Application Programmer's Library Reference Manual

Volume 2

SR–2165 2.0

Cray Research, Inc.

CONTENTS

Conversion routines

intro_conversion, INTRO_CONVERSION ..... Introduction to conversion routines 425

Common pages

cri2cray, CRI2CRAY, CRAY2CRI ..... Converts IEEE/MPP 64-bit data to Cray Research PVP 64-bit data and vice versa 433

PVP Pages

b2oct, B2OCT ..... Places an octal Hollerith representation of a Cray Research numeric value into a specified part of an integer array 437

cdc2cray, CDC2CRAY, CRAY2CDC ..... Converts CDC data to Cray Research format and vice versa 439

chconv, CHCONV ..... Converts decimal ASCII numerals to an integer value 443

dsasc, DSASC, ASCDC ..... Converts CDC display code character to ASCII character and vice versa 445

eta2cray, ETA2CRAY, CRAY2ETA ..... Converts ETA/CYBER 205 data to Cray Research format and vice versa 446

fp6064, FP6064, FP6460 ..... Converts between CDC 60-bit and Cray Research 64-bit numbers 449

ibm2cray, IBM2CRAY, CRAY2IBM ..... Converts IBM data to Cray Research format and vice versa 450

ieg2cray, IEG2CRAY, CRAY2IEG ..... Converts IEEE/Generic 32-bit data to Cray Research 64-bit data and vice versa 453

ieu2cray, IEU2CRAY, CRAY2IEU ..... Converts DEC ULTRIX/generic little-endian 32-bit data to Cray Research 64-bit data and vice versa 456

int6064, INT6064 ..... Converts CDC 60-bit integers to Cray Research 64-bit integers 460

int6460, INT6460 ..... Converts Cray Research 64-bit integers to CDC 60-bit integers 461

nve2cray, NVE2CRAY, CRAY2NVE ..... Converts NOS/VE data to Cray Research format and vice versa 462

rbn, RBN, RNB ..... Converts trailing blanks to nulls and vice versa 465

uscctc, USCCTC, USCCTI ..... Converts IBM EBCDIC data to ASCII data and vice versa 466

usdctc, USDCTC ..... Converts IBM 64-bit floating-point numbers to Cray Research 64-bit, single-precision numbers 468
usdcti, USDCTI ..... Converts Cray 64-bit single-precision, floating-point numbers to IBM 64-bit double-precision numbers 470
usictc, USICTC, USICTI ..... Converts between IBM INTEGER*2/INTEGER*4 and Cray Research 64-bit integer numbers 472
usictp, USICTP ..... Converts a Cray Research 64-bit integer to an IBM packed-decimal field 474
uslctc, USLCTC, USLCTI ..... Converts between IBM LOGICAL*1/LOGICAL*4 and Cray Research 64-bit logical values 475
uspctc, USPCTC ..... Converts specified number of bytes of IBM packed-decimal field to 64-bit integer field 477
ussctc, USSCTC ..... Converts IBM 32-bit floating-point numbers to Cray Research 64-bit single-precision numbers 478
usscti, USSCTI ..... Converts Cray Research 64-bit single-precision, floating-point numbers to IBM 32-bit single-precision numbers 480
vax2cray, VAX2CRAY, CRAY2VAX ..... Converts VAX data to Cray Research format and vice versa 482
vxdctc, VXDCTC ..... Converts VAX 64-bit, D-format numbers to Cray Research 64-bit single-precision numbers 485
vxdcti, VXDCTI ..... Converts Cray Research 64-bit single-precision, floating-point (real) numbers to VAX D-format, 64-bit, double-precision, floating-point numbers 486
vxgctc, VXGCTC ..... Converts VAX 64-bit G-format numbers to Cray Research 64-bit single-precision numbers 488
vxgcti, VXGCTI ..... Converts Cray Research 64-bit single-precision, floating-point (real) numbers to VAX G-format, 64-bit, single-precision, floating-point numbers 490
vxictc, VXICTC ..... Converts VAX INTEGER*2 (16 bits) or INTEGER*4 (32 bits) to Cray Research 64-bit integers 492
vxicti, VXICTI ..... Converts Cray 64-bit integers to VAX INTEGER*2 (16 bit) or INTEGER*4 (32 bit) numbers 494
vxlctc, VXLCTC ..... Converts VAX 32-bit logical numbers to Cray Research 64-bit logical numbers 496
vxlcti, VXLCTI ..... Converts Cray Research 64-bit logical numbers to VAX 32-bit logical numbers 498
vxsctc, VXSCTC ..... Converts VAX 32-bit floating-point numbers to Cray Research 64-bit single-precision real numbers 500

vxscti, VXSCTI ..... Converts Cray Research 64-bit single-precision, floating-point (real) numbers to VAX F format, 32-bit, single-precision, floating-point numbers 502
vxzctc, VXZCTC ..... Converts VAX 64-bit complex numbers to Cray Research 128-bit complex numbers 504
vxzcti, VXZCTI ..... Converts Cray Research 128-bit complex numbers to VAX 64-bit complex numbers 506

MPP pages

cry2cri, CRY2CRI, CRI2CRY ..... Converts Fortran data types between Cray floating-point and IEEE floating-point systems 509
ieg2cri, IEG2CRI, CRI2IEG ..... Converts Fortran data types between 32-bit generic IEEE and Cray Research IEEE. 512
ieg2cri_77, IEG2CRI_77, CRI2IEG_77 ..... Converts IEEE 32-bit data to Cray Research IEEE 64-bit data and vice versa 515

Heap routines

intro_heap, INTRO_HEAP ..... Introduction to heap, table, and data segment management 519

Common pages

hpalloc, HPALLOC ..... Allocates a block of memory from the heap 525
hpcheck, HPCHECK ..... Checks the integrity of the heap 527
hpclmove, HPCLMOVE ..... Extends a block or copies the contents of the block into a larger block 528
hpdeallc, HPDEALLC ..... Returns a block of memory to the list of available space (the heap) 530
hpdump, HPDUMP ..... Dumps the address and size of each heap block 531
hpnewlen, HPNEWLEN ..... Changes the size of an allocated heap block 532
hpshrink, HPSHRINK ..... Returns memory from the heap to the operating system 533
ihplen, IHPLEN ..... Returns the length of a heap block 534
ihpstat, IHPSTAT ..... Returns statistics about the heap 535
ihpvalid, IHPVALID ..... Returns validity of block address 536
tmadw, TMADW ..... Adds a word to a table 537
tmamu, TMAMU ..... Reports table management operation statistics 538
tmats, TMATS ..... Allocates table space 539
tminit, TMINIT ..... Initializes table descriptor vector and zeroes table length vector 540
tmmsc, TMMSC ..... Searches the table by using a mask to locate a specific field within an entry using an optional offset 541

tmmve, TMMVE ..... Moves memory (words) 542
tmptc, TMPTC ..... Processes table collisions 543
tmpts, TMPTS ..... Presets table space 544
tmsrc, TMSRC ..... Searches the table by using optional mask to locate specific field within entry and offset 545
tmvsc, TMVSC ..... Searches a vector table for the search argument 546

Programming Aid routines

intro_progaids, INTRO_PROGAIDS ..... Introduction to programming aids 547

Common pages

fsigctl, FSIGCTL ..... Specifies action on receipt of signal 551
getcallerinfo, GETCALLERINFO ..... Returns the name and line number of the calling routine 553
reprieve, REPRIEVE ..... Describes error handling under the UNICOS operating system 555
sigoff, SIGOFF, SIGON ..... Controls receipt of signals 558
tracebk, TRACEBK ..... Prints a traceback 559
trbk, TRBK ..... Lists all subroutines active in the current calling sequence 561

PVP Pages

getnameq, GETNAMEQ ..... Returns the name of calling routine 563
setplimq, SETPLIMQ ..... Initiates a detailed tracing of each call and return .. 564
shutdsav, SHUTDSAV ..... Sets up the calling program to be checkpointed on system shutdown 565
symdebug, SYMDEBUG ..... Produces a symbolic snapshot of a running process 567
symdump, SYMDUMP ..... Produces a symbolic snapshot dump of a running program 570
trbklvl, TRBKLVL ..... Returns information on current level of calling sequence 576
xpfmt, XPFMT ..... Produces a printable image of an exchange package 579

Interface routines

intro_interface, INTRO_INTERFACE ..... Introduction to system interface routines 581

Common pages

abort, ABORT ..... Requests abort with traceback 585

errexit, ERREXIT ..... Requests abort 586
exit, EXIT ..... Exits from a Fortran program 587
getcwd, GETCWD ..... Returns the current working directory 588
gethost, GETHOST ..... Returns name of host mainframe 589
getoargc, GETOARGC ..... Gets command-line arguments 590
getvargc, GETVARGC ..... Gets command-line arguments, allowing blanks or commas as delimiters 591
iargc, IARGC, GETARG ..... Returns number of command-line arguments or the argument itself 592
ishell, ISHELL ..... Executes a UNICOS shell command 594
remark, REMARK, REMARK2 ..... Enters a message in the stderr file 596
samefile, SAMEFILE ..... Checks to see whether two files have the same inode number 597

PVP Pages

clearbt, CLEARBT, SETBT ..... Temporarily disables or enables bidirectional memory transfers 599
clearfi, CLEARFI, SENSEFI, SETFI ..... Modifies floating-point interrupt status 600
getoarg, GETOARG ..... Gets command-line arguments 601
getvarg, GETVARG ..... Gets command-line arguments, allowing blanks or commas as delimiters 602
nlimit, NLIMIT ..... Provides an interface to setting or obtaining resource limit values 603
remarkf, REMARKF ..... Enters a formatted message in the stderr file 609
sensebt, SENSEBT ..... Determines whether bidirectional memory transfer is enabled or disabled 611
uname, UNAME ..... Returns name of current operating system (Fortran interface to uname(2)) 612

Sort/Search routines

intro_sortsearch, INTRO_SORTSEARCH ..... Introduction to sorting and searching routines 615

Common pages

ismax, ISMAX, ISMIN ..... Searches a real vector for the first occurrence of the maximum or minimum value 619

PVP Pages

cluseq, CLUSEQ, CLUSNE ..... Searches a vector for clusters of values equal or not equal to a target 621

clusflt, CLUSFLT, CLUSFLE, CLUSFGT, CLUSFGE ..... Searches a real vector for clusters of values with a specified logical relationship to a real target 623
clusilt, CLUSILT, CLUSILE, CLUSIGT, CLUSIGE ..... Searches an integer vector for clusters of values with a specified logical relationship to an integer target 625
iilz, IILZ, ILLZ, ILSUM ..... Returns number of leading occurrences of an object in a vector 627
inflmax, INFLMAX, INFLMIN ..... Searches for the maximum or minimum value in subfields of a vector element 629
intmax, INTMAX, INTMIN ..... Searches an integer vector for the maximum or minimum value 631
isamax, ISAMAX, ICAMAX, ISAMIN ..... Searches a vector for the first occurrence of the maximum or minimum absolute value 633
isortd, ISORTD ..... Performs a distribution counting sort on the elements of an integer vector 635
isrcheq, ISRCHEQ, ISRCHNE ..... Searches a vector for the first element equal or not equal to a target 641
isrchflt, ISRCHFLT, ISRCHFLE, ISRCHFGT, ISRCHFGE ..... Searches a real vector for the first element with a specified logical relationship to a real target 643
isrchilt, ISRCHILT, ISRCHILE, ISRCHIGT, ISRCHIGE ..... Searches an integer vector for the first element with a specified logical relationship to an integer target 645
isrchmeq, ISRCHMEQ, ISRCHMNE ..... Searches a vector for the first element whose subfield is equal or not equal to a target 647
isrchmlt, ISRCHMLT, ISRCHMLE, ISRCHMGT, ISRCHMGE ..... Searches a vector for the first element whose subfield has a specified logical relationship with a target 649
orders, ORDERS ..... Internal, fixed-length record-sorting routine optimized for Cray Research systems 651
osrchi, OSRCHI, OSRCHF ..... Searches an ordered vector for the first location that contains a target 664
osrchm, OSRCHM ..... Searches an ordered integer vector for the first element whose subfield is equal to an integer target 666
ssortb, SSORTB, ISORTB ..... Performs Batcher's Odd-Even Merge sort on the elements of a real or integer general vector 670
wheneq, WHENEQ, WHENNE ..... Searches a vector for all elements equal or not equal to a target 678

whenflt, WHENFLT, WHENFLE, WHENFGT, WHENFGE ..... Searches a real vector for all elements with a specified logical relationship to a real target 680
whenilt, WHENILT, WHENILE, WHENIGT, WHENIGE ..... Searches an integer vector for all elements that have a specified relationship to an integer target 682
whenmeq, WHENMEQ, WHENMNE ..... Searches a vector for all elements whose subfields are equal or not equal to a target 684
whenmlt, WHENMLT, WHENMLE, WHENMGT, WHENMGE ..... Searches a vector for all elements whose subfields have a specified logical relationship with a target 686

NAME

INTRO_CONVERSION – Introduction to conversion routines

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These Fortran-callable subroutines perform conversion of data residing in Cray Research systems memory. Conversion subprograms are listed under the following types of routines:

- Foreign data conversion
- Numeric conversion
- ASCII conversion
- IEEE conversion
- Other conversion

For more information regarding foreign data conversion, see the *Application Programmer's I/O Guide*, publication SG–2168.

Routines and Supported Machine Types

In the printed version of these routines, the man pages are grouped according to architecture type. The following list briefly describes this grouping:

- Common routines which run on all Cray Research systems:

  CRI2CRAY

- Routines that nrun only on Cray PVP systems:

B2OCT	CDC2CRAY
CHCONV	DSASC
ETA2CRAY	FP6064
IBM2CRAY	IEG2CRAY
IEU2CRAY	INT6064
INT6460	NVE2CRAY
RBN	USCCTC
USDCTC	USDCTI
USICTC	USICTP
USLCTC	USPCTC
USSCTC	USSCTI

```
VAX2CRAY        VXDCTC
VXDCTI          VXGCTC
VXGCTI          VXICTC
VXICTI          VXLCTC
VXLCTI          VXSCTC
VXSCTI          VXZCTC
VXZCTI
```

- Routines that run only on Cray MPP systems:

```
CRY2CRI
IEG2CRI
IEG2CRI_77
```

FOREIGN DATA CONVERSION ROUTINES

The foreign data conversion routines translate data in both directions between Cray Research internal representations and that of other vendors, including IBM, CDC, ETA, IEEE, and DEC (VAX systems). The following tables list these conversion routines by the data type on which they operate. Cray Research recommends the newer routines: IBM2CRAY, CDC2CRAY, VAX2CRAY, NVE2CRAY, IEG2CRAY, ETA2CRAY, CRAY2IBM, CRAY2CDC, CRAY2VAX, CRAY2NVE, CRAY2IEG, and CRAY2ETA. Generally, routines that are inverses of each other (that is, converted from Cray Research data types to IBM and IBM to Cray Research) are listed under a single entry. Routine descriptions follow later in this section, listed alphabetically by entry name.

The following foreign types can be converted using routines described in this section:

Type	Routine	Routine
IBM	IBM2CRAY	CRAY2IBM
CDC	CDC2CRAY	CRAY2CDC
VAX/VMS	VAX2CRAY	CRAY2VAX
NOS/VE	NVE2CRAY	CRAY2NVE
ETA/CYBER 205	ETA2CRAY	CRAY2ETA
IEEE (32-bit)	IEG2CRAY	CRAY2IEG
DEC	IEU2CRAY	WCRAY2IEU
IEEE (Cray MPP systems)	CRI2CRAY	CRAY2CRI

Superseded routines: Foreign to Cray Research

The following table lists superseded, older routines that are supported to maintain continuity. Cray Research recommends using the newer routines named in the preceding list.

Conversion	IBM	CDC	VAX/VMS
Foreign single precision to Cray single precision	USSCTC	FP6064	VXSCTC
Foreign double precision to Cray single precision	USDCTC	---	VXDCTC VXGCTC
Foreign double precision to Cray double precision	---	---	VXBCTC
Foreign integer to Cray integer	USICTC	INT6064	VXICTC
Foreign logical to Cray logical	USLCTC	---	VXLCTC
Foreign character to ASCII	USCCTC	DSASC	---
VAX 64-bit complex to Cray single precision	---	---	VXZCTC
IBM packed-decimal field to Cray integer	USPCTC	---	---

Superseded routines: Cray Research to Foreign

The following routines convert Cray Research types to foreign types. The man pages for these routines appear later in this section under the name of the inverse of each routine (for example, CRAY2IBM is described under entry IBM2CRAY).

Routine	Type
CRAY2IBM	IBM
CRAY2CDC	CDC
CRAY2VAX	VAX/VMS
CRAY2NVE	NOS/VE
CRAY2ETA	ETA/CYBER 205
CRAY2IEG	IEEE (32-bit)
CRAY2IEU	DEC
CRAY2CRI	IEEE (Cray MPP systems)

The following table lists older routines that convert Cray Research types to foreign types. These routines are supported to maintain continuity. Cray Research recommends using the newer routines named in the preceding list rather than these routines.

Conversion	IBM	CDC	VAX/VMS
Cray single precision to foreign single precision	USSCTI	FP6460	VXSCTI
Cray single precision to foreign double precision	USDCTI	---	VXDCTI, VXGCTI
Cray double precision to foreign double precision	---	---	VXBCTI
Cray integer to foreign integer	USICTI	INT6460	VXICTI
Cray logical to foreign logical	USLCTI	---	VXLCTI
ASCII character to foreign character	USCCTI	ASCDC	---
Cray complex to foreign complex	---	---	VXZCTI
Cray integer to foreign packed-decimal field	USICTP	---	---

NUMERIC CONVERSION ROUTINE

The numeric conversion routine, CHCONV, converts decimal ASCII numerals to an integer format.

ASCII CONVERSION FUNCTIONS

The ASCII conversion functions convert binary integers to or from 1-word ASCII strings (not Fortran character variables).

Note: The ASCII conversion functions are not intrinsic to Fortran. Their default type is real, even though their results are generally used as integers.

The ASCII conversion routines use one type integer argument. The DTB and OTB routines can also use a second optional argument as an error code. The resulting error codes (0 if no error; –1 if there are errors) are returned in the second argument for Fortran calls. If no error code argument is included in Fortran calls, the routine aborts if an error is encountered.

The following calls show how the ASCII conversion routines are used. These Fortran calls convert a binary number to decimal ASCII, then convert back from ASCII to binary.

result=BTD(*integer*)

result Decimal ASCII result (right-justified, blank-filled)

integer Integer argument

result=DTB(*arg,errcode*)

result Integer value

arg Decimal ASCII (left-justified, zero-filled)

errcode 0 if conversion successful; –1 if error

The following table lists the ASCII conversion functions.

Purpose	Name	Argument range	Result
Binary to decimal ASCII (right-justified, blank-filled)	BTD	$0 \le x \le 99999999$	One-word ASCII string (right-justified, blank-filled)
Binary to decimal ASCII (left-justified, zero-filled)	BTDL	$0 \le x \le 99999999$	One-word ASCII string (left-justified, zero-filled)
Binary to decimal ASCII (right-justified, zero-filled)	BTDR	$0 \le x \le 99999999$	One-word ASCII string (right-justified, zero-filled)
Binary to octal ASCII (right-justified, blank-filled)	BTO	$0 \le x \le 77777777_8$	One-word ASCII string (right-justified, blank-filled)
Binary to octal ASCII (left-justified, zero-filled)	BTOL	$0 \le x \le 77777777_8$	One-word ASCII string (left-justified, zero-filled)

Purpose	Name	Argument range	Result
Binary to octal ASCII (right-justified, zero-filled)	BTOR	$0 \leq x \leq 77777777_8$	One-word ASCII string (right-justified, zero-filled)
Decimal ASCII to binary	DTB	Decimal ASCII (left-justified, zero-filled)	One word containing decimal equivalent of ASCII string
Octal ASCII to binary	OTB	Octal ASCII (left-justified, zero-filled)	One word containing octal equivalent of ASCII string

The Cray Assembly Language (CAL) entry points are the same as the Fortran entry points for the preceding routines with the percent sign (%) appended. For example, BTDL is called as BTDL% from CAL.

IEEE CONVERSION ROUTINES

The IEEE conversion routines convert Cray single-precision real numbers to or from IEEE single- or double-precision real numbers.

The following list contains the purpose and name of the IEEE conversion routines. See the individual man pages for implementation details.

- IEG2CRAY: Converts IEEE/Generic 32-bit data to Cray data
- CRAY2IEG: Converts Cray data to IEEE/Generic 32-bit data
- IEU2CRAY: Converts DEC ULTRIX/Generic Little Endian 32-bit data to Cray data
- CRAY2IEU: Converts Cray data to DEC ULTRIX Generic Little Endian 32-bit data
- CRI2CRY: Converts Fortran data from Cray MPP systems to Cray PVP systems type
- CRY2CRI: Converts Fortran data from Cray PVP systems to Cray MPP systems
- CRI2CRAY: Converts IEEE/MPP 32-bit data to Cray Research PVP 64-bit data
- CRAY2CRI: Converts Cray Research PVP 64-bit data to IEEE/MPP 32-bit data

OTHER CONVERSION ROUTINES

These routines place the octal ASCII representation of a Cray word into a character area, convert trailing blanks to nulls or trailing nulls to blanks, and translate a string from one code to another, using a translation table.

The following list contains the purpose and name of these conversion routines.

- B2OCT: Places an octal ASCII representation of a Cray word into a character area
- CHCONV: Converts decimal ASCII to integers
- RBN: Converts trailing blanks to nulls
- RNB: Converts trailing nulls to blanks

NAME

CRI2CRAY, CRAY2CRI – Converts IEEE/MPP 64-bit data to Cray Research PVP 64-bit data and vice versa

SYNOPSIS

```
INTEGER CRI2CRAY, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) craych
DIMENSION foreign(*), cray(*)
ierr = CRI2CRAY(type, num, foreign, bitoff, native[, stride[, native]])

INTEGER CRAY2CRI, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) craych
DIMENSION foreign(*), cray(*)
ierr = CRAY2CRI(type, num, foreign, bitoff, native[, stride[, native]])
```

IMPLEMENTATION

All Cray Research systems (deferred on CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

CRI2CRAY converts Fortran data types on Cray MPP systems (using IEEE floating-point representation) to Cray PVP systems Fortran data types.

CRAY2CRI converts Cray PVP systems Fortran data types to Cray MPP systems (using IEEE floating-point representation) Fortran data types.

The following is a list of valid arguments for this routine.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

ierr An integer giving the returned function value, as follows:

- <0 Parameter error; no translation performed.
- =0 Translation complete; no errors.
- >0 Translation complete; return value is the number of values that completely overflowed (CRAY2CRI only).

type An integer giving the data type code, as follows (all float-to-float conversions are rounded):

Code	Description
0	Typeless (no translation)
1	Integer `CRI2CRAY`: 64-bit twos complement to 64-bit twos complement; no translation. `CRAY2CRI`: 64-bit twos complement to 64-bit twos complement; no translation.
2	Real `CRI2CRAY`: 64-bit IEEE floating point to 64-bit, single-precision, Cray Research real numbers. `CRAY2CRI`: 64-bit, single-precision, Cray Research real numbers to 64-bit IEEE floating point.
3	Double Precision `CRI2CRAY`: 64-bit IEEE floating point to 128-bit, double-precision, Cray Research floating point. `CRAY2CRI`: 128-bit, double-precision, Cray Research floating point to 64-bit IEEE floating point.
4	Complex `CRI2CRAY`: 2 x 64-bit, single-precision, IEEE floating point to 2 x 64-bit, single-precision, Cray Research floating point. `CRAY2CRI`: 2 x 64-bit, single-precision, Cray Research floating point to 2 x 64-bit IEEE floating point.
5	Logical `CRI2CRAY`: 64-bit logical to 64-bit Cray Research logical; all nonzero values are converted to Cray Research logical trues and all zero values are converted to Cray Research logical falses. `CRAY2CRI`: 64-bit Cray Research logical to 64-bit logical; all negative Cray Research values are converted to a value of 1, all positive Cray Research values remain unchanged.
6	Character ASCII to ASCII; no translation.
7	Short integer `CRI2CRAY`: 32-bit twos complement to 64-bit twos complement. `CRAY2CRI`: 64-bit twos complement to 32-bit twos complement.

num An integer giving the number of data items to convert.

foreign Variable or array of any noncharacter type or length containing Cray PVP systems or Cray MPP systems data, whichever is not native to the current system. This variable or array either receives the converted data or contains data to be converted.

bitoff An integer giving the bit offset within *foreign* to begin the conversion. Bits are numbers from 0 to 63, beginning at the leftmost bit of *foreign*. Normally, *bitoff* is zero.

native Variable or array of any noncharacter type or length containing Cray PVP systems or Cray MPP systems data, whichever is native to the current system. This variable or array either receives the converted data or contains data to be converted. This may be a strided array. This variable should be of a type corresponding to the *type* argument.

stride An integer variable or constant giving the memory increment for loading or storing data to the native array. For two and four-word items (complex and double-precision), this is a stride of items, not of words. For typeless, stride is always in words.

This parameter is ignored for CHARACTER (*type* = 6). Data in the foreign array is loaded or stored in a continuous bit stream regardless of this parameter. For 2-word items (complex and double precision), this is a stride of items, not of words. The default value is 1.

craych Type character.
`CRI2CRAY`: Optional character variable or array to receive the converted characters if *type* is 6 (character).

`CRAY2CRI`: Optional character variable or array containing the characters to be converted if *type* is 6 (character).

NAME

B2OCT – Places an octal Hollerith representation of a Cray Research numeric value into a specified part of an integer array

SYNOPSIS

```
CALL B2OCT(s, j, k, v, n)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

B2OCT puts an octal Hollerith representation of a Cray Research numeric value into a specified part of an integer array.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

s An integer array to contain the octal Hollerith representation of the integer input value.

j An integer variable, expression, or constant specifying the byte offset within array *s* where the first character of the octal representation is to be placed. A value of 1 indicates that the destination begins with the first (leftmost) byte of the first word of *s*. *j* must be greater than 0.

k An integer variable, expression, or constant containing the number of characters used in the Hollerith representation; *k* must be greater than 0. *k* indicates the size of the total area to be filled, and the area is blank-filled if necessary.

v An integer variable, expression, or constant containing a value to be converted. The low-order *n* bits of word *v* are used to form the Hollerith representation. The value in *v* must be less than or equal to $2^{63}-1$.

n An integer variable, expression, or constant containing the number of low-order bits of *v* to convert to octal Hollerith character representation; *n* must be within the range: $1 \leq n \leq 64$. If insufficient space is available in *s* ($3k<n$), the specified region in *s* is automatically filled with asterisks (*).

B2OCT places the octal Hollerith representation of the low-order *n* bits of a full Cray word into a specified part of *s*. The *k* bytes in array *s*, pointed to by *j*, are first set to Hollerith blanks. The low-order *n* bits of *v* are then converted to octal Hollerith, using leading zeros if necessary. The converted value (*n*/3 bytes, rounded up) is right-justified in the blanked-out destination area of *s*.

The B2OCT routine is an internal library routine that may not be available in a future release. Use the octal (O) edit descriptor with an internal file to create the octal representation of a numeric value. An example of the O edit descriptor with an internal file is:

```
      program testf
      integer ivar
      character*22 cvar(2)
      ivar=123456789
      write(cvar,2) ivar,ivar
    2 format (o22/o22.15)
      print *,'cvar(1)=',cvar(1)
      print *,'cvar(2)=',cvar(2)
      end
```

The octal values in cvar are:

```
cvar(1)=0000000000000726746425
cvar(2)=       000000726746425
```

NAME

CDC2CRAY, CRAY2CDC – Converts CDC data to Cray Research format and vice versa

SYNOPSIS

```
INTEGER  CDC2CRAY
iret=CDC2CRAY(type, num, foreign, bitoff, cray[, strd[, craychar]])
INTEGER  CRAY2CDC
iret=CRAY2CDC(type, num, foreign, bitoff, cray[, strd[, craychar]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

CDC2CRAY converts CDC data to Cray Research format. It accepts as input a bit string starting at the first element of the array or variable *foreign*, at bit *bitoff*, and converts the data according to *type*, placing the converted data in *cray*.

CRAY2CDC converts Cray Research data to CDC format. It accepts as input a bit string starting at the first element of the array or variable *cray*(1) and converts the data according to *type*, placing the converted data in *foreign* at bit *bitoff*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

type Type integer. The variable type code used by the libraries. The following lists these codes and shows the corresponding CDC and Cray Research formats (and the formats' mapping to one another). In the real, double, and complex conversions between CDC and the Cray Research format, all of the bits in the mantissas are retained; neither rounding nor truncation are required.

Code	CDC	Cray Research	CDC2CRAY	CRAY2CDC
1	INTEGER(60-bit)	INTEGER(64-bit)		
2	REAL(60-bit)	REAL(64-bit)	Neither	Neither
3	DOUBLE(120-bit)	DOUBLE(128-bit)	Neither	Neither

Code	CDC	Cray Research	CDC2CRAY	CRAY2CDC
4	COMPLEX(2*60-bit)	COMPLEX(2*64-bit)	Neither	Neither
5	LOGICAL(60-bit)	LOGICAL(64-bit)		
6	CHARACTER (Display code)	CHAR (ASCII)(8-bit)		
7	INTEGER(60-bit)	INTEGER(24)(64-bit)		

num Type integer variable, array, or constant. Number of data items to convert.

foreign CDC2CRAY: Variable or array of any noncharacter type and of any length containing the CDC format data to convert.

CRAY2CDC: Variable or array of any noncharacter type and of any length to receive the converted CDC data.

bitoff Type integer variable, expression, or constant in the range $0 \leq bitoff \leq 63$.

