User CALL and Driver Implementation Guide

Revision 4.2 (Preliminary)

Dynamic Concepts Inc.

18-B Journey

Aliso Viejo, CA 92656

www.dynamic.com

Chapter 1 - Introduction

This version (4.2) of the dL4 User CALL and Driver Implementation Guide is based on Version 4.2 of the dL4 product and covers all future releases, except for any new enhancements.

This guide is written for experienced BASIC and C programmers. It is a reference that describes the dL4 runtime libraries. Information concerning library functions and definitions used by the dL4 interpreter, dL4 drivers, and dL4 user defined CALLS/functions can be found on these pages. This guide is divided into topical sections which describe the various dL4 runtime libraries.

Typographical Conventions

This guide uses the following typographic conventions:

Example of convention	Description
GOSUB	Capitalized words in bold indicate language-specified reserved words. Refer to Appendix C.
KILL *filename*	Variables are shown in italic type for clarity and to distinguish them from elements of the language itself.
`LIST`	Mono-spaced type is used to display screen output and example input commands and program examples.
<letter>	Information inside angle brackets <> must be from specified group, e.g., a single letter.
WHILE | UNTIL	A vertical bar indicates that the user must choose one of the items.
[*expr*]	Items inside square brackets are mandatory.
{*expr*}	Items inside braces are optional.
stmt {\ stmt} ...	A series of three periods (...) indicates that the item preceding them can be repeated one or more times.

Syntax Notations

The following notations are used to describe dL4 BASIC syntax:

NOTATION	STANDS FOR	MEANING
args	Arguments	Expressions or variables passed to a procedure, a function, or used with an operator.
bool.expr	Boolean expression	An expression evaluated in a boolean context resulting in TRUE/FALSE.
chan.expr	Channel expression	An expression that combines a channel number followed by three optional numeric parameters, commonly indicating a record number, a field position, and a timeout value. It begins with a # and ends in a semicolon. e.g. #9, r, c, d; #9,5;
chan.no	Channel number	An integer value, between 0 and 99 inclusive, preceded by #, that the program uses for a logical connection between a BASIC program and a file. Refer to "Channel Expression" in Chapter 5 of this guide.
crt.expr	CRT expression	An expression used for cursor positioning, e.g. @x,y. Refer to "CRT Expressions", Chapter 6 of this guide.
expr	Expression	A valid series of constants, variables, functions, and operators to define a desired computation. Refer to Chapter 4 of this guide.

filename	Filename	A string literal or expression containing a name which is optionally preceded by a relative or absolute directory pathname. Refer to *Introduction to dL4.*
file.spec.items	File specification, items	A file specification expressed as a list of items.
file.spec.str	File specification, string	A file specification expressed as a string expression.
func.name	Function name	The valid name of a function.
label :	Label	A user-defined name identifying a statement line number. Refer to "Statements, Line Numbers and Labels", Chapter 7 of this guide.
num.const	Numeric constant	A numeric constant.
num.expr	Numeric expression	An expression yielding a number.
num.lit	Numeric literal	A numeric literal value, e.g. 1.23.
parm.list	Parameter	A list of variables associated with parameters passed, and optionally followed by three dots (...).
proc.name	Procedure name	The valid name of a procedure. Refer to Chapter 4 and Chapter 8 of this guide.
rel.op	Relational operator	A binary operator that compares its first operand to its second operand to test the validity of the specified relationship. Refer to "Relational Operators", Chapter 5 of this guide.
stmt.no	Statement number	Unique positive integer that identifies a statement line. Refer to "Statements, Line Numbers and Labels", Chapter 7 of this guide.
stmt	Statement	A single BASIC instruction along with parameters, e.g. PRINT A
str.expr	String expression	An expression yielding a string value or a string variable.
str.lit	String literal	A quoted sequence of characters, e.g. "string".
struct.name	Structure Name	The name of a pre-defined, fixed grouping of variables defined at compile-time. Refer to "Structure", Chapter 3 of this guide.
var.list	List of variables or expressions	An arbitrary number of comma separated variables of any dL4 data types. Refer to "Variables", Chapter 3 of this guide.
var.mat	Matrix Variable	Any numeric matrix variable name. Refer to "Variables", Chapter 3 of this guide.
var.name	Variable Name	A variable name. Refer to "Variables", Chapter 3 of this guide.
num.var	Numeric variable	A variable of numeric data type. Refer to Chapters 2 and 3 of this guide.
str.var	String variable	A variable of string data type. Refer to Chapters 2 and 3 of this guide.
struct.var	Structure variable	A variable of structure data type. Refer to "Structures", Chapter 3 of this guide.

