)

Prototyped in: rkxtra.h

Arguments: 'item' is a word containing the binary flags to be reported. (Items shorter than a full word can be typecast to ui32.)

'keys' is a string that specifies the codes to be reported. Only upper-case letters and digits are meaningful flag codes. Each character in 'keys' that corresponds, scanning from right-to-left, with a 1 bit in the 'item' is copied to the output. Minuses may be used to indicate bit positions that should not be reported. The number of bit positions reported is the lesser of olen, the length of 'keys', or 32, the number of bits in a ui32 item.

'olen' is the maximum length of the output. If coded as a positive integer, the output is padded with blanks if necessary to the exact length specified. If coded as a negative integer, the output will be whatever length it takes to report the item, up to a maximum of olen. If abs(olen) is greater than 32, the number of bits in a ui32 word, it is quietly replaced with that number. If olen is too short to contain all the output, the excess is simply dropped.

Return value: Character string containing the codes from 'keys' that correspond to 1 bits in 'item'. This string is contained in a static array and remains valid until the next call to **mcodprt**. The string is terminated with a NULL character that is not counted in the maximum length 'olen'.

RK.length is set to one less than the number of data characters (excluding padding) in the output string.

Errors: Execution is terminated with abexit code 25 if the length of the 'keys' string is greater than 32.

Function **KWSCAN** (formerly **XXSCAN**)

Function **kwscan** is used to match a set of keys on a control card and to read parameter values or set flags as directed by corresponding action codes. Before **kwscan** is called, the card must be read with **cryin** and **cdscan** must be called to initialize scanning. (The **cdscan** call can be made indirectly through **inform** with an S format code, see below.)

Usage: int **kwscan**(ui32 *ic, char *key, void *arg, ..., NULL)

Prototyped in: rkxtra.h

Arguments: 'ic' is a pointer to a ui32 integer. **kwscan** sets the n'th bit (bitset/bittst counting method) of 'ic' when the n'th key (n < BITSPERUI32) is encountered in the data. The caller can use this information to determine which options were found in the data. (The caller must zero 'ic' before the first call, thus allowing these bits to accumulate across '%X' exits.) If there are more than BITSPERUI32 (defined in sysdef.h) keys, the information for the excess ones is lost.

'key' and 'arg' specify keywords, actions, and variables. There is one 'key' argument for each keyword to be matched, consisting of a string (mixed case is acceptable) followed by a % separator, optionally the letter 'R', optionally the letter 'P' followed by an integer, and an action code. Keys may include '.' qualifying prefixes as described for **ssmatch**. The end of the key list must be marked by a NULL pointer. When an 'R' is present, and a data item matches that key, the next argument, assumed to be an int, is tested. If the result is 0, an error is generated, otherwise conversion proceeds normally. When a 'P' (for 'prefix length') is present, the following integer is the minimum number of characters needed to match that key (this is needed when keys recognized by a default processor (see below) have a common prefix with one of the enumerated keys). Following the 'R' and/or 'P', each 'key' is followed by 0 (code X), 2 (codes J,O) or 1 (all others) 'arg' pointers. The types of the args are implied by the codes. The input card is scanned and each field is compared with all of the keys using **ssmatch**. When a match is found, the action specified by the associated action code is performed. There may also be a default action to be invoked if no key is matched. This is coded by '%%' followed by an action as above. The default case must follow the last explicit keyword. If no match is found, and there is no default action, **ermark** error RK_IDERR is generated and another field is tested. The possible codes and actions are as follows:

Code	Action
X	Execute. **kwscan** returns to the caller for whatever action is necessary to process the key. The function value returned by **kwscan** is incremented by one for each 'X' code encountered up to and including the one matched, so a switch based on the return code can be

used to select the appropriate action. The delimiter after the keyword is not checked; the exit routine can use RK.scancode to check it if desired. There is no corresponding 'arg' argument. The exit routine can do anything necessary to process the keyword, including reading further fields with **scan**. When the exit routine is finished, it should branch back to the original **kwscan** call to resume scanning the card.

J[n]fmt JName. Executes a routine registered by a previous call to **kwjreg** to process the data field. 'n' is an integer from 0 to 9 that specifies which of 10 possible routines should be called. Two arguments are passed to the user routine. The first, presumably some sort of formatting information, is a pointer to the character following 'n' in the action code. The second, presumably a pointer to the result, is passed from the next 'arg' to the user routine.

N[n] Name. Same as 'J' except executes a routine registered by a previous call to **kwsreg** instead of **kwjreg** and only a single argument, presumably a pointer to the result, is passed from the next 'arg' to the user routine.

O[C|H|I|J|L|W][~] Or [now flag set/unset]. Two arguments must follow this code. Both are pointers to type char, short, int, ui32, long, ui64, or int, according to whether the O code is followed by C,H,I,J,L,W, or nothing, bzw. If the keyword is followed by a comma, or by an equal sign and one of '1', 'ON', 'TRUE', or 'YES', the second argument, typically an option flag, is OR'd into the first. If the keyword is followed by an equals sign and one of '0', 'OFF', 'FALSE', or 'NO', the bit represented by the second argument is cleared in the first argument. If the width code is followed by '~', these actions are reversed.

S[C|H|I|J|L|W]m Set. The variable pointed to by 'arg' is set to the value 'm', which may be positive or negative. The type of 'arg' defaults to int, but is taken to be char, short, int, si32, long, or si64 if 'S' is followed by C,H,I,J,L, or W, respectively. If 'm' is not given, an abexit occurs. The 'S' option is used to set program flags or switches The keyword must be followed in the input by a comma, not an equals sign. (For compatibility with earlier versions, 'm' may be followed by the type spec letter rather than preceded.)

For the remaining codes, information from the card is to be input into the variable 'arg' points to. The card being scanned must contain the construction 'key = value', where 'value' is a value that is converted and stored in 'arg'. The type of conversion is specified by the code as follows:

[V]An Alphanumeric. The value is a string, which is copied, up to a maximum of 'n' characters, to 'arg'. The string is terminated with a C end-of-string character ('\0'). If the string is longer than 'n' characters, **ermark** error RK_LENGERR is generated. Option code 'VA' causes the string to be converted to upper case.

[V]I[C|H|I|J|L|S|W] Integer. The value is an integer. The type of 'arg' defaults to int, but is taken to be char, short, int, si32, long, wseed, or si64 if the 'I' code is followed by C,H,I,J,L,S, or W, respectively. Option code 'V', used with I, U, F, D, or Q (see below), may be written 'V>[nnn]' or 'V>=[nnn]' (argument must be >nnn or >=nnn, respectively, nnn defaults to 0), 'V<[nnn]' or 'V<=[nnn]' (argument must be <nnn or <=nnn, respectively, nnn defaults to 0), 'V' (argument must be >= 0), or 'V~' (argument must be nonzero). 'V' may be replaced by 'W' to generate a warning instead of an error when the range checks fail. 'V' is unnecessary with 'IS'.

[V]BsI[C|H|I|J|L|W] Fixed point number with binary scale 's'. The value is converted as described above for code 'I' except that it may have an integral and a fractional part. The value is stored with 's' fraction bits. 's' may consist of two integers separated by a '/' or '|'. With '/', the second scale is used if **bscompat**(1) has been called; with '|', the second scale is used if **bscompat**(2) has been called; otherwise the first scale is used.

[V][Bs]U[C|H|I|J|L|W] Unsigned integer. Same as 'I' or 'BsI' except value is an unsigned integer.

[V]F[.d] Floating-point. A real value is converted and stored in an 'arg' of type float. If the input string has no decimal point, 'd' digits to the right of the point are assumed.

[V]Q[.d] or [V]D[.d] Double-precision. A real value is converted and stored in an 'arg' of type double. If the input string has no decimal point, 'd' digits to the right of the point are assumed.

T[C|H|I|J|L]n Text. The value is a string of maximum length 'n' characters. The string is read in and saved in the text cache and the text locator returned by **savetxt** is converted to an unsigned integer of type byte, ui16, int, ui32, long, or int according to whether the 'T' code is followed by C,H,I,J,L, or nothing, respectively, and saved in 'arg'. (I is optional.) Note: Text locators are consecutive integers starting at 1. Format 'TC' should be used only if the caller is certain that there can be no more than 255 different text strings in the application. If the string is longer than 'n' characters, **ermark** error RK_LENGERR is generated.

K[C|H|I|J|L]codes Codes. 'codes' represents a string of from one to 32 characters. The argument is taken to be an unsigned integer of type char, short, int, ui32, ui32, or int according to whether the K code is followed by C,H,I,J,L, or none of these four letters, respectively. ('I' may be used to specify int type when the first code letter is one of the other type specifiers. 'L' is a deprecated synonym for 'J' for compatibility with old programs. There is no current provision to handle 64-bit option fields, but if such a feature is added, 'L' and 'W' will be used to specify long or ui64 arguments, respectively. The input value must be a string, consisting of a subset of the characters in 'codes'. The characters in the string are matched to the characters in 'codes'. For each code character that is found in the input, the corresponding bit in 'arg', counting from the right, is selected. Selected bits in 'arg' are set or reset, or the entire arg is replaced, depending on whether the user string begins with '+', '-', or neither sign. (see **mcodes** above).

Codes I,U,F,Q,D may be followed, after any C,H,I,J,L,W or .d specification, by a metric units specification. A metric units specification consists of a '$' followed by a default units multiplier (a,f,p,u,m,c,d,-,D,H,M,K,G,T,P,E where a=atto, f=femto, etc. and '-' is required for unmodified base units), followed by a literal string to specify the name of the units to be entered. For example, "$mV" would indicate that a number of millivolts is to be entered. The user can then enter a bare number, which is scaled by any default decimal, or a number followed by any multiplier and a 'V', in which case the input is scaled appropriately, e.g. "4V" would be read in as 4000mV. If the unit name ends with a digit (2,3,etc.) it indicates that the units are taken to that power, and the input is scaled accordingly. E.g. if the units are "$cm2", an input of "30mm2" would be read as 0.3cm2. Also, a '$' can be followed by two unit specifiers separated by a '/', e.g. "cm/sec2". Both units in the quotient can be scaled individually. Also, variables read in with the I,U,F,Q,D formatw may be followed in the input by a '+' or '-' sign and the name of an adjustment value. If a value with that name was stored by a previous call to **svvadj**, the adjustment if added to the input value before it is returned. If any other text follows the numeric value, an error is generated.

When **kwscan** reaches the end of the input card, it returns zero. There is no guarantee that all or any of the keys were actually found on the card; all variables ('argn') should be set to default values before **kwscan** is called.

Errors: Execution is terminated with abexit code 12 if a key is not followed by a '%' or if the '%' is not followed by one of the documented action codes; with abexit code 21 if a metric units specification contains unknown multipler code; with abexit code 43 if a keyword with code 'J' or 'N' is matched and **kwjreg** or **kwsreg** has not previously been called; with abexit code 24 if an 'S', 'A', 'T', or 'B' code is not followed by a number, with abexit code 111 if a numeric parameter for a range value check was longer than 16 characters, with abexit code 112 if a range check specified both '>' and '>=' or both '<' and '<=' tests, and with abexit code 113 if a range check on an unsigned value specified either a maximum or a minimum of zero. These are considered to be development errors and hence do not produce text error messages.

Restriction: The internal scanning buffer in **kwscan** can contain no more than 32 characters. This restriction does not apply to alphanumeric fields read with the 'An' or 'Tn' code.

Example:

A DATASET control card might have the following fields: (a) 'ID=name', where 'name' is a 4 character identifier to be read into 'setid', (b) 'UNIT=tape', where 'tape' is a unit number to be read into an integer variable called 'iunit', and (c) 'OPT=flags', where 'flags' are a subset of the codes 'ABCD' to be used to set the four rightmost bits in integer variable 'flags'. This card could be read by the statements:

```
ui32 ic;
cdscan(card,7,16,0);
kwscan(&ic,"ID%a4",setid,"UNIT%i",&iunit,"OPT%KIABCD",&flags,
    NULL);
```

Subroutine **BSCOMPAT**

Subroutine **bscompat** is used to facilitate compatibility of binary scaling between different versions of an application. In all calls to **inform** or **eqscan** or **kwscan** subsequent to a call to **bscompat**, when a binary scale 'B' code is followed by two scales separated by a '/' or '|', the scale used depends on a previous call to **bscompat**. If **bscompat** has not been called, or was called with argument '0', the first scale is used. If the **bscompat** argument was '1', and the two scales are separated by '/', then the second scale is used. If the **bscompat** argument was '2', and the two scales are separated by '|', then the second scale is used.

Usage: void **bscompat**(int kbs)

Prototyped in: rocks.h

Argument: 'kbs' determines which scale is used when a fixed-point number is read and two possible binary scales are provided. Currently only the two low-order bits of 'kbs' are meaningful. See explanation above.

Subroutines **KWJREG** and **KWSREG**

Subroutine **kwjreg**, respectively, **kwsreg** is used to register (save pointers to) up to 10 routines that are to be called when a keyword with the 'Jn' or 'Nn' action code, respectively, is matched in a **kwscan** call or a format 'Jn' or 'Nn' is given in an **inform** call. The kwjfn() routines receive two arguments, the kwnfn() routines receive exactly one argument, as described above in the documentation for **kwscan**.

Usage: void **kwjreg**(void (*kwnfn)(char *, void *), int n)
Void **kwsreg**(void (*kwnfn)(void *), int n)

Prototyped in: rkxtra.h

Arguments: 'kwjfn', respectively, 'kwnfn' are pointers to functions that **kwscan** should call when a keyword with the 'Jn' or 'Nn' action codes is matched, where 'n' is an integer from 0 to 9. Registered functions can perform any action needed to establish a value for the input variable; in particular, they can make calls to **scan** to retrieve user information from the input card.

Function **EQSCAN**

Function **eqscan** reads from a control card a variable for which the keyword has already been identified. It is typically used in an exit routine entered from **kwscan**. Function **eqscan** simply combines the following frequently occurring actions: (a) verify required punctuation (ierr argument), (b) call **scan** to get the next field, and (c) convert the field according to a specified format.

Usage: int **eqscan**(void *item, char *code, int ierr)

Prototyped in: rkxtra.h

Arguments: 'item' is a pointer to the variable to be input. Its type is determined by 'code'.

'code' determines the type of conversion. Any of the codes defined for **kwscan**, other than 'J', 'N', 'O', 'S', or 'X', may be used. Codes may be given in mixed case.

'ierr' controls error checking. If 'ierr' is zero, the delimiter following the previous field is not checked. If RK_EQCHK is set and the previous field did not end with an equals sign, an RK_EQRQERR is generated. If RK_BEQCK is set and the previous field did not end with either a blank or an equals sign, an RK_PUNCERR error is generated. If both of these bits are set, the RK_BEQCK test is made if called from **(s)inform**, otherwise the RK_EQCHK test. If RK_PNCHK is set and the data are not enclosed in parentheses, an RK_PNRQERR error is generated. If RK_BPMCK is set, the test is the same as the RK_EQCHK test, except '+=', '-=' punctuation is acceptable as well as a single equals sign. In all error cases, RK.iexit is made nonzero, 'item' is not changed, and **eqscan** returns with a nonzero error code.

Value returned: 0 if there is no error, otherwise the **ermark** code of the particular error that was found.

Errors: Execution is terminated with abexit code 24 if an 'A', 'B', or 'T' code is not followed by a number or by abexit code 21 if an unrecognized metric units multiplier is encountered.

Restriction: The internal scanning buffer in **eqscan** can contain no more than 32 characters. This restriction does not apply to alphanumeric fields read with the 'An' or 'Tn' codes.

Subroutine **SSPRINTF**

Subroutine **ssprintf** ("subset sprintf") provides a small subset of the facilities available in the standard C library function **sprintf** in a completely self-contained unit. It is intended mainly for use in generating message texts on parallel computer nodes where other formatting facilities may require too much memory or may be unavailable altogether.

Usage: char *__ssprintf__(char *string, char *format, ...)

Prototyped in: sysdef.h

Arguments: 'string' is a character array large enough to hold the output generated by **ssprintf**, including a null terminator. If a NULL pointer is passed, **ssprintf** will use its own internal buffer, of size LNSIZE. The user is responsible to code the format string in such a way that the size of 'string' is not exceeded.

'format' is a message text possibly containing format control elements. The format string is copied to the output except as follows: (1) Each occurence of '%<n>s' results in copying up to a maximum of <n> characters from another string, taken from the next **ssprintf** argument, into the output. If the argument is a NULL pointer, nothing is copied and no error occurs. (2) Each occurence of '%<n>d' results in converting an integer, taken from the next **ssprintf** argument, into its decimal representation as a character string, and copying up to a maximum of <n> characters from this string into the output. If the code is '%<n>ld', the argument is a long rather than an int. If the code is '%<n>jd', the argument is a 32-bit integer even if the int type is smaller. If the code contains 'ed' rather than 'd', the length of the output is exactly <n> characters and any excess digits are discarded. (3) Each occurence of '%<n>X' or '%<n>x' results in converting an integer, taken from the next **ssprintf** argument, into its upper- or lower-case hexadecimal representation, respectively, and copying a maximum of <n> characters from this string into the output. If the code is '%<n>lx', the argument is a long rather than an int. If the code contains 'ex' or 'eX' rather than 'x' or 'X', the length of the output is exactly <n> characters and any excess digits are discarded. (4) Each occurrence of '%p' or '%P' results in converting a pointer, taken from the next **ssprintf** argument, into a hexadecimal representation of length appropriate to the processor and copying this string into the output (useful where pointers and ints are different lengths). (5) Each occurence of '%<n>m' results in converting <n> memory bytes from storage pointed to by the next **ssprintf** argument to hexadecimal and copying the resulting 2n characters to the output.

In all of the above cases, the length specification <n> can be an asterisk (*), in which case the length is taken from the next argument, which must be an integer. No escape characters are recognized. If an external buffer is used, then the string length specifications '<n>' are required. This is necessary to assure that the output cannot exceed the length of the 'string' argument.

Returns: A pointer to the result character string. If a NULL 'string' argument was used, the result is valid only until the next call to **ssprintf**.

Errors: **ssprintf** terminates with abexit code 58 if a length specification is missing (except for formats 'p' and 'P'), and with abexit code 59 if the format code is other than 's', 'd', 'p', 'P', 'x', 'X', or 'm'.

Note: The use of the count field is different than it is in standard **sprintf**. In **ssprintf**, it is the maximum field width, in **sprintf**, it is the minimum. Code 'ed' can be used to force longer than the minimum width to make aligned columns.

Note: When processing an unknown format string (e.g. in an implementation of **fatale** or **abexitm**, use the internal buffer to be sure buffer overflow is detected.

Note: There is no provision for formatting floating-point numbers or 64-bit integers in a 32-bit environment.

Subroutines **INFORM, SINFORM, CONVRT,** and **SCONVRT**

These subroutines provide facilities for formatted input (**inform**, **sinform**) and output (**convrt**, **sconvrt**) very similar to those provided in FORTRAN by formatted READ or WRITE statements. They are essentially interpreters which generate calls to the lower-level ROCKS interface routines under control of a pseudo-FORMAT statement. The various features of the interface routines are thus made available in a more convenient form. Subroutines **inform** and **convrt** perform I/O through **cryin** and **cryout**, respectively, while **sinform** and **sconvrt** operate on buffers in memory. When using **sinform** or **sconvrt**, it is an error to process more than one record or to use the L, P, or Y format codes. On return from any of these routines, a count of the number of items successfully converted is contained in RK.numcnv.

Input usage:
```
void inform(char *format, void *item, ..., NULL)
void sinform(char *card, char *format, void *item, ..., NULL)
```

Prototyped in: rkxtra.h

Output usage:
```
void convrt(char *format, void *item, ..., NULL)
void sconvrt(char *line, char *format, void *item, ..., NULL)
```

Prototyped in: rocks.h

Arguments: 'card' is a pointer to an input string to be converted by **sinform**. Subroutine **inform** can be made to read a card internally by including a '/' in the format code. Normally, however, **inform** will be used for scanning a card that has already been read by **cryin**. In that case, **inform** will use RK.last as the card location.

'line' is a buffer large enough to hold the output generated by **sconvrt**, including a null terminator character which is placed at the end of 'line' unless the 'O' format code is specified. Subroutine **convrt** does not use 'line'; it calls **cryout** internally to write each output record that is generated. The first character of each line is treated as an ANSI carriage control character, but if it is not recognized as such, then the previous line is continued. The longest line that can be generated by **convrt** or **sconvrt** is set by RK.pgcls.

'format' is the equivalent of a FORTRAN FORMAT statement (without the word "FORMAT") enclosed in parentheses and double quotes. It may contain any of the FORTRAN format codes except C, G, L, or P as well as additional codes defined below.

'item', ... represents a list of pointers to the variables or arrays to be converted. Control variables used with the L, R, T, ^, and # codes must be passed as pointers to integers. The list of items must be terminated with a NULL pointer.

There are several major differences in usage between **[s]inform** and **[s]convrt**, FORTRAN formatted I/O, and the C routines **[s]scanf** and **[s]printf**, which they resemble. Most important, the types and sizes of all the 'item' arguments must be explicitly coded in the format string. This is done as follows:

(1) A separate format code is provided for each type and length of variable, viz. A or T for string variables, F or E for float variables, Q or D for double variables, and I or U for integers. Codes I, K, T, and U may be modified by C for character, H for short, I for int, J for 32-bit integer, L for long integer, S for wide seed, and W for 64-bit integer, respectively. These distinctions must always be made or incorrect conversions and indexing will occur.

(2) To handle most cases involving arrays, the convention is adopted that a format repeat count implies that the corresponding I/O list variable is an array of the same dimension as the repeat count. For example, (I4,3I4) would specify the conversion of two list items, the first a single integer variable and the second an array of 3 integers. (The actual array may be equal to or larger than the number of items converted.) To handle other cases, the special format codes * and # are provided. '*', inserted between the repeat count and the format item, indicates that the corresponding list items are not arrays, and a separate list item is used for each conversion. For example, (3*I4) would specify that three separate variables are to be converted according to I4 format. '#' is used to indicate explicitly the dimension of an array, which may extend over several conversion codes. For

example (#10,4F6.2,6F9.4) would specify the conversion of a single array of 10 float elements, the first 4 according to F6.2 format, and the remaining 6 according to F9.4. If no dimension is given following the #, the next list item (assumed to be a pointer to an int) is picked up and used as the size of the following array.

Limited provision is made to handle intermixing of items from several arrays. This is done by enclosing the format codes to be repeated in <> with the repeat count before the <. Repetition of formats in <> is similar to that which occurs with () except that the same I/O list items are reused on each scan of the format. Each item is assumed to be an array, which is indexed to the next unused element on each iteration of the <>.

For example, given the declarations

```
int n; float a[3][3],b[5];
```

the call

```
convrt("(I6,3<3F6.2,F8.3>)",&n,a,b,NULL);
```

writes out n, then a[0][0], a[0][1], a[0][2], b[0], a[1][0], a[1][1], a[1][2], b[1], a[2][0], a[2][1], a[2][2], b[2] in that order. Note that a is processed in the order it exists in memory in C, effectively reversing the order of subscripts from what was implied in the FORTRAN version.

To step through array items referred to by formats in <> with stride greater than one, the = code can be placed before the <. The number following the = is the stride; if no number is found, the next list item, assumed to be a pointer to an int, is used for this purpose. The stride value is multiplied by the size of each item to obtain its indexing offset, which in turn is multiplied by the iteration number of the <> to obtain the total byte offset for that item, i.e. the stride is counted in numbers of items independent of the size of each item. The ^ code may be used to override the normal byte size assumed for a single item of any type.

For example, given float a[12],b[12], the call

```
convrt("(=R<F8.2F8.3>)",&is,&in,a,b);
```

with is = 4, in = 3 would print a[0], b[0], a[4], b[4], a[8], b[8]. The range of a # code may be entirely within or entirely outside a <> construct, but may not extend across the <>.

(3) On input, the last record read by **cryin** is processed first; additional records are read by calling **cryin** as needed.

The W format code causes the input card to be printed by **cdprt1**. Output prepared by **convrt** is always written; the W format code causes the output buffer to be flushed. This is different from the FORTRAN usage.

(4) A comma between two format specifications is optional except when needed to separate two numbers or letters, e.g. (I2I4,3F6.2). Commas in other positions are ignored. Blanks are treated as if they were commas.

(5) On output, decimal specifications must be one larger than the number of places to appear after the decimal point, as with **bcdout**. This permits .0 to be used to indicate that a real variable is to be printed in integer format (no decimal point printed). Decimals can be used with integer variables, with or without the binary point specification (B format). On input, decimal specifications work the same as in FORTRAN.

(6) The largest valid width specification is 32 characters except for the H and T format codes, which may have widths up to 256 characters. When input is scanned, width specifications for all formats except A and T may be omitted, as the actual widths of the scanned fields (up to DFLT_MAXLEN) are used. With A and T formats, a width is required; it is taken as the maximum that can be accommodated by the corresponding string item.