CDC2CRAY: Bit number within *foreign* to begin the conversion

CRAY2CDC: Bit number within *foreign* to place the converted data

Bits are numbered from 0, beginning at leftmost bit of *foreign*. (Bit 0 is the sign bit.)

cray Variable or array of any noncharacter type and of any length containing the Cray Research format data, word-aligned.

strd Type integer variable, expression, or constant.

CDC2CRAY: Memory increment for storing the conversion results in *cray*. If *strd* = 1, the items are placed in contiguous memory locations in *cray*. If *strd* > 1, the items are placed in *cray* in memory locations at intervals specified by *strd*. For example, if *strd* = 3, the input items are stored in *cray* at locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are taken from *foreign* in a continuous bit stream.

CRAY2CDC: Memory increment for loading the Cray Research items to be converted. If *strd* = 1, the items are taken from contiguous memory locations in *cray*. If *strd* > 1, the items are taken from *cray* memory locations at intervals specified by *strd*. For example, if *strd* = 3, the items are taken from *cray* locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are placed in *foreign* in a continuous bit stream.

For double-word items, this is a stride of items, not of words. Default stride for complex input is 1. For data of type character, *strd* must equal 1.

Default value is 1.

This is an optional argument.

craychar Type `character*N`. Variable or array containing the Cray Research format data. Must be used instead of *cray* when the variable type is character. If *craychar* is supplied, *cray* is ignored. This is an optional argument.

RETURN VALUES

The returned function values are as follows:

`=0` All values converted without error.

`<0` Error. An error was detected in the input parameters.

EXAMPLES

Example 1: The following code converts LENGTH CDC REAL(60-bit) numbers in array `ACDC` to Cray REAL(64-bit) numbers and places the results in array `ACRAY`.

```
INTEGER CDC2CRAY
IRET=CDC2CRAY(2,LENGTH,ACDC,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

Example 2: The following code converts the LENGTH CRAY REAL(64-bit) numbers in array ACRAY from the previous example back to CDC REAL(60-bit) numbers and places the results in array ACDC2.

```
INTEGER CRAY2CDC
IRET=CRAY2CDC(2,LENGTH,ACDC2,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

NAME

CHCONV – Converts decimal ASCII numerals to an integer value

SYNOPSIS

CALL CHCONV(*src*, *isb*, *num*, *ir*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CHCONV converts decimal ASCII numerals to integer values.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

src A variable or array of type Hollerith containing ASCII data or blanks.

isb Starting character in the *src* string. Specify an integer variable, expression, or constant. Characters are numbered from 1, beginning at the leftmost character position of *src.*

num Number of ASCII characters to convert. Specify an integer variable, expression, or constant.

ir Integer result.

Blanks in the input field are treated as zeros. A minus sign encountered anywhere in the input field produces a negative result. Input characters other than blank, digits 0 through 9, a minus sign, or more than one minus sign produce a fatal error.

The CHCONV routine is an internal library routine that may not be available in future releases. Use an internal file with a standard Fortran READ statement to convert the ASCII character representation to a numeric value. An example of the READ statement with an internal file is:

```
      program testf
      real avar
      integer ivar
      character*22 cvar(2)
      cvar(1)='123456789.012e0'
      cvar(2)='123456789'
      read(cvar,2) ivar,ivar
    2 format (e22.10/i22)
      print *,'avar=',avar
      print *,'ivar=',ivar
      end
```

The value for avar is 123456789.012 and the value for ivar is 123456789.

NAME

DSASC, ASCDC – Converts CDC display code character to ASCII character and vice versa

SYNOPSIS

```
CALL DSASC(src, sc, dest, num)
CALL ASCDC(src, sc, dest, num)
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

DSASC converts CDC display code characters to ASCII characters.

ASCDC converts ASCII characters to CDC display code characters.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src For DSASC, a variable or array of any noncharacter type and of any length containing CDC display code characters (64-character set), left-justified in a 64-bit Cray Research word. Contains a maximum of 10 display code characters per word. If the input string is packed 6-bit characters, use U6064(3F) to unpack characters 10 per word. For ASCDC, a variable or array of any type or length containing ASCII data.

sc Display code or ASCII character position to begin the conversion. Leftmost position is 1.

dest For DSASC, a variable or array of any noncharacter type and of any length to contain the converted ASCII data. Results are packed 8 characters per word. For ASCDC, a variable or array of any type or length to contain the converted CDC display code characters (64-character set). Results are packed 60 ASCII characters with 4 blank end bits per 64-bit word.

num Number of CDC display code or ASCII characters to convert. Specify an integer variable, expression, or constant.

NAME

ETA2CRAY, CRAY2ETA – Converts ETA/CYBER 205 data to Cray Research format and vice versa

SYNOPSIS

```
INTEGER  ETA2CRAY
iret=ETA2CRAY(type, num, foreign, bitoff, cray[, strd[, craychar]])

INTEGER  CRAY2ETA
iret=CRAY2ETA(type, num, foreign, bitoff, cray[, strd[, craychar]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

ETA2CRAY converts ETA/CYBER 205 data to Cray Research format. It accepts as input a bit string starting at *foreign*(1) at bit *bitoff*, converts the data according to *type*, and places the converted data in *cray*.

CRAY2ETA converts Cray Research data to ETA/CYBER 205 format. It accepts as input a bit string starting at *cray*(1), converts the data according to *type*, and places the converted data in *foreign* at bit *bitoff*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

type Type integer. The variable type code used by the libraries. The following lists these codes and shows the corresponding ETA/CYBER 205 and Cray Research formats (and the formats' mapping to one another).

Code	ETA	Cray Research
1	INTEGER	INTEGER(64-bit)
2	REAL	REAL(64-bit)
3	DOUBLE PRECISION	DOUBLE(128-bit)
4	COMPLEX	COMPLEX(2*64-bit)

Code	ETA	Cray Research
5	LOGICAL	LOGICAL(64-bit)
6	CHARACTER (ASCII)	CHAR (ASCII)(8-bit)
7	Half-precision REAL	REAL(64-bit)

num Type integer variable, array, or constant. Number of data items to convert.

foreign ETA2CRAY: Variable or array of any noncharacter type and of any length containing the ETA/CYBER 205 format data to convert.

CRAY2ETA: Variable or array of any type except CHARACTER and of any length to receive the converted ETA/CYBER 205 data.

bitoff Type integer variable, expression, or constant in the range $0 \leq bitoff \leq 63$.

ETA2CRAY: Bit number within *foreign* to begin the conversion.

CRAY2ETA: Bit number within *foreign* to place the converted data.

Bits are numbered from 0, beginning at leftmost bit of *foreign*. (Bit 0 is the sign bit.)

cray Variable or array of any noncharacter type and of any length containing the Cray Research format data, word-aligned.

strd Type integer variable, expression, or constant.

ETA2CRAY: Memory increment for storing the conversion results in *cray*. If *strd* = 1, the items are placed in contiguous memory locations in *cray*. If *strd* > 1, the items are placed in *cray* in memory locations at intervals specified by *strd*. For example, if *strd* = 3, the input items are stored in *cray* at locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are taken from *foreign* in a continuous bit stream.

CRAY2ETA: Memory increment for loading the Cray Research items to be converted. If *strd* = 1, the items are taken from contiguous memory locations in *cray*. If *strd* > 1, the items are taken from *cray* memory locations at intervals specified by *strd*. For example, if *strd* = 3, the items are taken from *cray* locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are placed in *foreign* in a continuous bit stream.

For double-word items, this is a stride of items, not of words. Default stride for complex input is 1. For data of type character, *strd* must equal 1.

Default value is 1.

This is an optional argument.

craychar Type `character*N`. Variable or array containing the Cray Research format data. Must be used instead of *cray* when the variable type is character. If *craychar* is supplied, *cray* is ignored. This is an optional argument.

RETURN VALUES

The returned function values are as follows:

`=0` All values converted without error.

`<0` Error. An error was detected in the input parameters.

EXAMPLES

Example 1: The following code converts LENGTH ETA/CYBER 205 REAL numbers in array `AETA` to Cray Research REAL(64-bit) numbers and places the results in array `ACRAY`.

```
INTEGER ETA2CRAY
IRET=ETA2CRAY(2,LENGTH,AETA,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

Example 2: The following code converts the LENGTH CRAY REAL(64-bit) numbers in array `ACRAY` from the previous example back to ETA REAL numbers and places the results in array `AETA2`.

```
INTEGER CRAY2ETA
IRET=CRAY2ETA(2,LENGTH,AETA2,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

NAME

FP6064, FP6460 – Converts between CDC 60-bit and Cray Research 64-bit numbers

SYNOPSIS

```
CALL FP6064(fpn, dest, num)
CALL FP6460(fpn, dest, num)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

FP6064 converts CDC 60-bit single-precision numbers to Cray Research 64-bit single-precision numbers.

FP6460 converts Cray Research 64-bit single-precision numbers to CDC 60-bit single-precision numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn For FP6064, a variable or array of any type or length containing CDC 60-bit, single-precision numbers, left-justified in a Cray Research 64-bit word. For FP6460, a variable or array of any length and of type real containing Cray Research single-precision numbers.

dest Variable or array of type real to contain the converted Cray Research 64-bit, single-precision or CDC 60-bit single-precision numbers. (In FP6460, each floating-point number is left-justified in a 64-bit word.)

num Number of CDC or Cray single-precision numbers to convert. Specify an integer variable, expression, or constant.

NAME

IBM2CRAY, CRAY2IBM – Converts IBM data to Cray Research format and vice versa

SYNOPSIS

```
INTEGER  IBM2CRAY
iret=IBM2CRAY(type, num, foreign, bitoff, cray [,strd [,craychar]])

INTEGER  CRAY2IBM
iret=CRAY2IBM(type, num, foreign, bitoff, cray [,strd [,craychar]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

IBM2CRAY converts IBM data to Cray Research format. It accepts as input a bit string starting at the first element of *foreign* at bit *bitoff* and converts the data according to *type*, placing the converted data in *cray*.

CRAY2IBM converts Cray Research data to IBM format. It accepts as input a bit string starting at the first element of *cray* and converts the data according to *type*, placing the converted data in *foreign* at bit *bitoff*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

type Type integer. The variable type code used by the libraries. The following lists these codes and shows the corresponding IBM and Cray Research formats (and the formats' mapping to one another). In the conversion from Cray Research to IBM floating-point format for real, double, and complex data, the mantissa is truncated rather than rounded.

Code	IBM	Cray Research	CRAY2IBM	IBM2CRAY
1	INTEGER*4	INTEGER(64-bit)		
2	REAL*4	REAL(64-bit)	Truncation	Not applicable
3	REAL*8	DOUBLE(128-bit)	Truncation	Not applicable

Code	IBM	Cray Research	CRAY2IBM	IBM2CRAY
4	COMPLEX*4	COMPLEX(2*64-bit)	Truncation	Not applicable
5	LOGICAL*4	LOGICAL(64-bit)		
6	CHARACTER (EBCDIC)	CHAR (ASCII)(8-bit)		
7	INTEGER*2	INTEGER(24)(64-bit)		

num Type integer variable, array, or constant. Number of data items to convert.

foreign IBM2CRAY: Variable or array of any noncharacter type and of any length containing the IBM format data to convert.

CRAY2IBM: Variable or array of any noncharacter type and of any length to receive the converted IBM data.

bitoff Type integer variable, expression, or constant in the range $0 \leq bitoff \leq 63$.

IBM2CRAY: Bit number within *foreign* to begin the conversion.

CRAY2IBM: Bit number within *foreign* to place the converted data.

Bits are numbered from 0, beginning at leftmost bit of *foreign*. (Bit 0 is the sign bit.)

cray Variable or array of any noncharacter type and of any length containing the Cray Research format data, word-aligned.

strd Type integer variable, expression, or constant.

IBM2CRAY: Memory increment for storing the conversion results in *cray*. If *strd* = 1, the items are placed in contiguous memory locations in *cray*. If *strd* > 1, the items are placed in *cray* in memory locations at intervals specified by *strd*. For example, if *strd* = 3, the input items are stored in *cray* at locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are taken from *foreign* in a continuous bit stream.

CRAY2IBM: Memory increment for loading the Cray Research items to be converted. If *strd* = 1, the items are taken from contiguous memory locations in *cray*. If *strd* > 1, the items are taken from *cray* memory locations at intervals specified by *strd*. For example, if *strd* = 3, the items are taken from *cray* locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are placed in *foreign* in a continuous bit stream.

For double-word items, this is a stride of items, not of words. Default stride for complex input is 1. For data of type character, *strd* must equal 1.

Default value is 1.

This is an optional argument.

craychar Type `character*N`. Variable or array containing the Cray Research format data. Must be used instead of *cray* when the variable type is character. If *craychar* is supplied, *cray* is ignored. This is an optional argument.

RETURN VALUES

The returned function values are as follows:

`=0` All values converted without error.

`<0` Error. An error was detected in the input parameters.

EXAMPLES

Example 1: The following code converts LENGTH IBM REAL*4 numbers in array `AIBM` to Cray Research REAL(64-bit) numbers and places the results in array `ACRAY`.

```
INTEGER IBM2CRAY
IRET=IBM2CRAY(2,LENGTH,AIBM,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

Example 2: The following code converts the LENGTH CRAY REAL(64-bit) numbers in array `ACRAY` from the previous example back to IBM REAL*4 numbers and places the results in array `AIBM2`.

```
INTEGER CRAY2IBM
IRET=CRAY2IBM(2,LENGTH,AIBM2,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

NAME

IEG2CRAY, CRAY2IEG – Converts IEEE/Generic 32-bit data to Cray Research 64-bit data and vice versa

SYNOPSIS

```
INTEGER IEG2CRAY, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) craych
DIMENSION foreign(*), cray(*)
ierr = IEG2CRAY(type, num, foreign, bitoff, cray[, stride[, craych]])

INTEGER CRAY2IEG, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) craych
DIMENSION foreign(*), cray(*)
ierr = CRAY2IEG(type, num, foreign, bitoff, cray[, stride[, craych]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

IEG2CRAY converts Fortran data types on a generic 32-bit platform (using IEEE floating-point representation) to Cray Research Fortran data types.

CRAY2IEG converts Cray Research Fortran data types to a generic 32-bit platform (using IEEE floating-point representation) Fortran data types.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ierr An integer giving the returned function value, as follows:

- <0 Parameter error; no translation performed.
- =0 Translation complete; no errors.
- >0 Translation complete; return value is the number of values that completely overflowed (CRAY2IEG only).

type An integer giving the data type code, as follows (all float-to-float conversions are rounded):

Code	Description
0	Typeless (no translation)
1	Integer
	IEG2CRAY: 32-bit twos complement to 64-bit twos complement.
	CRAY2IEG: 64-bit twos complement to 32-bit twos complement.
2	Real
	IEG2CRAY: 32-bit, single-precision, IEEE floating point to 64-bit, single-precision, Cray Research real numbers.
	CRAY2IEG: 64-bit, single-precision, Cray Research real numbers to 32-bit, single-precision, IEEE floating point.
3	Double
	IEG2CRAY: 64-bit, double-precision, IEEE floating point to 128-bit, double-precision, Cray Research floating point.
	CRAY2IEG: 128-bit, double-precision, Cray Research floating point to 64-bit, double-precision, IEEE floating point.
4	Complex
	IEG2CRAY: 2 x 32-bit, single-precision, IEEE floating point to 2 x 64-bit, single-precision, Cray Research floating point.
	CRAY2IEG: 2 x 64-bit, single-precision, Cray Research floating point to 2 x 32-bit, single-precision, IEEE floating point.
5	Logical
	IEG2CRAY: 32-bit generic logical to 64-bit Cray Research logical; all nonzero values are converted to Cray Research logical trues and all zero values are converted to Cray Research logical falses.
	CRAY2IEG: 64-bit Cray Research logical to 32-bit generic logical; all nonzero Cray Research values are converted to a value of 1, all zero Cray Research values remain unchanged.
6	Character
	ASCII to ASCII; no translation.
7	Short integer
	IEG2CRAY: 16-bit twos complement to 32-bit twos complement.

CRAY2IEG: 32-bit twos complement to 16-bit twos complement.

8 Special

IEG2CRAY: 64-bit, double-precision, IEEE floating point to 64-bit, single-precision, Cray Research real numbers.

CRAY2IEG: 64-bit, single-precision, Cray Research real numbers to 64-bit, double-precision, IEEE floating point.

num An integer giving the number of data items to convert.

foreign IEG2CRAY: Variable or array of any noncharacter type containing the data to be converted.

CRAY2IEG: Variable or array of any noncharacter type to receive the converted data.

bitoff An integer giving the bit offset within *foreign* to begin the conversion. Bits are numbers from 0 to 63, beginning at the leftmost bit of *foreign*.

stride Integer.

IEG2CRAY: Optional memory increment for storing the converted values into the *cray* array.

CRAY2IEG: Optional memory increment for loading the Cray Research values to be converted.

For 2-word items (complex and double-precision), this is a stride of items, not of words.

The default value is 1.

craych Type character.

IEG2CRAY: Optional character variable or array to receive the converted characters if *type* is 6 (character).

CRAY2IEG: Optional character variable or array containing the characters to be converted if *type* is 6 (character).

cray IEG2CRAY: Variable or array of any noncharacter type to receive the converted values.

CRAY2IEG: Variable or array of any noncharacter type containing the values to be converted.

This variable should be of a type corresponding to the *type* argument.

NAME

IEU2CRAY, CRAY2IEU – Converts DEC ULTRIX/generic little-endian 32-bit data to Cray Research 64-bit data and vice versa

SYNOPSIS

```
INTEGER IEU2CRAY, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) craych
DIMENSION foreign(*), cray(*)
ierr = IEU2CRAY(type, num, foreign, bitoff, cray[, stride[, craych]])

INTEGER CRAY2IEU, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) craych
DIMENSION foreign(*), cray(*)
ierr = CRAY2IEU(type, num, foreign, bitoff, cray[, stride[, craych]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

IEU2CRAY converts Fortran data types on a generic little-endian 32-bit platform with IEEE floating-point representation to Cray Research Fortran data types.

CRAY2IEU converts Cray Research Fortran data types to a Fortran data type on a generic little-endian 32-bit platform with IEEE floating-point representation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ierr An integer giving the returned function value, as follows:

- <0 Parameter error; no translation performed.
- =0 Translation complete; no errors.
- >0 Translation complete; return value is the number of values that completely overflowed (CRAY2IEU only).

type An integer giving the data type code, as follows (all float-to-float conversions are rounded).

Code	Description
0	Typeless (no translation)
1	Integer `IEU2CRAY`: 32-bit twos complement to 64-bit twos complement. `CRAY2IEU`: 64-bit twos complement to 32-bit twos complement.
2	Real `IEU2CRAY`: 32-bit, single-precision, IEEE floating point to 64-bit, single-precision, Cray Research real numbers. `CRAY2IEU`: 64-bit, single-precision, Cray Research real numbers to 32-bit, single-precision, IEEE floating point.
3	Double `IEU2CRAY`: 64-bit, double-precision, IEEE floating point to 128-bit, double-precision, Cray Research floating point. `CRAY2IEU`: 128-bit, double-precision, Cray Research floating point to 64-bit, double-precision, IEEE floating point.
4	Complex `IEU2CRAY`: 2 x 32-bit, single-precision, IEEE floating point to 2 x 64-bit, single-precision, Cray Research floating point. `CRAY2IEU`: 2 x 64-bit, single-precision, Cray Research floating point to 2 x 32-bit, single-precision, IEEE floating point.
5	Logical `IEU2CRAY`: 32-bit generic logical to 64-bit Cray Research logical; all nonzero values are converted to Cray Research logical trues and all zero values are converted to Cray Research logical falses. `CRAY2IEU`: 64-bit Cray Research logical to 32-bit generic logical; all nonzero Cray Research values are converted to a value of 1, all zero Cray Research values remain unchanged.
6	Character ASCII to ASCII; no translation.
7	Short integer `IEU2CRAY`: 16-bit twos complement to 32-bit twos complement. `CRAY2IEU`: 32-bit twos complement to 16-bit twos complement.

8 Special
IEU2CRAY: 64-bit, double-precision, IEEE floating point to 64-bit, single-precision, Cray Research real numbers.
CRAY2IEU: 64-bit, single-precision, Cray Research real numbers to 64-bit, double-precision, IEEE floating point.

num An integer giving the number of data items to convert.

foreign IEU2CRAY: Variable or array of any noncharacter type containing the data to be converted.
CRAY2IEU: Variable or array of any noncharacter type to receive the converted data.

bitoff An integer giving the bit offset within *foreign* to begin the conversion. Bits are numbers from 0 to 63, beginning at the leftmost bit of *foreign*.

stride An integer.
IEU2CRAY: Optional memory increment for storing the converted values into the *cray* array.
CRAY2IEU: Optional memory increment for loading the Cray Research values to be converted.

For 2-word items (complex and double-precision), this is a stride of items, not of words.

The default value is 1.

cray IEU2CRAY: Variable or array of any noncharacter type to receive the converted values.
CRAY2IEU: Variable or array of any noncharacter type containing the values to be converted.

This variable should be of a type corresponding to the *type* argument.

craych Type character.
IEU2CRAY: Optional character variable or array to receive the converted characters if *type* is 6 (character).
CRAY2IEU: Optional character variable or array containing the characters to be converted if *type* is 6 (character).

NOTES

The `IEU2CRAY` and `CRAY2IEU` functions are little-endian versions of `IEG2CRAY` and `CRAY2IEG`. Most generic IEEE systems use big-endian data representation and require the use of `IEG2CRAY` and `CRAY2IEG`.

NAME

INT6064 – Converts CDC 60-bit integers to Cray Research 64-bit integers

SYNOPSIS

CALL INT6064(*src*, *idest*, *num*)

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

INT6064 converts CDC 60-bit integer numbers to Cray Research 64-bit integer numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src Variable or array of any noncharacter type and any length containing CDC 60-bit integers, left-justified in a Cray Research 64-bit word.

idest Variable or array of type integer to receive the resultant Cray Research integer values. Each such integer is left-justified and zero-filled.

num Number of CDC integers to convert. Specify an integer variable, expression, or constant.

NOTES

INT6460(3F) is the inverse of this routine.

NAME

INT6460 – Converts Cray Research 64-bit integers to CDC 60-bit integers

SYNOPSIS

```
CALL INT6460(in, idest, num)
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

INT6460 converts Cray Research 64-bit integer numbers to CDC 60-bit integer numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

in Variable or array of any length and of type integer containing Cray Research integer numbers.

idest Variable or array of type integer to contain the converted values or CDC integer numbers. Each such integer is left-justified and zero-filled.

num Number of Cray Research integers to convert. Specify an integer variable, expression, or constant.

NOTES

INT6064(3F) is the inverse of this routine.

NAME

NVE2CRAY, CRAY2NVE – Converts NOS/VE data to Cray Research format and vice versa

SYNOPSIS

```
INTEGER  NVE2CRAY
iret=NVE2CRAY(type, num, foreign, bitoff, cray[, strd[, craychar]])

INTEGER  CRAY2NVE
iret=CRAY2NVE(type, num, foreign, bitoff, cray[, strd[, craychar]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

NVE2CRAY converts NOS/VE data to Cray Research format. It accepts as input a bit string starting at *foreign* at bit *bitoff* and converts the data according to *type*, placing the converted data in *cray*.

CRAY2NVE converts Cray Research data to NOS/VE format. It accepts as input a bit string starting at the first element in *cray* and converts the data according to *type*, placing the converted data in *foreign* at bit *bitoff*.

No precision is lost in either conversion.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

iret The returned function values are as follows:

- =0 All values converted without error.
- <0 Error. An error was detected in the input parameters.

type Type integer. The variable type code used by the libraries. The following table lists these codes, shows the corresponding NOS/VE and Cray Research formats (and the formats' mapping to one another).

Code	NOS/VE	Cray Research
1	INTEGER(64-bit)	INTEGER(64-bit)
2	REAL(64-bit)	REAL(64-bit)
3	DOUBLE(128-bit)	DOUBLE(128-bit)
4	COMPLEX(2*64-bit)	COMPLEX(2*64-bit)
5	LOGICAL(64-bit)	LOGICAL(64-bit)
6 (no-op)	CHARACTER (ASCII)	CHAR (ASCII)(8-bit)
7	INTEGER(64-bit)	INTEGER(24)(64-bit)

num Type integer variable, array, or constant. Number of data items to convert.

foreign NVE2CRAY: Variable or array of any noncharacter type and of any length containing the NOS/VE format data to convert.

CRAY2NVE: Variable or array of any noncharacter type and of any length to receive the converted NOS/VE data.

bitoff Type integer variable, expression, or constant in the range $0 \leq bitoff \leq 63$.

NVE2CRAY: Bit number within *foreign* to begin the conversion.

CRAY2NVE: Bit number within *foreign* to place the converted data.

Bits are numbered from 0, beginning at leftmost bit of *foreign*. (Bit 0 is the sign bit.)

cray Variable or array of any noncharacter type or length containing the Cray Research format data, word-aligned.

strd Type integer variable, expression, or constant.

NVE2CRAY: Memory increment for storing the conversion results in *cray*. If *strd* = 1, the items are placed in contiguous memory locations in *cray*. If *strd* > 1, the items are placed in *cray* in memory locations at intervals specified by *strd*. For example, if *strd* = 3, the input items are stored in *cray* at locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are taken from *foreign* in a continuous bit stream.

CRAY2NVE: Memory increment for loading the Cray Research items to be converted. If *strd* = 1, the items are taken from contiguous memory locations in *cray*. If *strd* > 1, the items are taken from *cray* memory locations at intervals specified by *strd*. For example, if *strd* = 3, the items are taken from *cray* locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are placed in *foreign* in a continuous bit stream.

For double-word items, this is a stride of items, not of words. Default stride for complex input is 1. For character data, *strd* must equal 1.

Default value is 1.

This is an optional argument.

craychar Type character*N. Variable or array containing the Cray Research format data. This must be used in place of *cray* when the variable type is character. If *craychar* is supplied, *cray* is ignored. This is an optional argument.

EXAMPLES

Example 1: The following code converts LENGTH NOS/VE REAL(64-bit) numbers in array ANVE to Cray Research REAL(64-bit) numbers and places the results in array ACRAY.

```
INTEGER NVE2CRAY
IRET=NVE2CRAY(2,LENGTH,ANVE,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

Example 2: The following code converts the LENGTH CRAY REAL(64-bit) numbers in array ACRAY from the previous example back to NOS/VE REAL(64-bit) numbers and places the results in array ANVE2.

```
INTEGER CRAY2NVE
IRET=CRAY2NVE(2,LENGTH,ANVE2,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

NAME

RBN, RNB – Converts trailing blanks to nulls and vice versa

SYNOPSIS

```
INTEGER RBN, RNB, blanks, noblanks
noblanks=RBN(blanks)
blanks=RNB(noblanks)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

RBN converts trailing blanks to nulls. RNB converts trailing nulls to blanks. These routines convert 1 word (up to 8 characters) for each invocation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

blanks For RBN, the argument to be converted. For RNB, the argument after conversion.

noblanks For RBN, the argument after conversion. For RNB, the argument to be converted.

NOTES

Fortran programs using RBN or RNB must declare the function to be an integer.

NAME

USCCTC, USCCTI – Converts EBCDIC character data to ASCII character data, and vice versa

SYNOPSIS

```
CALL USCCTC (src, isb, dest, num, npw [,val] )
CALL USCCTI (src, dest, isb, num, npw [,val] )
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

USCCTC converts EBCDIC character data to ASCII character data. USCCTI converts ASCII character data to EBCDIC character data. All non-printing characters are converted to blanks.

The following is a list of valid arguments for this routine.

src Variable or array of any type (except CHARACTER) containing the character data to be converted.

isb An integer variable or constant containing the starting byte offset to begin the data conversion. Bytes are numbered from left to right, with the leftmost byte as byte 1. The *isb* argument applies to the *src* (USCCTC) or *dest* (USCCTI) argument.

dest Variable or array of any type (except CHARACTER) to receive the character data to be converted.

num An integer variable or constant containing the number of characters to be converted.

npw An integer variable or constant containing the number of characters to be placed in each word of *dest* (USCCTC) or obtained from each word of *src* (USCCTI). A positive value for *npw* (1 to 8) indicates left-justification. A negative value (–1 to –8) indicates right-justification. For USCCTC, left-justified output is also blank-filled. Note that *npw* values of 8 and –8 are equivalent.

val An optional integer or logical variable or constant which, if specified and nonzero (.TRUE.), indicates that all lower case input is to be folded to upper case output. The default, if the argument is not specified, is no case folding.

NOTES

The following conditions must be met for any character data to be converted:

$num \geq 0$
$isb > 0$
$0 < |npw| < 9$

If *num* is not an even multiple of 8, the USCCTI routine will place the remaining converted characters in the final word of *dest* while preserving the rest of the original contents of the word.

CAUTIONS

The same variable or array can be specified for *src* (input) and *dest* (output) if and only if *isb* is 1 and *npw* is 8 (or −8). The results of overlapping conversions using any other values for *isb* and *npw* are undefined.

EXAMPLES

The following Fortran code converts 800 characters from EBCDIC to ASCII. The ASCII characters are placed one per word, right-justified.

```
INTEGER EBCDIC(100)
INTEGER ASCII (800)

CALL USCCTC(EBCDIC, 1, ASCII, 800, –1, .FALSE.)
```

NAME

USDCTC – Converts IBM 64-bit floating-point numbers to Cray Research 64-bit, single-precision numbers

SYNOPSIS

CALL USDCTC(*dpn*, *isb*, *dest*, *num*[, *inc*])

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

USDCTC converts IBM 64-bit floating-point numbers to Cray Research 64-bit, single-precision numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dpn Variable or array of any noncharacter type and of any length containing IBM 64-bit floating-point numbers to convert.

isb Byte number to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *fpn* or *dpn.*

dest Variable or array of type real to contain the converted values.

num Number of IBM 64-bit floating-point numbers to convert. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

`USDCTI`(3F) is the inverse of this routine.