Chapter 2 – Using the dL4 Runtime Libraries

Introduction

The dL4 programming language is implemented as a set of runtime libraries. These libraries, as documented in this guide, can be used to create user defined CALLs, functions, and channel drivers to customize dL4 for specific applications. This guide is a "C" language description of the functions and definitions provided by the dL4 libraries in the dL4 Development kits. This guide is not an introduction to the "C" programming language or to the various compilers and tools used for "C" programming.

The best way to learn how to write a user CALL, function, or driver for dL4 is to examine the source files included in the dL4 Development kits. The kits contain source files for many of the standard user defined CALLs, functions, and file drivers. The kits also supply the make (or nmake) files needed to control compiling and linking a customized version of dL4. The compilation and linking commands are described in the readme files in the development kit.

Licensing

A license is required to use the dL4 Development kit and the dL4 runtime libraries. The kit and libraries may only be used to create customized versions of dL4 for the purpose of executing dL4 BASIC application programs. Please see your dL4 license or our web site at www.unibasic.com for a full description of license rights and restrictions.

Libraries and Include Files

The following table lists all of the dL4 runtime libraries:

Library	Include files	Description
Mem	mem/memmgmt.h mem/table.h	Memory management
Strings	strings/strings.h strings/mnemonic.h	Unicode and ASCII string functions
Math	math/math.h	Math functions
Date	date/date.h	Date functions
Error	error/errors.h	Standard error codes and messages
Items	items/items.h	General purpose descriptor functions
Channel	channel/channel.h channel/classes.h	I/O and other channel oriented functions
Driver	channel/driver.h	Functions for use in drivers

In addition to these include files, there is also a shared include file, dcidef.h, that should be included in any source file that uses the dL4 runtime libraries. The dcidef.h include file provides standard definitions such as:

- BYTE Unsigned character
- UNICODE Unicode character, a 16-bit unsigned integer
- INT32 32-bit signed integer
- UINT32 32-bit unsigned integer
- BF16 16-bit bit field
- BF32 32-bit bit field
- Dimof(a) Macro to return the dimension of a "C" array variable
- MIN(x,y) Macro to return the minimum value of "x" and "y"
- MAX(x,y) Macro to return the maximum value of "x" and "y"

User CALLs and Functions

All user CALLs and functions must be declared as:

```
int UserProcNAME(ITEM *args, size_t argcnt)
```

where NAME is the name of the CALL or function. "args" is an array of **ITEM**s (see "Item Functions"), one for each parameter passed by the dL4 program. "argcnt" is the number of parameters. "args[n]" will be a character (UNICODE) **ITEM** for string parameters, a numeric **ITEM** for numeric parameters, a date **ITEM** for date parameters, a binary **ITEM** for binary parameters, an array **ITEM** for array parameters ("var[]"), or a structure **ITEM** for a structure variable parameter ("var.").

Chapter 3 – Memory and Table Functions

NAME

AddTableItems - Add items to a table

SYNOPSIS

```
#include <mem/table.h>

void *AddTableItems(TABLE *tbl, size_t index, size_t numitems)
```

DESCRIPTION

AddTableItems changes the size of table '*tbl*', inserting '*numitems*' items at table position '*index*', and moving subsequent items down, if necessary.

If the insertion is successful, the table is locked and a pointer to first item (not item #'*index*') is returned.
The new table items are not initialized to any particular values.

NOTES

Before calling **AddTableItems**, table '*tbl*' should be completely unlocked. If necessary, **AddTableItems** will attempt to increase the physical table allocation to accommodate the new items; if the table is locked, the reallocation will fail.

RETURN VALUE

`void *`	Pointer to first item in table; table is locked.
`NULL`	Table could not be resized; lock status is unchanged.