Individual format codes are interpreted as follows:

Aw Alphanumeric. Must be followed by 'w', the maximum number of characters to be transmitted. On input, shorter strings are truncated by an end-of-string character and longer strings give an error message. On output, shorter strings are padded with blanks to 'w' characters unless the 'A' code is preceded by 'Jn', in which case the actual string length is used, followed by 'n' blanks. 'w' may be followed by a '.d' specification, in which case 'd' is the displacement to the next variable (in characters) for indexing purposes (same as ^ code). If not specified, d=w.

.A The letter A appearing as a decimal point specification on output requests Automatic decimal placement. The decimal is placed to give the largest number of significant figures that will fit in the field after inserting a single blank at the start of the field.

Bn Placed before an I, M, or U format specification, B indicates Binary scaling. B must be followed by an integer, 'n', the number of bits to the right of the binary point ($0 \le n \le 63$). 'n' may also consist of two integers separated by a '/' or '|'. In the first case, the second scale is used if the RK_BSSLASH bit was set in a previous call to **bscompat**. In the second case, the second scale is used if the RK_BSVERTB bit was so set. Otherwise the first scale is used. The B specification follows any repeat count and applies only to the immediately following format code.

D Double exponential (same as FORTRAN).

E Exponential (same as FORTRAN).

F Floating (same as FORTRAN).

H Hollerith (same as FORTRAN).

I[C|H|I|J|L|B|S|W|Z] Integer. The I code may be followed by a type length modifier and a 'w.d' specification. The type modifiers C,H,I,J,L,B,S,W, and Z specify char, short, integer, si32, long, sbig, wseed, si64, and size_t types, respectively. The default is int. The '.d' specification is normally used only in conjunction with binary scaling.

J[n] Input: Calls routine 'n' (0 <= n <= 9) registered by a previous call to **kwjreg**. Everything after the 'n' until the the next comma is used as the first argument to the routine (max 7 chars).

J[n] Output: Left Justify. If a value for 'n' is given, 'n' blanks are inserted to the right of the output field-the width of the field is data dependent and the width given in the format is only a maximum width (which does not includes the 'n' blanks). If a value for 'n' is not given, enough blanks are inserted to fill out the width requested in the associated format item.

K Input: Keys expected. With scanned input (after an S code), indicates that nonnumeric keywords may appear after this point in the input. When an nonnumeric data item is found, **inform** calls **scanagn** and then returns so the calling program can call **kwscan**. Any remaining format items are ignored. Implies N, so no error message is generated for missing fields--these should be set to default values before calling **inform**. Without the 'K' code, nonnumeric items are treated as data.

K[C|H|I|J|L]w Output: Binary option Keys. This format requires two arguments in the argument list. The first is an ASCII string specifying the codes to be printed for each 1 bit in the second argument, which is a pointer to the data item to be converted. Characters other than uppercase letters and digits in the code specification indicate bit positions in the data that are not encoded in the output. The K code may be followed by a type modifier C,H,I,J, or L to specify that the corresponding argument is an unsigned character, short, int, ui32, or ui32, respectively (default unsigned int). This must be followed by 'w', the maximum number of characters to be transmitted. Shorter strings are padded with blanks to 'w' characters unless preceded by 'Jn', in which case the actual string length is used, followed by 'n' blanks. Not usable inside <> brackets. (L is a deprecated synonym for J for compatibility with old programs.)

L[n] Lines. The L code is used to force a page eject if the next 'n' Lines of output would not all fit on the current page. If a value for 'n' is not given in the format itself, a list item, assumed to be a pointer to an int, is read and used for 'n'. The L code results in a call to **tlines**, so the next 'n' lines must be printed with **cryout** or **convrt** for proper pagination.

M "Midi". M is an alternative code for halfword (short) integers, identical to 'IH'. This usage may be removed in a future version.

N[n|(] Input: Not-required or call to registered routine. Used with scanning input, 'N' followed by an digit 'n' (0-9) indicates that a routine registered by a previous call to **kwsreg** with value 'n' should be called to process this input. (If the variable is an array, use '^' to specify size.) 'N(' signifies that a '(' in the input ends the call to inform so the parenthesized information can be processed by some other means. 'N' alone signifies that following items in the input list are Not required, and will not be assigned if the end of the input card is reached. Additionally, If N is not specified, an error message is issued for each missing input variable--additional input cards, other than continuations, are never read to fill in missing variables.

N Output: Not-forced. N causes the RK_NFSUBTTL bit to be set in the next **cryout** call, i.e., the output is treated as a subtitle that does Not force a new page. If NS is specified, the RK_NTSUBTTL bit is set.

O Overwrite. With **sconvrt**, causes the null character at the end of the output string to be omitted. Otherwise, 'O' is ignored. 'O' may occur anywhere in the format string.

Pn The Priority of the printed output is set to 'n'. (Place this code before an output record is written. Default: 2)

Q "Quality". Q is the code for double-precision variables. It is otherwise identical to F.

R[~] Repeat. R may be used instead of a repeat count before (), <>, or a format code. It indicates that a list item, assumed to be a pointer to int, is read and used for the corresponding repeat count. Before format codes, but not () or <>, the repeat count may be zero to skip the next format code. If the list item is 0 or 1 and R is preceded by a number, that number is used as the number of items to skip or process, respectively. If R is followed by a '~', data are skipped on input and blanks are inserted on output corresponding to the skipped data items. If R is followed by a '?', the repeat count is saved but not used. Any following R code without '?' (for example, repeating a parenthesized format group), uses the saved count and a list item is not consumed. Both '?' and '~' modifiers cannot be used with the same 'R' code.

S[[W]n] Input: Scan. On input, S indicates that the card is to be scanned with **scan**. W, if present, indicates that 'n' is the number of words to skip; otherwise 'n' is in columns. **inform** calls **cdscan** unless 'n' is omitted, in which case **inform** assumes that **cdscan** was already called by the user (or by a previous **inform** call processing the same card). For each list item, one field is taken from the card and converted according to the corresponding format. With A or T format, the field will be truncated if it exceeds the width specification. With numeric conversion formats, the width specification is ignored and all characters found by **scan** are included in the conversion (max 32). Upon return from **inform**, **kwscan** may be called at once to interpret any keyword parameters that follow the fields read by **inform**.

S Output: Subtitle. On output, S indicates that the RK_SUBTTL bit is to be set on the next call to **cryout**, i.e. the output is to be treated as a subtitle. (See also the N format code.)

T[C|H|I|J|L]n Input: Text. The data item is a string of up to 'n' characters. The string is saved in the text cache and the text locator returned by **savetxt** is converted to an unsigned char, short, int, ui32, or long variable if the T code is followed by C,H,I,J, or L, respectively, (default: unsigned int) and stored in the location referenced by the next argument pointer. If the string is longer than 'n' characters, **ermark** error RK_LENGERR is generated.

T[n] Output: Sets the record poinTer to column 'n'. The output buffer is <u>not</u> blanked by **convrt**, so unfilled columns will contain whatever was left in them before the **convrt** call. This feature allows output from more than one call to be combined. The buffer can be blanked with nX if necessary. When printing occurs, the length of the line is computed from the most recent setting of the record pointer. Some output may be lost if the pointer is backspaced. If 'n' is not given in the format, a list item, assumed to be a pointer to an int, is read and used for the column number. This feature is useful for making graphs on the printer.

U[C|H|I|J|L|B|W] Unsigned integer. Same as I except the most significant bit of the numerical value on input or output is treated as data rather than as a sign. On input, a negative value produces an error. The type modifiers C, H,I,J,L,B, and W specify character, short, integer, ui32, long, ubig, and ui64 respectively.

V[<[=][nnn]|~|>[=][nnn]] Input: Verify or Underflow control. Placed before a format code, V<nnn or V<=nnn gives an error if items read by the following format spec are >=nnn or >nnn, respectively. V>nnn or V>=nnn gives an error if the data are <=nnn or <nnn, respectively. If 'nnn' is omitted, it defaults to 0. V~ gives an error if the value is zero. Multiple tests may be combined in one code. On alphabetic input, V causes the string to be converted to upper case.

V[(|)] Input: Verify data in parentheses. When a set of format codes is placed between 'V(' and 'V)', the input data items (if more than one) must be enclosed in parentheses. If there has not been an 'N' or '=' code and ')' or end of scan is found before all the data items have been read, an error occurs. If all the data items are satisfied before the ')' is found in the data, an error occurs and the excess data are skipped. When parentheses are found and 'V(' has not been coded (or the conditions of the '=' or 'K' format codes are satisfied), control returns to the caller and the rest of the format is skipped.

V Output: On decimal conversions, causes output generated by the following format specification to use E format if all precision otherwise would be lost. On hexadecimal conversions, causes the number of digits emitted to be determined by the size of the data item. (Code V+, never used, has now been removed.)

W Input: Warning. Same as 'V' except gives a warning rather than an error for data out of range. The value is replaced with a value just inside the allowed range. If followed directly by a comma, calls cdprt1() causing the card image to be printed, single-spaced.

W Output: Write. W causes the cryout buffer to be flushed every time it is written in the current call after the W is encountered.

X Blanks, same as FORTRAN.

Y[n] Output: The Y code causes the next 'n' lines to be duplicated in the **spout** output (when active). If 'n' is not present, a value of 1 is assumed.

Z[C|H|I|J|L|B|S|W|Z] Hexadecimal, same as FORTRAN.

@ Pad with zeros rather than blanks on output.

#[n] Gives dimension of next array as a value or list item.

' Output: Literal strings may be given, enclosed in single quotes. A quotation mark within such a string must be written as two quotation marks. Same as FORTRAN.

() Format codes enclosed in parentheses are repeated, as in FORTRAN. If the end of the I/O list is reached, interpretation of the format ends at the next right parenthesis and following literals are not printed. Only one level of nesting of parentheses is allowed.

* Indicates repeat count applies to separate list items.

\+ A + code appearing before an array specification following an 'N' or '=' format code indicates that values must be provided on the input card for either all members or the array or none. (Without the 'N' or '=', all values must be provided. With 'N' or '=' but not '+', any number of values can be provided.)

, Format separator. Optional except to separate numbers or ambiguous code combinations.

/ New record, same as FORTRAN. Calls **cryin** for input.

: Same as ;

; On output, causes interpretation of the format to terminate if all repeat counts have been satisfied and there are no more list items to be converted. It may be used to turn off the conversion of following literals.

<> See discussion above.

.<n Variant of automatic decimal in which the decimal parameter is the lesser of 'n' and the automatic value needed to just fit the value in the given width with one leading blank.

= When used with scanned input (after an S code), indicates that keyword parameters (fields ended by an equals sign) may appear after this point in the input. When an equals sign is found, **inform** calls **scanagn** and then returns so the calling program can call **kwscan**. Any remaining format items are ignored. '=' also implies 'N', so no error message is generated for missing fields--these should be set to default values before calling **inform**. Without the '=' code, an equals sign in the input does generate an error message. In situations other than scanned input, the '=' code gives the index increment for <> repeats as already described.

^[n] Stride (item size) override. Replaces the built-in item length with the value 'n' (bytes) for the next format item only. If 'n' is not given in the format, a list item, assumed to be a pointer to an int, is picked up and used. The ^ code is useful for indexing through an array of structures.

| Indent. On output only, records the current column location. Following this code, when the format scan returns to a left parenthesis on a new line, a sufficient number of blanks is inserted to cause the output to be indented to the recorded position.

Codes I,U,F,Q,D may be followed, after any C,H,I,J,L,W or .d specification, by a '$' metric units specification. A metric units specification consists of a '$' followed by a default units multiplier (a,f,p,u,m,c,d,-,D,H,M,K,G,T,P,E where a=atto, f=femto, etc. and '-' is required for unmodified base units), followed by a literal string to specify the name of the units to be entered. For example, "$mV" would indicate that a number of millivolts is to be entered. The user can then enter a bare number, which is scaled by any default decimal, or a number followed by any multiplier and a 'V', in which case the input is scaled appropriately, e.g. "4V" would be read in as 4000mV. If the unit name ends with a digit (2,3,etc.) it indicates that the units are taken to that power, and the input is scaled accordingly. E.g. if the units are "$cm2", an input of "30mm2" would be read as 0.3cm2. Also, a '$' can be followed by two unit specifiers separated by a '/', e.g. "cm/sec2". Both units in the quotient can be scaled individually. If any other text follows the numeric value, an error is generated.

Errors:

If an error occurs processing user input data, the appropriate **ermark** message is generated and RK.iexit is set nonzero. If there is an error in the construction of a format string, the application is terminated with one of the following abexit codes:

- 11 Format does not begin with '('.
- 12 Unrecognized format character.
- 13 End of I/O list reached while trying to pick up 'n' for L, R, T, #, or ^ format code.
- 14 Digit not found when required (e.g. width of an item or after B code).
- 15 Input-only function attempted on output or output-only function attempted on input (e.g. scanning attempted on output, justify attempted on input, or input attempted to Hollerith string).
- 16 Buffer length exceeded.
- 17 Negative or zero repeat count where not allowed.
- 18 Attempted I/O from sconvrt or sinform call.
- 19 Unmatched quotes in output format or V() in any format.
- 20 An output format contained an invalid decimal construction (not '.n', '.A', or '.<n' with 'n' an integer).
- 21 An unrecognized default unit multiplier is given.
- 43 Attempt to call an unregistered kwsreg or kwjreg routine.

Subroutines **FPRINTF, PRINTF, PUTS, RFPRINTF, SNPRINTF, SPRINTF**

These routines provide facilities for formatted output compatible with the C library formatted output functions. They include extra formatting codes to provide all the features of **convrt**/**sconvrt**, including overflow and underflow protection, fixed-point numbers with fractions, array indexing, indenting, etc. The idea is to allow programs and library functions that call **[fs]printf** to coexist with programs that need the extended features provided here. **fprintf** writes to an open file stream; **printf** writes to stdout via **cryout** for page numbering, title and subtitle control; **puts** is like the standard library **puts** except it counts lines for **cryout** pagination; **rfprintf** writes output to an rfdef stream; **snprintf** writes to a string with length control; **sprintf** writes to a string with the dangerous lack of length control found in the standard library **sprintf**.

Usage:

```
int fprintf(FILE *stream, const char *format, ...)
int printf(const char *format, ...)
int puts(const char *string)
int rfprintf(rkfd *rffile, const char *format, ...)
int snprintf(char *str, size_t size, const char *format, ...)
int sprintf(char *str, const char *format, ...)
```

Prototyped in: **rfprintf** in rkprintf.h; the others in the system header file stdio.h.

Arguments: 'stream' is a file stream opened with **fopen** for output.

'rffile' is a file stream opened with **rfopen** for output ('rkfd' is a typedef synonym for 'struct rfdef').

'str' is a pointer to a string where the output will be stored. In the case of **snprintf**, it must be long enough to contain no more than 'size' characters. In the case of **sprintf**, the user is responsible to provide enough space for whatever output will be generated, however, the ROCKS version of **sprintf** supplies an internal limit of 255 characters to protect somewhat against runaway outputs.

'format' is a format control string that is designed to obey all the conventions of the standard library **printf**, with many extensions as documented below. When the standard formatting constructs are used, the corresponding arguments in the variable-length argument list ('...' in the prototypes) are the single data items to be converted, assumed to be promoted to larger sizes per the standard C rules for argument promotion; but in the extensions, it is possible to specify that an argument is a pointer to the data item, in which case the type is specified in the format and the item is not promoted.

Format string description:

There are several major differences in usage between this ROCKS implementation of **[fs]printf** and the standard C library versions, but they are designed to work as expected in normal applications. Most important, additional '%' codes are provided in the format specifier to permit array indexing and other extensions. By design, these are codes that should never occur in traditional **[fs]printf** calls. To allow direct access to data values that may be arrays, and to avoid argument promotion that may differ on different machines, special flag code '&' is used to indicate that the argument is a pointer to a value of the type required by the size and type conversion codes. This is optional when an array address is implied by repetition codes.

Output widths in standard **[fs]printf** are minimum widths. Here they are exact widths, which permits predictable table formatting. Instead of expanding the width, values that are too large to fit in the specified width are converted to exponential format. If there is not room even for that, the field is filled with asterisks. However, if a width specification is omitted, or flag 'nJ' is used, the minimum necessary width will be used, as in the standard versions. The largest valid width specification for numeric output is (RK_MXWD+1) = 32.

Traditional size codes before a conversion character will be recognized whether or not the '&' flag (see below) is present. A few additional size codes have been added. The recognized size codes are: 'hh' (char), 'h' (short), 'j' (32-bit regardless of size of int or long on the machine), 'l' (long), 'll' (long long), 'w' (64-bit), and 'z' (size_t). Size codes L,j,q,Z defined in some implementations of **[fs]printf** are used for other purposes here, code 't' will result in abexit 180. Similarly, nonstandard flags 'I' and ''' (single quote) will result in abexit 180.

Because this program omits the 'E' in exponential-format output, the code pairs e,E and g,G are the same. Because a switch to exponential format always occurs when an 'f' format field overflows, 'g' is the same as 'f' except for also invoking underflow control. Conversion codes D,O,U,X have different definitions in different versions of the traditional library and are best not used. For convenience, 'D' and 'U' are implemented here as synonyms for 'd' and 'u', and 'x' and 'X' as hexadecimal conversions with lower- or uppercase 'a' through 'f', respectively, but 'O' is an independent code to skip the '\0' at the end of string output. Codes a,A,C,S will result in abexit 180 ('C' is reserved for future implementation of complex number output).

Other items that exist in some implementations of **[fs]printf** that are not implemented here include: Wide characters, locale control of decimal character and thousands separator, separate representations of infinity vs NAN and use of conversion codes 'f' vs 'F' to control case of NAN, 'm$' to indicate that the m'th argument is to be used, return of number of characters that would have been written when output field overflow occurs.

Individual format codes are interpreted as follows. Case is significant except for type codes, where either upper or lower case may be used for types d,e,f,g,i,q,u,x. Codes beginning with '%' in the documentation are not associated with numeric conversions and may appear where needed in the format specifier. Other codes are used with numeric conversions. In square brackets ('[]'), the words 'flag', 'prec', 'size', or 'type' indicate the position in the standard **[fs]printf** format specifier where this code may appear. 'w', 'n', and 'd' indicate numeric parameters which are optional if enclosed in '[]' in the writeup). '^' has two uses: Because '0' is defined as a flag in the standard library, '^' may be used wherever 0 is required as a numeric parameter. When '^' is used as a flag, it must be preceded by a nonzero number to disambiguate this usage. Numbered notes are at the end of the code descriptions:

.A [prec] Where a decimal precision would normally be given, 'A' indicates automatic decimal placement.
nb [type] Writes 'n' blanks to the output. [Note 3]
nB [flag] Binary scaling (with I or U). Must be preceded by 'n', the number of bits to the right of the binary point (0 <= n < 63). 'n' may consist of two integers separated by a '/' or '|'. In the first case, the second scale is used if the RK_BSSLASH bit was set by a previous call to bscompat(). In the second case, the second scale is used if the RK_BSVERTB bit was so set. The 'B' specification applies only to the immediately following format code.
c [type] Argument is an integer printed as a single character.
C [type] Reserved for future use for complex numbers.
[dD] [type] Convert a fixed-point (integer) value in decimal format. The d code may be preceded by a 'w.d' specification. The decimal is normally used only with binary scaling. [Notes 1,2]
[eE] [type] Convert a floating point value in exponential format with 6 decimal places if precision is not specified. There is no 'e' or 'E' in the output, unlike with the standard library. The standard library specifies that the argument is (or is coerced to) a double, but if a pointer is used, 'je' may be use to specify a pointer to a single-precision float. [Note 1]
[fF] [type] Convert a floating point value in decimal format with 6 decimal places if precision is not specified. If the width of the field would be exceeded, E format is used (similar to format 'g' in the standard C library). The standard library specifies that the argument is (or is coerced to) a double, but if a pointer is used ('&' or 'Z' code), a single-precision float is impled--use '[qQ]' for double. [Note 1]
[gG] [type] Same as f in this implementation except underflow conversion to exponential format is also provided. [Note 1]

h,hh [size] Variable is of type short or char, respectively. See explanation above.

[iI] [type] Same as d. [Notes 1,2]

j [size] Variable is a 32-bit fixed point number regardless of the size of an int or a long on the machine. The variable is a single-precision floating-point number if this code is used with conversion type 'e' or 'E' and a pointer to the argument. (In some implementations, 'j' is used for type 'intmax'.)

[n]J [flag] Left-Justify. This is an extension of the standard **[fs]printf** '-' flag. If a value for 'n' is given, 'n' blanks are inserted to the right of the output field--the width of the field is data dependent and the width given in the format is only a maximum width (which includes the 'n' blanks). If a value for 'n' is not given, enough blanks are inserted to fill out the width specified in the format item, if any. Overrides the '0' flag if both are given. [Note 3]

k [type] A wide random number seed (sysdef.h 'wseed' type) will be printed in format '(mm,nn)' where 'mm' and 'nn' are the two 32 bit components of the seed. This code may be preceded by a 'w' width specification. If the output will not fit in the specified space, the field is filled with asterisks.

K [type] Binary option Keys. This format requires two arguments in the argument list. The first is an ASCII string specifying the codes to be printed for each 1 bit in the second argument, which is an unsigned integer or (with '&' code) a pointer to the data item to be converted. Characters other than upper-case letters and digits in the code specification indicate bit positions in the data that are not encoded in the output. This code may be preceded by a 'w' width specification. The output is always left-justified--shorter strings are padded with blanks to 'w' characters unless preceded by 'nJ', in which case the actual string length is used, followed by 'n' blanks. [Notes 1,3]

l,ll [size] Variable is of type long or long long, bzw. See explanation above.

%[n]L Lines. Force a page eject if the next 'n' lines of output would not all fit on the current page (default: 1). [Notes 3,4]

n [type] Writes the number of characters written so far to the next argument, which should be a pointer to the type given by the preceding size code (default int).

%N Not-forced. Sets the RK_NFSUBTTL bit, so the output is interpreted as a subtitle which does not force a new page. [Note 4]

o [type] Convert a fixed-point (integer) value in octal format. The precision modifier is ignored. Binary or decimal scaling is an error. [Note 1]

%O Overwrite. Omit putting '\0' at end of the output string produced by **s[n]printf**. Ignored in other cases.

p [type] Writes the output as a hexadecimal pointer of whatever is the standard size of a pointer in the implementation. A size specification that differs from this is an error.

%[n]P Priority. Set the priority of the printed output to 'n' (1 <= n <= 5; default: 2). [Notes 3,4]

[qQ] [type] "Quality". Same as 'f' except always implies double precision whether or not '&' is given.

n[rR] [flag] Repeat count used as a flag before a format code. The repeat count may be zero to skip the next format code or the '?' flag may be used along with a repeat count to skip output. If the flag is 'r', there is a separate data item in the argument list for each repetition. If the flag is 'R', the data item is presented as a pointer to an array of size 'n'. [Note 3]

s [type] String. The argument is a pointer to a string, which is copied to the output. In accord with the standard specification, if a '.d' "precision" is given, no more than 'd' characters are copied. Otherwise the entire string is copied, except a buffer overflow is an error. If a width is given, it is the number of characters to be transmitted. Shorter strings are padded with blanks to 'w' characters unless the 's' code is preceded by 'nJ', in which case the actual string length is used, followed by 'n' blanks, up to a maximum of 'w' characters. Use the '^' flag to indicate the length of the string for indexing purposes if different. (Wide characters are not supported at this time.)

%S Subtitle. Sets the RK_SUBTTL bit, causing the output to be treated as a new subtitle. (If '%N' and '%S' both given, '%N' prevails.) [Note 4]

%nT Set the record poinTer to column 'n' (counting from 1). The output buffer is not blanked, allowing output from more than one call to be combined. Use 'nb' to blank the buffer if needed. When the line is printed, everything up to the current pointer is included. Some output may be lost if the pointer is backspaced. This feature is useful for making graphs on the printer. [Note 3]

[uU] [type] Convert an unsigned fixed-point value in decimal format. The 'u' code may be preceded by a 'w.d' specification. The decimal is normally used only with binary scaling. [Notes 1,2]

V [flag] If decimal, causes output generated by the following format specification to use E format if all precision otherwise would be lost. If hex, causes output to use the number of characters implied by the item width. 'v' is currently a synonym for 'V', but this usage may change in future versions.

w [size] Variable is a 64-bit fixed point number. (This is safer than 'll', which may indicate 64 or 128 bits depending on the system. At present, output of 128 bit long longs is not supported).

x,X [type] Hexadecimal output. In accord with the standard, type 'x' generates lowercase letters (a-f) for values 10-15, type 'X' generates uppercase (A-F). The precision modifier is ignored. Binary or decimal scaling is an error.