The conversion loses (truncates) 5 to 8 bits of precision from the coefficient. No rounding is performed.

NAME

USDCTI – Converts Cray 64-bit single-precision, floating-point numbers to IBM 64-bit double-precision numbers

SYNOPSIS

```
CALL USDCTI(fpn, dest, isb, num, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

USDCTI converts Cray Research 64-bit single-precision, real numbers to IBM 64-bit double-precision, floating-point numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any length and type real, containing Cray Research 64-bit single-precision, real numbers to convert.

dest Variable or array of type real to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of Cray Research real numbers to convert. Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. The value is 0 if all Cray Research values convert to IBM values without overflow; the value is nonzero if one or more Cray Research values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

Neither rounding nor truncation is required for the result. Precision is extended by introducing 8 more bits into the rightmost byte of the fraction from the Cray Research number being converted. Numbers that produce an underflow when converted to IBM format are converted to 64 binary 0's. Numbers that produce an overflow when converted to IBM format are converted to the largest IBM floating-point representation with the sign bit set if negative. An error parameter returns nonzero to indicate that one or more numbers converted produced an overflow. No such indication is given for underflow.

NOTES

USDCTC(3F) is the inverse of this routine.

NAME

USICTC, USICTI – Converts between IBM INTEGER*2/INTEGER*4 and Cray Research 64-bit integer numbers

SYNOPSIS

```
CALL USICTC(in, isb, dest, num, len[, inc])
CALL USICTI(in, dest, isb, num, len, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

USICTC converts IBM INTEGER*2 and INTEGER*4 numbers to Cray Research 64-bit integer numbers.

USICTI converts Cray Research 64-bit integer numbers to IBM INTEGER*2 or INTEGER*4 numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

in Variable or array of any noncharacter type and of any length containing IBM INTEGER*2 or INTEGER*4 numbers or Cray Research 64-bit integers to convert.

isb Byte number at which to begin the conversion or at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *in* (*dest* in USICTI).

dest Variable or array of type integer to contain the converted values.

num Number of IBM numbers or Cray Research integers to convert. Specify an integer variable, expression, or constant.

len Size of the IBM numbers to convert or of IBM result numbers. These values must be 2 or 4. A value of 2 indicates that input or output integers are INTEGER*2 (16-bit). A value of 4 indicates that input or output integers are INTEGER*4 (32-bit). Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest* or for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

ier Overflow indicator of type integer. The value is 0 if all Cray Research values converted to IBM values without overflow. The value is not 0 if one or more Cray Research values overflowed in the conversion.

Numbers that produce an overflow when converted to IBM format are converted to the largest IBM integer representation, with the sign bit set if negative. An error parameter returns nonzero to indicate that one or more of the numbers converted produced an overflow.

NAME

USICTP – Converts a Cray Research 64-bit integer to an IBM packed-decimal field

SYNOPSIS

CALL USICTP(*ian*, *dest*, *isb*, *num*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

USICTP converts a Cray Research 64-bit integer to an IBM packed-decimal field.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ian Cray Research integer to be converted to an IBM packed-decimal field. Specify an integer variable, expression, or constant.

dest Variable or array of any noncharacter type and of any length to contain the packed field generated.

isb Byte number within *dest* specifying the beginning location for storage. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of bytes to be stored. Specify an integer variable, expression, or constant.

If the input value contains more digits than can be stored in *num* bytes, the leftmost digits are not converted.

NOTES

USPCTC(3F) is the inverse of this routine.

NAME

USLCTC, USLCTI – Converts between IBM LOGICAL*1/LOGICAL*4 and Cray Research 64-bit logical values

SYNOPSIS

```
CALL USLCTC(src, isb, dest, num, len[, inc])
CALL USLCTI(src, dest, isb, num, len[, inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

USLCTC converts IBM LOGICAL*1 and LOGICAL*4 values to Cray Research 64-bit logical values.

USLCTI converts Cray Research logical values to IBM LOGICAL*1 or LOGICAL*4 values.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src Variable or array of any noncharacter type (type logical in USLCTI) and any length containing IBM LOGICAL*1, LOGICAL*4, or Cray Research logical values to convert.

isb Byte number to begin the conversion or, in USLCTI, specifying the beginning location for storage. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *src.*

dest Variable or array of any noncharacter type and of any length to contain the converted values.

num Number of IBM or Cray Research logical values to be converted. Specify an integer variable, expression, or constant.

len Size of the IBM logical values to convert or of the logical result value. These values must be 1 or 4. A value of 1 indicates that input or output logical values are LOGICAL*1 (8-bit). A value of 4 indicates that input or output logical values are LOGICAL*4 (32-bit). Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest* or for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

All arguments must be entered in the same order in which they appear in the SYNOPSIS section.

NAME

USPCTC – Converts specified number of bytes of IBM packed-decimal field to 64-bit integer field

SYNOPSIS

```
CALL USPCTC(src, isb, num, ian)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

USPCTC converts a specified number of bytes of IBM packed-decimal field to 64-bit integer field.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src Variable or array of any noncharacter type and of any length containing a valid IBM packed-decimal field.

isb Byte number to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *src.*

num Number of bytes to convert. Specify an integer variable, expression, or constant.

ian Returned integer result.

The input field must be a valid packed-decimal number less than 16 bytes long. Only the rightmost 15 digits are converted.

NOTES

USICTP(3F) is the inverse of this routine.

NAME

USSCTC – Converts IBM 32-bit floating-point numbers to Cray Research 64-bit single-precision numbers

SYNOPSIS

```
CALL USSCTC(fpn, isb, dest, num[, inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

USSCTC converts IBM real numbers to Cray Research real numbers. The result does not require rounding or truncation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any noncharacter type and of any length containing IBM 32-bit floating-point numbers to convert.

isb Byte number to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *fpn*.

dest Variable or array of type real to contain the converted values.

num Number of IBM 32-bit floating-point numbers to convert. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest*. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

`USSCTI`(3F) is the inverse of this routine.

NAME

USSCTI – Converts Cray Research 64-bit single-precision, floating-point numbers to IBM 32-bit single-precision numbers

SYNOPSIS

CALL USSCTI(*fpn*, *dest*, *isb*, *num*, *ier*[, *inc*])

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

USSCTI converts Cray Research 64-bit single-precision, floating-point numbers to IBM 32-bit single-precision, floating-point numbers. Numbers that produce an underflow when converted to IBM format are converted to 32 binary 0's. Numbers that produce an overflow when converted to IBM format are converted to the largest IBM floating-point representation, with the sign bit set if negative.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any length and type real, containing Cray Research 64-bit single-precision, floating-point numbers to convert.

dest Variable or array of type real to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of Cray Research floating-point numbers to convert. Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. Value is 0 if all Cray Research values convert to IBM values without overflow. Value is nonzero if one or more Cray Research values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

An error argument returns nonzero to indicate that one or more numbers converted produced an overflow. No such indication is given for underflow.

NOTES

USSCTC(3F) is the inverse of this routine. This routine truncates, rather than rounds, the result.

NAME

VAX2CRAY, CRAY2VAX – Converts VAX data to Cray Research format and vice versa

SYNOPSIS

```
INTEGER VAX2CRAY
iret=VAX2CRAY(type, num, foreign, bitoff, cray[, strd[, craychar]])
INTEGER CRAY2VAX
iret=CRAY2VAX(type, num, foreign, bitoff, cray[, strd[, craychar]])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

VAX2CRAY converts VAX data to Cray Research format. It accepts as input a bit string starting at *foreign* at bit *bitoff* and converts the data according to *type*, placing the converted data in *cray*.

CRAY2VAX converts Cray Research data to VAX format. It accepts as input a bit string starting at *cray* and converts the data according to *type*, placing the converted data in *foreign* at bit *bitoff*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

iret The returned function values are as follows:

=0 All values converted without error.

<0 Error. An error was detected in the input parameters.

type Type integer. The variable type code used by the libraries. The following table lists these codes, shows the corresponding VAX and Cray Research formats (and the formats' mapping to one another), and lists the rounding versus truncation strategy of the result.

Code	VAX	Cray Research	VAX2CRAY	CRAY2VAX
1	INTEGER*4	INTEGER(64-bit)		
2	REAL*4	REAL(64-bit)	Neither	Rounding
3	REAL*8	DOUBLE(128-bit)	Neither	Rounding
4	COMPLEX*4	COMPLEX(2*64-bit)	Neither	Rounding
5	LOGICAL*4	LOGICAL(64-bit)		
6 (no-op)	CHARACTER (ASCII)	CHAR (ASCII)(8-bit)		
7	INTEGER*2	INTEGER(24)(64-bit)		

num Type integer variable, array, or constant. Number of data items to convert.

foreign VAX2CRAY: Variable or array of any noncharacter type and of any length containing the VAX format data to convert.

CRAY2VAX: Variable or array of any noncharacter type and of any length to receive the converted VAX data.

bitoff Type integer variable, expression, or constant in the range $0 \leq bitoff \leq 63$.

VAX2CRAY: Bit number within *foreign* to begin the conversion.
CRAY2VAX: Bit number within *foreign* to place the converted data.

Bits are numbered from 0, beginning at leftmost bit of *foreign*. (Bit 0 is the sign bit.)

cray Variable or array of any noncharacter type and of any length containing the Cray Research format data, word-aligned.

strd Type integer variable, expression, or constant.

VAX2CRAY: Memory increment for storing the conversion results in *cray*. If *strd* = 1, the items are placed in contiguous memory locations in *cray*. If *strd* > 1, the items are placed in *cray* in memory locations at intervals specified by *strd*. For example, if *strd* = 3, the input items are stored in *cray* at locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are taken from *foreign* in a continuous bit stream.

CRAY2VAX: Memory increment for loading the Cray Research items to be converted. If *strd* = 1, the items are taken from contiguous memory locations in *cray*. If *strd* > 1, the items are taken from *cray* memory locations at intervals specified by *strd*. For example, if *strd* = 3, the items are taken from *cray* locations *cray*(1), *cray*(4), *cray*(7), *cray*(10), and so on. Regardless of this argument, the input bits are placed in *foreign* in a continuous bit stream.

For double-word items, this is a stride of items, not of words. Default stride for complex input is 1. For character data, *strd* must equal 1.

Default value is 1.

This is an optional argument.

craychar Type character*N. Variable or array containing the Cray Research format data. Must be used instead of *cray* when the variable type is character. If *craychar* is supplied, *cray* is ignored. This is an optional argument.

EXAMPLES

Example 1: The following code converts LENGTH VAX REAL*4 numbers in array AVAX to Cray Research REAL(64-bit) numbers and places the results in array ACRAY.

```
INTEGER VAX2CRAY
IRET=VAX2CRAY(2,LENGTH,AVAX,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

Example 2: The following code converts the LENGTH CRAY REAL(64-bit) numbers in array ACRAY from the previous example back to VAX REAL*4 numbers and places the results in array AVAX2.

```
INTEGER CRAY2VAX
IRET=CRAY2VAX(2,LENGTH,AVAX2,0,ACRAY)
IF(IRET.LT.0) GOTO 99     ! error
```

NAME

VXDCTC – Converts VAX 64-bit, D-format numbers to Cray Research 64-bit single-precision numbers

SYNOPSIS

```
CALL VXDCTC(dpn, isb, dest, num[, inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dpn Variable or array of any noncharacter type and of any length containing VAX D-format numbers to convert.

isb Byte number within *dpn* at which to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte of *dpn.*

dest Variable or array of type real to contain the converted values.

num Number of VAX D-format numbers to convert. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

VXDCTI(3F) is the inverse of this routine. This routine truncates, rather than rounds, the result.

NAME

VXDCTI – Converts Cray Research 64-bit single-precision, floating-point (real) numbers to VAX D-format, 64-bit, double-precision, floating-point numbers

SYNOPSIS

```
CALL VXDCTI(fpn, dest, isb, num, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any length and type real containing Cray Research 64-bit single-precision, real numbers to convert.

dest Variable or array of type real to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of Cray Research real numbers to convert. Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. The value is 0 if all Cray Research values convert to VAX values without overflow. The value is nonzero if one or more Cray Research values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant.

Numbers that produce an underflow when converted to VAX format are converted to 64 binary 0’s. Numbers that are in overflow on the Cray Research system are converted to a "reserved" floating-point representation, with the sign bit set if negative. Numbers that are valid on the Cray Research system, but overflow on the VAX system, are converted to the most positive possible number or most negative possible number, depending on the sign.

An error argument returns nonzero to indicate that one or more numbers converted produced an overflow. (Deferred implementation. At present, you must supply the argument, which is always returned as 0.) No such indication is given for underflow.

NOTES

VXDCTC(3F) is the inverse of this routine. This routine rounds, rather than truncates, the result.

NAME

VXGCTC – Converts VAX 64-bit G-format numbers to Cray Research 64-bit single-precision numbers

SYNOPSIS

```
CALL VXGCTC(dpn, isb, dest, num[, inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

VXGCTC converts VAX 64-bit G-format numbers to Cray Research 64-bit single-precision numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dpn Variable or array of any noncharacter type and of any length containing VAX G-format numbers to convert.

isb Byte number within *dpn* at which to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte of *dpn*.

dest Variable or array of type real to contain the converted values.

num Number of VAX G-format numbers to convert. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

VXGCTI(3F) is the inverse of this routine. This routine truncates, rather than rounds, the result.

NAME

VXGCTI – Converts Cray Research 64-bit single-precision, floating-point (real) numbers to VAX G-format, 64-bit, single-precision, floating-point numbers

SYNOPSIS

```
CALL VXGCTI(fpn, dest, isb, num, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

VXGCTI converts Cray Research 64-bit single-precision real numbers to VAX G-format single-precision, floating-point numbers. The result fits entirely in the target data structure, so neither rounding nor truncation is required.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any length and type real, containing Cray Research 64-bit single-precision, floating-point numbers to convert.

dest Variable or array of type real to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of Cray Research real numbers to convert. Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. The value is 0 if all Cray Research values convert to VAX values without overflow. The value is nonzero if one or more Cray Research values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

Numbers that produce an underflow when converted to VAX format are converted to 64 binary 0's. Numbers that are in overflow on the Cray Research system are converted to a "reserved" floating-point representation, with the sign bit set if negative. Numbers that are valid on the Cray Research system, but overflow on the VAX system, are converted to the most positive possible number or the most negative possible number, depending on the sign.

An error argument returns nonzero to indicate that one or more numbers converted produced an overflow. (Deferred implementation. At present, you must supply the parameter, which is always returned as 0.) No such indication is given for underflow.

NOTES

VXGCTC(3F) is the inverse of this routine.

NAME

VXICTC – Converts VAX INTEGER*2 (16 bits) or INTEGER*4 (32 bits) to Cray Research 64-bit integers

SYNOPSIS

CALL VXICTC(*in*, *isb*, *dest*, *num*, *len*[, *inc*])

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

VXICTC converts VAX INTEGER*2 (16 bits) or INTEGER*4 (32 bits) to Cray Research 64-bit integers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

in Variable or array of any noncharacter type and of any length containing VAX 16-bit or 32-bit integers.

isb Byte number at which to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *in.*

dest Variable or array of type integer to contain the converted values.

num Number of VAX integers to convert. Specify an integer variable, expression, or constant.

len Size of the VAX numbers to convert. This value must be 2 or 4. A value of 2 indicates that input integers are 16-bit integers. A value of 4 indicates that input integers are 32-bit integers. Specify an integer variable, expression, or constant.

inc Memory increment for storing conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

Instead of using the VXICTC routine, Cray Research, Inc. recommends using the newer VAX2CRAY routine. Select a *type* of 7 for an INTEGER*2 conversion and 1 for an INTEGER*4 conversion.

VXICTI(3F) is the inverse of this routine.

SEE ALSO

VAX2CRAY(3F)

NAME

VXICTI – Converts Cray 64-bit integers to VAX INTEGER*2 (16 bit) or INTEGER*4 (32 bit) numbers

SYNOPSIS

```
CALL VXICTI(in, dest, isb, num, len, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

VXICTI converts Cray 64-bit integers to VAX INTEGER*2 (16-bit) or INTEGER*4 (32-bit) numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

in Variable or array of any length and type integer, containing Cray integers to convert.

dest Variable or array of type integer to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest*.

num Number of Cray integers to convert. Specify an integer variable, expression, or constant.

len Size of the VAX result numbers. This value must be 2 or 4. A value of 2 indicates that output integers are INTEGER*2 (16-bit). A value of 4 indicates that output integers are INTEGER*4 (32-bit). Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. The value is 0 if all Cray values are converted to VAX values without overflow. The value is nonzero if one or more Cray values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

Numbers that produce an overflow when converted to VAX format are converted to the largest VAX integer representation, with the sign bit set if negative.

An error argument returns nonzero to indicate that one or more numbers converted produced an overflow. (Deferred implementation. At present, you must supply the parameter, which is always returned as 0.) No such indication is given for underflow.

NOTES

Instead of using the VXICTI routine, Cray Research, Inc. recommends using the newer CRAY2VAX routine, which is documented on the VAX2CRAY man page. Select a *type* of 7 for an INTEGER*2 conversion and 1 for an INTEGER*4 conversion.

VXICTC(3F) is the inverse of this routine.

SEE ALSO

VAX2CRAY(3F)

NAME

VXLCTC – Converts VAX 32-bit logical numbers to Cray Research 64-bit logical numbers

SYNOPSIS

CALL VXLCTC(*src*, *isb*, *dest*, *num*, *len*[, *inc*])

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

VXLCTC converts VAX 32-bit logical values to Cray Research 64-bit logical values.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

src Variable or array of any noncharacter type and of any length containing VAX logical values to convert.

isb Byte number at which to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *src.*

dest Variable or array of type logical to contain the converted values.

num Number of VAX logical values to be converted. Specify an integer variable, expression, or constant.

len Size of the VAX logical values to convert. At present, this argument must be set to 4, indicating that 32-bit logical values are to be converted. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

Instead of using the VXLCTC routine, Cray Research, Inc. recommends using the newer VAX2CRAY routine. Select a *type* of 5. VXLCTI(3F) is the inverse of this routine.

SEE ALSO

VAX2CRAY(3F)

NAME

VXLCTI – Converts Cray Research 64-bit logical numbers to VAX 32-bit logical numbers

SYNOPSIS

```
CALL VXLCTI(in, dest, isb, num, len, ier[,inc])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

VXLCTI converts Cray Research 64-bit logical numbers to VAX 32-bit logical numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

in Variable or array of any length and type logical, containing Cray Research logical numbers to convert.

dest Variable or array of type integer to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest*.

num Number of Cray Research integers to convert. Specify an integer variable, expression, or constant.

len Size of the VAX result. You must specify this as 4 (32-bit logical numbers).

ier A 0 is always returned (stored into) for this parameter. The parameter is required for consistency with similar subroutines.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

Instead of using the VXLCTI routine, Cray Research, Inc. recommends using the newer CRAY2VAX routine, which is documented on the VAX2CRAY man page. Select a *type* of 5.

VXLCTC(3F) is the inverse of this routine.

SEE ALSO

VAX2CRAY(3F)

NAME

VXSCTC – Converts VAX 32-bit floating-point numbers to Cray Research 64-bit single-precision real numbers

SYNOPSIS

```
CALL VXSCTC(fpn, isb, dest, num [,inc])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

VXSCTC converts VAX 32-bit floating-point numbers to Cray Research 64-bit single-precision real numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any noncharacter type containing VAX 32-bit floating-point numbers to convert.

isb Byte number at which to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *fpn.*

dest Variable or array of type real to contain the converted values.

num Number of VAX floating-point numbers to convert. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

NOTES

Instead of using the VXSCTC routine, Cray Research, Inc. recommends using the newer VAX2CRAY routine. Select a *type* of 2. VXSCTI(3F) is the inverse of this routine.

The resulting data structure is large enough to contain the whole number, so neither rounding nor truncation is required.

SEE ALSO

VAX2CRAY(3F)

NAME

VXSCTI – Converts Cray Research 64-bit single-precision, floating-point (real) numbers to VAX F format, 32-bit, single-precision, floating-point numbers

SYNOPSIS

```
CALL VXSCTI(fpn, dest, isb, num, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

VXSCTI converts Cray Research 64-bit single-precision, floating point (real) to VAX F format, 32-bit, single-precision, floating point.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any length and type real, containing Cray Research 64-bit real numbers to convert.

dest Variable or array of type real to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of Cray Research real numbers to convert. Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. The value is 0 if all Cray Research values convert to VAX values without overflow. The value is nonzero if one or more Cray Research values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

Numbers that produce an underflow when converted to VAX format are converted to 32 binary zeros. Numbers that are in overflow on the Cray Research system are converted to a "reserved" floating-point representation, with the sign bit set if negative. Numbers that are valid on the Cray Research system, but overflow on the VAX system, are converted to the most positive possible number or the most negative possible number, depending on the sign.

An error argument returns nonzero to indicate that one or more numbers converted produced an overflow (Deferred implementation. At present you must supply the argument, which is always returned as 0.) No such indication is given for underflow.

NOTES

Instead of using the VXSCTI routine, Cray Research, Inc. recommends using the newer CRAY2VAX routine, which is documented on the VAX2CRAY man page. Select a *type* of 2.

VXSCTC(3F) is the inverse of this routine.

This routine rounds, rather than truncates, the result.

SEE ALSO

VAX2CRAY(3F)

NAME

VXZCTC – Converts VAX 64-bit complex numbers to Cray Research 128-bit complex numbers

SYNOPSIS

CALL VXZCTC(*dpn*, *isb*, *dest*, *num*[, *inc*])

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

VXZCTC converts VAX 64-bit complex numbers to Cray Research 128-bit complex numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

dpn Variable or array of any noncharacter type and of any length containing complex numbers to convert.

isb Byte number within *dpn* at which to begin the conversion. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte of *dpn.*

dest Variable or array of type complex to contain the converted values.

num Number of complex numbers to convert. Specify an integer variable, expression, or constant.

inc Memory increment for storing the conversion results in *dest.* This is an optional argument specified as an integer variable, expression, or constant. Default value is 1.

NOTES

Instead of using the VXZCTC routine, Cray Research, Inc. recommends using the newer VAX2CRAY routine. Select a *type* of 4. VXZCTI(3F) is the inverse of this routine.

This routine truncates rather than rounds the result.

SEE ALSO

VAX2CRAY(3F)

NAME

VXZCTI – Converts Cray Research 128-bit complex numbers to VAX 64-bit complex numbers

SYNOPSIS

```
CALL VXZCTI(fpn, dest, isb, num, ier[, inc])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

VXZCTI converts Cray Research 128-bit complex numbers to VAX 64-bit complex numbers.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

fpn Variable or array of any length and of type complex, containing Cray Research complex numbers to convert.

dest Variable or array of any noncharacter type to contain the converted values.

isb Byte number at which to begin storing the converted results. Specify an integer variable, expression, or constant. Bytes are numbered from 1, beginning at the leftmost byte position of *dest.*

num Number of Cray Research complex numbers to convert. Specify an integer variable, expression, or constant.

ier Overflow indicator of type integer. The value is 0 if all Cray Research values convert to VAX values without overflow. The value is nonzero if one or more Cray Research values overflowed in the conversion.

inc Memory increment for fetching the number to be converted. This is an optional argument specified as an integer variable, expression, or constant. The default value is 1.

Numbers that produce an underflow when converted to VAX format are converted to two words of 32 binary zeros. Numbers that are in overflow on the Cray Research system are converted to a "reserved" floating-point representation, with the sign bit set if negative. Numbers that are valid on the Cray Research system but overflow on the VAX system are converted to the most positive possible number or the most negative possible number, depending on the sign.

An error argument returns nonzero to indicate that one or more numbers converted produced an overflow. (Deferred implementation. At present, you must supply the argument, which is always returned as 0.) No such indication is given for underflow.

NOTES

Instead of using the VXZCTI routine, Cray Research, Inc. recommends using the newer CRAY2VAX routine, which is documented on the VAX2CRAY(3F) man page. Select a *type* of 4.

VXZCTC(3F) is the inverse of this routine.

This routine rounds rather than truncates the result.

SEE ALSO

VAX2CRAY(3F)

NAME

CRY2CRI, CRI2CRY – Converts Fortran data types between Cray floating-point and IEEE floating-point systems

SYNOPSIS

```
INTEGER  CRY2CRI, CRI2CRY
ierr  =  CRI2CRY(type, num, foreign, bitoff, native, stride, natlen,
forlen[,nativech])
```

```
ierr  =  CRY2CRI(type, num, foreign, bitoff, native, stride, natlen,
forlen[,nativech])
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

CRY2CRI converts Cray Fortran data types for a system using Cray Research floating-point representation (abbreviated as "Cray" in the following text) to data for systems that use IEEE floating-point representation (abbreviated as "IEEE" in the following text).

CRI2CRY converts IEEE Fortran data types to Cray Fortran data types.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments, or double precision, which is KIND=16.

The following is a list of valid parameters for this routine.

type An integer giving the data type code, as follows.

Code	Description
1	Typeless (no translation; *natlen* and *forlen* must be equal and must be 64-, 128-, or 256-bits.)
2	Integer Cray: 64-bit twos complement IEEE: 32- or 64-bit twos complement

3 Floating-point
Cray: 64- or 128-bit Cray floating-point
IEEE: 32-, 64-bit (available on all systems), or 128-bit IEEE floating-point (CRAY T90 only)

4 Complex
IEEE: 2 x 32, 64-bit (all systems), or 128-bit floating-point (CRAY T90 only)
Cray: 2 x 64 or 128-bit floating-point

5 Logical
IEEE: 32- or 64-bit nonzero/zero logical
Cray: 64-bit or minus/positive logical

6 Character
ASCII to ASCII; no translation.

The *natlen* and *forlen* parameters select the size of the data.

num Number of data items to convert. Type integer variable, expression, or constant.

foreign Variable or array of any noncharacter type or length containing Cray PVP systems or Cray MPP systems data, whichever is not native to the current system. This variable or array either receives the converted data or contain data to be converted.

bitoff Integer giving the bit offset within the foreign data variable or array to begin the conversion. *bitoff* must be at least 0 and no more than 63.

native Variable or array of any noncharacter type or length containing Cray PVP systems or Cray MPP systems data, whichever is native to the current system. This variable or array either receives the converted data or contains data to be converted. This may be a strided array.

stride Integer variable or constant giving the memory increment for loading or storing data to the native array. For two and four-word items (complex and double-precision), this is a stride of items, not of words. For typeless, stride is always in words.

This parameter is ignored for CHARACTER (*type* = 6).
Data in the foreign array is loaded or stored in a continuous bit stream regardless of this parameter.

natlen Native storage length of an item, in bits. For COMPLEX data, *natlen* counts the total size of the real and imaginary component.

forlen Fortran storage length of an item, in bits. For COMPLEX data, *forlen* counts the total size of the real and imaginary components.

nativech Optional character parameter specifying *native* target variable if it is of type CHARACTER (*type* = 6). This parameter is ignored if type is not CHARACTER.

NOTES

`CRI2CRY` and `IEC2CRAY` are comparable functions.

`CRY2CRI` and `CRAY2IEC` are comparable functions.

These data conversion routines also correctly translate CRAY-2 data files with the exception of logical data.

RETURN VALUES

The returned function values are as follows:

`-1` Parameter error; too many arguments or *nativech* not specified with *type* = 6.

`-2` Parameter error; invalid *type*.

`-3` Parameter error; invalid *num*.

`-4` Parameter error; invalid *bitoff*.

`-5` Parameter error; invalid *intlen*.

`-6` Parameter error; invalid *extlen*.

`-7` Unable to `malloc`() memory for translation.

`0` Translation complete; no errors.

`>0` Translation complete; return value is the number of integer or real values that completely overflowed during conversion. Overflows and NaNs which exist before conversion, are not included.

NAME

IEG2CRI, CRI2IEG – Converts Fortran data types between 32-bit generic IEEE and Cray Research IEEE.

SYNOPSIS

```
INTEGER  IEG2CRI, CRI2IEG
ierr  =  CRI2IEG(type, num, foreign, bitoff, native, stride, intlen,
extlen[,nativech])
```

```
ierr  =  IEG2CRI(type, num, foreign, bitoff, native, stride, intlen,
extlen[,nativech])
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

IEG2CRI converts Fortran data types for Cray MPP systems using IEEE floating-point representation from equivalent generic IEEE 32-bit floating-point representation.

CRI2IEG converts Fortran data types for Cray MPP systems using IEEE into equivalent generic IEEE 32-bit floating-point representation. The data consists of floating-point, integer, and logical representations.

A *denormal* is a signed 32-bit IEEE value with magnitude between 1.1^{-38} and 0, a 64-bit IEEE value between 2.2^{-308} and 0, or a 128-bit IEEE value between 3.3^{-4932} and 0. IEG2CRI converts input denormals to normalized values of larger precision, or zero if the result precision is unchanged. CRI2IEG can create denormals.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid parameters for this routine.

type An integer giving the data type code, as follows.

Code	Description
1	Typeless (no translation) 64-bit

2 Integer
foreign: 16-bit, 32-bit or 64-bit twos complement
native: 32-bit or 64-bit twos complement

3 Real
foreign: 32-bit or 64-bit IEEE floating-point
native: 32-bit, 64-bit or 128-bit Cray Research IEEE floating-point (128-bit deferred)

4 Complex
foreign: 2 x 32-bit or 64-bit IEEE floating-point
native: 2 x 32-bit, 64-bit or 128-bit Cray Research IEEE floating-point (128-bit deferred)

5 Logical
foreign: 32-bit or 64-bit nonzero/zero logical
native: 32-bit or 64-bit one/zero logical

6 Character
ASCII to ASCII; no translation.

The *intlen* and *extlen* parameters select the size of the data.

num The number of data items to convert. Type integer variable, expression, or constant.

foreign Variable or array of any noncharacter type or length containing IEEE data. This variable or array either receives the converted data or contains data to be converted.

bitoff An integer giving the bit offset within the foreign data variable or array to begin the conversion. *bitoff* must be at least 0 and no more than 63.

native Variable or array of any noncharacter type or length containing Cray Research data. This variable or array either receives the converted data or contains data to be converted. This may be a strided array.

stride An integer variable or constant giving the memory increment for loading or storing data to the native array. For two and four-word items (complex and double-precision), this is a *stride* of items, not of words. For typeless, *stride* is always in 64-bit words.