SEE ALSO

AllocTable(), **DelTableItems**(), **LockTable**()

NAME

AllocMem - Allocate memory

SYNOPSIS

```
#include <mem/memmgmt.h>

MEMHND AllocMem(size_t size, int options )
```

DESCRIPTION

AllocMem() dynamically allocates '*size*' bytes of memory.

'*options*' is a combination of values indicating the intended usage of the memory in terms of "class", "permanence", and other miscellaneous options:

MEM_CLASS(c)	Indicates that the memory will be used for an object of class c, where c in an integer value between 0 and 1023 inclusive. The class number is intended for use in statistics and error reporting.
MEM_USETEMP	Indicates that the memory will be used only for a short time; i.e. its deallocation is certain and imminent, probably within the same function.
MEM_USEPERM	Indicates that the memory is expected to exist throughout the life of the calling process.
MEM_USESEMIPERM	Indicates any usage other than temporary or permanent.
MEM_USEUNKNOWN	Indicates that the expected memory usage is completely unknown.
MEM_ZERO	Initializes the allocated memory to zero.

The four MEM_USExxxx values are mutually exclusive. **AllocMem**() may use these values to optimize the arrangement of memory for efficient allocation and compaction. In addition to explicitly indicating "permanence", the MEM_USExxxx values implicitly indicate performance requirements, e.g. MEM_USETEMP regions are, if possible, managed more quickly than MEM_USEPERM regions.

The memory handle returned by AllocMem() is not a direct memory pointer; LockMem() must be called to convert a handle to a usable pointer by locking the region. Locked regions are held at a fixed address in memory until unlocked by UnlockMem(). All unlocked regions are subject to movement, usually due to compaction or reallocation.

The following example demonstrates the allocation of 1024 bytes of semi-perminent memory with a class of 10 initialized to zero:

```
MEMHND hld;

hld = AllocMem((size_t)1024, MEM_CLASS(10) | MEM_USETEMP |
MEM_ZERO);
```

RETURN VALUE

MEMHND	Handle of allocated memory.
0	Allocation failed.

SEE ALSO

ReallocMem(), **FreeMem**(), **LockMem**(), **UnlockMem**()

NAME

AllocTable - Allocate a table

SYNOPSIS

```
#include <mem/table.h>

int AllocTable(TABLE *tbl, size_t itemsize, size_t numitems,
size_t allocincr)
```

DESCRIPTION

The "Table" group of functions define a TABLE object, where a table is a one-dimensional array of fixed-length items (often structures). The assumption is that a table needs to expand and/or contract often and dynamically, therefore the physical allocation should be managed by a larger increment for the sake of performance and consistency.

AllocTable allocates a table of '*numitems*' items, where each item is '*itemsize*' bytes, and fills the given '*tbl*' structure to reference the table. '*allocincr*' is the allocation increment for the table in number of items. As the table changes in size, the physical memory will be managed such that the total allocated size is always a multiple of '*allocincr*' items.

The TABLE structure is defined as:

```
typedef struct {
    MEMHND  Handle;
    size_t  ItemSize;
    size_t  NextItem;
    size_t  NumItems;
    size_t  AllocIncr;
} TABLE;
```

To access the table data itself, the table must be locked to a fixed address using **LockTable**(). Applications should never change any value in the TABLE structure, nor use any values with the exception of 'TABLE.NextItem'.

TABLE.NextItem represents the next available item in a TABLE, and TABLE.NextItem - 1 is the array index to the last item used. Here is an example of how to traverse a table of TYPE items/structures/values...:

```
TYPE    *tp;
int     i;

tp = (TYPE *)LockTable(tbl);    /* tbl must point to a TABLE */
for (i = 0; i < tbl->NextItem; ++i) {
      /* data is in tp[i] */
}
(void)UnlockTable(tbl);
```

RETURN VALUE

`0`	Successful.
`-1`	Table allocation failed.