%[n]Y Spout the next 'n' lines of output (n < 256, default: 1). [Notes 3,4]

z [size] Variable is of type size_t. See explanation above.

%nZ Specifies that the next list item is a pointer to an array or structure of siZe 'n' items. The type (or types if a struct) are determined by the size and type codes of the following format specifiers. [Note 3]

0 [flag] The value is padded on the left with zeros rather than blanks. If the '0' and '-' flags both appear, '0' is ignored.

~ [flag] If an output is skipped because of a '?' flag or zero repeat count with the 'R' flag, blanks are written corres-ponding to the width of the items skipped.

%[n]! Error message. Sets iexit to 'n' (1 <= n <= 8, default 1) and priority to 1. [Notes 3,4]

[flag] Has as much as possible the same meaning as the '#' flag in the standard **[fs]printf**. Prefix octal output with '0'. Prefix hexadecimal output with '0x' or '0X' depending on whether the type code was 'x' or 'X', bzw. Always include a decimal point for e,f,g,q even if item is an integer.

%% Print a literal '%'.

n^ [flag] Overrides item size of the current format item for use in array or string indexing. This code is useful for indexing through an array of structures. [Note 3]

& [flag] This code specifies that the next numeric argument is provided as a pointer, not a value. The type is provided by the usual 'size' and 'type' codes.

* Indicates that an integer argument should be picked up and used for the integer value (n,w, or d) expected at that location. If both the integer and '*' are missing in the format, a default value of 1 is used except where documented otherwise (J,P codes).

- [flag] The converted value is left-justified on the field boundary. The converted value is padded on the right with blanks if a minimum width is specified. See 'J' for an extended version of this flag: '-' is equivalent to 'J' with no numeric argument.

+ [flag] A sign (+ or -) is always placed before the result of a signed numeric conversion. By default, a sign is used only for negative numbers. Overrides the ' ' flag if both are given.

' ' (a space) [flag] A blank is always left before a positive number produced by a signed conversion (preventing two adjacent numeric values from appearing concatenated in the output.

%n= Gives the index increment for arrays in <> repeats. Note this counts items, not bytes, unlike the '^' code. [Note 3]

%| Indent. Records the current column location. Following this code, when scanning returns to a left parenthesis or bracket on a new line, a sufficient number of blanks is inserted to cause the output to be indented to the recorded position. [If this code is preceded by a number, it is a binary scale, see 'B'.]

%; Causes interpretation of the format to terminate if there is no pending 'Z' array output or '()' or '<>' repetitions. The semicolon may be used to turn off the conversion of following literals when the last number in a series has been written.

%[n]\n Linefeeds. Write 'n' linefeeds (default 1) to the output. [Note 3]

.<n [prec] Variant of automatic decimal in which the decimal parameter is the lesser of 'n' and the automatic value needed to just fit the value in the given width with one leading blank. [Note 3]

? [flag] Conditional output. An int is read from the argument list and printing of variables controlled by this format code is skipped if the argument is 0.

%/ Flush. Flush the line buffer to the output device. An error occurs if used in **s[n]printf**. [If this code is preceded by a number, it is a binary scale, see 'B'.]

%[n](...%) Format codes enclosed in parentheses are repeated 'n' times. On each repetition, new argument variables are read unless a '%nZ' array length code is in effect. In that case, 'n' may be omitted and the format repeats until the 'Z' count is satisfied. Parentheses can not be nested. [Note 3]

%n<...%> Format codes enclosed in angle brackets are repeated 'n' times. On each repetition, the same argument variables are reused. Each item is assumed to be an array, which is indexed to the next unused element on each iteration of the '<>'. To step through array items referred to by formats with '<>' with stride greater than 1, use the '=' flag. The stride value is multiplied by the size of each item (possibly itself modified with the '^' flag) and further multiplied by the iteration number of the '<>' (minus 1) to obtain the total byte offset for that item, i.e. the stride is counted in numbers of items independent of the size of each item.

Note 1. Fixed-point arguments are assumed to be coerced to ints and floating-point arguments coerced to doubles unless the code '&' has been given or the format specifies an array, in which case a pointer must be passed. Arrays are assumed to be the type specified by the size and type codes, without coercion.

Note 2. Precision with fixed-point arguments in standard **[fs]printf** gives the minimum number of digits. Here, it gives the number of decimals, as with floating-point conversions. Padding with zeros can be accomplished with the '0' modifier and a specified width.

Note 3. 'n' or 'w' may be replaced by '*' to read an integer value from the argument list for this quantity. Use '^' to code a 0 numeric parameter.
Note 4. This code is ignored unless output is written with **cryout** (printf call).

Error actions: Abexit codes in the range 180-199 are allocated to this program. Because these errors can only be coding errors, no message text is provided. For explanation, type 'abend nnn' at the command line, where 'nnn' is the abexit error number.

Return value: Number of characters written, not including the trailing '\0'.

Subroutines **SVVADJ** and **CKVADJ**

A new mechanism referred to as "variable adjustment" has been added to the library. This mechanism is intended to handle situations where the zero point of some variable (membrane potential, temperature, etc.) needs to be added to or subtracted from some input values in order to put them on the correct scale. The application calls **svvadj** to enter an adjustment value and give it a name, "aname"; the user uses the notation "+aname" or "-aname" following a numeric value in the input to indicate that the adjustment should be applied; **kwscan**, **eqscan**, and **inform** call **ckvadj** when this construct is found, and **ckvadj** sets indicators that cause **bcdin** or **wbcdin** to apply the adjustment as the variable is converted to binary.

Usage: void **svvadj**(double value, char *aname)

Prototyped in: rkxtra.h

Arguments: 'value' is a scale adjustment value to be stored for future use. 'aname' is a name to be assigned to this value (max 7 characters). There is no provision to remove a stored adjustment value, but the stored value can be set to 0 so that future adjustments have no effect.

Usage: void **ckvadj**(void)

Prototyped in: rkxtra.h

Action: This routine checks whether the last scanned field was terminated with a plus or minus sign. If so, it scans the next field and compares it with the table of suffix names previously stored by **svvadj**. If a match is found, the named value is stored for one of the bcdin routines to apply. If a match is not found, an ermark RK_VANSERR error is generated. If there was no '+' or '-' delimiter, no adjustment is made.

Note: This routine is normally called internalled from **kwscan**, **eqscan**, or **inform** and should not need to be called directly from application programs.

Function **XXPOLY**

This function is used to read coefficients of polynomials supplied by users in literal form. Polynomials may be entered as a sum of terms preceded by plus or minus signs. Each term may consist of (1) a constant or (2) a product represented by a constant and the name of a variable connected in either order by an asterisk representing multiplication. The names are matched by **xxpoly** to a list of valid names supplied by the caller and the position of each match determines where the multiplying factor is stored in the output coefficient array. A comma or end-of-card terminates the scan and **xxpoly** returns to the caller. Function **xxpoly** assumes that **cryin** and **cdscan** were already called to read the card and to indicate the starting column for scanning. Function **xxpoly** itself calls **scan** with the RK_PMDELIM flag set to cause '+' and '-' to be treated as delimiters.

When an error occurs, a call is made to **ermark** to document the error, RK.iexit is set nonzero, and the program returns an error code identical to the **ermark** argument.

Usage: int **xxpoly**(float *coeffs, char **names, int len)

Prototyped in: rkxtra.h

Arguments: 'coeffs' is a real array of length 'len' to receive the result.

'names' is an array of 'len' strings giving names for the terms. When a name is matched, the numeric coefficient is stored in the corresponding location in the 'coeffs' array. The first name is always ignored--the constant term goes in the first position of the 'coeffs' array.

'len' is the number of coefficients and names.

Value returned: 0 indicates successful completion; nonzero values indicate errors and have the same meanings as the corresponding **ermark** codes.

Verification procedure: If the output priority is 5, **xxpoly** prints a table of the named coefficients and their assigned values so that the user may verify them.

Implementation: This routine has not yet been implemented in the C version of the ROCKS library, but the prototype and description are kept for use when needed.

SYSTEM INTERFACE ROUTINES

Subroutines **ABEXIT** and **ABEXITQ**

Subroutine **abexit** provides a standard method to terminate a program abnormally. It will, as a minimum, print an error message giving the termination code and return the termination code to the operating system. Applications may provide their own version of **abexit** when it is necessary to avoid using **cryout** or to perform required cleanup, complete plots, etc. Such routines must not call any ROCKS routines, such as **mallocv**, which themselves call **abexit[m][e]**.

Subroutine **abexitq** is the same except it returns to the caller if **e64qtest** indicates this is a test run. The reason for separating this function from traditional **abexit** is that **abexit** is now declared to be non-returning and that attribute is useful for optimization of codes that can never be tests. Tests that need to continue after an **abexit** call should use **abexitq** instead.

Usage: void **abexit**(int code)
Void **abexitq**(int code)

Prototyped in: sysdef.h

Argument: 'code' is an abnormal termination code. Under UNIX, codes larger than 255 cannot be recognized by the operating system. A value of 100 is returned for these codes after the original value has been printed. See the file "errtab" for a list of currently defined codes. All codes in the laboratory should be unique.

Subroutines **ABEXITM** and **ABEXITMQ**

Subroutine **abexitm** provides a second method to terminate a program abnormally. It prints a blank line and an error message supplied in the call, then calls **abexit**. Applications may provide their own version of **abexitm** when it is necessary to avoid using **cryout** or to perform required cleanup, complete plots, etc. Such routines must not call any ROCKS routines, such as **mallocv**, which themselves call **abexit[m][e]**.

Subroutine **abexitmq** is the same except it returns to the caller if **e64qtest** indicates this is a test run. The reason for separating this function from traditional **abexit** is that **abexit** is now declared to be non-returning and that attribute is useful for optimization of codes that can never be tests. Tests that need to continue after an **abexitm** call should use **abexitmq** instead.

Usage: void **abexitm**(int code, char *emsg)
Void **abexitmq**(int code, char *emsg)

Prototyped in: sysdef.h

Arguments: 'code' is an abnormal termination code. See CCRULES. CRK for a list of currently defined codes. All codes in the laboratory should be unique.

'emsg' is the text to be printed. The supplied version of **abexitm** prepends "***" to this message. The total length of the message may not exceed 128 characters. **ssprintf** may be used to generate the message in situations, such as parallel computers, where the full output formatting library is not available.

Subroutine **ABEXITME**

Subroutine **abexitme** provides a third method to terminate a program abnormally. It prints a blank line and an error message supplied in the call, then a third line giving the value of the global system variable **errno**, if defined under the operating system in use, then calls **abexit**. Applications may provide their own version of **abexitme** when it is necessary to avoid using **cryout** or to perform required cleanup, complete plots, etc. There are no ROCKS routines which themselves call **abexitme**.

Usage: void **abexitme**(int code, char *emsg)

Prototyped in: sysdef.h

Arguments: Same as for **abexitm**.

Subroutine **CRKVERS**

This routine may be called at any time by an application that is using the ROCKS library. It returns a pointer to a static string containing the current subversion revision number of the package in the form "ROCKS Library Rev nnn". This is usefult for documenting what software was used in a particular computation.

Usage: char ***crkvers**(void)

Prototyped in: rocks.h

Subroutine **RKSLEEP**

This subroutine causes the executing program to go into a sleep state for a specified time interval. It is similar to the standard UNIX routine **sleep**(), but provides greater resolution. The implementation under various operating systems may or may not provide the accuracy or resolution implied by the arguments.

Usage: void **rksleep**(int sec, int usec)

Prototyped in: rocks.h, rksubs.h

Arguments: The delay time is the sum of 'sec', a whole number of seconds, and 'usec', a whole number of microseconds.

Function **SECOND**

This function may be used to start a clock and then to determine the elapsed process time since the clock was started.

Usage: double **second**(void)

Prototyped in: rocks.h, rksubs.h

Value returned: Elapsed CPU process time in seconds. The first call initializes the clock to 0.0 and returns 0.0.

Function **GETMYIP**

This function returns an IP address at which the currently running system can be reached. (It does this by looking at the result returned by “ifconfig eth0", and if there is none, then “ifconfig ppp0". It is at present only provided on Linux systems.

Usage: char ***getmyip**(void)

Prototyped in: rksubs.h

Value returned: Pointer to a static character string containing the (or one of if there are more than one) IP address of the running system in the form xxx.xxx.xxx.xxx.

Functions **INVOKE** and **UPROG**

These functions for use under the IBM MVS and VM/CMS operating systems allow a program dynamically to load, then repeatedly execute a routine. Under MVS, the routine to be loaded must be link-edited into a load module and placed in the step library or a data set concatenated to the step library. Under VM/CMS, a compiled TEXT file can be dynamically invoked without loader or link-editor processing.

Under UNIX-like operating systems (including Linux), the application should call the standard library function **dlopen** to open a specified library containing user-written routines, **dlerror** to check for errors, then **dlsym** to locate a pointer the desired routine. That pointer is then used to call the user routine.

It is assumed that the routine conforms to the calling program in regard to number and type of arguments. The routine should not perform input/output on any unit used by the calling application.

Before the routine can be used, it must be loaded:

Usage: int **invoke**(char *routinename)

Prototyped in: rocks.h

Argument: 'routinename' is the name of the routine to be loaded, and must obey they conventions of MVS and CMS, where 'routinename' is an 8-character name assigned by a Linkage-Editor NAME statement.

Value returned: 0 if the routine has been successfully loaded; otherwise an implementation-dependent nonzero code.

Each time the routine is to be executed, it is called by:

Usage: int **uprog**(arguments,...)

Prototyped in: rocks.h

Arguments: 'arguments' are any arguments required by the particular situation. The user must be told in the application writeup what arguments will be passed to his **invoke** routine. Use of the name **uprog** gets around the fact that the name of the routine to be called is not known at compilation time.

Value returned: Whatever value is returned by the user-written routine is returned to the caller as the **uprog** function value. This value must be an integer.

When the routine is no longer required, it can be deleted from memory by:

Usage: int **invoke**(NULL)

Argument: 'routinename' is replaced by a null pointer.

Value returned: 0 if unloading was successful; otherwise an implementation-dependent nonzero error code.

A call to **uprog** after this call results in an immediate return. Only one **invoke** routine can be in memory at one time--each **invoke** call deletes the previous routine, if any.

Subroutine **NDRFLW**

This routine provides a way for an application to mask floating point underflow error traps on or off if permitted by the hardware. It has no effect on fixed-point operations. On systems that do not make this operation available, **ndrflw** is treated as a "no-operation". Note: On systems that do not permit user control of the action to be taken on underflow conditions, it is not predictable what action the system will take if an underflow occurs. In general, application code should be written to avoid underflows wherever possible.

Usage: void **ndrflw**(char *string)

Prototyped in: rocks.h

Argument: 'string' is the literal string "ON" to turn on trapping of underflows. If the C library provides a standard method of handling underflow exceptions, it will be used; otherwise, if possible, a routine will be provided that prints a message for each underflow (up to 100) and then sets the result to zero; otherwise, a routine will be provided that terminates the application when an underflow occurs.

'string' is the literal string "OFF" to turn off the trapping of underflows. The result of each operation that results in an underflow will be set to zero and no message will be produced. This method produces faster execution when underflows occur because no interrupt processing is necessary.

Error procedure: If 'string' is anything other than "ON" or "OFF", execution is terminated with abexit code 10.

Implementation: This routine has not yet been implemented in the C version of the ROCKS library, but the prototype and description are kept for use when needed.

Subroutines **PXON, PXQ**, and **PXOFF**

These routines provide an interface by which a user running an application interactively can interrupt it at controlled points to enter control cards or other information. The operation of this interface is operating-system dependent. Under CMS, the user enters "PX string" (PX = "pause execution") and the entire string is passed to the application when it is next able to process it. Under MS-DOS, the user must press the Ctrl-Break key to get the program's attention. A '>' prompt is given, to which the user may enter any command acceptable to the program. For compatibility with CMS, the command should begin with "PX".

To establish the PX interface:

Usage: int **pxon**(void)

Prototyped in: rocks.h

Value returned: 0 if the interactive interface was successfully established, otherwise, -1. (Some implementations may terminate with abexit code 9 if execution is unsuccessful.)

After this call is completed, the user may at any time enter an immediate command as prescribed above. The string 'string' will be read and passed to the user program at the next **pxq** call.

Usage: int **pxq**(char *string, int maxstr)

Prototyped in: rocks.h

Arguments: 'string' is a pointer to a string which will receive the command string if one has been entered. If no command has been entered, the first character of the string will be set to the standard C end-of-string character. The string should be long enough to contain the longest command line that the operating system permits to be entered (CDSIZE in sysdef.h).

'maxstr' gives the number of characters (not including the end-of-string marker) that will fit in 'string'. Function **pxq** will truncate the input command, if necessary, to fit in 'string'.

Value returned: If **pxon** was not called or failed to install the interface, or if no interactive command has been issued, -1 is returned. Otherwise, **pxq** returns the number of characters in the complete command string, including the "PX" but not the end-of-string character. If this value is larger than 'maxstr', truncation has occurred. The pending command state is cleared so that one command from the terminal can only give one positive **pxq** return.

To cancel an existing pause command interface:

Usage: void **pxoff**(void)

Prototyped in: rocks.h

After this call is completed, interactive user commands will no longer be recognized. The "PX" command is also automatically cancelled when the program terminates. If **pxon** was not called or was called but returned -1, **pxoff** is treated as a no-operation.

TEXT-CACHE ROUTINES

This set of routines may be used to save text strings, such as names and labels, in a cache. These items can then expand to any size (up to CDSIZE if scantxt is used) without having fixed allocations in the application. Further benefits are that duplicate strings are stored only once and, on parallel computers, the text and pointers to it are not copied/translated to comp nodes.

Function **SAVETXT**

This function is used to save a text string in the cache.

Usage: int **savetxt**(char *ptxt)

Prototyped in: rkxtra.h

Argument: 'ptxt' is a pointer to the text string to be stored. The length is not restricted by the text cache package.

Returns: An integer text locator that can be used in a call to **gettxt** to retrieve a pointer to the original text string.

Text locators are positive integers not greater than the total number of strings stored.

Errors: An abexit will be produced by one of the memory allocation routines in the unlikely event that all available heap memory is exhausted.

Function **SCANTXT**

This routine combines the function of a **scanck** call and a **savetxt** call.

Usage: int **scantxt**(int scflags, int badpn)

Prototyped in: rkxtra.h

Arguments: 'scflags' (scan function flags) and 'badpn' (mask indicating bits of RK.scancode that are considered erroneous) are used as arguments in a call to **scanck** to obtain the text string that is to be stored. An internal buffer is allocated for the text, which limits its length to no more than 80 characters (plus the NULL terminator).

Returns: An integer text locator that can be used in a call to **gettxt** to retrieve a pointer to the original text string.

Errors: An **ermark** error is generated by **scanck** if no field is found or if the specified punctuation test on the input field fails. An abexit is produced by one of the memory allocation routines in the unlikely event that all available heap memory is exhausted.

Function **FINDTXT**

This function may be used to determine whether a given text string has already been stored in the text cache. A hash algorithm is used for rapid searching.

Usage: int **findtxt**(char *ptxt)

Prototyped in: rkxtra.h

Argument: 'ptxt' is a pointer to a NULL-terminated text string that is to be found in the text cache.

Returns: 0 if the requested text string is not currently in the text cache, otherwise, the text locator for the string.

Errors: None

Function **GETTXT**

This function may be used to retrieve a pointer to a stored text string, given the text locator number returned by **savetxt**, **scantxt**, or **findtxt**.

Usage: char ***gettxt**(int txtloc)

Prototyped in: rkxtra.h

Argument: 'txtloc' is the text locator value of the string to be located.

Returns: A pointer to the requested text string. The string is NULL-terminated.

Errors: Abexit 93 is generated if the argument is not the locator number of a text string in the cache.

HASH-TABLE MANAGEMENT ROUTINES

This set of routines can be used to create and maintain generic hash tables. The caller provides a hash function suitable to the type of data at hand. Duplicate hash values are handled by building linked lists of items off each table entry. It is assumed that the items being accessed are data structures of some sort, and that each contains a field (or a pointer to a field) where the hash key is stored and also a reserved field of type (void *) that the hash routines can use as a place to store links when identical hash values occur.

There is provision for the routines to enlarge a hash table when it becomes too full. When this happens, each data item is rehashed using the new table length. The user-supplied hash function should return a 32-bit value. The remainder modulo the current table size is used to construct the index into the hash table.

Function **HASHINIT**

This function must be called to initialize a hash table before it can be used. This call provides a pointer to the user-supplied hash function and also gives offsets to where the key and link fields are stored in the user data structures being hashed.

Usage: struct htbl ***hashinit**(unsigned long (*hashfn)(void *), long nht, int lkey, int ohk, int ohl)

Prototyped in: rkhash.h

Arguments: 'hashfn' is a pointer to a user-supplied hash function that takes as argument a pointer to whatever it is that the user is hashing and returns a bit (unsigned long) hash value.

'nht' is the number of entries expected to be in the table. The actual length allocated to the table will be the lowest prime at least large as (2*nht). Each time the table becomes more than 3/4 full, it will be expanded by 50%. This action can be disabled by passing a negative value for nht.

'lkey' is the length of the keys, in bytes. If lkey == 0, the keys are NULL-terminated strings and 'ohk' is the offset to the keys in the user data blocks. If lkey < 0, 'ohk' is the offset of a pointer to a NULL-terminated string.

'ohk' is the offset in the user data structures where hash keys (or pointers to them if lkey < 0) are stored. This argument will be used to construct arguments to **hashfn**. It can be generated by an expression like:
(char *)&((struct mydata *)0)->key - (char *)0.

'ohl' is the offset in the user data structures where hash links can be stored. It should be a multiple of the alignment size for the CPU.

Returns: A pointer to an htbl structure that contains all the information needed to work with this hash table. The value is used as an argument to the remaining routines in this package, allowing the user application to maintain multiple hashes at one time.

Errors: Terminates execution if unable to allocate memory for the hash table.

Subroutine **HASHADD**

This function is used to add an entry to a hash table.

Usage: void **hashadd**(struct htbl *phtb, void *pdata)

Prototyped in: rkhash.h

Arguments: 'phtb' is a pointer to a hash table structure created by an earlier call to **hashinit**.

'pdata' is a pointer to the user data structure to be added to the hash table.

Returns: Nothing.

Errors: Terminates execution if unable to expand hash table when necessary.

Function **HASHLKUP**

This function is used to look up a data item stored in a hash table.

Usage: void ***hashlkup**(struct htbl *phtb, void *pkey)

Arguments: 'phtb' is a pointer to a hash table struct created by an earlier call to **hashinit**.

'pkey' is a pointer to the key identifying the data item to be looked up.

Returns: Pointer to the data item. In case of duplicate data items, the link field in the user data structure can be followed to find additional entries with the same key.

Errors: Returns NULL pointer if item is not in table.

Subroutine **HASHDEL**

This routine is used to remove an item from a hash table.

Usage: void **hashdel**(struct htbl *phtb, void *pdata)

Arguments: 'phtb' is a pointer to a hash table structure created by an earlier call to **hashinit**.

'pdata' is a pointer to the user data item to be deleted from the hash table.

Returns: Nothing. The reference to the data item is removed from the hash table. The data item itself is unaffected.

Errors: Terminates execution if the requested data item is not stored in the hash table.

Subroutine **HASHRLSE**

This routine is used to release storage allocated to a hash table.

Usage: void **hashrlse**(struct htbl *phtb)

Argument: 'phtb' is a pointer to the hash table structure that is to be released. The user data are not affected--the link fields in the user data can be reclaimed for other purposes if desired.

Returns: Nothing.

Errors: Execution terminated by **freev** if 'phtb' is an invalid pointer.

ITERATION LIST MANAGEMENT ROUTINES

An iteration list is a list of positive integers entered by a user. Iteration lists are used to select data items to be operated upon by a program, for example, a list of specific cells to be drawn in a complicated graph. The largest value allowed in an iteration list is 2**30 - 1 (32-bit systems) or 2**62 - 1 (64-bit systems).

Iteration lists may contain single entries, simple ranges (expressed as a starting number followed by a minus sign and an ending number, e.g. '21-47'), and ranges with stride (expressed as a starting number followed by a plus sign and a stride and a minus sign and an ending number, in either order, e.g. '21-47+13'). One can specify the first n or last n of the items in a set, or select n items at random. An iteration list may be set to repeat at some interval, e.g. '5,12 EVERY 20' would iterate over items 5, 12, 25, 32, 45, 52, etc. An existing iteration list may be edited by additions or deletions of numbers, ranges, or parts of ranges. Detailed instructions for constructing iteration lists are contained in the control card rules document.

Routines are provided to read iteration lists from control cards, to search for values in iteration lists, to count items in iteration lists, to use iteration lists to control program loops, and to destroy iteration lists. Multiple loops may be associated with a single iteration list. Header file **rkilst.h** contains type definitions for three kinds of objects used in conjunction with iteration lists: An **ilst** is a data structure where information regarding an iteration list is stored; applications need to store pointers to ilsts, but do not need to be aware of their contents. An **iter** is an iterator that is associated with an **ilst**. There is one **iter** for each loop that is being controlled by that **ilst**.