This parameter is ignored for *type* = 6 (character).

Data in the foreign array is loaded or stored in a continuous bit stream regardless of this parameter.

intlen Internal (*native*) storage length of an item, in bits. For *type* = 4 (complex) data, *intlen* counts the total size of the real and imaginary components.

extlen External (*foreign*) storage length of an item, in bits. For *type* = 4 (complex) data, *extlen* counts the total size of the real and imaginary components.

nativech Optional character parameter specifying *native* target variable if it is of *type* = 6 (character). This parameter is ignored if *type* is not character.

NOTES

Conversions from *native* to a larger *foreign* are not provided. Conversions from *foreign* to a smaller *native* are not provided. The internal length must be at least the external length. Floating-point conversions are rounded when necessary.

NaNs that are converted to another size have their rightmost bits UNDEFINED.

RETURN VALUES

ierr is a returned function value, which can be as follows:

`-1` Parameter error; too many arguments or *nativech* not specified with *type* = 6

`-2` Parameter error; invalid *type*

`-3` Parameter error; invalid *num*

`-4` Parameter error; invalid *bitoff*

`-5` Parameter error; invalid *intlen*

`-6` Parameter error; invalid *extlen*

`-7` Unable to `malloci`() memory for translation

`-8` Parameter error; invalid combination of *extlen* and *extlen*

`0` Translation complete; no errors

`>0` Translation complete; return value is the number of integer or real values that completely overflowed during conversion. Infinities and NaNs which exist before conversion are not included.

NAME

IEG2CRI_77, CRI2IEG_77 – Converts IEEE 32-bit data to Cray Research IEEE 64-bit data and vice versa

SYNOPSIS

```
INTEGER CRI2IEG_77, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) nativech
DIMENSION foreign(*), native(*)
ierr = IEG2CRI_77(type, num, foreign, bitoff, native[, stride[, nativech]])

INTEGER CRI2IEG_77, ierr, type, num, foreign, bitoff, stride
CHARACTER * (*) nativech
DIMENSION foreign(*), native(*)
ierr = CRI2IEG_77(type, num, foreign, bitoff, native[, stride[, nativech]])
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

IEG2CRI_77 converts from 32-bit generic IEEE floating-point representation to Cray Research IEEE 64-bit Fortran data types.

CRI2IEG_77 converts from Cray Research IEEE 64-bit Fortran data types to 32-bit generic IEEE floating-point representation.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ierr An integer giving the returned function value, as follows:

- <0 Parameter error; no translation performed. See IEG2CRI(3F).
- =0 Translation complete; no errors.
- >0 Translation complete; return value is the number of values that overflowed during conversion(CRI2IEG_77 only). Infinities existing before conversion (and NaNs) are not counted.

type An integer giving the data type code, as follows (all float-to-float conversions are rounded):

Code	Description
0	Typeless (no translation): 64-bit, 128-bit, and 256-bit data items.
1	Integer `IEG2CRI_77`: 32-bit twos complement to 64-bit twos complement; sign extension. `CRI2IEG_77`: 64-bit twos complement to 32-bit twos complement. Overflow results are undefined.
2	Real `IEG2CRI_77`: 32-bit IEEE floating point to 64-bit IEEE floating point. 32-bit IEEE denormal values are converted. `CRI2IEG_77`: 64-bit, IEEE floating point to 32-bit IEEE floating point. Overflows result in signed infinity. 32-bit IEEE denormal values can result. Underflow results in signed zero.
3	Double Precision `IEG2CRI_77`: 64-bit IEEE floating point to 128-bit IEEE floating point (deferred). `CRI2IEG_77`: 128-bit IEEE floating point to 64-bit IEEE floating point (deferred).
4	Complex `IEG2CRI_77`: 2 x 32-bit, IEEE floating point to 2 x 64-bit, IEEE floating point. `CRI2IEG_77`: 2 x 64-bit, IEEE floating point to 2 x 32-bit, IEEE floating point. Note that two overflows can result from each item.
5	Logical `IEG2CRI_77`: 32-bit logical to 64-bit Cray Research logical; all nonzero values are converted to Cray Research logical trues and all zero values are converted to Cray Research logical falses. `CRI2IEG_77`: 64-bit Cray Research logical to 32-bit logical; all nonzero values are converted to 1 and zero values are converted to zero.

6 Character

ASCII to ASCII; no translation.

7 Short integer

`IEG2CRI_77`: 16-bit twos complement to 64-bit twos complement.

`CRI2IEG_77`: 64-bit twos complement to 16-bit twos complement. Results are undefined on overflow.

8 Special

`IEG2CRI_77`: 64-bit IEEE floating point to 64-bit Cray Research IEEE floating point. IEEE denormal values become signed zero, other values are unchanged.

`CRI2IEG_77`: 64-bit, Cray Research IEEE floating point to 64-bit IEEE floating point. No conversion.

num An integer giving the number of data items to convert. A Fortran complex number is one item.

foreign `IEG2CRI_77`: Variable or array of any noncharacter type containing the data to be converted.

`CRI2IEG_77`: Variable or array of any noncharacter type to receive the converted data.

bitoff An integer giving the bit offset within *foreign* to begin the conversion. Bits are numbered from 0 to 63, beginning at the leftmost bit of *foreign*.

native `IEG2CRI_77`: Variable or array of any noncharacter type to receive the converted values.

`CRI2IEG_77`: Variable or array of any noncharacter type containing the values to be converted.

This variable should be of a type corresponding to the *type* argument.

stride An integer.

`IEG2CRI_77`: Optional memory increment for storing the converted values into the *native* array.

`CRI2IEG_77`: Optional memory increment for loading the Cray PVP systems values to be converted from the *native* array.

For 2-word items (complex and double-precision), this is a *stride* of items, not of words.

The default value is 1.

nativech Type character.

IEG2CRI_77: Optional character variable or array to receive the converted characters if *type* is 6 (character).

CRI2IEG_77: Optional character variable or array containing the characters to be converted if *type* is 6 (character).

SEE ALSO

IEG2CRI(3F)

NAME

INTRO_HEAP – Introduction to heap, table, and data segment management

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

These routines let you manage a block of memory (the heap) within your program area, manipulate tables, and manage the secondary data segment (SDS). The management routines are divided into three categories: heap management, table management, and SDS management.

All heap management routines run on all Cray Research systems.

HEAP MANAGEMENT ROUTINES

Heap management routines provide dynamic storage allocations by managing a block of memory, called the *heap*, within your job area. Each job has its own heap. The functions of the heap management routines include allocating a block of memory, returning a block of memory to the heap's list of available space, and changing the length of a block of memory. Heap management routines may also move a heap block to a new location if there is no room to extend it, return part of the heap to the operating system, check the integrity of the heap, and report heap statistics. See the *Segment Loader (SEGLDR) and* ld *Reference Manual*, publication SR–0066, for the location of the heap and a description of the parameters that affect the heap.

The heap management routines keep various statistics on the use of the heap. These statistics include values used to tune heap parameters specified by SEGLDR and information used in debugging.

The following routines are used for heap management:

- HPALLOC(3F): allocates a block of memory from the heap
- HPCHECK(3F): checks the integrity of the heap
- HPCLMOVE(3F): extends a block or copies block contents into a larger block
- HPDEALLC(3F): returns a block of memory to the heap
- HPSHRINK(3F): returns memory from the heap to the operating system
- HPDUMP(3F): dumps the address and size of each heap block

- HPNEWLEN(3F): changes the size of an allocated heap block
- IHPLEN(3F): returns the length of a heap block
- IHPSTAT(3F): returns statistics about the heap
- IHPVALID(3F): returns validity of block address

TABLE MANAGEMENT ROUTINES

The following list describes each Fortran-callable table management routine:

- TMADW: adds a word to a table
- TMAMU: reports table management operation statistics
- TMATS: allocates table space
- TMINIT: initializes table
- TMMSC: searches the table with a mask to locate a field within an entry
- TMMVE: moves memory
- TMPTC: processes table collisions
- TMPTS: presets table space
- TMSRC: searches the table with or without a mask to locate a field within an entry and an offset
- TMVS: searches a vector table for the search argument

The Fortran-callable versions of these routines use default BTAB and LTAB definitions from a common area in the library.

TMINIT zeroes all elements of the LTAB table length vector. You must preset each element of BTAB to contain the desired interspace value for the corresponding table (for example, s1 in the following example determines the interspace value for table 1). Interspace values determine how many words are added to a table when more room is needed for that table or for any table with a lower number.

```
INTEGER BTAB(64), LTAB(64)
COMMON /TM/ BTAB(64), LTAB(64)
DATA BTAB /s1,s2,s3,...,s64/
   .
   .
   .
CALL TMINIT
```

After the call to TMINIT, BTAB should not be changed. LTAB is only used to pass new table lengths from the user to the table manager.

You can use statements such as the following to access each table. In this example, TABLEi is accessed.

```
EQUIVALENCE (BTAB(i), PTRi)
INTEGER PTRi, TABLEi (0:0)
POINTER (PTRi, TABLEi)
   .
   .
   .
TABLEi (subscript) = ...
```

TM Common Block Name

The common block name TM is reserved for use by the Table Manager and must always contain 64 LTAB words.

```
COMMON /TM/ BTAB(64), LTAB(64)
```

EXAMPLES

The following program uses Cray pointers to access tables.

```
      PROGRAM TEST
      IMPLICIT INTEGER (A-Z)
      COMMON /TM/ BTAB (64), LTAB (64)
      COMMON /PTRS/ PTR1, PTR2, PTR63
      POINTER (PTR1, TAB1 (1))
      POINTER (PTR2, TAB2 (1))
      POINTER (PTR63, TAB63 (1))
      DATA BTAB/64*10/

C     Initialize table manager

      CALL TMINIT

C     Allocate space for first, second and last table of 40,
C     40 and 60 words

      CALL TMATS ( 1, 40)
      CALL TMATS ( 2, 40)
      CALL TMATS (63, 60)

C     Define CRAY POINTERS

      PTR1  = BTAB ( 1)
      PTR2  = BTAB ( 2)
      PTR63 = BTAB (63)

C     Zero tables

      DO I = 1, 40
        TAB1 (I) = 0
        TAB2 (I) = 0
        TAB63(I) = 0
      ENDDO

      DO I = 41, 60
        TAB63(I) = 0
      ENDDO
```

```
C     Assign values and preset to each half of the first table.

      DO I = 1, 10
        TAB1(I) = I
      ENDDO

      CALL TMPTS(PTR1 + 10, 10, 11)
      CALL TMPTS(PTR1 + 20, 0, 22)
      CALL TMPTS(PTR1 + 30, 1, 33)
      CALL TMPTS(PTR1 + 31, 9, 44)

C     Move the first 20 words of the first table to the first
C     20 words of the second table.

      CALL TMMVE (PTR1, PTR2, 20)

C     Preset last 20 elements of the 63rd table.

      CALL TMPTS (PTR63 + 40, 20, 12)

C     Move the last 20 words of the 63rd table to the last
C     20 words of the second table.

      CALL TMMVE (PTR63 + 40, PTR2 + 20, 20)

      END
```

SDS MANAGEMENT ROUTINES

The SDS management routines, SDSALLOC, SDSREALC, and SDSFREE, let you allocate, reallocate, and deallocate blocks of the SDS. They manage units of 512 words and are described on the SDSALLOC man page.

NAME

HPALLOC – Allocates a block of memory from the heap

SYNOPSIS

```
CALL HPALLOC(addr, length, errcode, abort)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

HPALLOC allocates a block of memory from the program's heap that is greater than or equal to the size requested. If the request cannot be satisfied from the free blocks currently in the heap, it will try to allocate more memory from the system.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

addr First word address of the allocated block (output).

length Number of words of memory requested (input).

errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).

abort Abort code; nonzero requests abort on error; 0 requests an error code (input).

By using the Fortran POINTER mechanism in the following manner, you can use array A to refer to the block allocated by HPALLOC.

```
POINTER (addr, A(1))
```

Error conditions are as follows:

Error code	Condition
-1	Length is not an integer greater than 0.
-2	No more memory is available from the system (checked if the request cannot be satisfied from the available blocks on the heap).

-8 The memory arena has been truncated by a user's sbreak(2) call with a negative value.

SEE ALSO

HPCHECK(3F), HPCLMOVE(3F), HPDEALLC(3F), HPDUMP(3F), HPNEWLEN(3F), HPSHRINK(3F)

NAME

HPCHECK – Checks the integrity of the heap

SYNOPSIS

CALL HPCHECK(*errcode*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

HPCHECK checks the integrity of the heap. Each control word is examined to ensure that it was not overwritten.

errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).

Error conditions are as follows:

Error code	Condition
-7	Memory arena has been corrupted.

NAME

HPCLMOVE – Extends a block or copies the contents of the block into a larger block

SYNOPSIS

CALL HPCLMOVE(*addr*, *length*, *status*, *abort*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine extends a block if it is followed by a large enough free block or copies the contents of the existing block to a larger block and returns a status code indicating that the block was moved. This routine can also reduce the size of a block if the new length is less than the old length. In this case, this routine has the same effect as HPNEWLEN(3F).

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

addr On entry, first word address of the block to change; on exit, the new address of the block if it was moved.

length Requested new total length (input).

status Status is 0 if the block was extended in place, 1 if it was moved, and a negative integer for the type of error detected (output).

abort Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Error conditions are as follows:

Error code	Condition
-1	Length is not an integer greater than 0.
-2	No more memory is available from the system (checked if the block cannot be extended and the free space list does not include a large enough block).
-3	Address is outside the bounds of the heap.

-4	Block is already free.
-5	Address is not at the beginning of a block.

NAME

HPDEALLC – Returns a block of memory to the list of available space (the heap)

SYNOPSIS

```
CALL HPDEALLC(addr, errcode, abort)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

HPDEALLC returns a block of memory to the list of available space (the heap).

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

addr First word address of the block to deallocate (input).

errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).

abort Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Error conditions are as follows:

Error code	Condition
-3	Address is outside the bounds of the heap.
-4	Block is already free.
-5	Address is not at the beginning of the block.

NAME

HPDUMP – Dumps the address and size of each heap block

SYNOPSIS

```
CALL HPDUMP(code, dsname)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

HPDUMP dumps the address and size of each heap block.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

code Code for the type of dump requested, as follows:

Code	Meaning
1	Dumps all heap blocks in storage order

dsname Name of the file to which the dump should be written; *dsname* may be in left-justified, Hollerith form, or it may be a Fortran character variable.

NAME

HPNEWLEN – Changes the size of an allocated heap block

SYNOPSIS

CALL HPNEWLEN(*addr*, *length*, *status*, *abort*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine changes the size of an allocated heap block. If the new length is less than the allocated length, the portion starting at *addr* + *length* is returned to the heap. If the new length is greater than the allocated length, the block is extended when followed by a free block. A status is returned, indicating if the change was successful.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

addr First word address of the block to change (input).

length Requested new total length of the block (input).

status Status is 0 if the change in length was successful, 1 if the block could not be extended in place, and a negative integer for the type of error detected (output).

abort Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Error conditions are as follows:

Error code	Condition
-1	Length is not an integer greater than 0.
-3	Address is outside the bounds of the heap.
-4	Block is already free.
-5	Address is not at the beginning of the block.

NAME

HPSHRINK – Returns memory from the heap to the operating system

SYNOPSIS

```
CALL HPSHRINK
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The unused portion of the heap is returned to the operating system if the blocks closest to the break value are free. This is a Fortran interface to the _memfree memory manager routine.

SEE ALSO

utilities(3C) in *UNICOS System Libraries Reference Manual*, publication SR–2080 for information about _memfree and ememmgr

NAME

IHPLEN – Returns the length of a heap block

SYNOPSIS

length=IHPLEN(*addr, errcode, abort*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

IHPLEN returns the length of a heap block.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

length Length of the block starting at *addr* (output).

addr First word address of the block (input).

errcode Error code is 0 if no error was detected; otherwise, it is a negative integer code for the type of error (output).

abort Abort code. Nonzero requests abort on error; 0 requests an error code (input).

Error conditions are as follows:

Error code	Condition
-3	Address is outside the bounds of the heap.
-4	Block is already free.
-5	Address is not at the beginning of the block.

NAME

IHPSTAT – Returns statistics about the heap

SYNOPSIS

value=IHPSTAT(*code*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

IHPSTAT returns statistics about the heap.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

value Requested information.

code Code for the type of information requested, as follows:

Code	Meaning
1	Current heap length
4	Number of allocated blocks
10	Size of the largest free block
11	Amount by which the heap can shrink
12	Amount by which the heap can grow
13	First word address of the heap; on Cray MPP systems, byte addresses are returned.
14	Last word address of the heap; on Cray MPP systems, byte addresses are returned.

All values returned by IHPSTAT are in words.

NAME

IHPVALID – Returns validity of block address

SYNOPSIS

ret=IHPVALID(*addr*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

IHPVALID returns the validity of a block address.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

ret Return code.

- =0 Block pointed to by *addr* is not a valid block in the heap.
- ≠0 Block pointed to by *addr* is a valid block in the heap.

addr First word address of the block (input).

NAME

TMADW – Adds a word to a table

SYNOPSIS

index=TMADW(*number*, *entry*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TMADW adds a word to a table.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

index Zero-based index of the added word

number Table number

entry Entry for the table

NAME

TMAMU – Reports table management operation statistics

SYNOPSIS

CALL TMAMU(*len*, *tabnum*, *tabmov*, *tabmar*, *nword*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TMAMU reports table management operation statistics. This routine defines the following values:

len	Allocated length of the tables
tabnum	Number of tables used
tabmov	Number of tables moved
tabmar	Maximum mark of the tables in memory
nword	Number of words moved

NAME

TMATS – Allocates table space

SYNOPSIS

index=TMATS(*number*, *incre*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TMATS allocates table space.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

index Zero-based index of the new area in the table

number Table number

incre Table increment (number of words to be added to the size of the table)

NAME

TMINIT – Initializes table descriptor vector and zeroes table length vector

SYNOPSIS

```
CALL  TMINIT
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TMINIT initializes the table manager, Table Base Address table (BTAB), and zeroes out all elements of the Table Length table (LTAB).

The BTAB array should contain the desired interspace value for the corresponding array before calling TMINIT. On exit, BTAB will contain the base address for the corresponding table.

You must call TMINIT before accessing the table manager.

SEE ALSO

INTRO_HEAP(3F)

NAME

TMMSC – Searches the table by using a mask to locate a specific field within an entry using an optional offset

SYNOPSIS

index=TMMSC(*tabnum*, *mask*, *sword*, *nword* [,*offset*])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine searches a table using a mask to locate a specific field.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

index Zero-based table index of the match, if found; -1 returned if no match is found.

tabnum Table number.

mask Mask defining a field within a word.

sword Search word.

nword Number of words per entry group.

offset Offset into the table. No offset is used if this argument is not specified. Optional argument.

NAME

TMMVE – Moves memory (words)

SYNOPSIS

```
CALL TMMVE(from, to, count)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine moves memory.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

from Address from which words are to be moved

to Address of the location to which words are to be moved

count Number of words to be moved

NOTES

The TMMVE routine expects indirect addresses to be passed for the *from* and *to* arguments. These can be obtained by specifying Cray pointers, using the LOC function, or by using the TBAB table manager array elements.

NAME

TMPTC – Processes table collisions

SYNOPSIS

```
CALL  TMPTC
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine checks all managed tables to ensure that the current size of the pad area is not less than zero and is not greater than the specified pad. Any tables in violation of this are adjusted so that they have the proper interspace.

NAME

TMPTS – Presets table space

SYNOPSIS

```
CALL TMPTS(start, len, preset)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine presets table space.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

start Starting address

len Length to preset

preset Preset value

NOTES

The TMPTS routine requires an indirect address for the *start* argument. An indirect address can be a Cray pointer, a BTAB table array element, or the result of a LOC function.

NAME

TMSRC – Searches the table by using optional mask to locate specific field within entry and offset

SYNOPSIS

index=TMSRC(*tabnum*, *arg*[, *nword*[, *offset*[, *mask*]]])

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine searches a table using an optional mask.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

index Zero-based table index of the match, if a match is found; -1 returned if no match is found.

tabnum Table number to search.

arg Search argument or key.

nword Optional argument; number of words per entry. Default is 1.

offset Optional argument; offset into the entry group. Default is 0.

mask Optional argument; mask for field within entries. Default is a mask of all 1's.

NAME

TMVSC – Searches a vector table for the search argument

SYNOPSIS

index=TMVSC(*tabnum, arg, nword*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

This routine searches a vector table for a search argument.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

index Zero-based table index of the match, if a match is found; -1 returned if no match is found.

tabnum Table number.

arg Search argument.

nword Number of words per entry group.

NAME

INTRO_PROGAIDS – Introduction to programming aids

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

Programming aids consist of the following types of routines:

- Tracing and timing routines
- Watchpointing routines
- Traceback routines
- Symbolic dump routines
- Exchange package processing routines
- Signal routines

In addition to these routines, this section includes the reprieve(3F) man page, which describes the traceback procedure after an error.

Routines and Supported Machine Types

In the printed version of these routines, the man pages are grouped according to architecture type. The following list briefly describes this grouping:

- Common routines which run on all Cray Research systems:
  FSIGCTL
  GETCALLERINFO
  REPRIEVE
  SIGOFF
  TRACEBK
  TRBK

- Routines that run only on Cray PVP systems:
  GETNAMEQ
  SETPLIMQ
  SHUTDSAV
  SYMDEBUG
  SYMDUMP
  TRBKLVL
  XPFMT
- Routines that run only on Cray MPP systems:
  none

TRACING AND TIMING ROUTINES

Tracing and timing of user programs is enabled on the `cf77` command with the Flowtrace option (`-F`), or on the `cft77` command with the `-e f` option. When these options are specified, the Fortran compiler automatically places calls to the trace libraries at the beginning and end of each routine. If default libraries are loaded, your program will be enabled for Flowtracing. This trace library times individual routines. After your program completes normally, you may execute the `flowview`(1) command to view and study the timing results.

If you load a traced program with `libperf.a`, this library will generate Perftrace, instead of Flowtrace, data. This trace library times individual routines, but it does so using the Hardware Performance Monitor (HPM), which is available only on Cray PVP systems. The HPM provides such performance statistics as megaflop ratings of each routine. After your program completes normally, you can execute the `perfview`(1) command to view and study the statistical timing results.

The Flowtrace library contains routines with names that start with the letters `ZZZF0` and `IZZF0`. You should avoid naming any routine in your program that starts with this prefix.

The Perftrace library contains routines with names that start with the letters `ZZZP0` and `IZZP0`. You should avoid these prefixes when naming routines in your program. You should also avoid naming a routine `SECOND` in your program, if you are using either of the tracing tools.

You can control individual routine event tracing by calling the library routine `SETPLIMQ`(3F). This routine only provides meaningful results under Flowtrace. However, for compatibility, it is also available as a dummy entry in Perftrace. You can time individual sections of your program by calling the `FLOWMARK` routine, which is available under both Flowtrace and Perftrace.

More information about the tracing tools is available in the *UNICOS Performance Utilities Reference Manual*, publication SR–2040.

WATCHPOINTING ROUTINES

As a debugging aid, you can control watchpointing of any variables or memory areas in your program. Watchpointing is enabled on the `cf77` command with the Flowtrace option `-F`, or on the `cft77` command with the `-e f` option. You must specify the special watchpoint library `libwatch.a` when you load your program.

In addition, you must include calls to the watchpointing facility in your program, specifying which variables are to be checked at every routine entry and exit. You should avoid beginning any routine names with the letters `WATCH` when you use this facility.

More information about the watchpointing facility is available in the *UNICOS Performance Utilities Reference Manual*, publication SR–2040.

TRACEBACK ROUTINES

The traceback routines list all subroutines active in the current calling sequence (using `TRBK`(3F)) and return information for the current level of the calling sequence (with `TRBKLVL`(3F)). `TRACEBK`(3F) is also available. It prints a traceback on `stdout`.

Traceback routines return unpredictable results when subroutine linkage does not use standard Cray Research calling sequences.

DUMP ROUTINES

The symbolic dump routines produce a symbolic dump of a running program. `SYMDUMP`(3F) is recommended for use on the UNICOS operating system because it has an interface designed for the UNICOS operating system.

EXCHANGE PACKAGE PROCESSING ROUTINES

The exchange package processing routine (`XPFMT`(3F)) provides a printable image of an exchange package, which is a 16-word block of memory associated with a particular program. This routine is available on Cray PVP systems only.

SIGNAL ROUTINES

Signals are an asynchronous process control and interprocess communication mechanism in the UNICOS operating system. As a control mechanism, signals can notify or terminate a process when certain hardware conditions occur. For instance, if a process gets a floating-point error or operand range error, the kernel sends that process a SIGFPE (floating-point exception) or SIGORE (operand range error), respectively. In addition, users can send certain signals to their processes by pressing <CONTROL-C> or <CONTROL-\>, which generate SIGINT and SIGQUIT, respectively.

Each signal has a default action associated with it that normally terminates its process, with or without a core dump. Processes may also choose to catch a signal; that is, jump to a user-defined signal handler routine when the signal is sent to the process. A signal may also be ignored by a process, in which case the delivery of the signal has no effect on the process.

See FSIGCTL(3F) for a list of the signals and a brief description of the meaning of each signal.

The SHUTDSAV routine sets up calling programs to be checkpointed on system shutdown.

The SIGOFF routine prevents receipt of all signals currently being caught. The SIGON routine allows receipt of signals after a SIGOFF or within a signal handler.

NAME

FSIGCTL – Specifies action on receipt of signal

SYNOPSIS

```
INTEGER FSIGCTL, oldact
CHARACTER*(*) action, sig
EXTERNAL func
oldact = FSIGCTL(action, sig, func)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

FSIGCTL is the Fortran-callable interface to the sigctl(2) system call.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

FSIGCTL lets you choose one of six ways to handle the receipt of a specific signal, as specified by the following arguments:

oldact If fsigctl completes successfully, it returns the previous action for the specified signal; otherwise –1 is returned.

action Character string that determines the action taken on receipt of the signal. Valid character string values for *action* are as follows:

´DUMPCORE´	When signal *sig* arrives, terminate the program and generate a core file (see signal(2)).
´IGNORE´	Ignore signal *sig*.
´KILL´	When signal *sig* arrives, terminate the program.
´REGISTER´	When signal *sig* arrives, transfer control to subroutine *func* to handle the signal, then return back to the point of interrupt.
´STOP´	When signal *sig* arrives, stop the program.
´CONTINUE´	When signal *sig* arrives, continue the program.

sig Character string that defines the signal to catch. Valid values are defined in the <sys/signal.h> file; the strings are the same as the signal names (for example, ´SIGHUP´, ´SIGABRT´, and so on).

func If *action* is ´REGISTER´, *func* is the user's signal-handling routine; otherwise, *func* should be 0. Usually, the name of the signal-handling routine must be declared as EXTERNAL, as in the EXAMPLES section.

NOTES

On Cray PVP systems (excluding the CRAY T90 series), *action* and *sig* can be integer values instead of character strings. Valid integer values for *action* are defined in <sys/signal.h> and on the sigctl(2) man page. Valid integer values for *sig* are the signal numbers defined in signal(2).

To promote portability, it is recommended that the sigctl function be called if you want to pass integer values for the *action* and *sig* arguments.

RETURN VALUES

FSIGCTL returns an integer value corresponding to the previous signal action for the specified signal. It returns –1 if an improper *action* or *sig* argument is given, or if any other errors occur.

EXAMPLES

A frequent use of FSIGCTL is to catch the shutdown signal, as shown in the following example:

```
EXTERNAL CHECKPT
CALL FSIGCTL('REGISTER','SIGSHUTDOWN',CHECKPT)
```

CHECKPT is a user-supplied routine that tells the running program to write a checkpoint file at the next possible time.