SEE ALSO

FreeTable(), **LockTable**(), **AddTableItems**(), **DelTableItems**(), **FreeTableToMem**()

NAME

CompareMem - Compare memory objects

SYNOPSIS

```
#include <mem/memmgmt.h>

int CompareMem(voidx *m1, voidx *m2, size_t size)
```

DESCRIPTION

CompareMem() compares memory object *m1* and *m2*, both containing *size* chars, and returns an integer which is negative if *m1* is less than *m2*, zero if *m1* equals *m2*, and positive if *m1* is greater than *m2*.

RETURN VALUE

0	if *m1* == *m2*
negative	if *m1* < *m2*
positive	if *m1* > *m2*

SEE ALSO

CopyMem(), **IsMemZero**(), **MoveMem**()

NAME

ConstMem - Fill memory objects with a byte value

SYNOPSIS

```
#include <mem/memmgmt.h>

void ConstMem(voidx *dest, BYTE value, size_t size)
```

DESCRIPTION

ConstMem() stores the *value* parameter into each of the first *size* bytes of *dest*.

SEE ALSO

CopyMem(), **MoveMem**()

NAME

CopyMem - Copy non-overlapping memory objects

SYNOPSIS

```
#include <mem/memmgmt.h>

void CopyMem(voidx *dest, voidx *src, size_t size)
```

DESCRIPTION

CopyMem() copies size bytes from *src* to *dest*. If the source and destination areas overlap, the result is undefined.

SEE ALSO

ConstMem(), **IsMemZero**(), **MoveMem**()

NAME

CopyTable - Copy a table

SYNOPSIS

```
#include <mem/table.h>

void CopyTable(TABLE *dtbl, TABLE *stbl)
```

DESCRIPTION

CopyTable() copies the contents of table 'stbl' to 'dtbl'. dtbl must already be allocated before calling this function. This function simply copies the data resident in **stbl* to **dtbl* without regard to its actual format. The amount of data copied is the lesser amount in-use of the two tables.

SEE ALSO

AllocTable(), **MakeTable**()

NAME

DelTableItems - Delete items from a table

SYNOPSIS

```
#include <mem/table.h>

void DelTableItems(TABLE *tbl, size_t index, size_t numitems)
```

DESCRIPTION

DelTableItems changes the size of table *'tbl'*, deleting *'numitems'* items at table position *'index'*, and moving subsequent items up, if necessary.

NOTES

Before calling **DelTableItems**, table '*tbl*' should be completely unlocked. According to the table's allocation increment, **DelTableItems** may attempt to decrease the physical table allocation; if the table is locked, the reallocation will fail. However, this condition is not fatal, albeit wasteful.

SEE ALSO

AllocTable(), **AddTableItems**(), **FreeTable**(), **LockTable**()

NAME

DuplicateMem - Duplicate a memory region

SYNOPSIS

```
MEMHND DuplicateMem(MEMHND original)
```

DESCRIPTION

DuplicateMem() generates a new memory object with contents and characteristics otherwise identical to the original object.

RETURN VALUE

(MEMHND)0	Error, memory overflow, unable to allocate new
(MEMHND)x	Handle of duplicate object

SEE ALSO

AllocMem(), **ReallocMem**()

NAME

ExchangeMem - switch positions of the two parts of an area

SYNOPSIS

```
#include <mem/memmgmt.h>

void ExchangeMem(void *area, size_t lowersize, size_t uppersize)
```

DESCRIPTION

ExchangeMem() switches the positions of the two parts of an area. For example, if *area* contained "1234567abcd", then after performing **ExchangeMem**(*area*, 7, 4), *area* would contain "abcd1234567". Note that the *area* consists ONLY of the upper and lower parts; there is no middle.

SEE ALSO

ReverseMem( )

NAME

FreeMem - Deallocate memory

SYNOPSIS

```
#include <mem/memmgmt.h>

void FreeMem(MEMHND handle)
```

DESCRIPTION

FreeMem deallocates the given memory region, making the space available for subsequent allocation by **AllocMem**. Once freed, the memory handle '*handle*' is invalid.

RESTRICTIONS

Regions must be completely unlocked with **UnlockMem**() in order to be deallocated with **FreeMem**. Failure to do so may produce an abortive program trap.

WARNINGS

Attempting to free the same memory region twice may corrupt the memory management package.