Finally, an **ilstitem** is an item that may be contained in an iteration list (it is an unsigned long integer). Data of type **ilstitem** are required as arguments to some of the routines described here:

Function **ILSTREAD**

Function **ilstread** generates a new iteration list or edits an existing iteration list based on user input. It is assumed that **cdscan** has been called and scanning is located at the start of the iteration list when **ilstread** is called. Iteration lists containing more than a single item are expected to be enclosed in parentheses--**ilstread** returns when it encounters the closing right paren.

Usage: ilst ***ilstread**(ilst *poldilst, int idop, int base, long seed)

Prototyped in: rkilst.h

Arguments: 'poldlist' is a pointer to an existing iteration list that may be modified by addition or deletion of elements. If the list is deleted (because of the 'OFF' keyword in the user input), then the storage pointed to by 'poldlist' is freed.

'idop' sets the default action in case none of the keywords (OFF, NEW, ADD, DEL) is found initially in the input list. It is one of the values:

```
IL_OFF (=1)  Default is OFF.
IL_NEW (=2)  Default is NEW.
IL_ADD (=3)  Default is ADD.
IL_DEL (=4)  Default is DEL.
```

To this value may be added IL_ELOK (=16) to prevent generating an error if an empty list is found (not the end of scan), and IL_ESOK (=32) to prevent generating an error if an empty list is found and the scan has reached the end of the control card.

'base' is 0 or 1 according to whether the base for counting indexes in an EVERY block is 0 or 1.

'seed' is a random number generating seed that will be used if the user enters the RANDOM keyword without also specifying the SEED keyword. If 'seed' is zero, a systemwide default is used.

Value returned: A pointer to the **ilst** constructed by this call. It may be the same as **poldlist** or may be different. If no list is constructed, a NULL pointer is returned. If an existing list is removed, RK.highrc is set to the larger of 1 or its current value (which will be 4 if there was a syntax error).

Errors: Errors in the user input result in calls to **ermark** or **cryout** to print error messages. RK.iexit will be set nonzero in the event of a terminal error. The contents of an existing list may or may not be changed, depending on the nature of the error, but it will always be left in a consistent state. An abexit is generated with code 81 if the 'idop' argument is not one of the specified values. Abexits with codes 82 and 83 are generated if certain internal logic errors are detected--these are very unlikely; consult GNR if one of them ever occurs.

Function **ILSTCHK**

Because the size of the set from which an iteration list chooses entries may not be known at the time the list is read, this function is provided to complete list processing after the set size is known. It checks that no entries in the list are larger than the set size and completes processing for the LAST and RANDOM user options. It should always be called after **ilstread** and before **ilstset**. It may be called multiple times for the same list without harm.

Usage: int **ilstchk**(ilst *pil, long nmax, char *msg)

Prototyped in: rkilst.h

Arguments: 'pil' is a pointer to an iteration list. It must be a value returned by a previous call to **ilstread**.

'nmax' is the number of items in the set from which the list is selecting. If the set size is not known, a value of IL_MASK (2**30-1) can be entered, which is the largest allowed set size.

'msg' is an identifying text of 32 or fewer characters that is appended to any error messages produced by **ilstchk** to identify the list being processed.

Value returned: 0 if the list passes all tests, otherwise, 1.

Error actions: An error message is printed, the 1 bit in RK.iexit is set, and 1 is returned if any value in the iteration list exceeds the size permitted by the 'nmax' argument. An abexit is generated with error code 84 if 'nmax' > 2**30 - 1.

Subroutine **ILSTREQ**

This routine marks the current scan location and prints an error message indicating that an expected iteration list was not found.

Usage: void **ilstreq**(void)

Prototyped in: rkilst.h

Subroutine **ILSTSALF**

In a parallel computer, the programmer may want **ilstread** to use alternative memory allocation routines that are not part of the ROCKS library in order to place iteration lists in shared memory. Subroutine **ilstsalf** stores pointers to allocation functions that will be used by subsequent **ilstread** and **ilstchk** calls until changed by another call to **ilstsalf**.

Usage:
```
void  ilstsalf(
    void * (*ilallocv)(size_t nitm, size_t size, char *msg),
    void * (*ilallorv)(void *block, size_t size, char *msg),
    void   (*ilfreev) (void *block, char *msg))
```

Prototyped in: rkilst.h

Arguments: 'ilallocv' is a routine to be used in place of **callocv** that has the same arguments as **callocv**.

'ilallorv' is a routine to be used in place of **reallocv** that has the same arguments as **reallocv**.

'ilfreev' is a routine to be used in place of **freev** that has the same arguments as **freev**.

Note: 'ilallocv' is called by **ilstread** both to allocate an ilst block and to allocate the ilstitem entries belonging to the ilst. The two cases can be distinguished if necessary by the 'size' argument, which is sizeof(ilst) or sizeof(ilstitem), respectively.

Function **ILSTHIGH**

This function returns the highest value contained in a given iteration list. If the list was constructed with the 'EVERY' option, then the highest value in the base list, before repetition with 'EVERY', is returned. If the list is empty, -1 is returned. This call may be used to check that a user has not entered values in an iteration list that are larger than the items available to be selected.

Usage: long **ilsthigh**(ilst *pil)

Prototyped in: rkilst.h

Arguments: 'pil' is a pointer to an iteration list. It must be a value returned by a previous call to **ilstread**.

Function **ILSTITCT**

This function counts the number of items in an iteration list that are less than a given value, i.e. the number of times **ilstiter** would need to be called to reach the last value just before the given value. It takes into account possible iterations of the list if the list was constructed with the 'EVERY' option. If the list is empty, 0 is returned.

Usage: long **ilstitct**(ilst *pil, ilstitem item)

Prototyped in: rkilst.h

Arguments: 'pil' is a pointer to an iteration list. It must be a value returned by a previous call to **ilstread**.

'item' is a positive integer value. It does not have to be a member of the list being tested.

Function **ILSTSRCH**

This function performs a binary search to find the offset to the largest value in an iteration list that is less than or equal to a given number. This routine always returns the offset to a single item or to the start of a range, never to an increment or range end item. It returns -1 if the request item is smaller than the first list item or if the list is empty. (**ilstsrch** is mainly for use by other routines in the package, not by applications.)

Usage: int **ilstsrch**(ilst *pil, ilstitem item)

Prototyped in: rkilst.h

Arguments: 'pil' is a pointer to the iteration list to be tested. It must be a value returned by a previous call to **ilstread.**

'item' is the item to be tested for.

Value returned: See description above.

Function **ILSTTEST**

This function determines whether a given value is included in an iteration list, either as an enumerated value or in a range.

Usage: int **ilsttest**(ilst *pil, ilstitem item)

Prototyped in: rkilst.h

Arguments: 'pil' is a pointer to the iteration list to be tested. It must be a value returned by a previous call to **ilstread**. It may be a NULL pointer.

'item' is the item to be tested for.

Value returned: **TRUE** if the item is on the list, otherwise **FALSE**.

Subroutine **ILSTSET**

This routine sets the starting position of an iteration to the smallest value in the iteration list that is greater than or equal to an argument value. Typically, this is used to start an iteration at the first item on the current node in a parallel computer.

Usage: void **ilstset**(iter *pit, ilst *pil, ilstitem item)

Prototyped in: rkilst.h

Arguments: 'pit' is a pointer to the iterator to be initialized. Storage for 'pit' must be allocated by the caller.

'pil' us a pointer to the iteration list that is to be used to control the iterator 'pit'. It must be a value returned by a previous call to **ilstread**.

'item' indicates the lowest value that the caller is prepared to process. Iteration will begin at the smallest value in the iteration list that is greater than or equal to 'item'.

Function **ILSTITER**

This function returns the current item in an iteration and advances the iteration so the next call will return the next item. Results are undefined if **ilstset** has not been called first to initialize the iterator.

Usage: long **ilstiter**(iter *pit)

Prototyped in: rkilst.h

Argument: 'pit' is a pointer to an iterator that has been initialized by a call to **ilstset**.

Value returned: The current value of the iteration. If this is the first call to **ilstiter** following a call to **ilstset**, the value returned is the first value selected from the iteration list. -1 is returned if the list is empty or all the items have been processed.

Function **ILSTNOW**

This functions returns the current item in an iteration but does not advance the current position in the list. Results are undefined if **ilstset** has not been called.

Usage: long **ilstnow**(iter *pit)

Prototyped in: rkilst.h

Argument: 'pit' is a pointer to an iterator that has been initialized by a call to **ilstset**.

Value returned: The current value of the iteration. If this is the first call to **ilstnow** following a call to **ilstset**, the value returned is the first value selected from the iteration list. -1 is returned if the list is empty or all the items have been processed.

Subroutine **FREEILST**

This routine frees the memory occupied by an existing iteration list.

Usage: void **freeilst**(ilst *pil)

Prototyped in: rkilst.h

Argument: 'pil' is a pointer to an iteration list or a NULL pointer. It must be a value returned by a previous call to **ilstread**.

GEOMETRICAL ITERATION ROUTINES

The geometrical iteration routines generate the x,y (2-D) or x,y,z (3-D) indices of points on a lattice contained within a geometrical figure. At present, 2-D circles, circular shells, rectangles (2 versions), and polygonal figures (3 versions) as well as 3-D cylinders, spheres, and spherical shells are supported. In each case, the caller provides a structured work area that is defined in a header file, initialized with the parameters of the particular lattice and geometrical figure to be computed, and accessed during the iterations. The actual iteration routine is then called at the top of a 'while' loop, returning TRUE when points remain in the figure and FALSE when all such points have been returned. The actual index values of the points are returned in the work structure. The user may carry out multiple iterations in parallel by providing separate work areas for each.

TWO-DIMENSIONAL ITERATORS

Subroutine **InitCircleIndexing**

This module returns all the lattice points inside a given circle.

Usage: void **InitCircleIndexing**(IterCirc *ICirc, double xc, double yc, double radius, long nx, long ny)

Prototyped in: itercirc.h

Arguments: 'ICirc' is a pointer to an IterCirc structure (prototyped in itercirc.h) that will be initialized by this program and used to hold the state of the circle indexing routine.

'xc' and 'yc' give the center of the circle in grid units along x and y.

'radius' is the radius of the circle in grid units.

'nx', 'ny' are the numbers of lattice units along x and y.

Returns: ICirc is initialized for subsequent calls to **GetNextPointInCircle**.

Notes:
Arguments 'nx' and 'ny' define the size of the rectangular lattice on which the circle is placed. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the circle, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the circle to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:

The axes must have equal spacings, the lattice must have right angles between the axes, and the lower bound of each axis is always taken to be zero. The ordering of indexes for computing lattice offsets is y slow moving, then x.

Function **GetNextPointInCircle**

Usage: int **GetNextPointInCircle**(IterCirc *ICirc)

Prototyped in: itercirc.h

Argument:
'ICirc' is a pointer to an IterCirc structure initialized by a previous call to **InitCircleIndexing**.

Return values:
The function value is TRUE if a grid point was found inside the given circle. Values in ICirc->ix, iy, and ioff are valid. The function value is FALSE if all grid points in the circle have already been returned. Values in ICirc are no longer valid in this case.
ICirc->ix and iy are the index values of the next lattice point in the circle along the x and y directions, counting from 0.
ICirc->ioff is the offset of the point ix, iy from the origin of the lattice.

Subroutine **InitShell2Indexing**

This module contains routines that find successively all the points on some rectangular lattice that fall in a shell between two given concentric circles. The center of the circles need not fall on a grid point. Points exactly on the outer radius are included. Points exactly on the inner radius are excluded. This prevents double counting when a circle is extended in shells.

Usage: void **InitShell2Indexing**(IterShl2 *IShl2, double xc, double yc, double r1, double r2, long nx, long ny)

Prototyped in: itershl2.h

Arguments: 'IShl2' is a pointer to an IterShl2 structure (prototyped in itershl2.h) that will be initialized by this program and used to hold the state of the shell indexing routine.

'xc' and 'yc' are the coordinates of center of the shell in grid units along x and y.

'r1' and 'r2' are respectively the radii of the inner and outer circles defining the desired shell. ('r1' may be 0.0 to initiate a series of expanding shells.)

'nx' and 'ny' are the numbers of lattice units along x and y.

Returns: IterShl2 is initialized for subsequent calls to **GetNextPointInShell2**.

Notes:
Arguments nx,ny define the size of the rectangular lattice. These values are used for two purposes: (1) to calculate the offset within the lattice to a point in the circle, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the shell to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
Currently the lattice must have right angles between the axes and the lower bound of each axis is always taken to be zero. The lattice must be Euclidean (x and y intervals equal). The itershl routines can be used if the lattice axes are unequal (set z size to 1 grid unit). The ordering of indexes for computing lattice offsets is y slow moving, then x.

Function **GetNextPointInShell2**

Usage: int **GetNextPointInShell2**(IterShl2 *IShl2)

Prototyped in: itershl2.h

Arguments: 'IShl2' is a pointer to an IterShl2 structure initialized by a previous call to InitShell2Indexing.

Return values:
The return function value is TRUE if a grid point was found inside the given shell. Values in Ishl2->ix,iy,ioff are valid. The return value is FALSE if all grid points in the shell have already been returned. Values in Ishl2 are no longer valid.

IShl2->ix,iy return the index values of the next lattice point in integer grid units along the x and y directions.

IShl2->ioff returns the offset (one-dimensional index) of the point (ix,iy) from the origin of the lattice (x fast moving).

Subroutines **InitRectangleIndexing** and **InitOIRectIndexing**

The rectangle iterator routines provide the coordinates of points in a rectangle generated in the form of a spiral from the center outwards to the edges (**InitRectangleIndexing** and **GetNext-PointInRectangle**) or a spiral from the upper left-hand corner inwards to the center (**InitOIRectIndexing** and **GetNextPoiintIn-OIRect**). This may be useful, for example, in searching around a point in an image for a certain desired feature. To perform a TV-type scan over a rectangle, use the polygon interator.

Usage: void **InitRectangleIndexing**(IterRect *Rect, long nsx, long nsy, long ix0, long iy0, long nrx, long nry)
void **InitOIRectIndexing**(IterRoi *Rect, long nsx, long nsy, long ix0, long iy0, long nrx, long nry)

Prototyped in: iterrect.h (**InitRectangleIndexing**) or iterroi.h (**InitOIRectIndexing**)

Arguments: 'Rect' is a pointer to an IterRect (**InitRectangleIn-dexing**) or IterRoi (**InitOIRectIndexing**) structure that will be initialized by this program and used to hold the state of the rectangle indexing routine.

'nsx' and 'nsy' give the size of the full rectangular lattice into which a smaller rectangle is to be embedded.

'ix0' and 'iy0' give the coordinates of the ULHC (upper-left-hand-corner) of the desired rectangle, counting from 0 in each direction.

'nrx', 'nry' give the size of the desired rectangle in lattice units along x and y.

Returns:
'Rect' is initialized for subsequent calls to **GetNextPointInRectangle** or **GetNextPointInOIRect**, respectively.

Notes:
Arguments 'nsx' and 'nsy' define the size of the rectangular lattice on which the smaller (or same size) rectangle is placed. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the small rectangle, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the specified rectangle to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:

The axes must have equal spacings, the lattice must have right angles between the axes, and the lower bound of each axis is always taken to be zero.

Functions **GetNextPointInRectangle** and **GetNextPointInOIRect**

Usage: int **GetNextPointInRectangle**(IterRect *Rect)
Int **GetNextPointInOIRect**(IterRoi *Rect)

Prototyped in: iterrect.h or iterroi.h, respectively.

Argument:
'Rect' is a pointer to an IterRect, respectively IterRoi structure initialized by a previous call to **InitRectangleIndexing**, respectively **InitOIRectIndexing**.

Return values:
The function value is TRUE if a grid point was found inside the specified rectangle. Values in Rect->ix, iy, and ioff are valid. The function value is FALSE if all grid points in the rectangle have already been returned. Values in Rect are no longer valid in this case.
Rect->ix and iy are the index values of the next lattice point in the rectangle along the x and y directions, counting from 0.
Rect->ioff is the offset of the point ix, iy from the origin of the lattice.

Subroutine **InitPolygonIndexing**

This module locates all points on some rectangular lattice within a simply-connected convex polygon. For figures that do not meet these conditions, use **InitPgfgIndexing** and **GetNextPointInPgfg** or **InitEpgfIndexing and GetNextPointInEpgf** (which may be slightly slower).

Usage: void **InitPolygonIndexing**(IterPoly *Poly, xyf *pgon, IPolyWk *work, long nsx, long nsy, int nvtx)

Prototyped in: iterpoly.h

Arguments: 'Poly' is a pointer to an IterPoly structure (prototyped in iterpoly.h) that will be initialized by this program and used to hold the state of the polygon indexing routine.

'pgon' is an array of pairs of x,y coordinates defining in order the edges of the polygon. 'xyf' is a typedef type declared in sysdef.h

'work' is a pointer to a work area large enough to hold 'nvtx' IPolyWk structures. This structure is declared in iterpoly.h

'nsx' and 'nsy' give the size of the full rectangular lattice into which a smaller polygon is to be embedded.

'nvtx' is the number of vertices (or edges) of the given polygon.

Returns:
'Poly' is initialized for subsequent calls to **GetNextPointInPolygon**.

Notes:
Arguments 'nsx' and 'nsy' define the size of the rectangular lattice on which the polygon is placed. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the polygon, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the specified polygon to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
The axes must have equal spacings, the lattice must have right angles between the axes, and the lower bound of each axis is always taken to be zero.

Function **GetNextPointInPolygon**

Usage: int **GetNextPointInPolygon**(IterRect *Poly)

Prototyped in: iterpoly.h

Argument:
'Poly' is a pointer to an IterPoly structure initialized by a previous call to **InitPolygonIndexing**.

Return values:
The function value is TRUE if a grid point was found inside the specified polygon. Values in Poly->ix, iy, and ioff are valid. The function value is FALSE if all grid points in the polygon have already been returned. Values in Poly are no longer valid in this case.

Poly->ix and iy are the index values of the next lattice point in the polygon along the x and y directions, counting from 0.
Poly->ioff is the offset of the point ix, iy from the origin of the lattice (x fast moving).

Subroutine **InitPgfgIndexing**

The pgfg module ('pgfg' stands for “polygon figure”) locates all points on some rectangular lattice within, or alternatively outside, a figure constructed from one or more polygons. The polygons may be concave or self-intersecting. Inside/outside decisions are made by a principle of parity switching upon line crossing, i.e. a point at a great distance from the figure is assumed to be outside it, and following along any scan line through the figure, the inside/outside state is reversed each time an edge of one of the polygons is crossed.

Usage: void **InitPgfgIndexing**(IterPgfg *Pgfg, xyf *pgon, IPgfgWk *work, si32 nsx, si32 nsy, int nvtx)

Prototyped in: iterpgfg.h

Arguments: 'Pgfg' is a pointer to an IterPgfg structure (prototyped in iterpgfg.h) that will be initialized by this program and used to hold the state of the polygon figure indexing routine.

'pgon' is an array of pairs of x,y coordinates defining in order the edges of the polygons. Coordinates must lie in the ranges -nsx <= x < 2*nsx, -nsy <= y < 2*nsy. To indicate that a point is the last vertex in its polygon, store PgfgPly(y,nsy) in place of the y coordinate. Subsequent vertices may define additional polygons or holes in previous polygons. (PgfgPly() is a macro defined in iterpgfg.h. 'xyf' is a typedef type declared in sysdef.h.)

'work' is a pointer to a work area large enough to hold 'nvtx' IPgfgWk structures. This structure is declared in iterpgfg.h

'nsx' and 'nsy' give the size of the full rectangular lattice into which a smaller polygonal figure is to be embedded. Note that these coordinates are of type si32 rather than long as used with the other geometric iterators, which implies the maximum lattice size is the same in 32-bit and 64-bit implementations. In fact, the lattice edges should be less than ~2^24 to keep round-off errors insignificant.

'nvtx' is the number of vertices in the 'pgon' array. A negative value indicates that points outside the figure defined by the polygons should be returned.

Returns:
'Pgfg' is initialized for subsequent calls to **GetNextPointInPgfg**.

Notes:
Arguments 'nsx' and 'nsy' define the size of the rectangular lattice on which the polygon is placed. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the polygon, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the specified polygon to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
The axes must have equal spacings, the lattice must have right angles between the axes, and the lower bound of each axis is always taken to be zero.

Function **GetNextPointInPgfg**

Usage: int **GetNextPointInPgfg**(IterPgfg *Pgfg)

Prototyped in: iterpgfg.h

Argument:
'Pgfg' is a pointer to an IterPgfg structure initialized by a previous call to **InitPgfgIndexing**.

Return values:
The function value is TRUE if a grid point was found inside (outside if 'nvtx' < 0) the specified polygonal figure. Values in Pgfg->ix, iy, and ioff are valid. The function value is FALSE if all grid points in the figure have already been returned. Values in Pgfg are no longer valid in this case.
Pgfg->ix and iy are the index values of the next lattice point in the figure along the x and y directions, counting from 0.
Pgfg->ioff is the offset of the point ix, iy from the origin of the lattice (x fast moving).

Subroutine **InitEpgfIndexing**

The Epgf module ('Epgf' stands for "extended polygon figure") locates all points on some rectangular lattice within, or alternatively outside, a figure constructed from one or more polygons, either extended or condensed by inclusion or exclusion of a border of specified width around the boundary polygon(s). The polygons may be concave or self-intersecting. Inside/outside decisions are made by a principle of parity switching upon line crossing, i.e. a point at a great distance from the figure is assumed to be outside it, and following along any scan line through the figure, the inside/outside state is reversed each time an edge of one of the polygons is crossed.

Usage: void **InitEpgfIndexing**(IterEpgf *pit, xyf *pgon, IEpgfWk *work, byte *pwim, float border, si32 nsx, si32 nsy, int nvtx, int mode)

Prototyped in: iterepgf.h

Arguments: 'pit' is a pointer to an IterEpgf structure (prototyped in iterepgf.h) that will be initialized by this program and used to hold the state of the extended polygon figure indexing routine.

'pgon' is an array of pairs of x,y coordinates defining in order the edges of the polygons. Coordinates must lie in the ranges -nsx <= x < 2*nsx, -nsy <= y < 2*nsy. To indicate that a point is the last vertex in its polygon, store EpgfPly(y,nsy) in place of the y coordinate. Subsequent vertices may define additional polygons or holes in previous polygons. (EpgfPly() is a macro defined in iterepgf.h. 'xyf' is a typedef type declared in sysdef.h.)

'work' is a pointer to a work area large enough to hold 'nvtx' IEPgfWk structures. This structure is declared in iterepgf.h

'pwim' is a pointer to a work area large enough to hold 'nsy' * (bytes to hold 'nsx' bits).

'border' is the Width of the border to be drawn on either side of the defining polygon (pixel units). ('border' may be 0 to allow this routine to behave like the iterpgfg routines, see description of drawing modes below.)

'nsx' and 'nsy' give the size of the full rectangular lattice into which a smaller polygonal figure is to be embedded. Note that these coordinates are of type si32 rather than long as used with the other geometric iterators, which implies the maximum lattice size is the same in 32-bit and 64-bit implementations. In fact, the lattice edges should be less than ~2^24 to keep round-off errors insignificant.

'nvtx' is the number of vertices in the 'pgon' array (which may include more than one polygon).

'mode' is the sum of bits from the following list describing the operations to be performed (bits left of the low-order three bits of 'mode' are ignored):

```
EPGF_INT  (=1)    Return points inside the pgon figure.
EPGF_EXT  (=2)    Return points outside the pgon figure.
EPGF_ADDB (=0)    Also return points within 'border'
                  pixels of the defining polygon
                  (this is the default).
EPGF_RMB  (=4)    Erase (do not return) points within
                  'border' pixels of the defining polygon.
```

These mode bits allow the following combinations: 0: draw a wide line around the borders of the defining polygons; 1: draw a filled figure with an extra border around it (if 'border' is 0, this is the same as a call to **InitPgfgIndexing** with positive 'nvtx'); 2: fill in the background outside the figure with an extra border within the polygons (if 'border' is 0, this is the same as a call to **InitPgfgIndexing** with negative 'nvtx'); 3: the entire lattice is filled; 4: the entire lattice is empty, i.e. **GetNextPointInEpgf** immediately returns FALSE; 5: a filled figure is returned except for points within a width 'border' of the defining boundary polygons; 6: the background outside the figure is filled except for a width 'border' outside the defining boundary polygons; 7: the entire lattice is filled except for a ribbon of width 2*'border' around the defining polygons.