SEE ALSO

SIGOFF(3F)

sigctl(2), signal(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

NAME

GETCALLERINFO – Returns the name and line number of the calling routine

SYNOPSIS

```
INTEGER ilen, lineno, ierror
CHARACTER*n, name
CALL GETCALLERINFO(name, ilen, lineno, ierror)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

The GETCALLERINFO subroutine returns the name of the subroutine and the line number from which the current routine was called.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine:

ilen An integer variable that will receive the actual number of characters in the caller's name

lineno An integer variable that will receive the line number of the calling subroutine

ierror An integer variable that will contain the exit status as defined in the EXIT STATUS section

name A variable of type CHARACTER that will receive the name of the calling subroutine

EXIT STATUS

Upon successful completion of this routine, *ierror* is set to 0. If any of the following conditions occur, *ierror* is set to the corresponding value:

EINVAL An internal error prevented recovery of the caller information.

ETRUNC The caller's name was longer than the length of the character variable provided by the user. As much of the caller's name as will fit is returned.

EXAMPLES

```
PROGRAM MAIN_PROGRAM
CALL SUB_ROUTINE
END
SUBROUTINE SUB_ROUTINE
CHARACTER * 50, NAME
CALL GETCALLERINFO(NAME, ILEN, LINENO, IERROR)
IF (IERROR .EQ. 0) THEN
  PRINT *,'Called from line ', LINENO, ' of ', NAME(1:ILEN)
ENDIF
RETURN
END
```

Execution of this program results in the following output:

```
Called from line 2 of MAIN_PROGRAM
```

NAME

REPRIEVE – Describes error handling under the UNICOS operating system

IMPLEMENTATION

All Cray Research Systems

DESCRIPTION

On Cray Research systems, program errors (like floating-point errors, operand range errors, and so on) can be handled by two mechanisms: core files or tracebacks. By default, all programs that use Fortran I/O are loaded with the traceback code, and all other programs do not contain this code.

If the traceback code is loaded with a program, error conditions that would normally cause a fatal signal to be sent are caught by a tracer routine. This tracer routine prints out the error condition and then gives a traceback before aborting the program with a core dump. If the enhanced traceback mechanism is used (only on Cray PVP systems), the tracer routine prints out a more complete error status (including the P, A, S, VM, and VL registers at the point of interrupt) and a traceback that includes routine arguments. It then flushes buffers and ends execution of the program without a core dump (so no post-mortem debugging is available).

The following environment variables are available to change the characteristics of the traceback mechanism:

TRACEBK If set to 0, errors will cause a core dump but no traceback. If you intend to use a debugger to debug your applications, or if you want to restart your core files, you should set this variable to 0.

TRACEBK2 If set to nonzero (and TRACEBK is not set to 0), programs on Cray PVP systems will use the enhanced traceback mechanism.

The variables below are used to customize an enhanced traceback and are available only on Cray PVP systems if the TRACEBK2 variable is also set:

TRBKCORE If set to nonzero, it will give both a traceback and a core dump on encountering an error.

TRBKVEC If set to nonzero, the vector registers will be printed along with the other registers.

TRBKSIGS This variable can be used to alter the set of signal numbers that will cause a traceback to occur. Its format is a list of numbers separated by commas; positive numbers in the list will be added to the set of signal numbers, and negative numbers will be removed from the set of signal numbers.

NOTES

The reprieve mechanism goes through all of the clean-up routines that would be run under `abort`(3C) processing, including all of the routines registered by `atabort`(3C). Because all open files are flushed and closed during this time, restarting a core file that is the result of reprieve processing will not work. For a checkpoint/restart mechanism that is controllable by a signal, see SHUTDSAV(3F).

EXAMPLES

Example 1: To get the enhanced traceback mechanism working for a C program, follow this example:

```
cc -c prog.c
segldr -D'hardref=_reprinit' prog.o
env TRACEBK2=1 a.out
```

Example 2:

```
cft77 prog.f
segldr prog.o
a.out
<Traceback+coredump produced>

env TRACEBK=0 a.out
<No traceback produced, just a core dump>

TRACEBK2=1; export TRACEBK2
<CRAY Y-MP systems only, to get an enhanced traceback>

a.out
<Enhanced style traceback, no core file produced>

env TRBKCORE=1 a.out
<Gets a traceback and a core file>

env TRBKVEC=1 a.out
<Prints out vector registers with traceback>

env TRBKSIGS=8,-7 a.out
<Adds signal 8, removes signal 7 from the set of
 traceback signals>
```

SEE ALSO

TRACEBK(3F) (the Fortran interface), signal(2) for a list of signals

abort(3C), SHUTDSAV(3F)

NAME

SIGOFF, SIGON – Controls receipt of signals

SYNOPSIS

```
INTEGER oldstat
CALL  SIGOFF
CALL  SIGON
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

SIGOFF and SIGON allow signals to be deferred without issuing a system call.

SIGOFF prevents the receipt of all signals that have a signal handler registered for them; however, all signals that terminate or cause a core dump of the process will be delivered.

Use these routines for coding critical sequences. Both routines return the previous signal status; a nonzero value means signals are deferred.

SIGON allows receipt of signals after a SIGOFF or within a signal handler.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

EXAMPLES

```
      INTEGER OLDSTAT, SIGOFF, SIGON
      OLDSTAT = SIGOFF()
C       < Critical region of code>
      IF(OLDSTAT .EQ. 0) CALL SIGON
```

SEE ALSO

sigoff(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

signal(2), pause(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

NAME

TRACEBK – Prints a traceback

SYNOPSIS

```
CALL  TRACEBK
        or
INTEGER  idepth
CALL  TRACEBK(idepth)
        or
INTEGER  idepth
CHARACTER*n  filenm
CALL  TRACEBK(idepth, filenm)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TRACEBK prints (on stdout) a traceback beginning with its caller and ending at "Start-up," or when *idepth* is reached. The traceback includes the arguments and their values and some data about the run-time stack.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

TRACEBK has the following optional arguments:

idepth Integer expression containing trace depth; if no argument is used, or *idepth* is not in the range 3 to 50, 25 is used. *idepth* can have the following values:

- <0 Trace depth equals 25
- 0 Print one line "Where am I" message
- 1 Print one line trace (caller's name and line number)
- 2 Print "Where am I" and one line trace
- >2 Trace depth (25 if above 50)

filenm Character variable containing the name of a file in which to write the traceback. *idepth* must be present if *filenm* is present.

The optional *filenm* argument in the CALL to TRACEBK is a character variable containing the name of a file in which TRACEBK( ) will write the traceback. *filenm* must not be open as a Fortran file when TRACEBK( ) is called because the results are unpredictable. *filenm* may be read with standard Fortran I/O.

MESSAGES

TRACEBK fails (with a message on stdout) if the trace data has been noticeably corrupted.

SEE ALSO

TRBK(3F) for information on listing the subroutines in a current calling sequence

REPRIEVE(3F) for a general description of reprieve processing, including descriptions of related environment variables

STKSTAT(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

TRBK – Lists all subroutines active in the current calling sequence

SYNOPSIS

```
CALL TRBK [(depth)]
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

TRBK prints a list of all subroutines that are active in the current calling sequence from the currently active subprogram. It also identifies the address of the reference. The list is printed to stderr.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

depth Optional argument that selects the maximum traceback depth. If not specified, the default is the value of the TRBKDPTH environment variable (if defined) or 25 if the variable is not defined.

SEE ALSO

REPRIEVE(3F) for a description of UNICOS error handling

TRACEBK(3F) for information about the ability to write traceback information to an alternate file

NAME

GETNAMEQ – Returns the name of calling routine

SYNOPSIS

```
SUBROUTINE  GETNAMEQ(name)
INTEGER name
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

GETNAMEQ returns the *name* of the caller to its caller.

NOTES

The GETNAMEQ routine is obsolete; use the GETCALLERINFO(3F) routine in its place.

BUGS

Until long-name support is added, GETNAMEQ returns only the first 8 characters of a name.

EXAMPLES

FOO calls BAR. If BAR calls GETNAMEQ, *name* is set to FOO. The result is left-justified and zero-filled in a Cray word.

SEE ALSO

GETCALLERINFO(3F)

NAME

SETPLIMQ – Initiates a detailed tracing of each call and return

SYNOPSIS

SETPLIMQ(*lines*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SETPLIMQ writes trace lines to stderr. Calling this routine only affects flowtracing. Perftrace contains this routine name, but calling it has no effect.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

lines Integer specifying the number of traces to display. Passing a negative number gives unpredictable results.

NAME

SHUTDSAV – Sets up the calling program to be checkpointed on system shutdown

SYNOPSIS

```
CALL SHUTDSAV(path, flags)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SHUTDSAV establishes a signal handler that catches the SIGSHUTDN signal, indicating that the system is going to shut down soon. The signal handler, upon receipt of the SIGSHUTDN signal, checkpoints the program using the chkpnt(2) system call, creating a restart file with the specified name.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

SHUTDSAV has the following arguments:

path Character string path name of the file to be created as the restart file. The *path* argument must not refer to a file that already exists, since the chkpnt(2) system call that performs the actual checkpoint work will not overwrite an existing file. If *path* does not designate an absolute path name, then the restart file will be created relative to the current working directory of the calling program at the time that the SIGSHUTDN signal is received. Finally, the directory in which the restart file is to be created must be writable by the caller for the checkpoint to be successful.

flags Control options. Specify an integer variable or constant. If the least-significant bit of *flags* is 0, the calling program will not checkpoint itself upon a system shutdown if it is running as part of an Network Queuing System (NQS) batch job. This is important because, by default, NQS checkpoints all of the jobs under its control upon a system shutdown, making any additional checkpoint work by the program unnecessary. Alternatively, if the least-significant bit of *flags* is set, the calling program will try to checkpoint itself, even when running as part of an NQS batch job. In all other cases, the calling program will try to checkpoint itself upon the receipt of a `SIGSHUTDN` signal. Finally, all bits other than the least-significant bit of *flags* are reserved for future use, and they should be set to 0.

NOTES

Certain conditions must be satisfied for a process to be successfully checkpointed. See `chkpnt`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012, for a discussion of the restrictions placed on a process that is to be checkpointed.

SEE ALSO

`chkpnt`(1), `restart`(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

`chkpnt`(2), `restart`(2), `sigctl`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

NAME

SYMDEBUG – Produces a symbolic snapshot of a running process

SYNOPSIS

```
CALL  SYMDEBUG('param{,param}.')
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SYMDEBUG is a library routine that produces the same sort of output as debug(1). SYMDEBUG is provided on the UNICOS operating system primarily to ease migration of applications from COS; new code should use SYMDUMP(3F), which is designed for use with the UNICOS operating system.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This routine takes the following arguments:

param SYMDEBUG parameters. *param* may be entered in either uppercase or lowercase.

Some SYMDEBUG parameters allow you to specify a value along with the parameter. In these cases, *param=value* substitutes for *param*. In *param=value*, in which *value* is a symbol, *value* is case-sensitive.

SYMDEBUG uses the following parameters:

S=*sfile* Specifies the file containing the Debug Symbol tables. The default is a.out.

L=*lfile* Specifies the file to receive the listing output from the symbolic debug routine. By default, SYMDEBUG output is written to stdout.

CALLS=*n* Number of routine levels to be looked at in a symbolic dump. For each known task, SYMDEBUG traces back through the active subprograms the number of levels specified by *n*. Routines for which no symbol table information is available are not counted for purposes of the CALLS count. If this parameter is omitted, or if CALLS is specified without a value, the default is 50.

SYMS=*sym*{:*sym*}

List of symbols to be dumped by SYMDEBUG. Up to 20 symbols (*sym*s) can be specified, and there is no limit on the length of the symbol names. Separate individual *sym*s with a colon. SYMS applies to all blocks dumped; by default, all symbols are dumped.

With regard to case, *sym*s must be entered in exactly the way the symbol names appear in the Debug Symbol table. Symbol names might not appear in the Debug Symbol table in the same way they appear in your program. Fortran and Pascal always convert names to uppercase.

NOTSYMS=*nsym*{:*nsym*}

List of symbols to skip. Up to 20 symbols (*nsym*s) can be specified, and there is no limit on the length of the symbol names. Separate individual *nsym*s with a colon. The default skips no symbols. This parameter takes precedence over the SYMS parameter.

With regard to case, *nsym*s must be entered in exactly the way the symbol names appear in the Debug Symbol table. Symbol names might not appear in the Debug Symbol table in the same way they appear in your program. Fortran and Pascal always convert names to uppercase.

MAXDIM=*dim*{:*dim*}

Maximum number of elements from each dimension of the arrays to be dumped. MAXDIM lets you sample the contents of arrays without creating huge amounts of output. When MAXDIM is specified, arrays are dumped in storage order (row, column for Pascal; column, row for Fortran). MAXDIM applies to all blocks dumped. The default is MAXDIM=20:5:2:1:1:1:1. No more than 7 dimensions can be specified.

BLOCKS=*blk*{:*blk*}

List of Fortran common blocks to be included from the symbolic dump. A maximum of 20 blocks can be specified. Separate the *blk*s with colons. All symbols (qualified by the SYMS and NOTSYMS parameters) in the named blocks are dumped. Default is no common blocks dumped; if you specify BLOCKS without any *blk*s, all common blocks declared in routines to be dumped are included in the symbolic dump.

blk must be entered in uppercase.

NOTBLKS=*nblk*{:*nblk*}
List of Fortran common blocks to be excluded in the symbolic dump. A maximum of 20 blocks can be specified. Separate the *nblk*s with colons. This parameter is used in conjunction with BLOCKS and takes precedence over the BLOCKS parameter.

nblk must be entered in uppercase.

RPTBLKS Repeat blocks; when this option is used, the contents of Fortran common blocks specified with the BLOCKS and NOTBLKS parameters are displayed for each subroutine in which they are declared. The default displays common blocks only once.

PAGES=*np* Page limit for the symbolic dump routine. Under the UNICOS operating system, SYMDEBUG does not format output in pages. However, this parameter can still be used to regulate the amount of output that SYMDEBUG generates. Each page is worth 45 lines of output from SYMDEBUG. The default *np* is 70.

NOTES

Specify library libdb.a, which contains SYMDEBUG, on the -l option when you load your program. See the examples on loading in the NOTES section of the SYMDUMP(3F) man page.

EXAMPLES

The following are example calls from Fortran to SYMDEBUG:

```
CALL SYMDEBUG('CALLS=40,RPTBLKS.')
CALL SYMDEBUG('BLOCKS=AA:BB:CC.')
```

SEE ALSO

SYMDUMP(3F)

debug(1) in the *UNICOS CDBX Symbolic Debugger Reference Manual*, publication SR–2091

NAME

SYMDUMP – Produces a symbolic snapshot dump of a running program

SYNOPSIS

CALL SYMDUMP ('–b *blklist* -B -c *calls* -d *dimlist* -l *lfile* -r -s *symfile* -V -y *symlist* -Y', *abort_flag*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SYMDUMP is a library routine that produces the same sort of output as debug(1). It accepts C character descriptors, Fortran Hollerith strings, and Pascal packed character arrays.

The method of calling library routines differs from language processor to language processor, but SYMDUMP accepts the same arguments regardless of the language processor. The argument string, if provided, must be enclosed in parentheses, and the options (excluding the Abort flag) must be enclosed in quotation marks. When calling SYMDUMP from Fortran or Pascal, the quotation marks must be single; when calling from C, the quotation marks must be double. All arguments are optional.

The options indicate the type and extent of information to be dumped by SYMDUMP. The options string is passed to SYMDUMP in one of the following forms:

- As a character descriptor, produced by Fortran and C for defined character strings
- As an address of a null-terminated string, such as an integer, Hollerith, or Pascal packed character array

The argument string can contain a maximum of 4096 characters. All options are optional, and they may appear in any order.

Unlike command lines, SYMDUMP option-arguments may not be grouped after one hyphen on the SYMDUMP call. That is, SYMDUMP('-V -r') is permitted, but SYMDUMP('-Vr') is not permitted.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are valid options and arguments:

`-b` *blklist*

`-B` These options control the displaying of Fortran common block symbols and may occur anywhere in the option string in any order. The symbols to be displayed from any particular common block depends on the use of the `-Y` and `-y` *symlist* options.

If neither option is specified, no Fortran common blocks are included in the symbolic dump. This is the default. If `-B` is specified, all Fortran common blocks are included in the symbolic dump. If `-b` *blklist* is specified, only the Fortran common blocks named in *blklist* are included in the symbolic dump. If both options are specified, all Fortran common blocks are included in the symbolic dump, except those in *blklist*.

blklist may have up to 20 Fortran common blocks named. There is no limit on the length of a common block name. The common blocks named in *blklist* must be separated by commas (for example: `-b c,d`).

The common blocks named by *blklist* must be entered in uppercase letters.

`-c` *calls*

An integer that specifies the number of routine levels to be displayed in the symbolic dump. For each known task, `SYMDUMP` traces back through active routines the number of levels specified by *calls*. Routines for which no symbol table information is available are not counted for purposes of the routine level count. The default is 50.

`-d` *dimlist*

A list of integers that specifies the maximum number of elements from each dimension of the arrays to be dumped. `SYMDUMP` can dump array elements from up to seven dimensions. The dimensions must be specified by integer values, and the values must be separated by commas (for example: `-d 4,6`).

This option lets you sample the contents of an array without creating huge amounts of output. *dimlist* applies to all blocks dumped, and the arrays are dumped in storage order. The default is 20,5,2,1,1,1,1.

`-l` *lfile* Names an output file. Specifying `-l` *file* directs `SYMDUMP` to write output to the specified file. If you call `SYMDUMP` more than once, and you specify `-l` with the same file each time, `SYMDUMP` output will be appended to the file each time. By default, `SYMDUMP` output is written to `stdout`.

`-r` Repeat blocks. When this option is used, `SYMDUMP` displays the contents of Fortran common blocks specified with `-B` and `-b` *blklist* for each subroutine in which they are declared. The default displays common blocks only once.

`-s` *symfile*
Names a file containing the Debug Symbol tables. There is no limit on the length of the *symfile* file name, and it may include a path name to the desired file. `segldr` puts both the symbol table information and the executable binary in the same file. By default, Debug Symbol tables are written to `a.out`.

`-V` With `-V` specified, `SYMDUMP` generates `SYMDUMP` release statistics.

`-y` *symlist*
`-Y` These options control the displaying of symbols and may occur anywhere in the option string in any order.

If neither option is specified, all symbols are displayed. This is the default. If only the `-Y` option is specified, no symbols are displayed. If only the `-y` option is specified, all symbols except those named in *symlist* are displayed. If both options are specified, only the symbols named in *symlist* are displayed.

The *symlist* variable may contain up to 20 named symbols, and there is no limit to the length of the symbol names. The symbols named in *symlist* must be separated by commas (for example: `-y a, b`).

Enter the symbols in the same case in which they appear in the symbol table. Names may not always appear in the symbol table in the same way they appear in your program. Fortran and Pascal always convert names to uppercase.

abort_flag
An optional *abort_flag* variable indicates to `SYMDUMP` whether or not to abort if it finds an error when parsing the `SYMDUMP` statement. An *abort_flag* with a value of 0 indicates no abort; an *abort_flag* with a nonzero value indicates abort.

You cannot enter an *abort_flag* variable if you have not entered any options.

By default, `SYMDUMP` examines all options, reports errors found, and generates a dump based on the options it could understand; the program does not abort.

Note that the *abort_flag* variable is not allowed when the option string is a Pascal variant array.

NOTES

Specify library libdb.a, which contains SYMDUMP, on the -l option when you load your program.

The following three examples show how to load programs that call SYMDUMP.

Example 1: If you are not expanding blank common and do not need to specify a segldr HEAP directive on the segldr command line for any other reason, you do not need to specify a segldr HEAP or STACK directive. The following example shows a segldr command line without HEAP or STACK directives:

```
segldr -l db.a *.o
```

Example 2: If you are expanding blank common, you need to specify stack and heap sizes to segldr. The following example shows a segldr command line that can be used if the program expands blank common.

```
segldr -l db.a -D "STACK=3000+0;HEAP=10000+0" *.o
```

This example shows settings that should provide enough stack and heap space for SYMDUMP to run, assuming that your program is an average large application that has as many as 1000 blocks. For applications with more blocks, 6 to 7 words per block over 1000 should be added to the heap setting. Optimal heap settings depend on the specific application.

If running the application causes SYMDUMP to exit with the following error message, the value on the HEAP directive is too small:

```
HPALLOC failed; return status = i
```

Example 3: If a segldr DYNAMIC directive is used, the stack and heap cannot expand, so a segldr STACK or HEAP directive may also be needed. See the previous example for information about expanding the stack and heap. To load the heap prior to blank common, use DYNAMIC=// on segldr's -D option, as shown in the following example:

```
segldr -l db.a -D "DYNAMIC=//" *.o
```

Both libdb.a and libp.a are required if ld(1) or cc(1) is used.

EXAMPLES

Example 1: The following example shows how to call SYMDUMP from a Fortran program when passing a character descriptor:

```
      character*30 string
      integer abtfl
      .
      .
      string = '-s test -B -b STRING'
      abtfl = 1
      .
      .
     C  CHARACTER VARIABLE
      call symdump (string, abtfl)
      .
      .
     C  CHARACTER CONSTANT
      call symdump ('-l outfile -V')
```

Example 2: The following example shows how to call SYMDUMP from C:

```
extern void SYMDUMP( );

int abt_flag = 1;
char *string;

string = "-s a.out -V";
SYMDUMP (string, &abt_flag);
```

Example 3: The following example shows how to call SYMDUMP from Pascal when passing a conformant array:

```
type
  string_type = packed array [1..30] of char;
var
  abort_flag: boolean;

procedure symdump (var string: string_type; var flag: boolean);
imported (SYMDUMP);

abort_flag := true;
string [1..20] := '-s test -y STRING -Y';
string [21] := chr (0);           (* must null terminate the string *)
symdump (string, abort_flag);
```

SEE ALSO

SYMDEBUG(3F)

For more information on segldr, see the *Segment Loader (SEGLDR) and* ld *Reference Manual*, publication SR–0066.

debug(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

NAME

TRBKLVL – Returns information on current level of calling sequence

SYNOPSIS

CALL TRBKLVL(*trbktab*, *arglist*, *status*, *name*, *calladr*, *entpnt*, *seqnum*, *numarg*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

TRBKLVL returns information on the current level of the calling sequence.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

trbktab Current level's traceback table address; on exit, current level caller's traceback table address. Returns a 0 if the current level is a main-level routine.

arglist Current level's argument list address; on exit, current level caller's argument list address. Returns a 0 if the current level is a main-level routine.

status *status* can have the following values:

- <0 Error
- =0 No error
- >0 No error and the current level is the main level

name Current level's name (ASCII, left-justified, blank-filled). This may not be a Fortran character variable.

calladr Parcel address from which the call to the current level was made.

entpnt Parcel address of the current level's entry point.

seqnum Line sequence number corresponding to the call address (0 indicates none).

numarg Number of arguments or registers passed to the current level.

EXAMPLES

Using TRBKLVL from a Fortran program requires knowledge of the values of the traceback table address and argument list address. This information is usually kept in B registers by the Fortran system. To get the necessary information, you must use the getb1 and getb2 functions. These should be declared as integer externals.

Example of use:

```
      integer getb1,getb2
      integer trbktab,arglist,status,calladr,entpnt
      integer seqnum
c
c      Get traceback table addresses for this
c      routine.
c
      arglist=getb1()
      trbktab=getb2()
c
c      Calling trbklvl will return us the name
c      and other information for this routine.
c
      call trbklvl(trbktab,arglist,status,name,calladr,
     x             entpnt,seqnum,numarg)
c
c      At this point, 'trbktab' and 'arglist' contain these
c      values for the routine using trbklvl. 'name' contains
c      the name of this routine, and so on.
c
c      Calling trbklvl again will return the information for
c      the caller of this routine.
c
      call trbklvl(trbktab,arglist,status,name,calladr,
     x             entpnt,seqnum,numarg)
c
c      At this point, 'trbktab' and 'arglist' contain these
c      values for the caller.  'name' contains the name of
c      the calling routine, and so on.
c
c       If we call trbklvl again, with the same parameters,
c       the returned values will then contain information about the
c       caller's caller.  We can continue this repeated process until
```

```
c       we reach the top of the calling tree.  At that point, 'status'
c       will be a number greater than zero, and the user should stop
c       calling trbklvl.  Otherwise, the user would loop indefinitely.
```

Note that the name returned is truncated to 8 characters if it is longer than 8 characters.

SEE ALSO

TRBK(3F)

NAME

XPFMT – Produces a printable image of an exchange package

SYNOPSIS

CALL XPFMT(*address*, *in*, *out*, *mode*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

XPFMT produces a printable image of an exchange package in a user-supplied buffer. A and S registers appear in the buffer in both octal and character form; in the character form, the contents of the register are copied unchanged to the printable buffer. The calling program is responsible for proper translation of unprintable characters. Parcel addresses have a lowercase a, b, c, or d suffixed to the memory address.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

address The nominal location of the exchange package to be printed as the starting exchange package address. The output buffer contains an 8-character field at the beginning of each line of the exchange package to indicate a Cray Research address. The binary number in *address* is used to fill these 8 characters of the first line of the exchange package in the output buffer and is incremented to fill each succeeding line of the output buffer. This is not the address of the 16-word buffer containing the exchange package to be formatted.

in A 16-word integer array containing the binary representation of the exchange package.

out An integer array, dimensioned (8,0:23), into which the character representation of the exchange package is stored. Line 0 is a ruler for debugging and is not usually printed.

The first word of each line is an address and need not always be printed.

mode An integer word indicating the mode in which the exchange package is to be printed. 'Y'L forces the exchange package to be formatted as an exchange package; 0 means that the subprogram is to use the exchange package contents to determine the machine type.

EXAMPLES

```
      SUBROUTINE SUB1(INTXP,OUTXP)
      INTEGER INTXP(16),OUTXP(8,0:23),IADDR,IMODE
*
*   address to use in output array
*
      IADDR = 8700
*
*   let processor deduce machine type
*
      IMODE = 0
*
*   pass the input exchange package to XPFMT and get the formatted
*   version to print in OUTXP
*
      CALL XPFMT(IADDR,INTXP,OUTXP,IMODE)
*
*   print the output of the XPFMT routine
*
      PRINT 1,OUTXP
  1   FORMAT(24(1X,8A8/))
      END
```

NAME

STOP_ALL – Stops all PEs in an application

SYNOPSIS

```
CHARACTER*n message
CALL STOP_ALL([message])
```

IMPLEMENTATION

Cray MPP systems

DESCRIPTION

The STOP_ALL subroutine causes a program to run all cleanup routines (including flushing I/O buffers), print out a message (saying where it was called from), and exit. If called from a single processing element (PE), STOP_ALL will cause all PEs in a multi-PE application to run their cleanup routines and exit, even if called from a parallel region. This differs from the STOP intrinsic in that STOP causes all PEs to exit if called from a master region, but will only cause the calling PE to exit if called from a parallel region.

The STOP_ALL function accepts the following optional argument:

message String to be printed as part of the STOP_ALL message line

RETURN VALUES

STOP_ALL returns no value.

SEE ALSO

exit(3C)

NAME

INTRO_INTERFACE – Introduction to system interface routines

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

System interface routines are grouped into the following categories:

- Job control routines
- Floating-point interrupt routines
- Special-purpose interface routines
- Miscellaneous routines

Routines and Supported Machine Types

In the printed version of these routines, the man pages are grouped according to architecture type. The following list briefly describes this grouping:

- Common routines which run on all Cray Research systems:

ABORT	ERREXIT
EXIT	GETCWD
GETHOST	GETOARGC
GETVARGC	IARGC
ISHELL	REMARK
SAMEFILE	

- Routines that run only on Cray PVP systems:

CLEARBT	CLEARFI
GETOARG	GETVARG
NLIMIT	REMARKF
SENSEBT	UNAME

- Routines that run only on Cray MPP systems:
  None

The following routines run on SPARC systems: ABORT, ISHELL, EXIT

JOB CONTROL ROUTINES

Job control routines perform functions relating to program termination, either causing a termination or instructing the system on how to handle a termination. Control is not returned.

The ABORT routine requests abort with traceback. The EXIT routine exits from a Fortran program. The ERREXIT routine requests an abort.

FLOATING-POINT INTERRUPT ROUTINES

Floating-point interrupt routines allow you to test, set, and clear the Floating-point Interrupt Mode flag. Subroutine linkage is call-by-address.

The CLEARFI routine temporarily prohibits floating-point interrupts; the SETFI routine temporarily permits floating-point interrupts; the SENSEFI routine determines if floating-point interrupts are permitted or prohibited. All of these routines are documented on the CLEARFI man page. These routines are not available on CRAY T90 systems that support IEEE arithmetic.

SPECIAL-PURPOSE INTERFACE ROUTINES

The following routines are used for special-purpose interfaces:

- GETCWD: returns the current working directory.
- GETHOST: gets name of host mainframe.
- GETARG: returns a command-line argument (documented on the IARGC man page).
- GETOARG: gets command-line arguments.
- GETOARGC: gets command-line arguments.
- GETVARG: gets command-line arguments, allowing blanks or commas as delimiters.
- GETVARGC: gets command-line arguments, allowing blanks or commas as delimeters.
- IARGC: returns the number of command-line arguments.
- ISHELL: executes a shell command.
- NLIMIT: sets or obtains resource limit values.
- REMARK: enters a message (preceded by a message prefix) in the stderr file.
- REMARK2: enters a message in the stderr file.

- REMARKF: enters a formatted message in the stderr file.
- UNAME: gets name of current operating system.

MISCELLANEOUS ROUTINES

The CLEARBT routine disables bidirectional memory transfers.

The SETBT routine enables bidirectional memory transfers.

The GETHOST routine returns name of host mainframe.

The SAMEFILE routine checks to see if two files have the same inode number.

The SENSEBT routine checks if bidirectional memory transfer is enabled or disabled.

NAME

ABORT – Requests abort with traceback

SYNOPSIS

```
CALL ABORT[(msg)]
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

ABORT requests an abort with traceback and provides an optional message written to the stderr file and creates a core dump.

msg Optional message of type character for stderr file.

NAME

ERREXIT – Requests abort

SYNOPSIS

```
CALL ERREXIT
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

ERREXIT provides an error exit from a Fortran program. It calls the abort(3C) function.

NAME

EXIT – Exits from a Fortran program

SYNOPSIS

```
CALL EXIT[(istat)]
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

EXIT ends the execution of a Fortran program.

istat EXIT ends the execution of a Fortran program optionally with status *istat*, which is an integer. The default exit status is 0.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

NOTES

The single parameter to EXIT is optional. Because EXIT is predeclared, any explicit call to it can optionally include a parameter. However, if EXIT is passed as an actual parameter corresponding to a dummy parameter that is a procedure, and if the corresponding dummy parameter is then called, one of the following must be done:

- The dummy procedure parameter must be declared in an interface block as having an optional integer parameter
- The call to the dummy procedure parameter (that is, the indirect call to EXIT) must include an actual parameter corresponding to the optional one for EXIT.