Returns:

'pit' is initialized for subsequent calls to **GetNextPointInEpgf.**

Notes:

Arguments 'nsx' and 'nsy' define the size of the rectangular lattice on which the polygon is placed. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the polygon, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the specified polygon to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:

The axes must have equal spacings, the lattice must have right angles between the axes, and the lower bound of each axis is always taken to be zero.

Function **GetNextPointInEpgf**

Usage: int **GetNextPointInEpgf**(IterEpgf *pit)

Prototyped in: iterepgf.h

Argument:
'pit' is a pointer to an IterEpgf structure initialized by a previous call to **InitEpgfIndexing**.

Return values:
The function value is TRUE if a grid point was found in the specified figure. Values in pit->ix, iy, and ioff are valid. The function value is FALSE if all grid points in the figure have already been returned. Values in 'pit' are no longer valid in this case.
pit->ix and iy are the index values of the next lattice point in the figure along the x and y directions, counting from 0.
pit->ioff is the offset of the point ix, iy from the origin of the lattice (x fast moving).

Subroutine **InitTapeIndexing**

The tape iterator routines may be used to scan along a rectangular "tape" that falls at some angle on a rectangular lattice. The tape may be divided into smaller rectangular boxes so as to be able to terminate the scan when all objects of interest have been found. The program carefully returns points that lie exactly on the lower and side borders, but on the leading edge, so as to avoid duplicates when the tape is extended.

Usage: void **InitTapeIndexing**(IterTape *T, xyf *pb1, xyf *pb2, float hgt, long nsx, long nsy)

Prototyped in: itertape.h

Arguments: 'T' is a pointer to an IterTape structure that will be initialized to hold the state of the tape indexing routine.

'pb1' and 'pb2' are pointers to pairs of x,y coordinates defining the base of the tape. The direction of tape extension will be along a direction 90 degrees clockwise to a line from b1 to b2.

'hgt' is the height of the first (or only) scan box along the tape.

'nsx' and 'nsy' give the size of the full rectangular lattice into which the tape is to be embedded.

Returns:
'T' is initialized for subsequent calls to **ExtendTapeIndexing** and **GetNextPointOnTape**.

Notes:
Arguments 'nsx' and 'nsy' define the size of the rectangular lattice on which the tape is placed. These values are used for two purposes: (1) to calculate the offsets within the lattice to points on the tape, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the specified tape to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
The axes must have equal spacings, the lattice must have right angles between the axes, and the lower bound of each axis is always taken to be zero.

<u>Function **GetNextPointOnTape**</u>

Usage: int **GetNextPointOnTape**(IterTape *T)

Prototyped in: itertape.h

Argument:
'T' is a pointer to an IterTape structure initialized by a previous call to **InitTapeIndexing**.

Return values:
The function value is TRUE if a grid point was found inside the specified portion of the tape. Values in T->ix, iy, and ioff are valid. The function value is FALSE if all grid points in the tape box have already been returned. Values in T are no longer valid in this case.
T->ix and iy are the index values of the next lattice point in the rectangle along the x and y directions, counting from 0.
T->ioff is the offset of the point ix, iy from the origin of the lattice at the ULHC.

Function **ExtendTapeIndexing**

This function may be called after **GetNextPointOnTape** returns FALSE in order to extend the tape with a further rectangular box build on the leading edge of the previous box. The width and angle of the tape remain unchanged, but the height of the new box may be specified.

Usage: int **ExtendTapeIndexing**(IterTape *T, float hgt)

Prototyped in: itertape.h

Argument: 'T' is a pointer to an IterTape structure initialized by a previous call to **InitTapeIndexing.**

'hgt' is the height of the next box to scan along the tape.

Return values:
The function value is TRUE if a box of the given height built on the end of the previous tape segment can still be fit at least partially within the given lattice. The function value is FALSE if the tape now extends entirely outside the given lattice, i.e. any further calls to **GetNextPointOnTape** are guaranteed to return FALSE.

THREE-DIMENSIONAL ITERATORS

Subroutine **InitCylinderIndexing**

Usage: void **InitCylinderIndexing**(IterCyl *Cyl, double dx, double dy, double dz, double xe1, double ye1, double ze1, double xe2, double ye2, double ze2, double radius, long nx, long ny, long nz)

Prototyped in: itercyl.h

Arguments: 'Cyl' is a pointer to an IterCyl structure (prototyped in itercyl.h) that this program will initialize and the following **GetNextPointInCyclinder** program will use to hold the state of the iteration.

'dx', 'dy', and 'dz' are the sizes of the lattice units along x, y, and z. (The grid axes must be perpendicular, but not necessarily all the same length.)

'xe1', 'ye1', and 'ze1' are the grid coordinates of one end of the cylinder axis.

'xe2', 'ye2', and 'ze2' are the grid coordinates of the other end of the cylinder axis.

'radius' is the radius of the cylinder in the same units as 'dx', 'dy', and 'dz'.

'nx', 'ny', and 'nz' are the number of lattice units along x, y,and z to be included in the model.

Value returned: 'Cyl' is initialized for later calls to **GetNextPointInCylinder**.

Notes:
Arguments nx,ny,nz define the size of the rectangular lattice. These values are used for two purposes: (1) to calculate the offset within the lattice to a point in the cylinder, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the cylinder to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
The lattice must have right angles between axes and the lower bound of each axis is always taken to be zero. The ordering of indexes for computing lattice offsets is z slow moving, then y, then x.

Function **GetNextPointInCylinder**

Usage: int **GetNextPointInCylinder**(IterCyl *Cyl)

Prototyped in: itercyl.h

Arguments: 'Cyl' is a pointer to an IterCyl structure initialized by a previous call to **InitCylinderIndexing**.

Values returned:
The function value is TRUE if a grid point was found inside the given cylinder. Values in Cyl->ix, iy, iz, and ioff are valid. FALSE is returned if all grid points in the cylinder have already been returned. In that case, values in Cyl->ix, iy, iz, and ioff are no longer valid.
Cyl->ix, iy, and iz are set to the grid coordinates of the next lattice point in the cylinder.
Cyl->ioff is set to the offset of point ix,iy,iz from the lattice origin.

Subroutine **InitShellIndexing**

Usage: void **InitShellIndexing**(IterShl, double dx, double dy, double double xc, double yc, double double r1, double long nx, long ny, long nz)

Prototyped in: itershl.h

Arguments: 'Ishl' is a pointer to an IterShl structure (prototyped in itershl.h) that will be used to hold the state of the shell indexing routine.

'dx', 'dy', and 'dz' are the sizes of the lattice units along x, y, and z. (The grid axes must be perpendicular, but not necessarily all the same length.)

'xc', 'yc', and 'zc' are the coordinates of the center of the sphere in grid units along x, y, and z.

'r1' and 'r2' are the inner and outer radii of the spherical shell.

'nx', 'ny', and 'nz' are the numbers of lattice units along x, y, and z.

Value returned: IShl is initialized for subsequent calls to **GetNextPointInShell**.

Notes:
Arguments nx,ny,nz define the size of the rectangular lattice. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the sphere, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the shell to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
The lattice must have right angles between the axes and the lower bound of each axis is always taken to be zero. The ordering of indexes for computing lattice offsets is z slow moving, then y, then x.

Function **GetNextPointInShell**

Usage: int **GetNextPointInShell**(IterShl *Ishl)

Prototyped in: itershl.h

Argument:
'Ishl' is a pointer to an IterShl structure initialized by a previous call to **InitShellIndexing**.

Values returned:
The function value is TRUE if a grid point was found inside the given shell. Values in Ishl->ix, iy, iz, and ioff are valid. The function value is FALSE if all grid points in the shell have already been returned. Values in IShl are no longer valid in this case.
Ishl->ix, iy, and iz are the index values of the next lattice point in the shell along the x, y, and z directions.
Ishl->ioff is the offset of the point ix,iy,iz from the origin of the lattice.

Note:
The routines in this module may be used to find successively all the points on some rectangular lattice that fall in a shell between two given concentric spheres. The center of the spheres need not fall on a grid point. Points exactly on the outer radius are included. Points exactly on the inner radius are excluded. This prevents double counting when a sphere is extended in shells.

<u>Subroutine **InitSphereIndexing**</u>

Usage: void **InitSphereIndexing**(IterSph *ISph, double dx, double dy, double dz, double xc, double yc, double zc, double radius, long nx, long ny, long nz)

Prototyped in: itersph.h

Arguments: 'ISph' is a pointer to an IterSph structure (prototyped in itersph.h) that will be initialized by this program and used to hold the state of the sphere indexing routine.

'dx', 'dy', 'dz' are the sizes of the lattice units along x, y, and z. (The grid axes must be perpendicular, but not necessarily all the same length.)

'xc', 'yc', and 'zc' give the center of the sphere in grid units along x, y, and z.

'radius' is the radius of the sphere in the same units as dx, dy, and dz.

'nx', 'ny', and 'nz' are the numbers of lattice units along x, y, and z.

Returns:
ISph is initialized for subsequent calls to **GetNextPointInSphere**.

Notes:
Arguments 'nx','ny','nz' define the size of the rectangular lattice. These values are used for two purposes: (1) to calculate the offsets within the lattice to points in the sphere, and (2) to limit the points returned to points inside the given bounds. It is valid for part or all of the sphere to lie outside the given lattice bounds--points outside the bounds are never returned.

Restrictions:
The lattice must have right angles between the axes and the lower bound of each axis is always taken to be zero. The ordering of indexes for computing lattice offsets is z slow moving, then y, then x.

Function **GetNextPointInSphere**

Usage: int **GetNextPointInSphere**(IterSph *Isph)

Prototyped in: itersph.h

Argument:
'ISph' is a pointer to an IterSph structure initialized by a previous call to **InitSphereIndexing**.

Return values:
The function value is TRUE if a grid point was found inside the given sphere. Values in Isph->ix, iy, iz, and ioff are valid. The function value is FALSE if all grid points in the sphere have already been returned. Values in Isph are no longer valid in this case.
Isph->ix, iy, and iz are the index values of next lattice point in the sphere along the x, y, and z directions.
ISph->ioff is the offset of the point ix, iy, iz from the origin of the lattice.

SORTING ROUTINES

Function **SORT**

Function **sort** performs rapid radix sorting of structures of any length. Keys may consist of any multiple of 4 bits.

Usage: void ***sort**(void *index, int keyoff, int n, int type)

Prototyped in: rocks.h, rksubs.h

Arguments: 'index' points to a linked list of structures containing the data to be sorted. It is assumed that the first element of each structure is a pointer to the next structure, and that the list is terminated by a NULL pointer. The pointers, and only the pointers, are modified by **sort**.

'keyoff' is the offset in bytes from the beginning of each structure to the beginning of the key. The remainder of each structure in the linked list may contain any data of any length. Normally, the keys are placed right after the linked list pointers and 'keyoff' = sizeof(void *). All sorts are in logical ascending sequence (i.e. keys are treated as unsigned integers). To sort in descending sequence, the keys should be negated before calling **sort**. (This argument was not present in the FORTRAN version.)

'n' is the number of hexadecimal digits in the keys.

'type' indicates whether the keys are numeric (type = 0) or alphabetic (type != 0). On machines in which numbers are stored in inverted byte order, this argument will cause the key bytes to be scanned in reverse during sorting.

Value returned: a pointer to the first structure in the sorted list. This pointer typically must be cast to the appropriate type.

Function **SORT2**

Function **sort2** is a revised and expanded version of **sort**. It performs radix sorting in increasing or decreasing order on data structures organized in a linked list. Keys may consist of character strings, fixed- or floating-point, positive or negative numeric data of any byte length. To reduce the number of passes through the data, 256 sorting bins are used, rather than 16 in **sort**.

Usage: void *sort2(void *pdata, void *work, int okeys, int lkeys, int ktype)

Prototyped in: rocks.h, rksubs.h

Arguments: 'pdata' points to a linked list of structures containing the data to be sorted. It is assumed that the first element of each structure is a pointer to the next structure, and that the list is terminated by a NULL pointer. The pointers, and only the pointers, are modified by sort2.

'work' is a pointer to a work area large enough to contain (2 << BITSPERBYTE) pointers of size PSIZE (defined in sysdef,h). The contents are modified during the operation of this routine.

'okeys' is the offset in bytes from the beginning of each structure to the beginning of the key. The remainder of each structure may contain data of any length. Normally, the keys are placed immediately after the linked-list pointers and okeys = sizeof(void *). However, note that in a 32-bit system, if keys are doubles, padding is inserted after the pointer and okeys will be 8, not 4.

'lkeys' is the length of the keys, in bytes.

'ktype' indicates the type of the keys and is the sum of any relevant bits from the following list (defined in rocks.h and rksubs.h):

KST_CHAR 1 Keys are character strings, to be sorted from left to right regardless of endian order of the machine. KST_APOS and KST_FLOAT are ignored. (Default: Keys are numbers, high-order byte first in big-endian, low-order byte first in little-endian machines.)

KST_APOS 2 All keys are known to be positive numeric values. Sorting is slightly faster than with mixed signs. (Default: Keys may include negative values.)

KST_FLOAT 4 Keys are floating-point numbers, that is, negative values are stored as sign and magnitude. (Default: Negative numbers (KST_APOS bit off) are stored as two's complements.)

KST_DECR 8 Sort in decreasing order of key values. Default: Sort increasing order.)

Value returned: a pointer to the first structure in the sorted list. This pointer typically must be cast to the appropriate type.

Performance: Execution time is proportional to lkeys*N, where N is the number of items to be sorted. There is additional time proportional to lkeys but not N which may be significant when N is very small. To eliminate sign testing in the inner loop, there is separate code for the case that all keys are positive and sort is ascending.

Subroutine **SHSORTUS**

Subroutine **shsortus** performs a "shell sort" (B.W. Kernighan & D.M. Ritchie, "The C Programming Language, Second Edition", Prentice Hall, p. 62) on an array of unsigned short integers. Unlike the case with **sort**, there are no additional data items associated with the keys. Additional routines in this family to handle different data types may be defined as needed; the last two letters in the name of the routine are intended to suggest the argument type.

Usage: void **shsortus**(unsigned short *us, int n)

Prototyped in: rocks.h, rksubs.h

Arguments: 'us' points to an array of 'n' unsigned short integers.

Value returned: The data are sorted in place. Nothing is returned.

OPTIMIZATION ROUTINES

The following functions and routines implement the Nelder-Mead algorithm for function optimization when it is not practical to obtain derivatives. Implementations are provided to optimize single- or double-precision functions with or without simulated annealing. Methods are based on the algorithms in Press (1992) with some enhancements described in the next paragraph.

The optimization process is divided into separate initialization and iteration routines. This allows (actually, requires) the user to initialize the simplex according to geometrical and dimensional constraints appropriate to the problem at hand. It also allows an optimization to be continued with minimal startup cost after an interruption, for example, a temperature change during simulated annealing. The sums of the vertex coordinates are recalculated at nmasri (default 250) iteration intervals to avoid accumulation of round-off errors. When the user function detects a singular point (NM_SING code), if option NMKA_MOVE was set, the point is moved in an expanding see-saw along the line being explored to another point picked at random until a valid point is found. This procedure maintains the rank of the simplex. The best-ever vertex is recorded even if it is not accepted due to added noise. To enhance execution speed, code is repeated where needed to avoid extra 'if' statements. These ideas, the test for convergence on the unthermalized data, and the application of thermal noise as a multiplier rather than an addition in simulated annealing, are all original with this implementation by G.N.R.

All routines with names ending in 'd' relate to the double-precision version; those names ending in 'f' relate to the single-precision version. Both may be used in one application. All double-precision variables used in these routines are typedef'd to nmxyd and all single-precision variables are typedef'd to nmxyf.

This allows the types to be changed easily if necessary. For use outside the rocks library, all convrt() calls can be changed to printf() calls by compiling with -DUSE_PRINTF.

Reference: Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, Paul E. Wright, "Convergence Properties of the Nelder-Mead Simplex Method in Low Dimensions", SIAM Journal of Optimization, 9(1): p.112-147, 1998.

Functions **NMALLOD** and **NMALLOF**

These functions allocate a global workspace for a Nelder-Mead optimization. This allows multiple optimizations to be run in parallel without mutual interference. **nmfreed** or **nmfreef**, respectively, must be called to free the storage when it is no longer needed.

Usage: struct nmglbld ***nmallod**(int N, si32 seed, nmxyd (*ufn)(nmxyd *x, void *usrd))
struct nmglblf ***nmallof**(int N, si32 seed, nmxyf (*ufn)(nmxyf *x, void *usrd)

Prototyped in: **nmallod** and the types 'nmglbld' and 'nmxyd' are declared in nelmeadd.h. **nmallof** and the types 'nmglblf' and 'nmxyf' are declared in nelmeadf.h.

Arguments: 'N' is the number of dimensions to the problem.

'seed' is a random number seed. If zero, a value based on the system clock is chosen.

'ufn' is a pointer to the function, ufn(nmxyd *x, void *usrd), that is to be minimized, where x is a vector of length N defining a point where the function is to be evaluated and usrd is any user data (typically a structure) that is to be passed to ufn each time it is called. ufn should return NM_SING if the result is a singularity.

Returns: These functions return pointers to a nmglbld or nmglblf structure, respectively, that will contain all the data needed to maintain the state of the ongoing optimization. This pointer must be passed to all the other routines in the package. Its contents are of no concern to the caller.

Errors: These functions generate an abexit error if required memory is not available.

Subroutines **NMPARMSD** and **NMPARMSF**

These routines modify the standard Nelder-Mead search parameters as needed for special applications. The nmasrr, nmasxr, nmascr, and nmassr parameters are described in the literature references given.

Usage: void **nmparmsd**(struct nmglbld *nmG, nmxyd nmasrr, nmxyd nmasxr, nmxyd nmascr, nmxyd nmassr, nmxyd nmasar, int nmasri)
void **nmparmsf**(struct nmglblf *nmG, nmxyf nmasrr, nmxyf nmasxr, nmxyf nmascr, nmxyf nmassr, nmxyf nmasar, int nmasri)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: 'nmG' is a pointer to an nmglbld (nmglblf) struct created by a previous call to **nmallod** (**nmallof**).

'nmasrr' is the reflection ratio (default 1.0).

'nmasxr' is the expansion ratio (default 2.0).

'nmascr' is the contraction ratio (default 0.5).

'nmassr' is the shrinkage ratio (default 0.5).

'nmasar' is the avoidance ratio (used to avoid placing a simplex vertex in a singularity. Sets the relative amount of each successive movement. Should be negative and a little less than -1.0, default is -1.083333. This parameter is not used if option NMKA_MOVE is not set.

'nmasri' is the interval at which sums of x vectors are recalculated rather than just corrected (used to prevent roundoff errors from huilding up indefinitely, default 250).

Functions **NMGETXD** and **NMGETXF**

These functions return the location of the storage allocated for the simplex by a previous call to **nmallod** or **nmallof.** This should be viewed as a two-dimensional array of (N+1) vectors each of length N. It is the responsibility of the user to initialize the simplex with an appropriate starting configuration before beginning optimization.

Usage: nmxyd ***nmgetxd**(struct nmglbld *nmG)
nmxyf ***nmgetxf**(struct nmglblf *nmG)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: 'nmG' is a pointer to an nmglbld (nmglblf) struct created by a previous call to **nmallod** (**nmallof**).

Return value: See description above.

Functions **NMINITD** and **NMINITF**

These functions calculate and store the initial function values at the vertices of a double-precision (single-precision) simplex in readiness for a round of optimization by one of **nmitr[ns]a[df]**. Before calling **nminitd** (**nminitf**), the user should call **nmallod** (**nmallof**) to allocate a global minimization work-space, **nmparmsd** (**nmparmsf**) if desired to change any default optimization parameters, then **nmgetxd** (**nmgetxf**) to locate the storage allocated for the simplex. The user must generate an initial simplex of (N+1) vertices in N-dimensional space at that location. Any vertex that is in a singularity may be moved by **nminitd** (**nminitf**) to a safe nearby position if option NMKA_MOVE is set. **nminitd** (**nminitf**) also stores the options argument for use by **nmitr[ns]a[df]**. It should be followed by one or more calls to one of these routines to carry out the actual optimization. In deference to custom, the ufn function value is minimized--ufn should return the complementary value if maximization is desired.

Usage: nmxyd **nminitd**(struct nmglbld *nmG, void *usrd, char *msgid, int options)
nmxyf **nminitf**(struct nmglblf *nmG, void *usrd, char *msgid, int options)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: 'nmG' is a pointer to an nmglbld (nmglblf) struct created by a previous call to **nmallod** (**nmallof**).

'usrd' ('usrf') is a pointer to any arbitrary data that is to be passed to the user-defined function being optimized. May be NULL if no such data are expected.

'msgid' is a pointer to identifying text to be inserted in error messages and reports from **nmitr[ns]a[df]**. Only the pointer, not the text, is copied, so the value should remain valid until **nmfreed** (**nmfreef**) has been called.

'options' is an OR of any of the codes defined in nelmeadd.h (nelmeadf.h) for various reporting and algorithmic options. These codes are as follows:

NMKH_LAST Return nmhist for last iteration.
NMKH_EVRY Return nmhist for every iteration.
NMKP_LAST Print nmhist for last iteration.

NMKP_EVRY   Print nmhist for every iteration.
NMKP_DETAIL Print shrink & avoidance info.
NMKS_THERM Shrink around thermalized (default: unthermalized) best vertex. Ignored **nmitrna[df].**
NMKA_MOVE   Try to move vertex away from singularity when ufn() returns NM_SING.

Returns: The best value, y, of the user-defined function at any vertex of the initial simplex.

Functions **NMITRNAD** and **NMITRNAF**

These functions perform Nelder-Mead simplex optimization without simulated annealing. They minimize the user-provided function ufn beginning with the starting guess x. **nmitrnad** is for double-precision functions, **nmitrnaf** for single-precision. **nmallod** (**nmallof**) must be called first to allocate working memory, then the starting guess must be stored in the array returned by a call to **nmgetxd** (**nmgetxf**), then **nminitd** (**nminitf**) must be called to initialize relevant internal variables.

Usage: int **nmitrnad**(struct nmglbld *nmG, void *usrd, nmxyd **ppxbest, nmxyd *pybest, ui32 *pniter, nmxyd ftol, nmxyd xtol, ui32 mxiter)
int **nmitrnaf**(struct nmglblf *nmG, void *usrd, nmxyf **ppxbest, nmxyf *pybest, ui32 *pniter, nmxyf ftol, nmxyf xtol, ui32 mxiter)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: 'nmG' is a pointer to an nmglbld (nmglblf) struct created by a previous call to **nmallod** (**nmallof).**

'usrd' is a pointer to any arbitrary data that is to be passed to the user-defined function being optimized. May be NULL if no such data are expected.

'ppxbest' is a pointer to a pointer that is filled in on return with the location of an array containing the best solution found in possibly a series of **nmitrnad** (**nmitrnaf**) calls since the last call to **nminitd** (**nminitf**). N.B. Swapping this vector back into the simplex before another call to **nmitrnad** (**nmitrnaf**) can in principle produce a singularity, so don't do it (unless rebuilding an entirely new simplex).

'pybest' is a pointer to a location that is filled in on return with the best value of the function ufn found since the last call to **nminitd** (**nminitf**).

'pniter' is a pointer to a ui32 that will be filled in on return with the number of iterations completed since the last call to **nminitd** (**nminitf**).

'ftol' is the fractional tolerance in the y value. When the magnitude of the difference between the largest and smallest values of y at vertices of the simplex becomes less than ftol times the mean of the magnitudes of those values of y, the optimization terminates. (Caution: This is not necessarily a very good test.)

'xtol' is the absolute tolerance in the radius of the simplex. After a shrink only (to save time), when the difference of x coords at lowest vs highest y is less than xtol, optimization terminates.

'mxiter' is the maximum number of iterations allowed in this call. When this number of iterations is reached, the optimization terminates. (Note: the MATLAB fminsearch() default for this parameter is 200*N.)

Returns: A return code defined in nelmeadd.h (nelmeadf.h) as follows:

- NM_MXITER Completed the specified maximum number of iterations without converging.
- NM_CONVERGED Converged by the ftol test.
- NM_SHRUNK Shrunk to a figure with a radius less than specified by the xtol test.
- NMERR_NOAVOID NMKA_MOVE option was specified and the program failed to avoid a singular vertex.
- NMERR_ALLSING All vertices became singular.

Additionally, the variables pointed to by the ppxbest, pybest, and pniter arguments are filled in as described above. **nmgethd** (**nmgethf**) can be called to locate history records requested by the NMKH_LAST or NMKH_EVRY option codes. **nmgetsd** (**nmgetsf**) can be called to locate counts of the operations performed. Finally, **nmgetxd** (**nmgetxf**) can be called to locate the updated final state of the simplex. This matrix can be used to restart an optimization that was halted prematurely for any reason.

Functions **NMITRSAD** and **NMITRSAF**

These functions are similar to **nmitrnad** and **nmitrnaf** except that optimization is performed with simulated annealing as suggested in the Press (1992) reference.

Usage: int **nmitrsad**(struct nmglbld *nmG, void *usrd, nmxyd **ppxbest, nmxyd *pybest, ui32 *pniter, nmxyd T, nmxyd ftol, nmxyd xtol, ui32 mxiter)
int **nmitrsaf**(struct nmglblf *nmG, void *usrd, nmxyf **ppxbest, nmxyf *pybest, ui32 *pniter, nmxyf T, nmxyf ftol, nmxyf xtol, ui32 mxiter)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: Same as for **nmitrnad** (**nmitrnaf**) with the addition of 'T', the annealing temperature for this series of iterations. N.B. The original article adds -T*log(rand) to each y value. This program multiples y by (1-T*log(rand)).

Functions **NMGETHD** and **NMGETHF**

These functions returns the location of the storage allocated for the history record kept by a previous call to **nmitr[ns]a[df]**. The result will point to one nmhistd (nmhistf) struct if the NMKH_LAST options bit was set, or to an array containing one nmhistd (nmhistf) struct for each iteration since the last **nminitd** (**nminitf**) call if the NMKH_EVRY options bit was set, or NULL if neither bit was set.

Usage: struct nmhistd ***nmgethd**(struct nmglbld *nmG)
struct nmhistf ***nmgethf**(struct nmglblf *nmG)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: 'nmG' is a pointer to an nmglbld (nmglblf) struct created by a previous call to **nmallod** (**nmallof)**.

Returns: See explanation above.

Functions **NMGETSD** and **NMGETSF**

These functions return the location of the storage allocated for the optimization statistics by a previous call to **nmallod** (**nmallof**). This array contains NNMST counts with individual meanings as defined in nelmeadd.h (nelmeadf.h).

Usage: ui32 ***nmgetsd**(struct nmglbld *nmG)
ui32 ***nmgetsf**(struct nmglblf *nmG)

Prototyped in: nelmeadd, nelmeadf, respectively.

Arguments: 'nmG' is a pointer to an nmglbld (nmglblf) struct created by a previous call to **nmallod** (**nmallof)**.

Returns: See explanation above.

Subroutines **NMFREED** and **NMFREEF**

These subroutines free global workspace storage allocated by **nmallod** (**nmallof**). After this call, the nmG pointer is invalid and no more calls to routines in the Nelder-Mead package should be made using this pointer.

Usage:
```
void nmfreed(struct nmglbld *nmG)
void nmfreef(struct nmglblf *nmG)
```

Arguments: 'nmG' is a pointer to an nmglbl (nmglblf) struct created by a previous call to **nmallod** (**nmallof**).

MATRIX OPERATIONS

The following routines, mainly derived from the IBM Scientific Subroutine Package or Numerical Recipes in C, have been included in the ROCKS system. Eventually a more complete set of matrix operations is expected to be provided. Algorithms are discussed in the comments in the source code of each subroutine.

Subroutine **MINV**

To invert an arbitrary single precision square matrix (Gauss-Jordan method):

Usage:
```
void minv(float *matrix, int n, float *determinant,
    int *iwork1, int *iwork2)
```

Prototyped in: rkarith.h

Arguments: 'matrix' is the matrix to be inverted, stored in standard C order, i.e., M(1,1), M(1,2), . . .M(1,n), M(2,1), ... On return, it contains the result. In typical applications, it is a rotation matrix. Symmetric least-squares matrices generally need to be double precision and are inverted with **dsinv**.

'n' is the order of the matrix.

'determinant' is returned as the determinant of the result matrix.

'iwork1' and 'iwork2' are integer arrays of length 'n' used as work areas during the calculations.

Implementation: This routine has not yet been implemented in the C version of the ROCKS library, but the prototype and description are kept for use when needed.

Function **DMFSD**

This function factors a given symmetric positive definite matrix using the square-root method of Cholesky. The given matrix is represented as product of two triangular matrices, where the left hand factor is the transpose of the returned right hand factor. It is called internally by **dsinv**.

Usage: int **dmfsd**(double *a, long n, float eps)

Arguments: 'a' is the upper triangular part of the matrix to be factored, stored columnwise in N*(N+1)/2 successive storage locations (see more detailed description below under **dsinv**). On return 'a' contains the resultant upper triangular matrix in double precision.

'n' is the size of (number of rows and columns in) 'a'.

'eps' is the relative tolerance for a test on loss of significance.

Value returned: 0 if no error; RK_POSDERR (=-1) if no result because of wrong input parameter (N <= 0 or matrix 'a' not positive definite); a positive integer, K, if loss of significance has occurred--the radicand formed at factorization step K+1 was still positive but no longer greater than abs(eps*a[k+1][k+1]).

Remarks: The product of the returned diagonal terms is equal to the square root of the determinant of the given matrix.

Function **DSINV**

To invert a symmetric double-precision matrix stored in triangular form:

Usage: int **dsinv**(double *matrix, long n, float tolerance)

Prototyped in: rkarith.h

Arguments: 'matrix' is the matrix to be inverted. It must be declared a one dimensional double-precision array of length n*(n+1)/2 and it must contain the upper triangular half of the symmetric input matrix arranged as follows:

```
    M(1)    M(2)    M(4)    .
            M(3)    M(5)    .
                    M(6)    .
                            .
```

On return, the original matrix is destroyed and replaced by the inverse in the same format.

'n' is the order of the matrix.

'tolerance' gives a relative tolerance for a test on loss of significance. 1E-9 is often a good value for this argument.

Value returned: Function **dsinv** returns the value returned to it by **dmfsd**: 0 if no error; RK_POSDERR (=-1) if no result because of wrong input parameter (N <= 0 or matrix 'a' not positive definite); a positive integer, K, if loss of significance has occurred--the radicand formed at factorization step K+1 was still positive but no longer greater than abs(eps*a[k+1][k+1]).

Function **JACOBI**

Function **jacobi** computes eigenvalues and, optionally, eigenvectors of a double-precision symmetric real matrix using Jacobi's method as adapted by Von Neumann and described in "Mathematical Methods for Digital Computers", A. Ralston and H.S. Wilf, eds, John Wiley and Sons, New York, 1962, Chapter 7, with improvements included in the Jacobi routine in "Numerical Recipes in C", 2nd edition, W.H. Press et al., Cambridge University Press, 1992. It uses the triangular matrix storage scheme of the IBM "Scientific Subroutines Package" as described above in connection with subroutine **dsinv** but is more accurate than the routine **deigen** from that package that was previously included with the ROCKS library. Corrections to diagonal elements are accumulated in an auxiliary vector to reduce errors. The C code was written by GNR and is free of copyright. However, it should be noted that more efficient methods (e.g. Householder) are available for large matrices and use of a routine from a commercial library should be considered for large applications.

Usage: int **jacobi**(double *a, double *val, double *vec, double dd, int N, int kvec, int ksort)

Prototyped in: rkarith.h

Arguments: 'a' is a pointer to the double precision upper triangular part of the input symmetric N x N matrix stored columnwise (A11, A12, A22, A13, etc) in N*(N+1)/2 storage locations. On return, the diagonal elements of 'a' are replaced with the eigenvalues of 'a' in arbitrary order. The rest of matrix 'a' is destroyed (set to 0) during the computation.

'val' is a pointer to a double precision vector of order N which will contain the eigenvalues of 'a' on return. Sorting is controlled by ksort (see below).

'vec' is a pointer to a double precision matrix of order NxN which will contain the eigenvectors of 'a' on return (may be a NULL pointer if 'kvec' is 0). Eigenvectors are stored columnwise in the same order as the corresponding eigenvalues.

'dd' is a pointer to a double precision work vector of order N for internal use by the routine.

'N' is the order of matrix 'a' and of the resulting eigenvalues and vectors.

'kvec' is a code which controls the computation of eigenvectors. If 'kvec' is 0, only eigenvalues are computed. If 'kvec' is 1, both eigenvalues and eigenvectors are computed.

'ksort' is a code which controls the sorting of the results. If 'ksort' is 0, eigenvalues and eigenvectors are returned in arbitrary (but the same) order. If 'kvec' is 1, eigenvalues and eigenvectors are sorted in decreasing order of eigenvalue magnitude.

Return values: Function **jacobi** returns -1 if the calculation failed to converge after 50 iterations. Otherwise, it returns the number of iterations required for convergence.

Other functions required: **sqrt**.

Note: Storage for 'a', 'val', 'vec', and 'dd' must not overlap in memory.

Subroutine **MATM33**

To multiply two 3x3 matrices:

Usage: void **matm33**(float *in1, float *in2, float *out)

Prototyped in: rkarith.h

Arguments: 'in1' and 'in2' are the input matrices, each of which must be declared single-precision real of dimension 3x3.

'out' is the result matrix, also 3x3.

Implementation: This routine has not yet been implemented in the C version of the ROCKS library, but the prototype and description are kept for use when needed.

64-BIT FIXED-POINT ARITHMETIC

The macros and functions in this section provide a full set of basic and some combined arithmetic operations using 64-bit operands or producing 64-bit results. These functions have also been used to provide C implementations of an older set of routines which use 64-bit intermediate results but which have only 32-bit arguments and return only 32-bit results. C implementations of all these routines are provided, using native 64-bit arithmetic where available, otherwise using multiprecision algorithms with 32-bit arithmetic. In the latter case, it may be advisable for performance reasons to write Assembler implementations for any functions that are heavily used in a particular application.

Two data types are typedef'd in sysdef.h or rkarith.h for use with these routines: si64 is a signed 64-bit integer and ui64 is an unsigned 64-bit integer. Variables of types si64 and ui64 may be declared freely in applications, but they should be accessed only with the functions and macros defined here. Routines in the **bem** and **lem** families are provided to communicate variables of these types to and from files and messages. Some macros generate multiline code and therefore must be coded as statements, not as functions: There are indicated by names ending in 'm'.

Because si64 and ui64 variables may be implemented as native longs, long longs, or as structures, they cannot be tested directly in **if** statements. Use the macros **qsw** and **quw**, respectively, to test whether a 64-bit value is greater than, equal to, or less than zero.

Macros are provided in rkarith.h to obtain the absolute values of any of the arithmetic types defined in the ROCKS library, namely, **absb**, **absj**, **absm**, and **absw** (or **jabs**) with arguments of type sbig, si32, smed, or si64, respectively. These are in addition to the macros **abs32** and **abs64** provided in sysdef.h.

Routines that perform operations that can result in overflow errors are provided in alternative versions that do and do not check for these errors. The error-checking versions all have names that end either in 'd' (or 'dm') or 'e' (or 'em'). The 'd' versions use a default error action specified in an earlier call to **e64dec**, whereas the 'e' versions include as argument a code that may either select an action to be performed or provide a numeric value to an action previously specified by subroutine **e64set**. If no default action has been specified, the fallback default is to terminate execution with an **abexit** call. When the current error-handling state is unknown, but needs to be restored after performing some operation, **e64push** changes the **e64set** setting but saves the previous settings on a push-down stack; **e64pop** restores the previous settings in the reverse of the order stored. The stack holds up to E64_STKDPTH (currently 5) entries. The errors detected are overflows on addition, subtraction, complementation, truncation, and left-shifting. Divide-by-zero errors always cause program termination.

Routines to preset or perform error actions

Subroutines **E64SET, E64PUSH,** and **E64POP**

```
Usage:  void e64set(int act, void *p)
        void e64push(int act, void *p)
        void e64pop(void)
```

Prototyped in: rkarith.h

```
Arguments:  'act' is a code that defines the action to be taken
    when an error is detected.  It is one of:
        E64_ABEXIT   (0)    Terminate with abexit (default).
        E64_COUNT    (1)    Count the errors in p[ec].
        E64_FLAGS    (2)    Set bit (1<<ec) in word at *p.
        E64_CALL     (3)    Call user-written routine
                            void (*p)(char *fnm, int ec).
    where 'ec' is an error code provided by the caller of the
    original e-suffix arithmetic routine that finds the error.

        'p' is a pointer to an ui32 or a user-written routine as
    specified by the 'act' argument.  The user is responsible for
```

seeing to it that sufficient storage is allocated at p to hold the specified counts or flags.

Subroutine **E64DEC**

e64dec provides a value of 'ec' that will be used by subsequent **e64dac** error calls from any of the d-suffix arithmetic routines. The 'act' code still comes from a previous call to **e64set** or **e64push**.

Usage: void **e64dec**(int ec)

Prototyped in: rkarith.h

Arguments: 'ec' is an error code as described above.

Subroutines **E64ACT** and **E64DAC**

These are the routines to be called when an overflow or other arithmentic error occurs. The d-suffix routines call **e64dac**; the e-suffix routines call **e64act**.

Usage: void **e64act**(void *fnm, int ec)
void **e64dac**(void *fnm)

Prototyped in: rkarith.h

Arguments: 'fnm' is a literal string that gives the name of the calling routine when the action is E64_ABEXIT. When the action is E64_CALL, 'fnm' can point to any information the user wants to pass to the preset routine. This argument is not used when the action is E64_COUNT or E64_FLAGS.

'ec' is an error code supplied by the caller of the routine that finds the error. If just an integer ('ec' < 2^24) is coded, the default action specified by the most recent call to **e64set** or **e64push** is performed with argument 'ec'. Alternatively, one of the following macros may be used to specify the action and the code in one int:

Action/code macros:

EAabx(ec)	Perform E64_ABEXIT action with code 'ec'.
EAct(ec)	Perform E64_COUNT action with code 'ec'.
EAfl(ec)	Perform E64_FLAGS action with code 'ec'.
EAcb(ec)	Perform E64_CALL action with code 'ec'.

Routines to generate 64-bit numbers

Functions **JCSW** and **JCUW**

These functions create signed or unsigned 64-bit fixed-point numbers, respectively, by "just concatenating" the high- and low-order 32 bits, which are provided as separate arguments.

```
Usage:  si64 jcsw(si32 hi, ui32 lo)
        ui64 jcuw(ui32 hi, ui32 lo)
```

Prototyped in: rkarith.h

Arguments: 'hi' is the high-order 32 bits of the desired 64-bit number; 'lo' is the low-order 32 bits.

Functions **JESL** and **JEUL**

These functions create signed or unsigned 64-bit fixed-point numbers, respectively, by "just extending" the low-order 32 bits.

```
Usage:  si64 jesl(si32 lo)
        ui64 jeul(ui32 lo)
```

Prototyped in: rkarith.h

Arguments: 'lo' is the low-order 32 bits of the desired number.

Macros **SL2W** and **UL2W** and functions **SW2LD, SW2LE, UW2LD,** and **UW2LE**

The macros convert respectively a signed long or an unsigned long, which may be 32 bits or 64 bits depending on the system, into the corresponding signed or unsigned 64-bit long or long long. The functions perform the opposite operations, checking for overflows.

```
Usage:  si64 sl2w(long x)
        ui64 ul2w(unsigned long x)
        long sw2ld(si64 x)
        long sw2le(si64 x, int ec)
        unsigned long uw2ld(ui64 x)
        unsigned long uw2le(ui64 x, int ec)
```

Defined in: rkarith.h

Arguments: 'x' is the quantity whose type is to be converted; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when a 64-bit value cannot be contained in a 32-bit long variable. 'ec' is ignored on systems with 64-bit longs.

Functions **DBL2SWD, DBL2SWE, DBL2UWD, DNL2UWE**

These functions convert a double-precision floating-point number to a 64-bit signed or unsigned integer with full overflow checking.

```
Usage:  si64 dbl2swd(double dx)
        si64 dbl2swe(double dx, int ec)
        ui64 dbl2uwd(double dx)
        ui64 dbl2uwe(double dx, int ec)
```

Defined in: rkarith.h

Arguments: 'dx' is the quantity whose type is to be converted; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow or conversion of a negative float to an ui64 occurs.

Routines to test 64-bit numbers

Macros **QSW** and **QUW**

These macros are used to test si64 or ui64 values, respectively, in **if** statements. On systems that implement native 64-bit integers, these macros simply return their arguments. On machines where 64-bit values are implemented as structures, these macros return a value of type si32 or ui32, respectively, that tests the same as the argument for greater-than, equal-to, or less-than (**qsw** only) zero.

```
Usage:  si64 qsw(si64 x)    /* Machine that has native si64 */
        ui64 quw(ui64 x)    /* Machine that has native ui64 */
        si32 qsw(si64 x)    /* Machine that has si64 struct */
        ui32 quw(ui64 x)    /* Machine that has ui64 struct */
```

Defined in: rkarith.h

Arguments: 'x' is the 64-bit quantity whose value is to be tested. For example, the following code is valid regardless of whether x is implemented as a native 64-bit long long or as a structure: `if (qsw(x) < 0) x = jesl(0);`

Macros **JCKSLO, JCKULO**

These macros "just" check whether a 64-bit signed or unsigned value will fit in the corresponding 32-bit signed or unsigned value, i.e. the high-order 32 bits are 0 (or all 1's in the case of a negative value).

Usage: int **jckslo**(si64 x)
int **jckulo**(ui64 x)

Defined in: rkarith.h

Argument: 'x' is the 64-bit signed or unsigned value to be tested.

Return value: An integer (or si32 or ui32) that is zero if the 64-bit argument can be truncated to 32 bits without loss, nonzero if the truncation would lose significant bits.

Macros **JCKSD, JCKSS, JCKUD, JCKUS**

These macros "just" check whether a signed or unsigned sum or difference of 32-bit numbers will overflow.

Usage: int **jcksd**(si32 dif, si32 a, si32 b)
int **jckss**(si32 sum, si32 a, si32 b)
int **jckud**(ui32 a, ui32 b)
int **jckus**(ui32 a, ui32 b)

Defined in: rkarith.h

Arguments: 'a' and 'b' are the two values to be added (**jckss** or **jckus**) or subtracted (**jcksd** or **jckud**); 'dif' is the naively computed difference; 'sum' is the naively computed sum.

Return value: An integer that is nonzero if the 32-bit sum or difference would overflow the 32-bit result, zero if the result does not overflow.

Routines to extract components from 64-bit numbers

Macros **SWHI, SWLO, SWLODM, SWLOEM, SWLOU, UWHI, UWLO, UWLODM,** and **UWLOEM** and functions **SWLOD, SWLOE, UWLOD** and **UWLOE**

These macros and functions return the high-order 32 bits of a signed 64-bit number (**SWHI**), the low-order 32 bits of a signed 64-bit number (**SWLO** family), the high-order 32 bits of an unsigned 64-bit number (**UWHI**), and the low-order 32 bits of an unsigned 64-bit number (**UWLO** family). When the magnitude of the 64-bit argument is too large to fit in the 32-bit result, the versions with 'D' in the name call **e64dac** and the versions with 'E' in the name call **e64act** with error code 'ec' to report the error. However, **swlou** returns the low-order 32 bits of a signed 64-bit number without regard to sign or overflow checking. This is generally used to implement multiprecision arithmetic routines. The macros with overflow checking generate code blocks; the corresponding functions perform the same actions at a slight cost in speed but can be used in more complex arithmetic expressions, i.e. anywhere a function call can be used.

Usage:
```
si32 swhi(si64 x64)
si32 swlo(si64 x64)
si32 swlod(si64 x64)
swlodm(si32 x32, si64 x64)
si32 swloe(si64 x64, int ec)
swloem(si32 x32, si64 x64, int ec)
ui32 swlou(si64 x64)
ui32 uwhi(ui64 x64)
ui32 uwlo(ui64 x64)
ui32 uwlod(ui64 x64)
uwlodm(ui32 x32, ui64 x64)
ui32 uwloe(ui64 x64, int ec)
uwloem(ui32 x32, ui64 x64, int ec)
```

Defined in: rkarith.h

Arguments: 'x64' is the 64-bit quantity whose components are to be extracted; 'x32' is the 32-bit result returned by the macros (these generate statements, not function calls); 'ec' is an error code to be passed to the action specified by the last call to **e64set** when the 64-bit value cannot be contained in a 32-bit variable.

Macros **SWDBL, SWFLT, UWDBL, UWFLT**

These macros convert 64-bit signed or unsigned numbers to double- or single-precision floating point numbers as indicated by the names. There are no overflow conditions.

Usage:
```
double swdbl(si64 x)
float swflt(si64 x)
double uwdbl(ui64 x)
float uwflt(ui64 x)
```

Defined in: rkarith.h

Arguments: 'x' is the 64-bit value to be converted.

Routines to manipulate the signs of numbers

Macros **ABS32, ABS64, ABSB, ABSJ, ABSM,** and **ABSW** (or **JABS**)

These macros return the absolute value of a typedef'd argument. **abs32** and **absj** are synonymous, as are **abs64** and **absw.**

Usage:
```
si32 abs32(si32 x)
si64 abs64(si64 x)
sbig absb(sbig x)
si32 absj(si32 x)
smed absm(absm x)
si64 absw(si64 x)
si64 jabs(si64 x)
```

Defined in: sysdef.h (**abs32** and **abs64**), rkarith.h (the others)

Arguments: 'x' is a signed quantity of the indicated type whose absolute value is to be determined.

Note: The definition of **jabs** is irregular and is kept only for compatibility with earlier versions of the library. Use **absw** or **abs64** in new code.

Functions **JNSW, JNSWD,** and **JNSWE**

These functions "just negate" a signed 64-bit number.

Usage:
```
si64 jnsw(si64 x)
si64 jnswd(si64 x)
si64 jnswe(si64 x, int ec)
```

Prototyped in: rkarith.h

Arguments: 'x' it the signed 64-bit number to be negated; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs (this happens only when an attempt is made to negate the largest negative number).

Addition and subtraction routines

Macros **JASIDM** and **JASIEM**

These macros "just add" two signed 32-bit integers and return a sum of the same type with overflow checking. (The corresponding unsigned sums can be performed with **jauadm** or **jauaem.**) Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jasidm**(int sum, int a, int b)
jasiem(int sum, int a, int b, int ec)

Defined in: rkarith.h

Arguments: 'sum' is the result; 'a' and 'b' are the quantities to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Macros **JASJDM** and **JASJEM**

These macros "just add" two signed 32-bit fixed-point quantities and return a sum of the same type with overflow checking. (The corresponding unsigned sums can be performed with **jauadm** or **jauaem**.) Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jasjdm**(si32 sum, si32 a, si32 b)
jasjem(si32 sum, si32 a, si32 b, int ec)

Defined in: rkarith.h

Arguments: 'sum' is the result; 'a' and 'b' are the quantities to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when a overflow occurs.

Macros **JASLDM** and **JASLEM**

These macros "just add" two signed fixed-point quantities declared as long integers and return a sum of the same type with overflow checking. (The corresponding unsigned sums can be performed with **jauadm** or **jauaem**.) Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jasldm**(long sum, long a, long b)
jaslem(long sum, long a, long b, int ec)

Defined in: rkarith.h

Arguments: 'sum' is the result; 'a' and 'b' are the quantities to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when a overflow occurs.

Macros **JAUADM** and **JAUAEM**

These macros "just add" two unsigned fixed-point numbers and return a sum of the same type with overflow checking. These macros for unsigned types can accept any base type (any integer type except ui64, which may be implemented as a struct in 32-bit systems). Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jauadm**(unsigned sum, unsigned a, unsigned b)
       jauaem(unsigned sum, unsigned a, unsigned b, int ec)

Defined in: rkarith.h

Arguments: 'sum' is the result; 'a' and 'b' are the quantities to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Functions **JASW, JASWD, JASWE, JAUW, JAUWD,** and **JAUWE**

These functions "just add" two 64-bit quantities and return a 64-bit sum. **jaswd**, **jaswe**, **jauwd**, and **jauwe** check for overflow.

Usage: si64 **jasw**(si64 x, si64 y)
       ui64 **jauw**(ui64 x, ui64 y)
       si64 **jaswd**(si64 x, si64 y)
       si64 **jaswe**(si64 x, si64 y, int ec)
       ui64 **jauwd**(ui64 x, ui64 y)
       ui64 **jauwe**(ui64 x, ui64 y, int ec)

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the two numbers to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Functions **JASL, JASLD, JASLE, JAUL, JAULD,** and **JAULE**

These functions add a 32-bit quantity to a 64-bit quantity and return a 64-bit sum.

Usage: si64 **jasl**(si64 x, si32 y)
       ui64 **jaul**(ui64 x, ui32 y)
       si64 **jasld**(si64 x, si32 y)
       si64 **jasle**(si64 x, si32 y, int ec)
       ui64 **jauld**(ui64 x, ui32 y)
       ui64 **jaule**(ui64 x, ui32 y, int ec)

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the two numbers to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Macros **JRSIDM** and **JRSIEM**

These macros "just reduce" (subtract) two signed 32-bit integers and return a difference of the same type with overflow checking. (The corresponding unsigned differences can be performed with **jruadm** or **jruaem**.) Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jrsidm**(int diff, int a, int b)
jrsiem(int diff, int a, int b, int ec)

Defined in: rkarith.h

Arguments: 'diff' is the result; 'a' and 'b' are the quantities to be subtracted; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Macros **JRSJDM** and **JRSJEM**

These macros "just reduce" (subtract) two signed 32-bit fixed-point quantities and return a difference of the same type with overflow checking. (The corresponding unsigned differences can be performed with **jruadm** or **jruaem**.) Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jrsjdm**(si32 diff, si32 a, si32 b)
jrsjem(si32 diff, si32 a, si32 b, int ec)

Defined in: rkarith.h

Arguments: 'diff' is the result; 'a' and 'b' are the quantities to be added; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when a overflow occurs.