NAME

GETCWD – Returns the current working directory

SYNOPSIS

```
CHARACTER cwd*n
CALL  GETCWD(cwd)
```

or

```
CHARACTER cwd*n
INTEGER  GETCWD, i
i  =  GETCWD(cwd)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

GETCWD returns the current working directory of the process in a character string. It can be called as either a subroutine or a function.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The arguments are as follows:

cwd Character variable to receive the current working directory. The *cwd* argument must be of type character and must be at least 1 character larger than the current working directory. Extra characters are set to blanks.

i Integer return value (when called as a function); *i* can have the following values:

- -2 GETCWD was called with no argument or with an argument not of type character
- -1 The getcwd(3C) request failed, probably because the *cwd* variable was not long enough to contain the complete path name (*cwd* will be set to all blanks)
- >0 Number of characters in the name of the current working directory

SEE ALSO

`getcwd`(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

GETHOST – Returns name of host mainframe

SYNOPSIS

```
CHARACTER host*n
INTEGER  GETHOST, i
...
i = GETHOST(host)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

GETHOST returns the name of the host mainframe in a character string. It can be called either as a subroutine or a function.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

GETHOST has the following arguments:

host Variable of type character to receive the host name. It should be at least as large as the host name. Extra characters will be set to blanks.

i Return value of type integer (when called as a function); *i* can have the following values:

- -2 GETHOST was called with no argument or with an argument not of type character.
- -1 The gethostname(2) request failed. *host* will be set to all blanks.
- >0 Number of characters in the name of the host.

SEE ALSO

hostname(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

gethostname(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

NAME

GETOARGC – Gets command-line arguments

SYNOPSIS

```
INTEGER  GETOARGC
CHARACTER  cbuf*m
iret=GETOARGC(cbuf)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

GETOARGC places the next command-line argument in *cbuf*. Each call to GETOARGC selects the next argument.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

cbuf Variable, array, or array element of type character to receive the argument. If the argument is shorter than *cbuf*, it is blank padded.

iret The return value; *iret* can have the following values:

- 1 If successful
- 0 If there are no more command-line arguments

NAME

GETVARGC – Gets command-line arguments, allowing blanks or commas as delimiters

SYNOPSIS

```
INTEGER  GETVARGC
CHARACTER  cbuf*m
iret=GETVARGC(cbuf)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

GETVARGC places the next command-line argument in *cbuf*. Each call to GETVARGC selects the next argument.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

cbuf Variable, array, or array element of type character to receive the argument. If the argument is shorter than *cbuf*, it is blank-filled.

iret Return value; *iret* can have the following values:

- 1 If successful
- 0 If there are no more command-line arguments

NOTES

The PXFGETARG(3F) routine provides similar functionality.

NAME

IARGC, GETARG – Returns number of command-line arguments or the argument itself

SYNOPSIS

```
iargs= IARGC( )

INTEGER GETARG
ichars = GETARG(i,c)
ichars = GETARG(i,c,size)
```

IMPLEMENTATION

GETARG is available on Cray PVP systems (except CRAY T90 series)

IARGC is available on all Cray Research systems

DESCRIPTION

These routines return the *i*th command-line argument of the current process.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following are valid arguments for this routine.

iargs An integer specifying the number of command-line arguments passed to the program.

ichars An integer specifying the number of nonnull characters in the string returned.

i Integer specifying the number of the argument to return.

c Character variable or integer array in which to return the command-line argument.

size If *c* is an array, an integer giving the number of elements in that array.

If a program is invoked with the following command line, IARGC returns 3:

```
foo arg1 arg2 arg3
```

NOTES

PXFGETARG(3F) provides similar functionality and is available on Cray MPP systems and Cray PVP systems, including the CRAY T90 series.

SEE ALSO

GETOPT(3) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

ISHELL – Executes a UNICOS shell command

SYNOPSIS

```
ISTAT = ISHELL(command)
```

IMPLEMENTATION

All Cray Research systems

SPARC systems

DESCRIPTION

ISHELL executes a UNICOS shell command. ISHELL has the following argument:

command Command to be given to the shell; can be of type character or some numeric type. If it is a noncharacter type, the command should consist of packed characters terminated by a null byte.

ISHELL passes *command* to the shell sh(1) as input, as if *command* were entered at a terminal.

NOTES

In UNICOS 6.0, the pshell(1) command was used to reduce the memory usage of processes using ISHELL. In UNICOS 7.0 and later, this memory usage problem has been eliminated. The pshell(1) command is obsolete in UNICOS 7.0; it is useful only for users running UNICOS 6.0 or 6.1 binary files on a 7.0 system.

RETURN VALUES

ISHELL returns the termination status filled in by the waitpid(2) system call, which is used to wait for the child shell process. Unless the command was interrupted by a signal, its exit status is contained in bits 8 through 15 (bit 0 being the least significant bit) of the value returned by ISHELL (see the waitpid(2) man page for more information). However, if any errors occur in running the shell or collecting its exit status, ISHELL returns a negative number; this number is the negative value of errno corresponding to the error.

EXAMPLES

```
      WRITE(TPMNT,500) DTV, XDT, VOL, XVOL, DSN, PDN, MBSV
      CALL ISHELL(TPMNT)
500   FORMAT('tpmnt -l sl -F U -T -g ',a6,' -x ',a6,
     +       '-v ',a6,'=',a6,' -P ',a8,' -f ',a, ' -b ',a)
```

SEE ALSO

`pshell`(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011

`system`(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

REMARK, REMARK2 – Enters a message in the stderr file

SYNOPSIS

```
CALL REMARK(message)
CALL REMARK2(message)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

REMARK and REMARK2 send a message to the stderr file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

message Character entity sent to the stderr file. The character entity must be of type character or a string of characters terminated by a null byte.

Maximum message length: (REMARK): 71 characters; (REMARK2): 79 characters.

NAME

SAMEFILE – Checks to see whether two files have the same inode number

SYNOPSIS

```
LOGICAL  SAMEFILE
i=SAMEFILE(path1, path2)
```

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

SAMEFILE returns .TRUE. if two file names point to the same inode; otherwise, it returns .FALSE.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

EXAMPLES

In this example, F1 and F2 represent the path names of files to be tested. The names must start at a word boundary and be null terminated. On Cray MPP systems and CRAY T90 series systems, a character variable may not be used.

```
IF (SAMEFILE(F1,F2)) PRINT*,'files are the same!'
```

NAME

CLEARBT, SETBT – Temporarily disables or enables bidirectional memory transfers

SYNOPSIS

```
CALL  CLEARBT
CALL  SETBT
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CLEARBT temporarily disables bidirectional memory transfers and SETBT temporarily enables bidirectional memory transfers.

These routines are local to the current process (such as a program). The system restores the most recent mode setting at the start of the next process. No arguments are required or returned.

NAME

CLEARFI, SENSEFI, SETFI – Modifies floating-point interrupt status

SYNOPSIS

```
INTEGER modefi
CALL  CLEARFI
CALL  SENSEFI(modefi)
CALL  SETFI
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 systems that support IEEE arithmetic)

DESCRIPTION

These Fortran-callable routines let you modify the floating-point interrupt status.

SETFI sets the interrupt bit.

CLEARFI clears the interrupt bit.

SENSEFI returns the current interrupt status in *modefi*.

modefi Returns a 1 if floating-point interrupts are enabled; if floating-point interrupts are disabled, it returns a 0.

NOTES

Under the UNICOS operating system, CLEARFI and SETFI are local to the current process. The system restores the most recent mode setting at the start of the next process. No arguments or parameters are required.

NAME

GETOARG – Gets command-line arguments

SYNOPSIS

```
INTEGER  GETOARG
iret=GETOARG(cbuf [,length])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 series)

DESCRIPTION

GETOARG places the next command-line argument in *cbuf*. Each call to GETOARG selects the next argument.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

cbuf Variable, array, or array element of any type to receive the argument. If *cbuf* is character, it is blank-padded; otherwise, *cbuf* is null-padded.

length The number of words available in *cbuf* if *cbuf* is not a character variable.

iret The return value; *iret* can have the following values:

- 1 If successful
- 0 If there are no more command-line arguments

NOTES

The GETOARGC(3F) routine provides similar functionality and is available on Cray MPP systems and Cray PVP systems, including the CRAY T90 series.

NAME

GETVARG – Gets command-line arguments, allowing blanks or commas as delimiters

SYNOPSIS

```
INTEGER  GETVARG
iret=GETVARG(cbuf [,length])
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 series)

DESCRIPTION

GETVARG places the next command-line argument in *cbuf*. Each call to GETVARG selects the next argument.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The following is a list of valid arguments for this routine.

cbuf Variable, array, or array element of any type to receive the argument. If *cbuf* is character, it is blank-padded. Otherwise, *cbuf* is null-padded.

length The number of words available in *cbuf* if *cbuf* is not a character variable.

iret Return value; *iret* can have the following values:

- 1 If successful
- 0 If there are no more command-line arguments

NOTES

The GETVARGC(3F) routine provides similar functionality and is available on Cray MPP systems and Cray PVP systems, including the CRAY T90 series.

NAME

NLIMIT – Provides an interface to setting or obtaining resource limit values

SYNOPSIS

```
INTEGER id, rsarray(10), errstat
CALL NLIMIT(id, rsarray, errstat)
   or
iret=NLIMIT(id, rsarray, errstat)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This library interface provides a means to establish or view resource limit information from the kernel based on the following arguments:

id The *pid*, *sid*, or *uid* corresponding to the *rsarray(2)* field. A 0 indicates the current *pid*, *sid*, or *uid*.

rsarray An integer array that matches the resclim structure defined in /usr/include/sys/resource.h. The array elements and their possible values are as follows:

Array Element	Values
rsarray(1)	Specified resource. Valid options are as follows: 1 – CPU time limits (see CAUTIONS)
rsarray(2)	Resource category. Valid options are as follows: 1 – Process 3 – Session 4 – User ID 6 – Session process

rsarray(3) Resource type. Valid options are as follows:
0 – Absolute limit
1 – Hard limit
2 – Soft limit

rsarray(4) Resource action. Valid options are:
1 – Terminate
2 – Checkpoint and terminate

rsarray(5) Resource used. The accumulated resource used for the resource category. For CPU time, this value is in seconds (see CAUTIONS).

rsarray(6) Resource limit value for the absolute limit type. Used for setting the absolute resource limit value (super user only) or for returning the absolute resource limit value.

rsarray(7) Resource limit value for the hard limit type. Used for setting the hard resource limit value or for returning the hard resource limit value.

rsarray(8) Resource limit value for the soft limit type. Used for setting the soft resource limit value or for returning the soft resource limit value.

rsarray(9) Reserved for future use.

rsarray(10) Reserved for future use.

errstat An integer that contains the error status as defined in `/usr/include/sys/errno.h`. *errstat* is set if an error has occurred with either setting a limit or returning limit information.

Typical Use

The *rsarray* argument contains fields used to establish or view resource limits. `NLIMIT` calls the `nlimit`(3C) routine to set up the `resclim` structure according to the *rsarray* passed to it. `nlimit` then calls `getlim`(2) to obtain information about resource limit values or `setlim`(2) to change limit values.

Obtaining Information about Resource Limits

The *rsarray* elements *rsarray(1)* and *rsarray(2)* must be set in order to return limit values. Element *rsarray*(1) represents the resource to be queried. Currently, only CPU resources are supported, so the value of *rsarray*(1) must be 1.

Element *rsarray*(2) identifies the category of resource to be queried. *rsarray(2)* determines if the *id* argument is a *pid*, *sid*, or a *uid*. An *rsarray(2)* value of 6 (session process) requires an *sid*. Acceptable values are as follows:

Value	Return Information
1	Process limits
3	Session limits
4	User limits
6	Default process limits for the session

If the call succeeds, NLIMIT fills in the missing information in elements *rsarray(4)* through *rsarray(10)*. This includes the following elements:

Array	Return Information
rsarray(4)	Resource action. Returns a value of 1 for terminate or 2 for checkpoint and terminate. This value determines whether, when a hard limit is reached, the process is checkpointed before termination.
rsarray(5)	Resource used. Returns the amount of resource currently accumulated at the time of the call. For CPU time, this value is the amount of CPU seconds accumulated.
rsarray(6)	Absolute resource limit. Returns the absolute resource limit for the specified resource. For CPU time, this value is in seconds.
rsarray(7)	Hard resource limit. Returns the hard resource limit for the specified resource. For CPU time, this value is in seconds.
rsarray(8)	Soft resource limit. Returns the soft resource limit for the specified resource. For CPU time, this value is in seconds.
rsarray(9)	Reserved for future use.
rsarray(10)	Reserved for future use.

Setting Resource Limits

To set a limit value, all *rsarray* elements must be set to either a value or 0.

Element *rsarray(1)* represents the resource for which a limit is to be established. Currently, only CPU resources are supported, so the value of *rsarray(1)* must be 1.

Element *rsarray(2)* identifies the category of resource to be set. *rsarray(2)* determines if the *id* argument is a *pid*, *sid*, or *uid*. An *rsarray(2)* value of 6 (session process) requires an *sid*. Acceptable values are as follows:

Value	Limits to be Set
1	Process limits
3	Session limits
4	User limits
6	Default process limits for the session

Element *rsarray(3)* identifies the type of limit to be set. Acceptable values are as follows:

Value	Limit Type to be Set
0	Absolute limit
1	Hard limit
2	Soft limit

Element *rsarray(4)* determines whether or not the process is checkpointed before termination when a hard limit is reached. By default, when a hard limit is reached, the process is terminated. For the core file to be restartable, the environment variable TRACEBK should be set to 0 prior to running the application. Acceptable values are as follows:

Value	Determination
0	No change
1	Terminate
2	Checkpoint then terminate

In order to set *rsarray(4)*, *rsarray(3)* must be set to limit type of hard (1).

Element *rsarray(5)*, resource accumulated element, is not used when setting limits.

Based on the value in *rsarray(3)* (limit type), the corresponding *rsarray(6)* (absolute limit value), *rsarray(7)* (hard limit value), or *rsarray(8)* (soft limit value) must be set. Only the super user can set absolute limits. Only one limit value can be set with each NLIMIT call.

The NLIMIT library call fails and no information is updated in the *rsarray* array or no resource limits are set if one or more of the following error conditions occur:

Error code	Description
[EFAULT]	The address specified for *rptr* was invalid.
[EINVAL]	One of the arguments contains an invalid value.
[EPERM]	The user ID of the requesting process is not that of a super user.
[EPERM]	An attempt was made to change a limit on a system process; this is not allowed.
[ESRCH]	No processes were found that matched the request.

RETURN VALUES

On successful completion, a value of 0 indicates that the call succeeded and either the *rsarray* array was filled in with appropriate returned values or a new limit was set. An unsuccessful completion returns a value of –1 and *errstat* is set to indicate the error. See intro(2) for an explanation of error numbers.

CAUTIONS

The CPU time limit does not apply when running as root.

EXAMPLES

Following is sample Fortran code sequence that sets the current process CPU hard limit to 500 seconds and the hard limit action to checkpoint and terminate.

```
integer id, rsarr(10), err

id =0
rsarr(1) = 1
rsarr(2) = 1
rsarr(3) = 1
rsarr(4) = 2
rsarr(5) = 0
rsarr(6) = 0
rsarr(7) = 500
rsarr(8) = 0
rsarr(9) = 0
rsarr(10) = 0

iret=nlimit(id,rsarr,err)
if (iret .EQ. -1) then
  write (0,*)' nlimit call failed'
  write (0,*)' errno = ',err
endif
```

SEE ALSO

`nlimit`(1) in the *UNICOS User Commands Reference Manual*, publication SR–2011, for an overview of resource limits

`intro`(2), `getlim`(2), `setlim`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012

`nlimit`(3C) in the *UNICOS System Libraries Reference Manual*, publication SR–2080

NAME

REMARKF – Enters a formatted message in the stderr file

SYNOPSIS

CALL REMARKF(*var*, *fvar*, [*fvar2*, ... *fvar13*])

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

REMARKF enters a formatted message in the stderr file.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

var Variable containing the address of a format statement for ENCODE.

fvar Address of variable.

Up to 12 variables can be passed in arguments 2 through 13. The variables must be of type integer, real, or logical, so that they each occupy only 1 word. The message is prefixed by 'UT009 - ' unless you supply a prefix. To supply the prefix, the characters '*b*-*b*' (*b* equals blank) must appear in columns 6 through 8 of the formatted message.

CAUTIONS

Variables are passed without any data type information. Because of this, variables printed with the A data edit descriptor print as integer Hollerith variables. Variables printed with the D edit descriptor print as a single-precision variable. Variables printed with the G edit descriptor print as octal (typeless) variables.

EXAMPLES

Sample Fortran calling sequences with user-supplied prefixes:

```
10030 FORMAT ('CA001 - ', I4, ' errors')
      ASSIGN 10030 TO LABEL
      CALL REMARKF (LABEL, IERRCNT)

10770 FORMAT ('PD001 - ACCESS ', A8,A7,' ED=', I4, ';')
      ASSIGN 10770 TO LABEL
      CALL REMARKF (LABEL, DN(1), DN(2), ED)
```

Sample Fortran calling sequence without prefix:

```
10550 FORMAT ('LOOP EXECUTED ', I4, ' TIMES')
      ASSIGN 10550 TO LABEL
      CALL REMARKF (LABEL, LOOPCNT)
```

NAME

SENSEBT – Determines whether bidirectional memory transfer is enabled or disabled

SYNOPSIS

CALL SENSEBT(*mode*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SENSEBT determines if bidirectional memory transfer is enabled or disabled.

mode Transfer mode:

= 1 Bidirectional memory transfer is enabled

= 0 Bidirectional memory transfer is disabled

NAME

UNAME – Returns name of current operating system (Fortran interface to uname(2))

SYNOPSIS

The uname(2) system call can be called from Fortran as a function:

```
CHARACTER sys*n1, node*n2, rel*n3, ver*n4, mach*n5
INTEGER  UNAME, i
i =  UNAME(sys, node, rel, ver, mach)
```

IMPLEMENTATION

Cray PVP systems (except CRAY T90 series)

DESCRIPTION

UNAME returns information identifying the current operating system. It can be called as either a subroutine or a function.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

The arguments are as follows:

sys Character or integer variable to receive the name of the current operating system.

node Character or integer variable to receive the name by which this system is known on a communications network.

rel Character or integer variable to receive the name of the operating system release.

ver Character or integer variable to receive the name of the operating system release version.

mach Character or integer variable to receive the name of the hardware on which this system is running.

i Return value, of type integer (when called as a function):

- `-2` `UNAME` called with too many parameters (the first 5 parameters remain)
- `-1` The `uname`(2) system call failed
- `0` `UNAME` called with no parameters
- `1-5` `UNAME` returned 1 to 5 values

The character variables receive as much of the selected field as will fit. If the field is longer it will be truncated; if shorter, the character variable will be padded with blanks.

The integer variables receive no more than the first 8 characters of the appropriate field, left-justified and zero-filled.

The `PXFUNAME`(3F) routine provides similar functionality and is available on all Cray Research systems.

NOTES

`PXFUNAME`(3F) provides similar functionality and is available on Cray PVP and Cray MPP systems.

SEE ALSO

`uname`(2) in the *UNICOS System Calls Reference Manual*, publication SR–2012 for a description of the content and length of the various `uname` fields

NAME

INTRO_SORTSEARCH – Introduction to sorting and searching routines

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

The ISMAX routine runs on all Cray Research systems. The remainder of the sorting and searching routines run on Cray PVP systems only.

Sorting Routines

The following table contains the purpose, name, and entry of each sorting routine. The *entry* is the name of the man page that contains documentation for the routine(s) listed.

Sorting routines:

Purpose	Name	Entry
Performs a distribution counting sort on the elements of an integer vector	ISORTD	ISORTD
Performs a Batcher's Odd/Even Merge sort on the elements of a real or integer general vector	SSORTB ISORTB	SSORTB
Performs an internal fixed-length record sort using radix sort algorithm	ORDERS	ORDERS

Searching Routines

The searching routines are grouped into maximum or minimum element search routines and vector search routines.

Maximum or Minimum Element Search Routines

The maximum and minimum element search routines search a vector for the largest or smallest element and return its index.

Note: The index location returned by each of these routines is in relation to the number of elements chosen to be searched, rather than the total number of elements in the vector. For example, if you choose to search every second element in a 15-element vector, 8 values will be searched and the index location of the target returned will be the target's position relative to the 8 searched values, not relative to the 15 total elements. That is, the returned index is relative, not absolute.

The following table contains the purpose, name, and entry of each maximum or minimum element search routine. The *entry* is the name of the man page containing documentation for the routine(s) listed.

Maximum or minimum element search routines:

Purpose	Name	Entry
Searches for the maximum or minimum value in subfields of a vector element	`INFLMAX` `INFLMIN`	`INFLMAX`
Searches an integer vector for the maximum or minimum value	`INTMAX` `INTMIN`	`INTMAX`

Searches a vector for the first occurrence of the maximum or minimum absolute value	`ISAMAX`
	`ICAMAX`
	`ISAMIN`
		`ISAMAX`

Searches a real vector for the first occurrence of the maximum or minimum value	`ISMAX` `ISMIN`	`ISMAX`

Vector Search Routines

Vector search routines have one of the following functions:

- To return the number of leading occurrences of an object in a vector
- To search for an object in a vector

To return the number of leading occurrences of an object in a vector: These integer functions return the number of occurrences of a given relation in a vector.

Note: The number of occurrences returned is relative to the number of elements chosen to be searched, not the absolute number of elements in the vector.

IILZ returns the number of zero values in a vector before the first nonzero value.

ILLZ returns the number of leading elements of a vector that do not have the sign bit set.

ILSUM returns the number of .TRUE. values in a vector declared logical.

To search for an object in a vector: These routines return the index (or indices) of the element(s) found.

ISRCH functions search a vector for the first element (or element subfield) that has a specified logical relationship to a target.

Note: As previously described for the Maximum or Minimum Element Search Routines, the index returned by these functions and routines is relative, not absolute.

The WHEN routines search a vector for all elements (or element subfields) that have a specified logical relationship to a target.

The CLUS routines search a vector for clusters of values that have a specified logical relationship to a target.

The OSRCHI and OSRCHF routines search an ordered vector for the first element (or element subfield) that has a specified logical relationship to a target.

The following table contains the purpose, name, and entry of each vector search routine. The *entry* is the name of the man page containing documentation for the routine(s) listed.

Vector search routines:

Purpose	Name	Entry
Returns the number of leading occurrences of an object in a vector	IILZ ILLZ ILSUM	IILZ
Searches a vector for clusters of values equal or not equal to a target	CLUSEQ CLUSNE	CLUSEQ
Searches a real vector for clusters of values that are less than, less than or equal to, greater than, or greater than or equal to a target	CLUSFLT CLUSFLE CLUSFGT CLUSFGE	CLUSFLT

Purpose	Name	Entry
Searches an integer vector for clusters of values that are less than, less than or equal to, greater than, or greater than or equal to a target	CLUSILT CLUSILE CLUSIGT CLUSIGE	CLUSILT
Searches a vector for the first element equal or not equal to a target	ISRCHEQ ISRCHNE	ISRCHEQ
Searches a real vector for the first element that is less than, less than or equal to, greater than, or greater than or equal to a real target	ISRCHFLT ISRCHFLE ISRCHFGT ISRCHFGE	ISRCHFLT
Searches an integer vector for the first element that is less than, less than or equal to, greater than, or greater than or equal to an integer target	ISRCHILT ISRCHILE ISRCHIGT ISRCHIGE	ISRCHILT
Searches a vector for the first element whose subfield is equal or not equal to a target	ISRCHMEQ ISRCHMNE	ISRCHMEQ
Searches a vector for the first element whose subfield is less than, less than or equal to, greater than, or greater than or equal to a target	ISRCHMLT ISRCHMLE ISRCHMGT ISRCHMGE	ISRCHMLT
Searches an ordered vector for the first location that contains a target	OSRCHI OSRCHF	OSRCHI
Searches an ordered integer vector for the first element having a subfield equal to an integer target	OSRCHM	OSRCHM
Searches a vector for all elements equal or not equal to a target	WHENEQ WHENNE	WHENEQ
Searches a real vector for all elements that are less than, less than or equal to, greater than, or greater than or equal to a real target	WHENFLT WHENFLE WHENFGT WHENFGE	WHENFLT

Purpose	Name	Entry
Searches an integer vector for all elements that are less than, less than or equal to, greater than, or greater than or equal to an integer target	WHENILT WHENILE WHENIGT WHENIGE	WHENILT
Searches a vector for all elements whose subfields are equal or not equal to a target	WHENMEQ WHENMNE	WHENMEQ
Searches a vector for all elements whose subfields are less than, less than or equal to, greater than, or greater than or equal to a target	WHENMLT WHENMLE WHENMGT WHENMGE	WHENMLT

NAME

ISMAX, ISMIN – Searches a real vector for the first occurrence of the maximum or minimum value

SYNOPSIS

index = ISMAX (*n*, *x*, *incx*)

index = ISMIN (*n*, *x*, *incx*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

ISMAX searches a real vector for the first occurrence of the maximum value; ISMIN searches a real vector for the first occurrence of the minimum value.

ISMAX returns the first index *i* such that

$x_i = \max x_j \quad for\ all\ j = 1, \ldots, n$

where x_j is an element of a real vector.

ISMIN returns the first index *i* such that

$x_i = \min x_j \quad for\ all\ j = 1, \ldots, n$

where x_j is an element of a real vector.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
First index of maximum or minimum value.

n Integer. (input)
Number of elements to process in the vector to be searched.
If $n \leq 0$, ISMAX and ISMIN return 0.

x Real array containing the vector to be searched. (input)
Array *x* is of dimension:

$$1+(n-1)*|incx|$$

incx Integer. (input)
Increment between elements of *x*.

NOTES

This subroutine executes on a single processor and uses private data only.

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$$

The desired value is at:

$x(1+(index-1)*incx)$ when $incx > 0$
$x(1+(index-n)*incx)$ when $incx < 0$

NAME

CLUSEQ, CLUSNE – Searches a vector for clusters of values equal or not equal to a target

SYNOPSIS

```
CALL CLUSEQ (n, x, incx, target, index, nn)
CALL CLUSNE (n, x, incx, target, index, nn)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CLUSEQ searches a real or integer vector for clusters of values equal to a real or integer target.

CLUSNE searches a real or integer vector for clusters of values not equal to a real or integer target.

The indices of the beginning and end of these clusters are returned in the array *index*. The number of occurrences of these clusters is returned in *nn*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements (64-bit words) of the searched array.

target Real or integer. (input)
Value for which array is searched.

index Integer array of dimension (2, *n*). (output)
On exit, contains the indices in *x* at which each cluster starts and stops. The index of the first element is 1.

nn Integer. (output)
Number of matches found.

NOTES

Searching for the cluster allows vectorization. Before using these routines, users should know that the logical search results in clusters of finds.

If the size of each cluster is 1, this routine operates the same as WHENEQ(3F).

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

NAME

CLUSFLT, CLUSFLE, CLUSFGT, CLUSFGE – Searches a real vector for clusters of values with a specified logical relationship to a real target

SYNOPSIS

```
CALL CLUSFLT (n, x, incx, target, index, nn)
CALL CLUSFLE (n, x, incx, target, index, nn)
CALL CLUSFGT (n, x, incx, target, index, nn)
CALL CLUSFGE (n, x, incx, target, index, nn)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CLUSFLT searches a real vector for clusters of values less than a real target.

CLUSFLE searches a real vector for clusters of values less than or equal to a real target.

CLUSFGT searches a real vector for clusters of values greater than a real target.

CLUSFGE searches a real vector for clusters of values greater than or equal to a real target.

The indices of the beginning and end of these clusters are returned in the array *index*. The number of occurrences of these clusters is returned in *nn*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements to be searched.

x Real array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

target Real. (input)
Value for which array is searched.

index Integer array of dimension (2, *n*). (output)
On exit, contains the indices in *x* where the cluster(s) starts and stops. The index of the first element is 1.

nn Integer. (output)
Number of matches found.

NOTES

Searching for the cluster allows vectorization. Before using these routines, you should know that the logical search results in clusters of finds.