Macros **JRSLDM** and **JRSLEM**

These macros "just reduce" (subtract) two signed long integers and return a difference of the same type with overflow checking. (The corresponding unsigned differences can be performed with **jauadm** or **jauaem**.) Note that because these functions are implemented as macros, the result is an argument, not a function return.

Usage: **jasldm**(long diff, long a, long b)
jaslem(long diff, long a, long b, int ec)

Defined in: rkarith.h

Arguments: 'diff' is the result; 'a' and 'b' are the quantities to be subtracted; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when a overflow occurs.

Macros **JRUADM** and **JRUAEM**

These macros "just reduce" (subtract) two unsigned fixed-point quantities and return a difference of the same type with overflow checking. These macros for unsigned types can accept any base type (any integer type except ui64, which may be implemented as a struct in 32-bit systems). Note that because these functions are implemented as macros, the result is an argument, not a function return.

```
Usage:  jruadm(ui32 diff, ui32 a, ui32 b)
        jruaem(ui32 diff, ui32 a, ui32 b, int ec)
```

Defined in: rkarith.h

Arguments: 'diff' is the result; 'a' and 'b' are the quantities to be subtracted; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when a overflow occurs.

Functions **JRSW, JRSWD, JRSWE, JRUW, JRUWD,** and **JRUWE**

These functions "just reduce" (subtract) two 64-bit quantities and return a 64-bit remainder. **jrswd**, **jrswe**, **jruwd**, and **jruwe** check for overflow.

```
Usage:  si64 jrsw(si64 x, si64 y)
        ui64 jruw(ui64 x, ui64 y)
        si64 jrswd(si64 x, si64 y)
        si64 jrswe(si64 x, si64 y, int ec)
        ui64 jruwd(ui64 x, ui64 y)
        ui64 jruwe(ui64 x, ui64 y, int ec)
```

Prototyped in: rkarith.h

Arguments: 'x' is the subtrahend; 'y' is the minuend; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Functions **JRSL, JRSLD, JRSLE, JRUL, JRULD, JRULE**

These functions subtract a 32-bit quantity from a 64-bit quantity and return a 64-bit difference.

```
Usage:  si64 jrsl(si64 x, si32 y)
        si64 jrsld(si64 x, si32 y)
        si64 jrsle(si64 x, si32 y, int ec)
        ui64 jrul(ui64 x, ui32 y)
        ui64 jruld(ui64 x, ui32 y)
        ui64 jrule(ui64 x, ui32 y, int ec)
```

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the two numbers to be subtracted; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Shift routines

Functions **JSSW, JSSWD, JSSWE, JSUW, JSUWD,** and **JSUWE**

These routines "just shift" a 64-bit number by a given amount in a direction determined at runtime by the sign of the shift. The versions ending in 'd' or 'e' perform overflow checking. Left shifts without overflow checking should be used only when there is no possibility of overflow or overflow can be neglected due to program design. Note that versions of these routines that can be used when the sign of the shift is known at compile time are also provided (next section) and these will be slightly faster because they do not need to check the sign of the shift.

```
Usage:  si64 jssw(si64 x, int s)
        si64 jsswd(si64 x, int s)
        si64 jsswe(si64 x, int s, int ec)
        ui64 jsuw(ui64 x, int s)
        ui64 jsuwd(ui64 x, int s)
        ui64 jsuwe(ui64 x, int s, int ec)
```

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit number to be shifted; 's' is the amount of shift (positive for left shift, negative for right shift); 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Functions **JSLSWD, JSLSWE, JSLUWD, JSLUWE, JSRSW, JSRUW**

These functions perform a left (**jslswd**, **jslswe**, **jsluwd**, **jsluwe**) or right (**jsrsw**, **jsruw**) shift on a 64-bit value when the sign of the shift is known at compile time. There is full overflow checking for left shifts; overflow is not possible with right shifts and so overflow-checking versions are not provided. **jsrsw** and **jsruw** are implemented as macros if the system has 64-bit arithmetic.

Usage: si64 **jslswd**(si64 x, int s)
si64 **jslswe**(si64 x, int s, int ec)
ui64 **jsluwd**(ui64 x, int s)
ui64 **jsluwe**(ui64 x, int s, int ec)
si64 **jsrsw**(si64 x, int s)
ui64 **jsruw**(ui64 x, int s)

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit number to be shifted; 's' is the amount of shift, which must be in the range 0 < 's' < 64 (not checked); 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Functions **JSRRSW** and **JSRRUW**

These functions perform a right shift on a 64-bit value with rounding. A value equal to the largest bit to be shifted out is added to the argument before shifting.

Usage: si64 **jsrrsw**(si64 x, int s)
ui64 **jsrruw**(ui64 x, int s)

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit number to be shifted with rounding; 's' is the amount of shift, which must be in the range 0 < 's' < 64 (not checked).

Macros **JSLSW** and **JSLUW**

These macros perform a left shift on a 64-bit value with no overflow checking and no checking of the value of 's'. (In systems without 64-bit arithmetic, these macros are implemented as calls to **jssw** or **jsuw**, respectively). These macros should be used only when there is no possibility of overflow or when overflow can be neglected by program design.

Usage: si64 **jslsw**(si64 x, int s)

ui64 **jsluw**(ui64 x, int s)

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit value to be shifted, 's' is the amount of shift (assumed to be in the range 0 < 's' < 64).

Multiply; multiply and shift; multiply, shift, and round

Functions **JMSW** and **JMUW**

These functions "just multiply" two 32-bit numbers to generate a 64-bit product. Overflow is not possible.

Usage: si64 **jmsw**(si32 x, si32 y)
ui64 **jmuw**(ui32 x, ui32 y)

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the numbers to be multiplied.

Functions **JMSLD, JMSLE, JMULD,** and **JMULE**

These functions multiply two 32-bit numbers and return the low-order 32 bits of the product but check for overflow into the high order.

Usage: si32 **jmsld**(si32 x, si32y)
si32 **jmsle**(si32 x, si32 y, int ec)
ui32 **jmuld**(ui32 x, ui32 y)
ui32 **jmule**(ui32 x, ui32 y, int ec)

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the 32-bit numbers to be multiplied; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Functions **JMUWJD, JMUWJE, JMUWWD,** and **JMUWWE**

These functions multiply an unsigned 64-bit number by a 32-bit unsigned number (**jmuwjd** and **jmuwje**) or by another 64-bit unsigned number (**jmuwwd** and **jmuwwe**) and return the product with full overflow checking.

Usage: ui64 **jmuwjd**(ui64 x, ui32 y)
ui64 **jmuwje**(ui64 x, ui32 y, int ec)
ui64 **jmuwwd**(ui64 x, ui64 y)

```
        ui64 jmuwwe(ui64 x, ui64 y, int ec)
```

Defined in: rkarith.h

Arguments: 'x' and 'y' are the values that are to be multiplied; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Function **JMUWB**

This function multiplies a 64-bit unsigned integer by a 32-bit unsigned integer and returns the low-order 64 bits of the product as the function value. It also returns the high-order 32 bits of the product via a pointer argument. There are no error conditions. To keep a 96-bit product, the multiplication has to be carried out in pieces even if the machine has 64-bit arithmetic. (This routine is mainly useful for converting the fraction part of a fixed-point number to decimal.)

Usage: ui64 jmuwb(ui64 x, ui32 y, ui32 *phi)

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the numbers to be multiplied. 'phi' is a pointer to a location where the high-order product should be stored.

Functions **MRSSLD, MRSSLE, MRSSWD, MRSSWE, MRSSWJ, and MRSULD, MRSULE, MRSUWD, MRSUWE, MRSUWJ**

These routines multiply two numbers and right shift the result with full overflow checking (overflow is not possible with **mrsswj** and **mrsuwj**; these routines are implemented as macros on systems that have 64-bit arithmetic). See the next section for routines that round the result after shifting.

```
Usage:  si32 mrssld(si32 x, si32 y, int s)
        si32 mrssle(si32 x, si32 y, int s, int ec)
        si64 mrsswd(si64 x, si32 y, int s)
        si64 mrsswe(si64 x, si32 y, int s, int ec)
        si64 mrsswj(si32 x, si32 y, int s)
        ui32 mrsuld(ui32 x, ui32 y, int s)
        ui32 mrsule(ui32 x, ui32 y, int s, int ec)
        ui64 mrsuwd(ui64 x, ui32 y, int x)
        ui64 mrsuwe(ui64 x, ui32 y, int s, int ec)
        ui64 mrsuwj(ui32 x, ui32 y, int s)
```

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the numbers to be multiplied; 's' is a right shift to be applied, 0 <= s < 64, values of 's' are never checked for validity; and 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Note: Earlier versions of these routines were named **mssld**, etc. These versions allowed the shift 's' to be coded as a negative number by analogy with **jssw**, etc., where a negative shift indicated a right shift. 'r' has been added as the second letter in each routine name to indicate that the shift is always a right shift encoded as a positive integer. Macros have been added to provide the function of those earlier versions that were actually used, viz:

```
#define msswe(x,y,s,ec)  mrsswe(x,y,abs(s),ec)
#define msuwe(x,y,s,ec)  mrsuwe(x,y,abs(s),ec)
#define mssle(x,y,s,ec)  mrssle(x,y,abs(s),ec)
```

Note: Versions of **mrssle**, **mrsswe**, **mrsule**, and **mrsuwe** that do not check for overflow (**mrssl**, **mrssw**, **mrsul**, **mrsuw**, bzw) are available and are implemented as macros on systems that have 64-bit arithmetic for greater speed. Failure to detect overflow will usually cause incorrect results and these routines should be used only when there is no possibility of overflow or overflow discard is desired by design.

Note: Values of 'x' or 'y' equal to the most negative number are allowed--they can be handled if the shift brings the results back into the representable range.

Note: Shifting followed by negation of odd products gives an answer one bit different than SRA-type shifting of negatives, although one is hard-pressed to say which is "correct", e.g. SRA(-3,1) yields -2, not -1. Routines in this set were revised, Oct. 2014, to yield the same results as an SRA shift in all versions.

Functions **MRSRSLD, MRSRSLE, MRSRSWD, MRSRSWE, MRSRSWJ, MRSRULD, MRSRULE, MRSRUWD, MRSRUWE,** and **MRSRUWJ**

These routines multiply two numbers and shift the result to the right with rounding and full overflow checking (overflow is not possible with **mrsrswj** and **mrsruwj**). Rounding is accomplished by adding before shifting a value equal to the largest bit to be shifted out.

Usage:
```
si32 mrsrsld(si32 x, si32 y, int s)
si32 mrsrsle(si32 x, si32 y, int s, int ec)
si64 mrsrswd(si64 x, si32 y, int s)
si64 mrsrswe(si64 x, si32 y, int s, int ec)
```

```
        si64 mrsrswj(si32 x, si32 y, int s)
        ui32 mrsruld(ui32 x, ui32 y, int s)
        ui32 mrsrule(ui32 x, ui32 y, int s, int ec)
        ui64 mrsruwd(ui64 x, ui32 y, int s)
        ui64 mrsruwe(ui64 x, ui32 y, int s, int ec)
        ui64 mrsruwj(ui32 x, ui32 y, int s)
```

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the numbers to be multiplied; 's' is a right shift to be applied, 0 <= s < 64, values of 's' are never checked for validity; 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Note: Earlier versions of these routines were named **mssld**, etc. These versions allowed the shift 's' to be coded as a negative number by analogy with **jssw**, etc., where a negative shift indicated a right shift. 'r' has been added as the second letter in each routine name to indicate that the shift is always a right shift encoded as a positive integer. Macros have been added to provide the function of those earlier versions that were actually used, viz:

```
#define msrsle(x,y,s,ec)  mrsrsle(x,y,abs(s),ec)
#define msrswe(x,y,s,ec)  mrsrwe(x,y,abs(s),ec)
#define msrule(x,y,s,ec)  mrsrule(x,y,abs(s),ec)
```

Note: Values of 'x' or 'y' equal to the most negative number are allowed--they can be handled if the shift brings the results back into the representable range.

Note: Shifting followed by negation of odd products gives an answer one bit different than SRA-type shifting of negatives, although one is hard-pressed to say which is "correct", e.g. SRA(-3,1) yields -2, not -1. Routines in this set were revised, Oct. 2014, to yield the same results as an SRA shift in all versions.

Functions **MLSSWJD, MLSSWJE, MLSUWJD,** and **MLSUWJE**

These functions multiply two 32-bit numbers to produce a 64-bit product, then shift left with full error checking.

```
Usage:  si64 mlsswjd(si32 x, si32 y, int s)
        si64 mlsswje(si32 x, si32 y, int s, int ec)
        ui64 mlsuwjd(ui32 x, ui32 y, int s)
        ui64 mlsuwje(ui32 x, ui32 y, int s, int ec)
```

Prototyped in: rkarith.h

Arguments: 'x' and 'y' are the numbers to be multiplied; 's' is a left shift to be applied, 0 <= s < 64 (The shift is always a left shift, values of 's' are never checked for validity); 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Divide; shift, divide, and round

Functions or macros **JDSWQ, JDSWB, JDUWQ,** and **JDUWB**

These functions "just divide" a 64-bit number by a 32-bit number. **jdswq** and **jduwq** return only the 32-bit quotient; **jdswb** and **jduwb** return both quotient and remainder. Routines are not currently provided to return just the remainder, but these could easily be added. Divide by zero and quotient overflow always result in termination of execution--use **jdswqd**, **jdswqe**, **jdswbd**, **jdswbe**, **jduwqd**, or **jduwqe** to instead call **e64dac** or **e64act** when a quotient overflow occurs (overflow cannot occur with **jduwb** because it returns a 64-bit quotient).

```
Usage:  si32 jdswq(si64 x, si32 y)
        ui32 jduwq(ui64 x, ui32 y)
        si32 jdswb(si64 x, si32 y, si32 *pr)
        ui64 jduwb(ui64 x, ui32 y, ui32 *pr)
```

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit dividend; 'y' is the 32-bit divisor; 'pr' is a pointer to a 32-bit variable where the remainder is to be stored.

Functions **JDSWQD, JDSWQE, JDSWBD, JDSWBE, JDUWQD,** and **JDUWQE**

These functions "just divide" a 64-bit number by a 32-bit number. When a quotient overflow would occur, the error is reported by a call to **e64dac** (routine names ending in 'd') or **e64act** (routine names ending in 'e') and the largest possible numerical value is returned. **jdswqd**, **jdswqe**, **jduwqd**, and **jduwqe** return only the 32-bit quotient; **jdswbd** and **jdswbe** returns both quotient and remainder. Divide by zero always results in termination of execution.

```
Usage:  si32 jdswqd(si64 x, si32 y)
        si32 jdswqe(si64 x, si32 y, int ec)
        ui32 jduwqd(ui64 x, ui32 y)
        ui32 jduwqe(ui64 x, ui32 y, int ec)
        si32 jdswbd(si64 x, si32 y, si32 *pr)
        si32 jdswbe(si64 x, si32 y, int ec, si32 *pr)
```

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit dividend; 'y' is the 32-bit divisor; 'pr' is a pointer to a 32-bit variable where the remainder is to be stored, 'ec' is an error code to be passed to **e64act** when a quotient overflow occurs.

Functions **DSRSJQD** and **DSRSJQE**

These functions scale a signed 32-bit dividend by a specified left or right shift amount, perform rounding by adding half the divisor to the dividend, then perform the division and return the quotient. Quotient overflow is reported via **e64dac** (**dsrsjqd**) or **e64act** (**dsrsjqe**). Division by 0 is a terminal error. There is no version to return the remainder, which is of no interest after rounding has been applied.

```
Usage:  si32 dsrsjqd(si32 x, int s, si32 y)
        si32 dsrsjqe(si32 x, int s, si32 y, int ec)
```

Prototyped in: rkarith.h

Arguments: ‘x’ is the 32-bit dividend, ‘s’ is the shift to be performed before division (positive for left shifts, negatvie for right shifts), 'y' is the 32-bit divisor, and 'ec' is the error code to be passed to **e64act** when an overflow occurs.

Functions **DSRSWQ, DSRSWQD, DSRSWQE, DSRUWQ, DSRUWQD,** and **DSRUWQE**

These functions shift a 64-bit number right or left by a specified scale, then divide by a 32-bit number. The result is rounded by adding half the divisor to the dividend before dividing. Only the 32-bit quotient is returned; the remainder is generally of no interest after rounding has been performed. When a quotient overflow occurs, routines **dsrswqd** and **dsruwqd** call **e64dac**, **dsrswqe** and **dsruwqe** call **e64act**, and return respectively the largest 32-bit signed or unsigned number; **dsrswq** and **dsruwq** terminate execution on all overflows. All six routines terminate with abend 72 on a divide-by-zero error.

```
Usage:  si32 dsrswq(si64 x, int s, si32 y)
        si32 dsrswqd(si64 x, int s, si32 y)
        si32 dsrswqe(si64 x, int s, si32 y, int ec)
        ui32 dsruwq(ui64 x, int s, ui32 y)
        ui32 dsruwqd(ui64 x, int s, ui32 y)
        ui32 dsruwqe(ui64 x, int 2, ui32 y, int ec)
```

Prototyped in: rkarith.h

Arguments: 'x' is the 64-bit dividend; 's' is the shift to be performed before division (positive for left shifts, negative for right shifts), 'y' is the 32-bit divisor, and 'ec' is the error code to be passed to **e64act** when an overflow occurs.

Functions **DSRSWWJD, DSRSWWJE, DSRSWWQD, DSRSWWQE, DSRUWWJD, DSRUWWJE, DSRUWWQD,** and **DSRUWWQE**

These functions scale a 64-bit signed (**DSRSWWxx** routines) or unsigned (**DSRUWWxx** routines) value by a given left shift amount to get a 96-bit dividend, divide by a 64-bit divisor, round by adding the high-order bit of the remainder to the quotient, and return a 32-bit (**DSRxWWJx** routines) or 64-bit (**DSRxWWQx** routines) quotient, with quotient overflow checking. These programs terminate with abexit code 74 if the shift is outside the range 0 <= s <= 32. Division is performed with **vdivl.** A divisor of 0 is not considered an error, but returns a result of 0.

Usage: si32 **dsrswwje**(si64 x, si64 y, int s, int ec)
si32 **dsrswwjd**(si64 x, si64 y, int s)
si64 **dsrswwqe**(si64 x, si64 y, int s, int ec)
si64 **dsrswwqd**(si64 x, si64 y, int s)
ui32 **dsruwwje**(ui64 x, ui64 y, int s, int ec)
ui32 **dsruwwjd**(ui64 x, ui64 y, int s)
ui64 **dsruwwqe**(ui64 x, ui64 y, int s, int ec)
ui64 **dsruwwqd**(ui64 x, ui64 y, int s)

Prototyped in: rkarith.h

Arguments: ‘x’ is the dividiend, ‘y’ the divisor, ‘s’ the shift (0 <= s <= 32), and ‘ec’ is the error code to be passed to **e64act** when a quotient overflow occurs.

Routines that perform multiplication followed by division

Macros **DMSJQD, DMSJQE, DMUJQD,** and **DMUJQE**

These macros multiply two 32-bit signed (**dmsjqd** and **dmsjqe**) or unsigned (**dmujqd** and **dmujqe**) numbers, then divide the product by another 32-bit number. The 32-bit quotient is returned. **dmsjqd** and **dmujqd** call **e64dac** and **dmsjqe** and **dmujqe** call **e64act** and return respectively the largest 32-bit signed or unsigned number when an overflow occurs; all four terminate execution with abend 72 on a divide-by-zero error.

Usage: si32 **dmsjqd**(si32 m1, si32 m2, si32 div)
si32 **dmsjqe**(si32 m1, si32 m2, si32 div, int ec)
ui32 **dmujqd**(ui32 m1, ui32 m2, ui32 div)

```
ui32 dmujqe(ui32 m1, ui32 m2, ui32 div, int ec)
```

Prototyped in: rkarith.h

Arguments: 'm1' and 'm2' are the values to be multiplied, 'y' is the 32-bit divisor, and 'ec' is the error code to be passed to **e64act** when an overflow occurs.

Functions **DMRSWJWD** and **DMRSWJWE**

These functions multiply a 64-bit signed integer ‘x’ by a 32-bit signed integer ‘y’ to get a 94-bit intermediate product, divide that by a 32-bit divisor ‘d’, round by adding one-half the smallest bit that will be retained after scaling, scale by a specified right shift ‘s’, and return the signed 64-bit result with full overflow checking. (This sequence of operations might be useful where a sum is to be converted to an average, then scaled with rounding.)

Usage:
```
si64 dmrswjwd(si64 x, si32 y, si32 d, int s)
si64 dmrswjwe(si64 x, si32 y, si32 d, int s, int ec)
```

Prototyped in: rkarith.h

Arguments” ‘x’ is the dividend, ‘y’ the multiplier, ‘d’ is the divisor, ‘s’ the scale (|s| < 64, ‘s’ may be coded with a positive or a negative value, but a right shift is always performed), and ‘ec’ is the error code passed to **e64act** when an overflow occurs.

Notes: Abexit 74 occurs if |s| > 64. If the divisor is 0, a result of 0 is returned and no error occurs.

Functions for computing sums of products

Functions **AMSSW, AMSSWD, AMSSWE, AMSUW, AMSUWD,** and **AMSUWE**

These routines multiply two 32-bit numbers, shift the product by a given amount, and add the result into a 64-bit accumulator.

Usage:
```
si64 amssw(si64 sum, si32 x, si32 y, int s)
ui64 amsuw(ui64 sum, ui32 x, ui32 y, int s)
si64 amsswd(si64 sum, si32 x, si32 y, int s)
si64 amsswe(si64 sum, si32 x, si32 y, int s, int ec)
si64 amsuwd(ui64 sum, ui32 x, ui32 y, int s)
ui64 amsuwe(ui64 sum, ui32 x, ui32 y, int s, int ec)
```

Defined in: rkarith.h

Arguments: 'sum' is the 64-bit accumulator; 'x' and 'y' are the 32-bit values that are to be multiplied; 's' is the shift that is to be applied to the product (positive for left shift, negative for right shift); 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs. **amssw** and **amsuw** do not check for overflow.

Macros **AMLSSW, AMLSUW, AMRSSW, AMRSUW**

These macros perform the functions of **amssw** or **amsuw** when the sign of the shift is known at compilation time. **amlssw** and **amlsuw** perform a left shift before adding; **amrssw** and **amrsuw** perform a right shift. There is no error checking.

Usage:
```
si64 amlssw(si64 sum, si32 x, si32 y, int s)
si64 amrssw(si64 sum, si32 x, si32 y, int s)
ui64 amlsuw(ui64 sum, ui32 x, ui32 y, int s)
ui64 amrsuw(ui64 sum, ui32 x, ui32 y, int s)
```

Defined in: rkarith.h

Arguments: As above, except the shift values, 's', are always positive; the sign of the shift is set by the name of the macro.

Functions **AMUWWD** and **AMUWWE**

These functions (may be implemented as macros) multiply two unsigned 64-bit numbers, add the product to a third unsigned 64-bit number, and return the sum with full overflow checking.

Usage:
```
ui64 amuwwd(ui64 sum, ui64 x, ui64 y)
ui64 amuwwe(ui64 sum, ui64 x, ui64 y, int ec)
```

Prototyped in: rkarith.h

Arguments: 'sum' is the 64-bit accumulator; 'x' and 'y' are the 64-bit values that are to be multiplied; and 'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

ARITHMETIC WITH 64-BIT INTERMEDIATE RESULTS

The routines in this section perform arithmetic operations on 32-bit quantities that produce 32-bit results but require 64-bit quantities during the computation. These routines constitute an older set that was developed before 64-bit arithmetic became available in hardware; more flexibility may be achieved with the routines in the sets described above. A systemic method of naming such routines is first presented, followed by descriptions of some specific members of the family that have so far been implemented. Additional routines may be added when needed. While many of these routines are currently implemented as macros, it is anticipated that it may often be advantageous to implement them in Assembler language, as C implementations will require complex algorithms for multi-precision arithmetic on systems that do not have 64-bit arithmetic in the hardware.

Two 32-bit quantities are used to form an 'initial 64-bit number'. A 'main operation' (possibly involving another 32-bit argument) is carried out on this number to produce a second 64-bit number which will be referred to as the 'intermediate 64-bit number'. An optional shift is performed on the intermediate 64-bit number to yield a 'final 64-bit number'. 32 bits of the final 64-bit number are returned to the caller. The other 32 bits may either be ignored or loaded into a variable referenced by a pointer argument.