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

NAME

CLUSILT, CLUSILE, CLUSIGT, CLUSIGE – Searches an integer vector for clusters of values with a specified logical relationship to an integer target

SYNOPSIS

```
CALL CLUSILT (n, x, incx, itarget, index, nn)
CALL CLUSILE (n, x, incx, itarget, index, nn)
CALL CLUSIGT (n, x, incx, itarget, index, nn)
CALL CLUSIGE (n, x, incx, itarget, index, nn)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

CLUSILT searches an integer vector for clusters of values less than an integer target.

CLUSILE searches an integer vector for clusters of values less than or equal to an integer target.

CLUSIGT searches an integer vector for clusters of values greater than an integer target.

CLUSIGE searches an integer vector for clusters of values greater than or equal to an integer target.

The indices of the beginning and end of these clusters are returned in the array *index*. The number of occurrences of these clusters is returned in *nn*.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements to be searched.

x Integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

itarget Integer. (input)
Value for which the array is searched.

index Integer array of dimension (2, *n*). (output)
On exit, contains the indices in *x* where the cluster(s) starts and stops. The index of the first element is 1.

nn Integer. (output)
Number of matches found.

NOTES

Searching for the cluster allows vectorization. Before using these routines, you should know that the logical search will result in clusters of finds.

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

NAME

IILZ, ILLZ, ILSUM – Returns number of leading occurrences of an object in a vector

SYNOPSIS

kount = IILZ (*n*, *x*, *incx*)

kount = ILLZ (*n*, *x*, *incx*)

kount = ILSUM (*n*, *x*, *incx*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

IILZ returns the number of zero values in a vector before the first nonzero value. The vector can be of type integer, real, or logical.

ILLZ returns the number of leading elements of a vector that do not have the sign bit set. The vector can be of type integer, real, or logical.

ILSUM returns the number of .TRUE. values in a vector declared logical.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

kount Integer. (output)
Number of leading occurrences of an object in the vector.

n Integer. (input)
Number of vector elements to process.

x Integer, real, or logical array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of *x*.

NOTES

When scanning backward (*incx* < 0), both `IILZ` and `ILLZ` start at the end of the vector and move backward, as follows:

$x(1 - incx*(n-1)), x(1-incx*(n-2)),\ldots, x(1)$

If *x* is of type logical, `IILZ` returns the number of FALSE values before encountering the first .TRUE. value.

NAME

INFLMAX, INFLMIN – Searches for the maximum or minimum value in subfields of a vector element

SYNOPSIS

index = INFLMAX (*n*, *x*, *incx*, *mask*, *shift*)

index = INFLMIN (*n*, *x*, *incx*, *mask*, *shift*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

INFLMAX searches for the maximum value in subfields of a vector element.

INFLMIN searches for the minimum value in subfields of a vector element.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index in *x* where maximum or minimum occurs. The index of the first element is 1.

n Integer. (input)
Number of elements to be searched; length of the array.

x Real, integer, or logical array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the table to be searched.

incx Integer. (input)
Increment between elements of *x*.

mask Integer. (input)
Right-justified mask used for masking the table vector.

shift Integer. (input)
Number of bits to right shift the table vector before masking.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

x(1 – *incx** (n –1)), x (1–*incx** (n –2)),. . ., x (1)

The desired element is at:

x(1+(*index*– 1)**incx*) when *incx* > 0

x(1+(*index*– *n*)**incx*) when *incx* < 0

NAME

INTMAX, INTMIN – Searches an integer vector for the maximum or minimum value

SYNOPSIS

index = INTMAX (*n*, *x*, *incx*)

index = INTMIN (*n*, *x*, *incx*)

IMPLEMENTATION

All Cray Research systems

DESCRIPTION

INTMAX searches an integer vector for the maximum value. INTMIN searches an integer vector for the minimum value.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index in *x* where maximum or minimum occurs. The index of the first element is 1.

n Integer. (input)
Number of elements to be searched.

x Integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of *x*.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

x(1 – *incx** (n –1)), x (1–*incx** (n –2)),. . ., x (1)

The desired element is at:

x(1+(*index*– 1)**incx*) when *incx* > 0

x(1+(*index*– *n*)**incx*) when *incx* < 0

NAME

ISAMAX, ICAMAX, ISAMIN – Searches a vector for the first occurrence of the maximum or minimum absolute value

SYNOPSIS

index = ISAMAX (*n*, *x*, *incx*)

index = ICAMAX (*n*, *x*, *incx*)

index = ISAMIN (*n*, *x*, *incx*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ISAMAX searches a real vector for the first occurrence of the maximum absolute value.

ICAMAX searches a complex vector for the first occurrence of the maximum absolute value.

ISAMIN searches a real vector for the first occurrence of the minimum absolute value.

ISAMAX returns the first index *i* such that

$$|x_i| = \max |x_j| : j = 1, \ldots, n$$

where x_j is an element of a real vector.

ICAMAX determines the first index *i* such that

$$|Real(x_i)| + |Imag(x_i)| = \max \left(|Real(x_j)| + |Imag(x_j)| \right) : j = 1, \ldots, n$$

where x_j is an element of a complex vector.

ISAMIN returns the first index *i* such that

$$|x_i| = \min |x_j| : j = 1, \ldots, n$$

where x_j is an element of a real vector.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These integer functions have the following arguments:

index Integer. (output)
First index of maximum or minimum absolute value.

n Integer. (input)
Number of elements to process in the vector to be searched.
If $n \leq 0$, ISAMAX, ICAMAX, and ISAMIN return 0.

x ISAMAX, ISAMIN: Real array of dimension $(n-1)*|incx|+1$. (input)
ICAMAX: Complex array of dimension $(n-1)*|incx|+1$. (input)

Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of *x*.

NOTES

This subroutine executes on a single processor and uses private data only.

When scanning backward ($incx < 0$), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

The largest absolute value is:

abs $(x(1+(index-1)*incx))$ when $incx > 0$

abs $(x(1+(index-n)*incx))$ when $incx < 0$

ISAMAX and ICAMAX are Level 1 Basic Linear Algebra Subprograms (Level 1 BLAS).

NAME

ISORTD – Performs a distribution counting sort on the elements of an integer vector

SYNOPSIS

CALL ISORTD (*ad*, *n*, *l*, *h*, *x*, *incx*, *index*, *incd*, *count*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

This routine performs an indexed sort on the elements of an integer vector in ascending or descending order. An *indexed* sort returns the result in an index array, leaving the input vector unchanged. This sort is applicable when all values fall into the range $l \le x(i) \le h$ and $h - l$ is small. When this is true, the ISORTD algorithm is of O(*n*) complexity and does not entail the startup found in ORDERS.

The following calls illustrate the sorts provided by this routine.

CALL ISORTD('A', N, L, H, X, INCX, INDEX, INCD, COUNT) – Performs an indexed sort in ascending order.

CALL ISORTD('D', N, L, H, X, INCX, INDEX, INCD, COUNT) – Performs an indexed sort in descending order.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This routine has the following arguments:

ad Character. (input)
Indicates whether to sort in ascending or descending order.
Valid values are 'a', 'A', 'd', and 'D'.
'a' and 'A' signify ascending order, and 'd' and 'D' signify descending order.
Only the first character is significant; therefore, to enhance clarity, you may use 'ascending' or 'descending' as the value for *ad*, to enhance clarity.

n Integer. (input)
Number of elements to be sorted.
$n > 0$.

l Integer. (input)
Lower bound on the values of the input array.

h Integer. (input)
Upper bound on the values of the input array.

x Integer array of dimension $(n-1)*incx+1$. (input)
Array *x* contains the values to be sorted.

incx Integer. (input)
Increment between elements of the input array.
$incx > 0$.

index Integer array of dimension $(n-1)*incd+1$. (output)
Array *index* holds the results of the sort.

On exit, *x* may be accessed in sorted order using *index*, as follows:

$$xsorted(k) = x(1 + (index(1 + (k-1)*incd) - 1) * incx), \text{ for all } k = 1, 2, \ldots, n$$

incd Integer. (input)
Increment for the elements of *index*.
$incd > 0$.

count Integer. (output (scratch))
Work array of dimension at least $w = (h - l)$ used to count occurrences of values.

NOTES

Negative increments are never required because *ad* specifies ascending or descending order.

The increment in *count* is 1.

This sort is stable.

EXAMPLES

Suppose that you want to sort every third element of the array `X` into ascending order. The values of `X` range from 61 to 1010 and consist of a small subset of these. You want these to go into column 4 of a matrix of indices, `MID`. `INCD`=1, the increment between elements of a column in `MID`. The call is as follows:

```
CALL ISORTD( 'A', N, 61, 1010, X, 3, MID( 1,4 ), INCD, COUNT )
```

Following are two examples that show how to sort rows or columns of a matrix into the corresponding rows or columns of an index matrix. After the sort is complete, you can access the matrix in sorted order through the matrix `MID`. The examples in the code are sorting each

row of a matrix into ascending order and then sorting each column of a matrix into descending order.

Example 1: Row-by-row sort (ascending order):

```
      PROGRAM TISORTD1
*-----------------------------------------------------------------------
*     ..Parameters..
        INTEGER ROW, COL, L, H
        PARAMETER( ROW=4, COL=5, L=1, H=4 )

*     ..Scalar vars..
        INTEGER I, J

*     ..Vector vars..
        INTEGER COUNT( L:H ), MID( ROW,COL ), IA( ROW,COL )

*-----------------------------------------------------------------------
*     ..Begin execution..

*       Initialize IA() with numbers between L and H.
        DO I = 1, ROW
          DO J = 1, COL
            IA( I,J ) = 1 + MOD( I*J, H )
          END DO
        END DO

*-----------------------------------------------------------------------
*       Sort each row in ascending order.
        DO I = 1, ROW
          CALL ISORTD( 'ASCENDING', COL, L, H, IA( I,1 ), ROW,
     &                 MID( I,1 ), ROW, COUNT )
        END DO

*-----------------------------------------------------------------------
*     ..I/O..
        PRINT *, "     ... IA() prior to row-by-row sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( IA( I,J ), J=1,COL )
        END DO
```

```
        PRINT *, "     ... IA() after row-by-row sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( IA( I,MID( I,J ) ), J=1,COL )
        END DO

*     ..Format Statements..
10      FORMAT( 5X, 5( I5, 2X ) )
      END
```

The results of program `TISORTD1` are as follows:

```
      ... IA() prior to row-by-row sort ...
         2      3      4      1      2
         3      1      3      1      3
         4      3      2      1      4
         1      1      1      1      1
      ... IA() after row-by-row sort ...
         1      2      2      3      4
         1      1      3      3      3
         1      2      3      4      4
         1      1      1      1      1
```

Example 2: Column-by-column sort (descending order):

```
      PROGRAM TISORTD2
*-----------------------------------------------------------------------
*     ..Parameters..
        INTEGER ROW, COL, L, H
        PARAMETER( ROW=5, COL=4, L=1, H=4 )

*     ..Scalar vars..
        INTEGER I, J

*     ..Vector vars..
        INTEGER COUNT( L:H ), MID( ROW,COL ), IA( ROW,COL )

*-----------------------------------------------------------------------
*     ..Begin execution..

*       Initialize IA() with numbers between L and H.
        DO I = 1, ROW
          DO J = 1, COL
```

```
            IA( I,J ) = 1 + MOD( I*J, H )
          END DO
        END DO

*-----------------------------------------------------------------------
*       Sort each column in descending order.
        DO J = 1, COL
          CALL ISORTD( 'DESCENDING', ROW, L, H, IA( 1,J ), 1,
     &                 MID( 1,J ), 1, COUNT )
        END DO

*-----------------------------------------------------------------------
*     ..I/O..
        PRINT  *, "     ... IA() prior to column-by-column sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( IA( I,J ), J=1,COL )
        END DO

        PRINT *, "     ... IA() after column-by-column sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( IA( MID( I,J ),J ), J=1,COL )
        END DO

*     ..Format statements
10      FORMAT( 5X, 4( I5, 2X ) )
      END
```

The results of program `TISORTD2` are as follows:

```
... IA() prior to column-by-column sort ...
   2      3      4      1
   3      1      3      1
   4      3      2      1
   1      1      1      1
   2      3      4      1
... IA() after column-by-column sort ...
   4      3      4      1
   3      3      4      1
   2      3      3      1
   2      1      2      1
   1      1      1      1
```

NAME

ISRCHEQ, ISRCHNE – Searches a vector for the first element equal or not equal to a target

SYNOPSIS

index = ISRCHEQ (*n*, *x*, *incx*, *target*)

index = ISRCHNE (*n*, *x*, *incx*, *target*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ISRCHEQ searches a real or integer vector for the first element that is equal to a real or integer target.

ISRCHNE searches a real or integer vector for the first element that is not equal to a real or integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index of the first element equal or not equal to *target*.
If *target* is not found, *n*+1 is returned.
If $n \leq 0$, 0 is returned.

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

target Real or integer. (input)
Value for which to search in the array.

The Fortran equivalent code for ISRCHEQ is as follows:

```
      FUNCTION ISRCHEQ(N,X,INCX,TARGET)
      INTEGER X(*), TARGET
      J=1
      ISRCHEQ=0
      IF(N.LE.0) RETURN
      IF(INCX.LT.0) J=1-(N-1)*INCX
      DO 100 I=1,N
         IF(X(J).EQ.TARGET) GO TO 200
         J=J+INCX
100   CONTINUE
200   ISRCHEQ=I
      RETURN
      END
```

Although used as integers internally, you can use real values of *x* and *target*, because ISRCHEQ and ISRCHEQ are matching bit patterns.

NOTES

ISRCHEQ replaces the ISEARCH routine, but it has an entry point named ISEARCH as well as ISRCHEQ.

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1))$, $x(1-incx*(n-2))$,. . ., $x(1)$

The desired value is at:

$x(1+(index-1)*incx)$ when $incx > 0$

$x(1+(index-n)*incx)$ when $incx < 0$

NAME

ISRCHFLT, ISRCHFLE, ISRCHFGT, ISRCHFGE – Searches a real vector for the first element with a specified logical relationship to a real target

SYNOPSIS

index = ISRCHFLT (*n*, *x*, *incx*, *ftarget*)

index = ISRCHFLE (*n*, *x*, *incx*, *ftarget*)

index = ISRCHFGT (*n*, *x*, *incx*, *ftarget*)

index = ISRCHFGE (*n*, *x*, *incx*, *ftarget*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ISRCHFLT searches a real vector for the first element that is less than a real target.

ISRCHFLE searches a real vector for the first element that is less than or equal to a real target.

ISRCHFGT searches a real vector for the first element that is greater than a real target.

ISRCHFGE searches a real vector for the first element that is greater than or equal to a real target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index of the first element with the specified logical relationship to *ftarget*.
If *ftarget* is not found, *n*+1 is returned.
If $n \leq 0$, 0 is returned.

n Integer. (input)
Number of elements to be searched.

x Real array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

ftarget Real. (input)
Value for which to search in the array.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

The desired value is at:

$x(1+(index-1)*incx)$ when $incx > 0$

$x(1+(index-n)*incx)$ when $incx < 0$

NAME

ISRCHILT, ISRCHILE, ISRCHIGT, ISRCHIGE – Searches an integer vector for the first element with a specified logical relationship to an integer target

SYNOPSIS

index = ISRCHILT (*n*, *x*, *incx*, *itarget*)

index = ISRCHILE (*n*, *x*, *incx*, *itarget*)

index = ISRCHIGT (*n*, *x*, *incx*, *itarget*)

index = ISRCHIGE (*n*, *x*, *incx*, *itarget*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ISRCHILT searches an integer vector for the first element that is less than an integer target.

ISRCHILE searches an integer vector for the first element that is less than or equal to an integer target.

ISRCHIGT searches an integer vector for the first element that is greater than an integer target.

ISRCHIGE searches an integer vector for the first element that is greater than or equal to an integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index of the first element with the specified logical relationship to *itarget*.
If *itarget* is not found, *n*+1 is returned.
If $n \leq 0$, 0 is returned.

n Integer. (input)
Number of elements to be searched.

x Integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

itarget Integer value searched for in the array. (input)

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

The desired value is at:

$x(1+(index-1)*incx)$ when $incx > 0$

$x(1+(index-n)*incx)$ when $incx < 0$

NAME

ISRCHMEQ, ISRCHMNE – Searches a vector for the first element whose subfield is equal or not equal to a target

SYNOPSIS

index = ISRCHMEQ (*n*, *x*, *incx*, *itarget*, *mask*, *iright*)

index = ISRCHMNE (*n*, *x*, *incx*, *itarget*, *mask*, *iright*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ISRCHMEQ searches a real or integer vector for the first element whose subfield is equal to a real or integer target.

ISRCHMNE searches a real or integer vector for the first element whose subfield is not equal to a real or integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index of the first element whose subfield is equal or not equal to *itarget*.
If *itarget* is not found, *n*+1 is returned.
If $n \leq 0$, 0 is returned.

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

itarget Integer. (input)
Value for which to search in the array.

mask Integer. (input)
Mask of 1's (set bits) from the right. The number of set bits in *mask* is the number of bits in the subfield, which is searched in each element of the input array.

iright Integer. (input)
Number of bits to shift each element of the input array to the right so as to right justify the subfield searched.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

x(1 – *incx** (n –1)), x (1–*incx** (n –2)),. . ., x (1)

The desired value is at:

x(1+(*index*– 1)**incx*) when *incx* > 0

x(1+(*index*– *n*)**incx*) when *incx* < 0

NAME

ISRCHMLT, ISRCHMLE, ISRCHMGT, ISRCHMGE – Searches a vector for the first element whose subfield has a specified logical relationship with a target

SYNOPSIS

index = ISRCHMLT (*n*, *x*, *incx*, *itarget*, *mask*, *iright*)

index = ISRCHMLE (*n*, *x*, *incx*, *itarget*, *mask*, *iright*)

index = ISRCHMGT (*n*, *x*, *incx*, *itarget*, *mask*, *iright*)

index = ISRCHMGE (*n*, *x*, *incx*, *itarget*, *mask*, *iright*)

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ISRCHMLT searches a real or integer vector for the first element whose subfield is less than a real or integer target.

ISRCHMLE searches a real or integer vector for the first element whose subfield is less than or equal to a real or integer target.

ISRCHMGT searches a real or integer vector for the first element whose subfield is greater than a real or integer target.

ISRCHMGE searches a real or integer vector for the first element whose subfield is greater than or equal to a real or integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These functions have the following arguments:

index Integer. (output)
Index of the first element whose subfield has the specified logical relationship with *itarget*.
If *itarget* is not found, *n*+1 is returned.
If $n \leq 0$, 0 is returned.

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

itarget Integer. (input)
Value for which to search in the array.

mask Integer. (input)
Mask of 1's (set bits) from the right. The number of set bits in *mask* is the number of bits in the subfield, which is searched in each element of the input array.

iright Integer. (input)
Number of bits to shift each element of the input array to the right so as to right justify the subfield searched.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

The desired value is at:

$x(1+(index-1)*incx)$ when *incx* > 0

$x(1+(index-n)*incx)$ when *incx* < 0

NAME

ORDERS – Internal, fixed-length record-sorting routine optimized for Cray Research systems

SYNOPSIS

```
CALL ORDERS (mode, iwork, data, index, n[, ireclth, ikeylth, iradsiz])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

ORDERS is an internal, fixed-length record sort routine. ORDERS uses the *radix* sort, which is more commonly known as a *bucket* or *pocket* sort. For this type of sort, the length, in bytes, of the key, and the length, in words, of the record, determines the number of passes made through all of the records. You can perform the sort in O(*n*) operations, and it is stable.

For definitions of major terms used in this document, see the Glossary subsection of the NOTES section.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This routine has the following arguments:

mode Integer. (input/output)
On input, *mode* describes the type of key and indicates an initial ordering of the records, as follows:

- 0 Keys are positive integers or ASCII characters
- 1 Keys are treated as two's complement (signed) integers over the range $[-2^{63}, 2^{63}-1]$
- 2 Keys are treated as floating-point numbers
- 10 Same as *mode* = 0, but the array *index* contains the initial ordering of the records
- 11 Same as *mode* = 1, but the array *index* contains the initial ordering of the records

12 Same as *mode* = 2, but the array *index* contains the initial ordering of the records

On output, if an error is encountered, ORDERS returns a value < 0 in *mode*; otherwise, *mode* remains unchanged. The error values are as follows:

−1 Too few arguments (must be greater than 4)

−2 Too many arguments (must be less than 9)

−3 *ireclth* is not valid (must be > 0)

−4 Invalid *ikeylth* (must be in the range $0 < ikeylth \leq 8*ireclth$)

−5 *iradsiz* is not valid (must be 1 or 2)

−6 *ikeylth* is not valid (must be > 0)

−7 *n* is not valid (must be > 0)

−8 *mode* input value is not valid (must be 0, 1, 2, 10, 11, or 12)

−9 *ikeylth* is not valid (must be 8 for real or integer sort)

iwork Integer array of dimension 256 or 65,536. (workspace)
User-supplied working storage array. The length depends on *iradsiz*, as follows:
If *iradsiz* = 1, the length of *iwork* should be 256.
If *iradsiz* = 2, the length of *iwork* should be 65,536.

data Integer array of dimension (*ireclth, n*). (input)
data contains *n* records of length *ireclth* each.

index Integer array of dimension *n*. (output)
If *mode* = 10, 11, or 12, *index* contains an initial ordering of the records in the array *data*. Thus, *data*(*i*,*index*(1)) is the *i*th word of the first record, and *data*(*i*,*index*(*n*)) is the *i*th word of the last record. Each *index*(*i*) should be in the closed interval (1 . . . *n*) and unique with respect to all other *index*(*i*).

n Integer. (input)
Number of records to be sorted.

ireclth Integer. (input)
Optional argument.
Length of each record in words.
If specified, $1 \leq ireclth$.
The default is 1.

ikeylth Integer. (input)
Optional argument.
Length of each key as a number of 8-bit bytes. (8 bytes = 1 word.)
If specified, must be in the interval ($1 \leq$ *ikeylth* ≤ 8**ireclth*).
If *mode* = 1, *ikeylth* must = 8.
The default is 8.

iradsiz Integer. (input)
Optional argument.
Radix of the sort (the number of 8-bit bytes processed per pass).
If specified, must be 1 or 2.
A radix-2 sort requires fewer passes through the record, but it uses more workspace and has a higher overhead per pass.
The default is 1.

NOTES

ORDERS has the option of processing 1 or 2 bytes of the key per pass through the records. This process halves the number of passes through the record, but at the expense of increased working storage and overhead per pass. ORDERS can sort on several keys within a record by using its multipass capability.

Large Radix Sorting

The number of times the key of each record is read from memory is proportional to *ikeylth*/*iradsiz*. Using ORDERS with *iradsiz* = 2 halves this ratio, because 2 bytes instead of 1 are processed each time the key is read. The disadvantage of halving the number of passes is that the user-supplied working storage array goes from 256 words to 65,536 words and the initialization time of this work array is greater than for the 256-word array. From a performance standpoint, this favors a 1-byte pass for sorting smaller arrays up to approximately 5000 records. For more than 5000 records, however, a 2-byte pass is faster.

Multipass Sorting

Because the array *index* can define an ordering of the records, several calls can be made to ORDERS where the order of the records is that of the previous call. *mode* = 10, 11, or 12 specifies that the array *index* contains an ordering from a previous call to ORDERS. This specification allows sorting of text keys that extend over more than 1 word or keys involving double-precision numbers. This algorithm performs the radix sort over all keys. When using the multipass capability, sort the least-significant word first.

Glossary

This subsection contains definitions for some of the terms used in this man page.

Internal sort

An *internal sort* means that the sort can be performed entirely in core memory.

Record

A *record* is a generalized array element. The term *record* corresponds to the definition used in Pascal. Record also corresponds to the term *struct* in C. From a C or Pascal programmer's perspective, a struct (record) is allowed to contain heterogeneous elements; it can be any mix of integer, real, logical, or character data. It can even contain another struct (record).

This idea translates in limited fashion to Fortran. Although Fortran does not allow this general type of storage scheme, you could envision each element of an array of real numbers as being a record. Furthermore, if you are sorting a character array in which each element is 24 bytes wide, each 24-byte element is also a record.

A *fixed-length record* means that the length of the record cannot vary. The programming construct that allows the length of a data structure to vary is known as a *union* in C. Fixed-length records do not allow this. The length of the record is always the same.

Key

Suppose a Pascal code contains the declaration of an array of records in which each record consists of an integer and a real number. Each component of the record is called a *key*. If you wanted to sort this array based on the values of the integers in each record, the sort is done on the integer key. For Fortran, a limited notion of key applies. In this case, if you were sorting the character array consisting of 24-byte elements, a sort could be done in which a key consists of the high-order 8 bytes. That is, the sort does not necessarily have to apply to an entire element. Rather, the array can be sorted on the high-order 8 bytes of each array element. For example, suppose that you are sorting an array of character strings in dictionary order and that each string consists of 16 letters (bytes). If it were known that the last 8 letters (bytes) consists of the letters xxxxxxxx, the sort need not proceed beyond the eighth letter of the element. Table 1 shows that you do not have to look beyond the first 8 letters to sort these strings.

Table 1: Example of sorting on keys

Unsorted	Sorted
ccccccaaxxxxxxxx	bcccccabxxxxxxxx
cccccccbxxxxxxxx	bcccccgbxxxxxxxx
bcccccgbxxxxxxxx	cccccccaxxxxxxxx
bcccccabxxxxxxxx	ccccccccbxxxxxxxx

Indexed sort

An *indexed sort* means that the original data is not disturbed. The indices in an index array are permuted such that on completion of the sort, the data can be accessed in sorted order through this array. Because ORDERS sorts the keys in increasing order, *data*(*index*(1)) will be the first word of the record with the smallest key, *data*(*index*(2)) will be the first word of the record with the next smallest key, and so on. See Table 2.

Table 2: Example of an indexed sort

Initial Data				After Indexed Sort			
i	x(i)	index(i)	x(index(i))	i	x(i)	index(i)	x(index(i))
1	2	1	2	1	2	1	2
2	3	2	3	2	3	3	2
3	2	3	2	3	2	2	3
4	10	4	10	4	10	6	9
5	17	5	17	5	17	4	10
6	9	6	9	6	9	5	17
7	17	7	17	7	17	7	17

Multipass sorting
Sorting can be done in multiple passes. For example, suppose you wanted to sort a character array consisting of 16 byte-wide elements. The sort could be done in two passes. The first pass could sort on the low-order byte of each element. This byte could be thought of as a key. During the second pass, the sort is done on the high-order byte (key).

Stable sort
A *stable* sort is one in which the original order of records with equal keys is not disturbed. This matters for multipass sorting. In a multipass sort, elements are sorted whose previous byte is the same. For example, suppose you have an array of 2-character strings to be sorted. Table 3 illustrates the progression of a multipass sort of the character array. When complete, the array *inx* will contain the permuted indices.

Table 3: Progression of a multipass sort

	Initial Ordering		First Pass		Second Pass	
i	cx(i)	inx(i)	cx(inx(i))	inx(i)	cx(inx(i))	inx(i)	cx(inx(i))
1	cb	1	cb	2	aa	2	aa
2	aa	2	aa	4	xa	3	ac
3	ac	3	ac	5	ba	5	ba
4	xa	4	xa	6	ca	7	bx
5	ba	5	ba	1	cb	6	ca
6	ca	6	ca	3	ac	1	cb
7	bx	7	bx	7	bx	4	xa

Note the two stages of the sort. During the first stage or pass, the strings are arranged in dictionary order based on the least significant letter only (in this case, a letter is a key); therefore, all of the strings ending with an "a" come first, then those ending with a "b", and so forth. The most-significant byte is sorted on during the second pass. This is called *least significant digit first radix sorting*.

Also notice how this data was moved around. During the first pass, the strings were rearranged so that if originally the string "xa" preceded the string "ba", that ordering is preserved when moving the data. This is what is meant by a *stable sort*.

EXAMPLES

This section contains examples of the following uses of ORDERS:

- Sorting an array of 20 floating-point numbers, using default values for *ireclth*, *ikeylth*, and *iradsiz*
- Sorting double-precision numbers

Sorting an array of 20 floating-point numbers, using default values for *ireclth*, *ikeylth*, and *iradsiz*

```
      PROGRAM TRORD
      IMPLICIT NONE

*     ..Parameters..
        INTEGER LDIM, WKSPACE
        PARAMETER( LDIM=20 )
        PARAMETER( WKSPACE=65536 )

*     ..Variables..
        INTEGER I, MODE, INDEX( LDIM ), IWORK( WKSPACE )
        REAL    VEC( LDIM )

*     ..Data initialization..
        DATA MODE / 2 /     ! Sort floating point numbers
        DATA VEC / 55.,54.,53.,52.,51.,
     &             -1.,-2.,-3.,-4.,-5.,
     &             105.,104.,103.,102.,101.,
     &             -201.,-202.,-203.,-204.,-205 /

*     ..External routines..
        EXTERNAL ORDERS

*     ..Begin Execution..
        PRINT *, "     ... VEC() prior to sort ..."
        DO I = 1, LDIM
          WRITE( 6,10 ) VEC( i )
        END DO

        CALL ORDERS( MODE, IWORK, VEC, INDEX, LDIM )

        PRINT *, " "
```

```
          PRINT *, "     ... VEC() after sort ..."
          DO I = 1, LDIM
            WRITE( 6,10 ) VEC( INDEX( i ) )
          END DO

10        FORMAT( 5X, F10.3 )
        STOP
        END
```

The results of program TRORD are as follows:

```
... VEC() prior to sort ...
  55.000
  54.000
  53.000
  52.000
  51.000
  -1.000
  -2.000
  -3.000
  -4.000
  -5.000
 105.000
 104.000
 103.000
 102.000
 101.000
-201.000
-202.000
-203.000
-204.000
-205.000

... VEC() after sort ...
-205.000
-204.000
-203.000
-202.000
-201.000
  -5.000
  -4.000
```

```
  -3.000
  -2.000
  -1.000
  51.000
  52.000
  53.000
  54.000
  55.000
 101.000
 102.000
 103.000
 104.000
 105.000
```

Sorting Double-precision Numbers

This code example presents the sort of double-precision numbers. Double-precision numbers consist of 2 words per number. To sort them, you must use the multipass capability of ORDERS.

The data is equivalenced to an integer array so that for each pass the data can be treated as unsigned integers. If the number is negative, the negative sign has to be carried temporarily to the low-order component so that the direction of the sort will be consistent for both passes. This example uses random numbers.

```
      PROGRAM TDORD
*     ..Parameters..
        INTEGER LDIM, WKSPACE
        DOUBLE PRECISION BOT, TOP
        PARAMETER( LDIM=20 )
        PARAMETER( WKSPACE=256 )
        PARAMETER( BOT=1.0D-12 )
        PARAMETER( TOP=2.0D-12 )

*     ..Variables..
        INTEGER I, MODE, INDEX( LDIM ), IWORK( WKSPACE )
        INTEGER IVEC( 2*LDIM )
        DOUBLE PRECISION VEC( LDIM )
        EQUIVALENCE( VEC,IVEC )

*     ..External routines..
        EXTERNAL ORDERS

*     ..Data initialization..
        DATA IRECLTH / 2 /  ! Length of a record.  In this case, a
*                           !  record is two CRI words long.
*                           !  Length of a record is in terms of
*                           !  CRI words.
        DATA IKEYLTH / 8 /  ! Length of a key is 8 bytes.
        DATA IRADSIZ / 1 /  ! Do sort 1 byte at a time.

*     ..Statement function..
*       Generate random double precision numbers between the value A
*       and the value B not including A or B.
        DOUBLE PRECISION A, B, DRAN
        DRAN( A,B ) = ( B-A )*DBLE( RANF() ) + A

*     ..Begin execution..

*       Load VEC() with positive random numbers that differ only
*       in the low-order bits.
        DO I = 1, LDIM/2
          VEC( I ) = 1.0D2 + DRAN( BOT,TOP )
        END DO
```

```
*       Load VEC() with negative random numbers that differ only
*       in the low-order bits.
        DO I = LDIM/2+1, LDIM-1
          VEC( I ) = -1.0D2 - DRAN( BOT,TOP )
        END DO

*       Make sure there is a number and it's negative version in
*       the array.
        VEC( LDIM ) = -VEC( 1 )

*       Change the second word to have the same sign as the first
*       word.  Otherwise, if the number is negative, the
*       sort of the second word would be backwards.
        DO I = 1, 2*LDIM, 2
          IF( IVEC( I ) .LE. 0 ) IVEC( I+1 ) = -IVEC( I+1 )
        END DO

*       Sort low-order bits, first.
*       IVEC() had to be used because there is no way to access
*       the low-order bits using VEC().
        MODE = 1
        CALL ORDERS( MODE, IWORK, IVEC(2), INDEX, LDIM, IRECLTH,
     &               IKEYLTH, IRADSIZ )

*       Restore second word to original form.
        DO I = 1, 2*LDIM, 2
          IVEC( I+1 )= ABS( IVEC( I+1 ) )
        END DO

*       Sort high-order bits.
        MODE = 12
        CALL ORDERS( MODE, IWORK, VEC, INDEX, LDIM, IRECLTH,
     &               IKEYLTH, IRADSIZ )

*       I/O
        WRITE( 6,10 )
        DO I = 1, LDIM
          WRITE( 6,30 ) VEC( I ), IVEC(2*I-1), IVEC(2*I)
        END DO
```

```
      PRINT *, " "
      WRITE( 6,20 )
      DO I = 1, LDIM
        J = INDEX(I)
        WRITE( 6,30 ) VEC( J ), IVEC(2*J-1), IVEC(2*J)
      END DO

10    FORMAT( 4X, "...Values in VEC() before sort...", 5X,
     &  "...Equivalent hex values in IVEC()..." )
20    FORMAT( 4X, "...Values in VEC() after sort...", 6X,
     &  "...Equivalent hex values in IVEC()..." )
30    FORMAT( D40.30, 2X, Z16, 1X, Z16 )
      END
```

The following output presents an array of random numbers that were generated. This array is sorted using the multipass option. The result is then presented. Other than sign, the numbers differ from one another only in the low-order bits. This is to show that, other than the sign of the numbers, the sort will really take place based on these values.

```
 ...Values in VEC() before sort...     ...Equivalent hex values in IVEC()...
 0.100000000000001580113648579586E+03  4007C80000000003 00007986602A5AAD
 0.100000000000001950512734980764E+03  4007C80000000004 00004A0A82782AC9
 0.100000000000001786371425330602E+03  4007C80000000003 0000EDA34101DFE7
 0.100000000000001297620264003728E+03  4007C80000000002 0000DA7EC9D47284
 0.100000000000001453699900298492E+03  4007C80000000003 0000325C3C0AA41E
 0.100000000000001006261941606187E+03  4007C80000000002 00003679A1040E89
 0.100000000000001275736426383874E+03  4007C80000000002 0000CE2CFEB9DCBA
 0.100000000000001305650943870477E+03  4007C80000000002 0000DF04219F5276
 0.100000000000001689100710749872E+03  4007C80000000003 0000B6E1110D45E2
 0.100000000000001382662238656297E+03  4007C80000000003 00000A5EA0E9EB86
-0.100000000000001132902705496380E+03  C007C80000000002 00007DC47C8993AF
-0.100000000000001831857903209073E+03  C007C80000000004 0000073E8BD6FD21
-0.100000000000001582979795830740E+03  C007C80000000003 00007B236E55C827
-0.100000000000001098625338337415E+03  C007C80000000002 00006A7898E5FDEB
-0.100000000000001276548455133567E+03  C007C80000000002 0000CEA2054C9046
-0.100000000000001620446027796948E+03  C007C80000000003 0000903AE251EF64
-0.100000000000001083502966833808E+03  C007C80000000002 000061F53BDD54E3
-0.100000000000001990377120595685E+03  C007C80000000004 0000607B92B524B0
-0.100000000000001979346943443064E+03  C007C80000000004 00005A45F4FF5F3C
-0.100000000000001580113648579586E+03  C007C80000000003 00007986602A5AAD

 ...Values in VEC() after sort...      ...Equivalent hex values in IVEC()...
-0.100000000000001990377120595685E+03  C007C80000000004 0000607B92B524B0
-0.100000000000001979346943443064E+03  C007C80000000004 00005A45F4FF5F3C
-0.100000000000001831857903209073E+03  C007C80000000004 0000073E8BD6FD21
```

```
-0.10000000000000016204460277969 48E+03  C007C80000000003 0000903AE251EF64
-0.10000000000000015829797958307 40E+03  C007C80000000003 00007B236E55C827
-0.10000000000000015801136485795 86E+03  C007C80000000003 00007986602A5AAD
-0.10000000000000012765484551335 67E+03  C007C80000000002 0000CEA2054C9046
-0.10000000000000011329027054963 80E+03  C007C80000000002 00007DC47C8993AF
-0.10000000000000010986253383374 15E+03  C007C80000000002 00006A7898E5FDEB
-0.10000000000000010835029668338 08E+03  C007C80000000002 000061F53BDD54E3
 0.10000000000000010062619416061 87E+03  4007C80000000002 00003679A1040E89
 0.10000000000000012757364263838 74E+03  4007C80000000002 0000CE2CFEB9DCBA
 0.10000000000000012976202640037 28E+03  4007C80000000002 0000DA7EC9D47284
 0.10000000000000013056509438704 77E+03  4007C80000000002 0000DF04219F5276
 0.10000000000000013826622386562 97E+03  4007C80000000003 00000A5EA0E9EB86
 0.10000000000000014536999002984 92E+03  4007C80000000003 0000325C3C0AA41E
 0.10000000000000015801136485795 86E+03  4007C80000000003 00007986602A5AAD
 0.10000000000000016891007107498 72E+03  4007C80000000003 0000B6E1110D45E2
 0.10000000000000017863714253306 02E+03  4007C80000000003 0000EDA34101DFE7
 0.10000000000000019505127349807 64E+03  4007C80000000004 00004A0A82782AC9
```

NAME

OSRCHI, OSRCHF – Searches an ordered vector for the first location that contains a target

SYNOPSIS

```
CALL OSRCHI (n, x, incx, target, index, iwhere, inum)
CALL OSRCHF (n, x, incx, target, index, iwhere, inum)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

OSRCHI searches an ordered integer vector for the first location that contains an integer target.

OSRCHF searches an ordered real vector for the first location that contains a real target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of array elements to be searched.

x OSRCHI: Integer array of dimension $(n-1)*|incx|+1$. (input)
OSRCHF: Real array of dimension $(n-1)*|incx|+1$. (input)

Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

target OSRCHI: Integer. (input)
OSRCHF: Real. (input)

Value for which to search in the array.

index Integer. (output)
Index of the first element equal to *target*.
If *target* is not found, *index* = *n*+1 .
If $n \leq 0$, *index* = 0 .

iwhere Integer. (output)
Index of the first element greater than or equal to *target*.
If *target* is found, *iwhere* = *index* .
If *target* is greater than the last element of the array, *iwhere* = *n*+1 .
If $n \leq 0$, *iwhere* = 0 .

inum Integer. (output)
Number of copies of *target* found in the array

NOTES

Searching always begins at the lowest value in the ordered array. Even if the target is not found, the index of the location that would contain the target is returned in *iwhere*. The total number of occurrences of the target in the array (*inum*) can also be returned.

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

Thus, a positive increment indicates an ascending array, while a negative increment indicates a descending array.

The *index* value is at:

$x(1+(index-1)*incx)$ when $incx > 0$

$x(1+(index-n)*incx)$ when $incx < 0$

The *iwhere* value is at:

$x(1+(iwhere-1)*incx)$ when $incx > 0$

$x(1+(iwhere-n)*incx)$ when $incx < 0$

NAME

OSRCHM – Searches an ordered integer vector for the first element whose subfield is equal to an integer target

SYNOPSIS

```
CALL OSRCHM (n, x, incx, itarget, mask, iright, index, iwhere, inu)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

OSRCHM searches an ordered integer vector and returns the index of the first element for which the subfield defined by *mask* and *iright* is equal to an integer target. For the search to be successful, the input vector must be ordered **by subfield**, not by the whole integer.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

This routine has the following arguments:

n Integer. (input)
Number of array elements to be searched

x Integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the array to be searched.

itarget Integer. (input)
Value for which to search in the array.

mask Integer. (input)
Mask set from the right side of the field of interest in array *x*.

iright Integer. (input)
Amount to right-shift array *x* to position the field of interest at right side of word.

index Integer. (output)
Index of the first element whose subfield is equal to *itarget*.
If *itarget* is not found, *index* = *n*+1 .
If $n \leq 0$, *index* = 0 .

iwhere Integer. (output)
Index of the first element whose subfield is greater than or equal to *itarget*.
If *itarget* is found, *iwhere* = *index* .
If *itarget* is greater than the subfield of the last element in the array, *iwhere* = *n*+1 .
If $n \leq 0$, *iwhere* = 0 .

inum Integer. (input and output)
If *inum* equals 0 on input, *inum* is still 0 on output. If *inum* does not equal 0 on input, *inum* is the number of array elements whose subfields match *itarget* on output; that is, *inum* returns a value only if requested and at least one target value is found in the array; otherwise, it will always be 0.

NOTES

Searching always begins at the lowest value in the ordered array. Even if the target is not found, the index of the location that would contain the target is returned in *iwhere*. The total number of occurrences of the target in the array (*inum*) can also be returned.

When scanning backward (*incx* < 0), this routine starts at the end of the vector and moves backward, as follows:

$x(1-incx\cdot(n-1)),\ x(1-incx\cdot(n-2)),\ldots,\ x(1)$

Thus, a positive increment indicates an ascending array (with respect to the subfield), while a negative increment indicates a descending array (with respect to the subfield).

The *index* value is at:

$x(1+(index-1)*incx)$ when $incx > 0$

$x(1+(index-n)*incx)$ when $incx < 0$

The *iwhere* value is at:

$x(1+(iwhere-1)*incx)$ when $incx > 0$

$x(1+(iwhere-n)*incx)$ when $incx < 0$

EXAMPLES

As an example of a vector that is ordered by the whole integer value, but not ordered by subfield, consider the following Fortran program:

```
      PARAMETER (N=128)
      PARAMETER (ITARGET=20)
      PARAMETER (INCX=1)
      INTEGER X(N),XWHERE,INDEX,IWHERE,INUM
C
C     EACH SUBFIELD IS THE FIVE RIGHTMOST BITS OF EACH INTEGER
C     ( SUBFIELD(I) = MOD(X(I), 2**5), FOR ALL I = 1, 2, ..., 128 )
C
      PARAMETER (MASK=(2**5)-1,IRIGHT=0)
C
C     X IS AN ASCENDING SEQUENCE
C
      PRINT*
      PRINT*, 'X(i) = 127*i, for all I = 1, 2, ..., 128'
      PRINT*
      DO 10 I=1,128
        X(I)=127*I
10    CONTINUE
C
C     LIST OF ALL LOCATIONS WHERE SUBFIELD MATCHES TARGET
C
      INUM=0
      PRINT*, 'LOCATIONS WHERE MOD(X(i), 2**5) = ',ITARGET
      PRINT*, '(SUBFIELD MATCHES TARGET)'
      DO I=1,128
        IF (IAND(X(I),MASK).EQ.ITARGET) THEN
          PRINT*, 'i = ',I, '     X(',I,') = ', X(I)
          INUM=INUM+1
        ENDIF
      ENDDO
      PRINT*
      PRINT*, 'NUMBER OF SUCH LOCATIONS = ',INUM
      PRINT*
C
C     ATTEMPT TO FIND TARGET WITH OSRCHM
C
```

```
      CALL OSRCHM(N,X,INCX,ITARGET,MASK,IRIGHT,INDEX,IWHERE,INUM)
      XWHERE=X(1+(IWHERE-1)*INCX)
      PRINT*, 'LOCATIONS FOUND BY OSRCHM WITH INCX = ', INCX
      PRINT*,'  INDEX     IWHERE     INUM    XWHERE  MOD(XWHERE,2**5)'
      PRINT50, INDEX, IWHERE, INUM, XWHERE, IAND(XWHERE,MASK)
50    FORMAT(I7,4(I10))
      END
```

The output of this program is as follows:

```
     X(i) = 127*i, for all i = 1, 2, ..., 128

     LOCATIONS WHERE MOD(X(i), 2**5) = 20
     (SUBFIELD MATCHES TARGET)
     i = 12     X(12) = 1524
     i = 44     X(44) = 5588
     i = 76     X(76) = 9652
     i = 108     X(108) = 13716

     NUMBER OF SUCH LOCATIONS = 4

     LOCATIONS FOUND BY OSRCHM WITH INCX = 1
       INDEX     IWHERE     INUM    XWHERE  MOD(XWHERE,2**5)
        129         1         0       127        31
```

NAME

SSORTB, ISORTB – Performs Batcher's Odd-Even Merge sort on the elements of a real or integer general vector

SYNOPSIS

```
CALL SSORTB (ad, n, x, incx [, index, incd])
CALL ISORTB (ad, n, x, incx [, index, incd])
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

SSORTB sorts the elements of a real general vector.

ISORTB sorts the elements of an integer general vector.

These routines can perform an in-place or indexed sort in ascending or descending order.

An *in-place* sort returns the result by overwriting the input vector. An *indexed* sort returns the result in another array, leaving the input vector unchanged. The in-place sort offers about 25% better performance.

The following calls illustrate the various sorts provided by these routines.

`CALL SSORTB('A', N, X, INCX)` – Performs an in-place sort in ascending order.
`CALL ISORTB('A', N, X, INCX)` – Performs an in-place sort in ascending order.

`CALL SSORTB('D', N, X, INCX)` – Performs an in-place sort in descending order.
`CALL ISORTB('D', N, X, INCX)` – Performs an in-place sort in descending order.

`CALL SSORTB('A', N, X, INCX, ID, INCD)` – Performs an indexed sort in ascending order.
`CALL ISORTB('A', N, X, INCX, ID, INCD)` – Performs an indexed sort in ascending order.

`CALL SSORTB('D', N, X, INCX, ID, INCD)` – Performs an indexed sort in descending order.
`CALL ISORTB('D', N, X, INCX, ID, INCD)` – Performs an indexed sort in descending order.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is `KIND=8` for integer, real, complex and logical arguments.

These routines have the following arguments:

ad Character. (input)
Indicates whether to sort in ascending or descending order.
Valid values are 'a', 'A', 'd', and 'D'.
'a' and 'A' signify ascending order, and 'd' and 'D' signify descending order.
Because only the first character is significant, to enhance clarity, you may use 'ascending' or 'descending' as the value for *ad*.

n Integer. (input)
Number of elements to be sorted.
$n > 0$.

x `SSORTB`: Real array of dimension $(n-1)*incx+1$. (input and output)
`ISORTB`: Integer array of dimension $(n-1)*incx+1$. (input and output)
Array *x* contains the values to be sorted.
If you specify only four arguments, these routines perform an in-place sort and use *x* for output.

incx Integer. (input)
Increment between elements of the array *x*.
$incx > 0$.

index Integer array of dimension $(n-1)*incd+1$. (output, optional)
If specified, array *index* holds the results of the indexed sort.

In this case, on exit, *x* may be accessed in sorted order by using *index*, as follows:

$$xsorted(k) = x(1 + (index(1 + (k-1)*incd) - 1) * incx), \quad \text{for all } k = 1, 2, \ldots, n$$

incd Integer. (input)
Increment for the elements of *index*.
$incd > 0$.

NOTES

Negative increments are not required because *ad* specifies ascending or descending order.

These sorts are *unstable*. An unstable sort is one in which the indices of two elements whose values are the same could be inverted in the sorted output. That is, if $x(j) = x(k)$ for some $j < k$, and (at the end of the sort) $index(ij) = j$ and $index(ik) = k$ for some ij and ik, it is possible that $ij > ik$.

For example, if $x(6) = x(15) = 10$, at the end of sorting, the subscript at which the array *index* takes on the value 6 may be larger than the subscript at which *index* takes on the value 15.

If the number of arguments is 4, the routine performs an in-place sort. If the number of arguments is 6, the routine performs an indexed sort.

EXAMPLES

In-place Sorting

Suppose you have a real array X, every other element of which to sort into descending order, then the call is as follows:

```
CALL SSORTB( 'D', N, X, 2 )
```

Indexed Sorting

Suppose you have a real array X, every third element of which to sort into ascending order without destroying X, then an indexed sort is done, as follows:

```
CALL SSORTB( 'A', N, X, 3, ID, INCD )
```

The argument *incd* allows you extra flexibility in arranging the indexing. For instance, suppose you want to sort row #2 of matrix A in descending order. Furthermore, suppose you want the result to be stored into the corresponding row of a matrix of indices, I, with leading dimension LDI. The call is as follows:

```
CALL SSORTB( 'D', N, A( 2,1 ), LDA, I( 2,1 ), LDI )
```

The following program is an example of a row-by-row indexed sort of a 4-by-5 rectangular matrix into ascending order. This example shows how to access the ordered version of A by using the index array, MID.

Following the program is its output. The output demonstrates the values to which SSORTB initializes MID. It also presents the values in the unsorted matrix, A. It then presents the results of a call to SSORTB. First the values in A are presented in sorted order; then the corresponding values in MID are output. That way you can apply the final results in MID to the unsorted version of A and see how the indexed sort works. The final values in MID are such that

$A(i,MID(i,1)) \le A(i,MID(i,2)) \ldots \le A(i,MID(i,5))$.

```
      PROGRAM TSSORTB
*-----------------------------------------------------------------------
*     ..Parameters..
        INTEGER ROW, COL, MAX
        PARAMETER( ROW=4, COL=5, MAX=9 )

*     ..Scalar vars..
        INTEGER I, J

*     ..Vector vars..
        REAL    A( ROW,COL )
        INTEGER MID( ROW,COL )

*-----------------------------------------------------------------------
*     ..Begin execution..

*       Load A() with numbers between 1. and max.
        DO I = 1, ROW
          DO J = 1, COL
            A( I,J ) = 1. + REAL( MOD( I*J, MAX ) )
          END DO
        END DO

*       Initialize MID()
        DO I = 1, ROW
          DO J = 1, COL
            MID( I,J ) = J
          END DO
        END DO

*-----------------------------------------------------------------------
        PRINT *,  "       ... Indices in MID() prior to indexed sort ",
     &            "of each row of A() ..."
        DO I = 1, ROW
          WRITE( 6,20 ) ( MID( I,J ), J=1,COL )
        END DO

        PRINT *, "     ... A() prior to row-by-row sort ..."
        DO I = 1, ROW
```

```
          WRITE( 6,10 ) ( A( I,J ), J=1,COL )
        END DO

*-----------------------------------------------------------------------
*       Sort each column in ascending order.
        DO I = 1, ROW
          CALL SSORTB( 'ASCENDING', COL, A( I,1 ), ROW,
     &                 MID( I,1 ), ROW )
        END DO


*-----------------------------------------------------------------------
        PRINT *, "     ... A() after row-by-row sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( A( I,MID( I,J ) ), J=1,COL )
        END DO

        PRINT  *, "    ...  Indices in MID() after the indexed sort ",
     &            "of each row of A() ..."
        DO I = 1, ROW
          WRITE( 6,20 ) ( MID( I,J ), J=1,COL )
        END DO

*-----------------------------------------------------------------------
*     ..Format statements..
10      FORMAT( 5X, 5( F5.1, 2X ) )
20      FORMAT( 5X, 5( I5, 2X ) )
      END
```

The results of program TSSORTB are as follows:

```
  ... Indices in MID() prior to indexed sort of each row of A() ...
     1      2      3      4      5
     1      2      3      4      5
     1      2      3      4      5
     1      2      3      4      5
  ... A() prior to row-by-row sort ...
   2.0    3.0    4.0    5.0    6.0
   3.0    5.0    7.0    9.0    2.0
   4.0    7.0    1.0    4.0    7.0
   5.0    9.0    4.0    8.0    3.0
  ... A() after row-by-row sort ...
   2.0    3.0    4.0    5.0    6.0
   2.0    3.0    5.0    7.0    9.0
   1.0    4.0    4.0    7.0    7.0
   3.0    4.0    5.0    8.0    9.0
  ... Indices in MID() after the indexed sort of each row of A() ...
     1      2      3      4      5
     5      1      2      3      4
     3      4      1      2      5
     5      3      1      4      2
```

The following example illustrates the use of ISORTB to perform a column-by-column in-place sort of a 5-by-4 rectangular matrix into descending order.

```
      PROGRAM TISORTB
*-----------------------------------------------------------------------
*     ..Parameters..
        INTEGER ROW, COL, MAX
        PARAMETER( ROW=5, COL=4, MAX=9 )

*     ..Scalar vars..
        INTEGER I, J

*     ..Vector vars..
        INTEGER IA( ROW,COL )
```

```
*---------------------------------------------------------------------
*     ..Begin execution..

*       Initialize IA() with numbers between 1 and max.
        DO I = 1, ROW
          DO J = 1, COL
            IA( I,J ) = 1 + MOD( I*J, MAX )
          END DO
        END DO

*---------------------------------------------------------------------
        PRINT  *,  "     ... IA()  prior  to  column-by-column sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( IA( I,J ), J=1,COL )
        END DO

*---------------------------------------------------------------------
*       Perform an in-place sort on IA in descending order on
*       each column.
        DO J = 1, COL
          CALL ISORTB( 'DESCENDING', ROW, IA( 1,J ), 1 )
        END DO

*---------------------------------------------------------------------
        PRINT *, "     ... IA() after column-by-column sort ..."
        DO I = 1, ROW
          WRITE( 6,10 ) ( IA( I,J ), J=1,COL )
        END DO

*---------------------------------------------------------------------
*     ..Format statements..
10      FORMAT( 5X, 4( I5, 2X ) )
      STOP
      END
```

The results of program TISORTB are as follows:

```
... IA() prior to column-by-column sort ...
   2      3      4      5
   3      5      7      9
   4      7      1      4
   5      9      4      8
   6      2      7      3
... IA() after column-by-column sort ...
   6      9      7      9
   5      7      7      8
   4      5      4      5
   3      3      4      4
   2      2      1      3
```

NAME

WHENEQ, WHENNE – Searches a vector for all elements equal or not equal to a target

SYNOPSIS

```
CALL WHENEQ (n, x, incx, target, index, nn)
CALL WHENNE (n, x, incx, target, index, nn)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WHENEQ searches a real or integer vector for all elements equal to a real or integer target.

WHENNE searches a real or integer vector for all elements not equal to a real or integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* containing the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

target Real or integer. (input)
Value for which to search in the array.

index Integer array of dimension *n*. (output)
Array *index* contains the indices in the array elements that match *target*.

nn Integer. (output)
Number of values put in the *index* array.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

x(1 – *incx** (n –1)), x (1–*incx** (n –2)),. . ., x (1)

The desired values are at:

x(1+(*index*(*k*)– 1)**incx*) when *incx* > 0

x(1+(*index*(*k*)– *n*)**incx*) when *incx* < 0

for all *k* = 1, 2, . . ., *nn* .

NAME

WHENFLT, WHENFLE, WHENFGT, WHENFGE – Searches a real vector for all elements with a specified logical relationship to a real target

SYNOPSIS

```
CALL WHENFLT (n, x, incx, target, index, nn)
CALL WHENFLE (n, x, incx, target, index, nn)
CALL WHENFGT (n, x, incx, target, index, nn)
CALL WHENFGE (n, x, incx, target, index, nn)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WHENFLT searches a real vector for all elements that are less than a real target.

WHENFLE searches a real vector for all elements that are less than or equal to a real target.

WHENFGT searches a real vector for all elements that are greater than a real target.

WHENFGE searches a real vector for all elements that are greater than or equal to a real target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements of the array to be searched.

x Real array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

target Real. (input)
Value for which array is searched.

index Integer array of dimension *n*. (output)
Array *index* contains the indices in the array elements that match *target*.

nn Integer. (output)
Number of values put in the *index* array.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)), x(1-incx*(n-2)),\ldots, x(1)$

The desired values are at:

$x(1+(index(k)-1)*incx)$ when *incx* > 0

$x(1+(index(k)-n)*incx)$ when *incx* < 0

for all $k = 1, 2, \ldots, nn$.

NAME

WHENILT, WHENILE, WHENIGT, WHENIGE – Searches an integer vector for all elements that have a specified relationship to an integer target

SYNOPSIS

```
CALL WHENILT (n, x, incx, itarget, index, nn)
CALL WHENILE (n, x, incx, itarget, index, nn)
CALL WHENIGT (n, x, incx, itarget, index, nn)
CALL WHENIGE (n, x, incx, itarget, index, nn)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WHENILT searches an integer vector for all elements that are less than an integer target.

WHENILE searches an integer vector for all elements that are less than or equal to an integer target.

WHENIGT searches an integer vector for all elements that are greater than an integer target.

WHENIGE searches an integer vector for all elements that are greater than or equal to an integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements in the array to be searched.

x Integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

incx Integer. (input)
Increment between elements of the searched array.

itarget Integer. (input)
Value for which the array is searched.

index Integer array of dimension *n*. (output)
Array *index* contains the indices in the array elements that match *target*.

nn Integer. (output)
Number of values put in the *index* array.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

x (1 – *incx** (n –1)), x (1–*incx** (n –2)),. . ., x (1)

The desired values are at:

x(1+(*index*(*k*)– 1)**incx*) when *incx* > 0

x(1+(*index*(*k*)– *n*)**incx*) when *incx* < 0

for all *k* = 1, 2, . . ., *nn*.

NAME

WHENMEQ, WHENMNE – Searches a vector for all elements whose subfields are equal or not equal to a target

SYNOPSIS

```
CALL WHENMEQ (n, x, incx, itarget, index, nn, mask, iright)
CALL WHENMNE (n, x, incx, itarget, index, nn, mask, iright)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WHENMEQ searches a real or integer vector for all elements for which the subfield defined by *mask* and *iright* is equal to an integer target.

WHENMNE searches a real or integer vector for all elements for which the subfield defined by *mask* and *iright* is not equal to an integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

itarget Integer. (input)
Value for which to search in the array.

index Integer array of dimension *n*. (output)
Array *index* contains the indices of the array elements whose subfield is equal or not equal to *itarget*.

nn Integer. (output)
Number of values put in the *index* array.

mask Integer. (input)
Mask of 1's (set bits) from the right. The number of set bits in *mask* is the number of bits in the subfield, which is searched in each element of the input array.

iright Integer. (input)
Number of bits to shift each element of the input array to the right so as to right justify the subfield searched.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

The desired values are at:

$x(1+(index(k)-1)*incx)$ when $incx > 0$

$x(1+(index(k)-n)*incx)$ when $incx < 0$

for all $k = 1, 2, \ldots, nn$.

NAME

WHENMLT, WHENMLE, WHENMGT, WHENMGE – Searches a vector for all elements whose subfields have a specified logical relationship with a target

SYNOPSIS

```
CALL WHENMLT (n, x, incx, itarget, index, nn, mask, iright)
CALL WHENMLE (n, x, incx, itarget, index, nn, mask, iright)
CALL WHENMGT (n, x, incx, itarget, index, nn, mask, iright)
CALL WHENMGE (n, x, incx, itarget, index, nn, mask, iright)
```

IMPLEMENTATION

Cray PVP systems

DESCRIPTION

WHENMLT searches a real or integer vector for all elements for which the subfield defined by *mask* and *iright* is less than an integer target.

WHENMLE searches a real or integer vector for all elements for which the subfield defined by *mask* and *iright* is less than or equal to an integer target.

WHENMGT searches a real or integer vector for all elements for which the subfield defined by *mask* and *iright* is greater than an integer target.

WHENMGE searches a real or integer vector for all elements for which the subfield defined by *mask* and *iright* is greater than or equal to an integer target.

CF90 compiler only: Unless documented otherwise, when using the CF90 compiler on Cray PVP systems and Cray MPP systems, all arguments must be of default kind, which is KIND=8 for integer, real, complex and logical arguments.

These routines have the following arguments:

n Integer. (input)
Number of elements to be searched.

x Real or integer array of dimension $(n-1)*|incx|+1$. (input)
Array *x* contains the vector to be searched.

itarget Integer. (input)
Value for which to search in the array.

index Integer array of dimension *n*. (output)
Array *index* contains the indices of the array elements whose subfield is equal or not equal to *itarget*.

nn Integer. (output)
Number of values put in the *index* array.

mask Integer. (input)
Mask of 1's (set bits) from the right. The number of set bits in *mask* is the number of bits in the subfield, which is searched in each element of the input array.

iright Integer. (input)
Number of bits to shift each element of the input array to the right so as to right justify the subfield searched.

NOTES

When scanning backward (*incx* < 0), each routine starts at the end of the vector and moves backward, as follows:

$x(1-incx*(n-1)),\ x(1-incx*(n-2)),\ldots,\ x(1)$

The desired values are at:

$x(1+(index(k)-1)*incx)$ when *incx* > 0

$x(1+(index(k)-n)*incx)$ when *incx* < 0

for all $k = 1, 2, \ldots, nn$.