In all cases in which shifts are specified, a positive shift is to the left, and a negative shift is to the right. Some of the routines may specify that only one or the other shift direction is allowed.

Routine names are arrived at by concatenating 5 fields, each of which relates to an aspect of the operation carried out by the routine:

<Main Op><Initial 64 Bit><64><SHFT><RET>

<Main Op> = { d | m | j }
Main operation performed on initial 64-bit number:

- m Multiplication.
- d Division.
- j ("Just"). No main operation, just formation.

<Initial 64 Bit> = { m | c | s }
Manner in which initial 64-bit number is formed:

- m Multiply first two args.

```
        c    Concatenate first two args.  First arg is higher
                order.
        s    Like c except a shift is also performed.  Note that
                this shift precedes the main operation.

    <64> = 64

    <SHFT> = { n | s }
        Shifting of intermediate 64-bit number (i.e. shifting of
        the result of the main operation):

        n    No shift is performed.
        s    A shift is performed.

    <RET> = { h | l | r | q | b }
        Manner in which pieces of the final 64-bit number are
        returned to the caller:

        h     Return value is high order 32 bits of final 64-bit
                result (for <Main Op> = j or m only).
        l     Return value is low order 32 bits of final 64-bit
                result (for <Main Op> = j or m only).
        r     Return value is remainder (for <Main Op> = d only).
        q     Return value is quotient (for <Main Op> = d only).
        b     ("Both").  In this case part of the result is
                returned as the function value and an additional
                argument pointer is used to receive the rest of
                the result.  If <Main Op> = m or j, the high 32
                bits are returned, and the low 32 bits are loaded
                into an ui32 argument.  If <Main Op> = d, the
                quotient is returned and the remainder is loaded
                into a si32 argument.

Usage:  xx64xx(arg1,arg2,[ishft],[arg3],[fshft],[result2])

Arguments:
    arg1:   First 32-bit signed integer used to create initial
            64-bit number.
    arg2:   Second 32-bit integer used to create initial 64-bit
            number.  Signed if <Initial 64 bit> = m, unsigned
            otherwise.
    ishft:  Signed integer.  This argument only appears in the
            case of <Initial 64 Bit> = c.  It specifies the shift
            to be applied to the initial 64-bit number.
    arg3:   Signed 32-bit integer used as second operand of main
            operation along with 64-bit result of combining arg1
            and arg2.  This argument does not appear if <Main Op>
            = j.
```

fshft: Signed integer. Specifies final shift to be applied to intermediate 64-bit number to obtain result. This argument does not appear if <SHFT> = n.

result2: Address of 32-bit signed or unsigned integer which receives the remainder or low-order 32 bits of a 64-bit product as specified above. This argument appears only if <RET> = b.

Note: For maximum efficiency, none of these routines checks for invalid arguments, e.g. divide by zero or excessive shifts, or for arithmetic overflow. These are the responsibility of the caller.

Macro **DM64NB**

Function **dm64nb** multiplies two 32-bit integers, producing a 64-bit intermediate product, then divides the product by a 32-bit divisor and returns the 32-bit quotient as function value and the remainder via a pointer.

Usage: si32 **dm64nb**(si32 mul1, si32 mul2, si32 div, si32 *rem)

Prototyped in: rkarith.h

Macro **DM64NQ** (formerly **MDIV**)

Function **dm64nq** multiplies two 32-bit integers, producing a 64-bit intermediate product, then divides the product by a 32-bit divisor and returns the 32-bit quotient. This function was formerly called **mdiv**.

Usage: si32 **dm64nq**(si32 mul1, si32 mul2, si32 div)

Prototyped in: rkarith.h

Value returned: (mul1 * mul2)/div.

Function **DM64NR**

Function **dm64nr** multiplies two 32-bit integers, producing a 64-bit intermediate product, then divides the product by a 32-bit divisor and returns the 32-bit remainder.

Usage: si32 **dm64nr**(si32 mul1, si32 mul2, si32 div)

Prototyped in: rkarith.h

Macro **DS64NQ**

This function generates a 64 bit dividend by concatenating high- and low-order 32-bit values and shifting the resulting number left or right by a specified amount. Division is then performed and the 32-bit quotient is returned.

```
Usage:  si32 ds64nq(si32 hi32, ui32 lo32, int ishft,
            si32 div)
```

Prototyped in: rkarith.h

Function **JM64SB**

Function **jm64sb** multiplies two 32-bit integers, producing a 64-bit intermediate product, then performs an arithmetic shift on the product to yield a scaled result and returns the high-order 32 bits as the function value and the low-order 32 bits via a pointer argument.

```
Usage:  si32 jm64sb(si32 mul1, si32 mul2, int fshft,
    ui32 *lowbits)
```

Prototyped in: rkarith.h

Macro **JM64SH**

Function **jm64sh** multiplies two 32-bit integers, producing a 64-bit intermediate product, then performs an arithmetic shift on the product to yield a scaled result and returns the high-order 32 bits as the function value.

```
Usage:  si32 jm64sh(si32 mul1, si32 mul2, int fshft)
```

Prototyped in: rkarith.h

Macro **JM64SL** (formerly **MSHFT**)

Function **jm64sl** multiplies two 32-bit integers, producing a 64-bit intermediate product, then performs an arithmetic shift on the product to yield a scaled result, of which only the unsigned low-order 32 bits are returned. This function replaces **mshft**, but differs in that the convention for designating shifts has been rationalized to agree with the other xx64xx routines: positive or negative shifts are allowed and positive shifts are now to the left. Function **jm64sl** is equivalent to **dm64nq** with a divisor that is a power of two, but faster.

Usage: ui32 **jm64sl**(si32 mul1, si32 mul2, int fshft)

Prototyped in: rkarith.h

Value returned: (mul1 * mul2) << fshft.

Function **JM64NB**

Function **jm64nb** multiplies two 32-bit numbers to obtain a 64-bit product and returns the high-order 32 bits of the product as the function value and the low-order 32 bits via a pointer.

Usage: si32 **jm64nb**(si32 mul1, si32 mul2, ui32 *lowbits)

Prototyped in: rkarith.h

Macro **JM64NH**

Function **jm64nh** multiples two 32-bit numbers and returns the high-order 32 bits of the product.

Usage: si32 **jm64nh**(si32 mul1, si32 mul2)

Prototyped in: rkarith.h

Functions **UI32POW, UI32POWD,** and **UI32POWE**

These functions compute a positive integer power of a positive fixed point number. The result is returned on the same binary scale as the first argument. **ui32pow** is a synonym for **ui32powe;** this routine is obsolete--it was named before the convention of adding ‘e’ to a routine name to indicate use of an error code was developed.

Usage: ui32 **ui32pow**(ui32 val, int pow, int scale, int ec)
ui32 **ui32powd**(ui32 val, int pow, int scale)
ui32 **ui32powe**(ui32 val, int pow, int scale, int ec)

Arguments: 'val' is the value whose power is to be found.

'pow' is the desired power (must be a positive integer).

'scale' is the binary scale of (number of fraction bits in) 'val'.

'ec' is an error code to be passed to the action specified by the last call to **e64set** when an overflow occurs.

Prototyped in: rkarith.h

RANDOM NUMBER AND PERMUTATION GENERATORS

Function **UDEV**

Function **udev** generates a positive 32-bit integer pseudorandom number drawn from a uniform distribution. On most systems, this function will be implemented in Assembler language for fastest possible execution.

Usage: si32 **udev**(si32 *seed)

Prototyped in: rkarith.h

Argument: 'seed' is a positive 32-bit integer (1 <= seed <= (2**31-1)). It is replaced on return by a new seed for the next call. In a typical application, the user is allowed to enter a seed to 'prime' the calculation. When the same seed is used, the same random numbers are generated.

Value returned: The random number returned is the same as the value stored in 'seed'.

Algorithm: ir(i) = ir(i-1)*(7**5) % (2**31-1). This method is fast and simple, and produces reasonably good results (see Knuth, 'The Art of Computer Programming', Vol. I).

Function **NDEV**

Function **ndev** generates a scaled 32-bit integer pseudorandom number drawn from a normal distribution. On most systems, this function will be implemented in Assembler language for fastest possible execution.

Usage: si32 **ndev**(si32 *seed, si32 cmn, si32 csg)

Prototyped in: rkarith.h

Arguments: 'seed' is a positive 32-bit integer (1 <= seed <= (2**31-1)). It is replaced on return by a new seed for the next call. In a typical application, the user is allowed to enter a seed to 'prime' the calculation. When the same seed is used, the same random numbers are generated.

'cmn' is the signed fixed-point mean value of the normal distribution to be used. Any desired bit scale can be used.

'csg' is the signed fixed-point standard deviation of the normal distribution to be used. The number of fraction bits in 'csg' must be four more than the number of fraction bits in 'cmn'.

Value returned: The normal variate returned has the same scale as 'cmn'. Scales S24 (7 integer bits and 24 fraction bits) for 'cmn' and S28 for 'csg' are often useful. (Due to a limitation imposed to make the algorithm fast, the value returned is never more than 3.0 standard deviations from the mean.)

Algorithm: **udev** is called to generate a uniform variate. The first eight bits of the value returned are used to select from a look-up table an interval of width 0.125 in the range 0.0 - 3.0 with appropriate frequency. The remaining bits specify the location within the chosen interval and the sign. The resulting value is multiplied by 'sigma' and finally 'mean' is added. The result is a piecewise rectangular approximation to the true Gaussian distribution, good enough for most purposes. See Knuth, "The Art of Computer Programming", Vol. 2.

Subroutine **UDEVSKIP**

Subroutine **udevskip** may be used to advance the seed for a random number sequence as if **udev** or **ndev** had been called some given number of times. This operation is frequently needed in parallel programming to obtain deterministic results equivalent to those obtained with the corresponding serial program by priming the random number generator on a computational node according to the number of calls executed on lower-numbered nodes. When the number of skips is large, **udevskip** will perform this operation much more rapidly than performing the individual calls to **udev**.

Usage: void **udevskip**(si32 *seed, si32 n)

Prototyped in: rkarith.h

Arguments: 'seed' is the random number generating seed to be adjusted. The new value replaces the current value of 'seed'.

'n' is the number of calls to **udev** that are to be simulated.

Algorithm: The formula for the k'th number that would be returned by the 'udev' function is x(k) = (R**k)*x (mod M) = (R**k (mod M))*x (mod M) where R = 7**5 = 16807 and M = 2**31 - 1 = 147483647 (See Knuth's chapter on random number generators if this is not obvious.) This is expressed as x(k) = P(k)*x and the values of P are tabulated for values of k that are exact powers of two. x(k) is then obtained by repeated application of the above formula for each tabulated value that corresponds to a one in the binary representation of k. This method runs in time proportional to log(2)(k), and represents a compromise between speed and the space that would be required to store larger tables of P values. The tabulated values of P were calculated using the REXX program 'udevtabl exec', which did the calculation with explicit multiprecision arithmetic.

Acknowldegment: Thanks to Joe Brandenberg of Intel Scientific Computers for the idea used in this routine.

Subroutine **RANNUM** (formerly **RANDOM**)

Subroutine **rannum** is used to generate one or more pseudorandom integers drawn from a uniform distribution. Provision is made to generate positive numbers in various ranges or mixed positive and negative numbers. If integer values in the range 1 <= n <= (2**31-1) are needed, **udev** is more efficient.

Usage: void **rannum**(si32 *ir, int n, si32 *seed, int bits)

Prototyped in: rkarith.h

Arguments: 'ir' is a 32-bit integer array of dimension 'n' into which the results are stored.

'n' is the number of random integers to be generated. Use of 'n' > 1 can improve efficiency by reducing the overhead of multiple calls.

'seed' is an si32 integer in the range from 1 to (2**31)-1. It is replaced on return by a new seed for the next call. In a typical application, the user is allowed to enter a seed to 'prime' the calculation. When the same seed is used, the same random numbers are generated.

'bits' is equal to 31 minus the number of bits required in the output values. Set 'bits' = -1 to produce signed (32-bit) random numbers. For 'bits' ≥ 0, 'bits' is the number of right shifts to apply to a value returned by **udev**, and output values in 'ir' cover the range 1 to 2**(31-bits)-1. (This definition is reversed from the convention used with the xx64xx routines, but is for compatibility with the FORTRAN version.)

Algorithm: ir(i) = ir(i-1)*(7**5) % (2**31-1). Result is shifted right 'bits' bits. If 'bits' == -1, the low-order bit of the quotient in the above formula is used to set the sign of the result. Results are the same as given by the IBM Assembler routine **random**.

Errors: Execution is terminated with abexit code 70 if $n \leq 0$.

Subroutine **RANNOR**

Subroutine **rannor** is used to generate a sequence of normally distributed pseudorandom numbers.

Usage: void **rannor**(float *r, int n, si32 *seed, float mean, float sigma)

Arguments: 'r' is an array large enough to hold the results.

'n' is the number of values to be generated.

'seed' is an si32 integer in the range from 1 to (2**31)-1. It is replaced on return by a new seed for the next call. In a typical application, the user is allowed to enter a seed to 'prime' the calculation. When the same seed is used, the same random numbers are generated.

'mean' is the desired mean of the normal distribution of the random values to be produced.

'sigma' is the desired standard deviation of the normal distribution of random values to be produced.

Value returned: Due to a limitation imposed to make the algorithm fast, the values returned are never more than 3.0 standard deviations from the mean.

Algorithm: Same as **ndev**, except calculations are carried out in floating point arithmetic.

Errors: The application is terminated with abexit code 71 if $n < 1$.

Function **RAND**

This function provides a crude random number generator which returns only a 15 bit result. **rand** is taken directly from Kernigan & Ritchie, second edition, page 46, and therefore may be useful for comparing results with other standand programs. It also may be useful for testing on new systems before the 64 bit arithmetic routines needed for **udev**, etc. have been implemented. The seed may be set by calling **srand** (see below).

Usage: int **rand**(void)

Value returned: A pseudorandom positive integer in the range 0 <= rand <= 32767.

Algorithm:
```
seed = seed * 1103515245 + 12345;
rand = (unsigned int)(seed/65536) % 32768;
```
Note that after 2**30 calls, this random number generator enters a simple arithmetic sequence from which it never escapes. This condition is not important for the uses to which the this routine will be put.

Subroutine **SRAND**

Subroutine **srand** is used to set the seed used by function **rand** to a new value.

Usage: void **srand**(unsigned int seed)

Argument: 'seed' is an unsigned integer in the range from 1 to (2**31)-1.

Subroutine **RANDSKIP**

Subroutine **randskip** may be used to advance the seed for a random number sequence as if **rand** had been called some given number of times (see discussion of **udevskip**).

Usage: void **randskip**(ui32 n)

Argument: 'n' is the number of calls to **rand** that are to be simulated.

Function **NRAND**

Function **nrand** generates a pseudorandom number from a normal distribution with zero mean and unit standard deviation. This distribution can be converted by the caller to any other normal distribution by multiplying the value returned by the desired standard deviation and adding the desired mean. Deviations from the mean greater than 3.0 standard deviations will never be generated. This function uses the method of **rand** to generate two uniform random numbers used internally; otherwise, it is similar to **ndev**.

Usage: float **nrand**(si32 *seed)

Argument: 'seed' is an si32 integer in the range from 1 to (2**31)-1. It is replaced on return by a new seed for the next call.

Value returned: The desired normally distributed pseudorandom number.

Subroutines **SI16PERM** and **SI32PERM**

These routines generate a random permutation of a stored list of n si16 or si32 values, respectively, given n with 2 <= n <= (2^31-1) and a udev seed. (The same routines can be used to permute an array of unsigned values by casting the argument pointer.)

Usage: void si16perm(si16 *pval, si32 *seed, si32 n)
void si32perm(si32 *pval, si32 *seed, si32 n)

Prototyped in: rkarith.h

Arguments: 'pval' is a pointer to the array to be permuted.

'seed' is a pointer to a random number seed that will be updated with the result of n calls to udev(seed).

'n' is the size of the permutation.

Errors: Abexit 79 if n < 2.

MISCELLANEOUS MATHEMATICAL ROUTINES

Function **BESSI0**

Function **bessi0** calculates $I_0(x) = J_0(ix)$, the modified Bessel function of the first kind, where 'x' is any real argument.

Usage: float **bessi0**(float x)

Prototyped in: rkarith.h

Algorithm: For |x| < 3.75, an optimized polynomial is used; for $|x| \geq 3.75$, exp(x)/sqrt(2*pi*x) times another polynomial. See M. Abramowitz and I.A. Stegun, Handbook of Mathematical Functions. Applied Mathematics Series, Vol. 55 (Washington, National Bureau of Standards; reprinted 1968 by Dover Publications, New York), Section 9.8 (1964).

Errors: This function will overflow when the exponential function used for large |x| overflows; the error handling will be whatever is provided by the library exp() function.

Functions **BITSZS32, BITSZU32, BITSZSW,** and **BITSZUW**

These functions calculate the number of bits needed to store a value of one less than the argument. Functions **bitszs32** and **bitszsw** operate on the magnitude of the signed argument.

Usage: int **bitszs32**(si32 x)
int **bitszu32**(ui32 x)
int **bitszsw**(si64 x)
int **bitszuw**(ui64 x)

Prototyped in: rkarith.h

Algorithm: Returns (int)(log2(|x|-1)+1) if x > 1, otherwise 0.

Macros **BITSZ, BITSZSL,** and **BITSZUL**

These macros call one of the above bit-size functions according to the type of the argument. **bitsz** is equivalent to **bitszs32** and obsolete; it is retained only for compatibility with older versions of this library. **bitszsl** calls either **bitszs32** or **bitszsw** according to whether a long is 32 or 64 bits; similarly for **bitszul**.

Usage: int **bitsz**(si32 x)
int **bitszsl**(long x)
int **bitszul**(unsigned long x)

Protytped in: rkarith.h

Function **GETPRIME**

Function **getprime** returns the smallest prime number that is equal to or larger than the argument. This function is intended mainly for such uses as determining a suitable size for a hash table.

Usage: ui32 **getprime**(ui32 minval)

Prototyped in: rkarith.h

Algorithm: Beginning with the next odd number equal to or larger than minval, checks each odd number in turn until a prime is found. Checking consists of dividing by 3 and all numbers of the form 6n+5 and 6n+7, n=0,1,... up to the square root of the current candidate prime.

Errors: None.

Functions **LMULUP** and **UMULUP**

These functions compute the next integer greater than or equal in magnitude to the first argument that is an exact multiple of (at
least one times) the second argument.

Usage: si32 **lmulup**(si32 val, si32 base)
ui32 **umulup**(ui32 val, ui32 base)

Prototyped in: rkarith.h

Note: 'ui32' refers to an unsigned 32-bit integer quantity. The appropriate 'typedef' is in sysdef.h

Errors: Abexit 77 if base is zero or result would overflow.

Function **ERFCF**

Function **erfcf** computes the single-precision complement of the error function of a single precision argument x, erfc(x) = 1.0 - erf(x). Intermediate calculations are in double precision. The name is chosen to avoid conflict with any other erfc function on a system.

Usage: float **erfcf**(float x)

Argument: 'x' is the argument to the erfc() function.

Note: The approximation polynomial is taken from M. Abramowitz and I.A. Stegun (1964), Handbook of Mathematical Functions, Applied Mathematics Series, vol. 55 (Washington: National Bureau of Standards; reprinted (1968) by Dover Publications, New York) Par. 7.1.26. The absolute error is claimed to be less than 1.5E-7.

Function **RERFC**

Function **rerfc** calculates the complement of the standard error integral, erfc(x) = 1 - erf(x) = (2/sqrt(pi))*integral(from x to infinity)(exp(-t*t) dt), where x is any real argument.

Usage: float **rerfc**(float x)

Prototyped in: rkarith.h

Algorithm: Separate Chebyshev approximation polynomials are used in the ranges 0 <= x < 0.5, 0.5 <= x < 4.0, and 4.0 <= x < 10.0. erfc(x > 10.0) = 2.0 and erfc(-x) = 2 - erfc(x). Coded by G.N.R. See W.J. Cody, Math. of Comp. ??, 631-637 (1969).

Errors: None--returns a value over the entire range of x.

Function **RMBF10**

Function **rmbf10** calculates the ratio I1(x)/I0(x), where I1(x) and I0(x) are modified Bessel functions of the first kind and 'x' is an argument $\geq$ 0.

Usage: float **rmbf10**(float x)

Prototyped in: rkarith.h

Algorithm: For 'x' $\leq$ 9.8: Gauss's continued fraction; for 'x' > 9.8: Perron's continued fraction; coded by G.N.R. See W. Gautschi and J. Slavik, Math. of Comp. 32, 865-875 (1978).

Errors: Execution is terminated with abexit code 80 if x < 0. Otherwise, accuracy is equal to that of a single precision real variable.

Implementation: This routine has not yet been implemented in the C version of the ROCKS library, but the prototype and description are kept for use when needed.

Subroutine **VDIVL**

Multi-precision division. This subroutine divides an unsigned integer, x, composed of nx ui32 'digits', by an unsigned divisor, y, composed of ny ui32 'digits'. It returns a quotient of nx 'digits' (the number of nonzero digits is actually lx-ly+1 where lx and ly are the numbers of nonzero digits in x and y, respectively), and optionally a remainder of ny 'digits', or the remainder may be used to round up the quotient. **vdivl** can handle leading zeros in the arguments x and y and generates leading zeros as necessary in the quotient and remainder.

vdivl is designed to perform long division in a 32- or 64-bit environment to support various other ROCKS routines such as **wbcdin**, **udevw**, **jduwb**, etc. with a single invariant and thoroughly tested piece of code. To handle arguments of various lengths, they are all accessed with pointers. The only requirements are that the arguments all be multiples of 32 bits in length, and arranged in memory in big-endian or little-endian order as specified by the BYTE_ORDRE compile-time definition in sysdef.h. The caller is responsible for dealing with signed arithmetic.

If the divisor is larger than 2^16 (2^32 if the machine has 64-bit arithmetic), **vdivl** performs classical long division in base 2^16 (2^32 with 64-bit arithmetic) using notation and a renormalizing trick found at

http://www.imsc.res/in/~kapil/crypto/notes/node4.html

which assures that the 'guess' obtained by short division is off by not more than 2 from the correct answer digit. The minimum number of iterations dictated by the sizes of the arguments is performed.

Usage: void **vdivl**(ui32 *x, ui32 *y, ui32 *pq, ui32 *pr, ui32 *u, int nx, int ny, int kr)

Prototyped in: rkarith.h

Arguments: 'x' is a pointer to the dividend, x. 'x' is an array of 'nx' ui32 variables. If the variables are other types, e.g. ui64, the pointer can be cast to a ui32 pointer.

'y' is a pointer to the divisor, y. 'y' is array of 'ny' ui32 variables.

'pq' is a pointer to an array of length 'nx' ui32 variables where the quotient will be returned. Can be the same array as 'x'.

'pr' is a pointer to an array of length 'ny' ui32 variables where the remainder will be returned. May be same array as y. May be NULL if the remainder is not needed.

'u' is a pointer to a work area of size at least 4*nx+2*ny+2 ui32 variables.

'nx' is the size of x in units of 32 bits.

'ny' is the size of y in units of 32 bits.

'kr' is TRUE if the quotient should be rounded up when the remainder is equal to or greater than 1/2 the divisor. FALSE to omit rounding.

The order in storage of all argument arrays is specified by the compile-time parameter BYTE_ORDRE.

By definition, the code returns quotient = remainder = 0 if the divisor is 0. This eliminates the need for zero checking in the caller, for example, if an average statistic is being calculated. If 0 divisor is an error, the caller needs to check for it. There are no other possible errors.

Results of tests with 1,000,000 randomly generated problems (64-bit numbers divided by 32-bit numbers) on a 2400 MHz Pentium were as follows:

were as follows:		Binary division	vdivl algorithm
-g	NO_I64	0.715 sec	0.698 sec
-O2	NO_I64	0.112	0.255
-g	HAS_I64	0.459	0.574
_O2	HAS_I64	0.104	0.199

So the binary algorithm, when optimized, is faster, but as written, that algorithm cannot handle general-length problems. Tests in which the work area was defined as a ui64 array (to get rid of many of the explicit type conversions in the HAS_I64 code) gave no improvement, so the work area was left as ui32.

INDEX

INDEX TO SUBROUTINE CALLS

(Routines that require use of the ROCKS formatted I/O routines (cryin, cryout, etc.) are marked with a '+' in column 1. The unmarked remaining routines can be used in any environment and are guaranteed to perform error exits via the **abexit**/**abexitm**/**abexitme** routines and memory allocations via the **callocv**/**mallocv**/**reallocv**/**freev** family of routines, which can be replaced by simple user-written routines when necessary in certain environments, e.g. MATLAB mex routines.)

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

INDEX

- End -