SEE ALSO

AllocMem(), **ReallocMem**()

NAME

FreeTable - Deallocate table

SYNOPSIS

```
#include <mem/table.h>

void FreeTable(TABLE *tbl)
```

DESCRIPTION

FreeTable deallocates the given table, making the space available for subsequent allocation. Once freed, the contents of the given '*tbl*' structure are invalid.

RESTRICTIONS

Tables must be completely unlocked with **UnlockTable**() in order to be deallocated with **FreeTable**. Failure to do so may produce an abortive program trap.

WARNINGS

Attempting to free the same table twice may corrupt the memory management package.

SEE ALSO

AllocTable(), **UnlockTable**(), **FreeTableToMem**()

NAME

FreeTableToMem - Extract data and discard table

SYNOPSIS

```
#include <mem/table.h>

MEMHND FreeTableToMem(TABLE *tbl)
```

DESCRIPTION

FreeTableToMem removes the memory handle from a table and marks the table as deallocated. The memory handle is returned to the caller with any unused table space deallocated.

RESTRICTIONS

Tables must be completely unlocked with **UnlockTable**() in order to be deallocated with **FreeTableToMem**. Failure to do so may produce an abortive program trap.

RETURN VALUE

(MEMHND)x Handle to table memory

SEE ALSO

AllocTable(), **UnlockTable**()

NAME

GetMemClassName - Get name of memory class

SYNOPSIS

```
#include <mem/memmgmt.h>

const char *GetMemClassName(unsigned int classname)
```

DESCRIPTION

Lookup class number in the MemClassTable and return its name. This debug version is used if NDEBUG is defined.

RETURN VALUE

char * Pointer to class name

SEE ALSO

RegisterMemClass()

NAME

GetMemFlags - Return option flags for a region of allocated memory

SYNOPSIS

```
#include <mem/memmgmt.h>

int GetMemFlags(MEMHND handle)
```

DESCRIPTION

GetMemFlags returns an integer containing the flags currently in use for memory region '*handle*'. The region's "permanence" value can be extracted from the flags with the mask MEM_USE. See **AllocMem**() for possible flag values.

RETURN VALUE

Int	Flag values.

SEE ALSO

AllocMem(), **GetMemSize**()

NAME

GetMemSize - Return current size for a region of allocated memory

SYNOPSIS

```
#include <mem/memmgmt.h>

size_t GetMemSize(MEMHND handle)
```

DESCRIPTION

GetMemSize returns the current size in bytes of memory region '*handle*'.

RETURN VALUE

size_t Current size in bytes.

SEE ALSO

AllocMem(), **GetMemFlags**()

NAME

ExportBYTE - Store BYTE value in portable format
ExportBYTEArray - Store BYTE array in portable format
ExportUINT16 - Store UINT16 value in portable format
ExportUINT32 - Store UINT32 value in portable format
ExportSize_t - Store size_t value in portable format

ImportBYTE - Convert BYTE value from portable format
ImportBYTEArray - Convert BYTE array from portable format
ImportUINT16 - Convert UINT16 value from portable format
ImportUINT32 - Convert UINT32 value from portable format
ImportSize_t - Convert size_t value from portable format

SYNOPSIS

```
BYTE *ExportBYTE(BYTE *op, UINT32 value)
BYTE *ExportBYTEArray(BYTE *op, BYTE *array, size_t size)
BYTE *ExportUINT16(BYTE *op, UINT32 value)
BYTE *ExportUINT32(BYTE *op, UINT32 value)
BYTE *ExportSize_t(BYTE *op, size_t value)

BYTE ImportBYTE(BYTE *ip)
void ImportBYTEArray(BYTE *array, BYTE *ip, size_t size)
UINT16 ImportUINT16(BYTE *ip)
UINT32 ImportUINT32(BYTE *ip)
size_t ImportSize_t(BYTE *ip)
```

DESCRIPTION

ExportXXXX()

Store the value as a string of one to four bytes, most significant byte first.

ImportXXXX()

Convert a string of one to four bytes, most significant byte first, to an integer value.

These functions are meant to provide for portable integers and byte arrays between systems, usually in a file. Normally, 16- and 32-bit numbers generated on one system are unusable on another, especially if the other system has a different processor type from the originating system. These functions solve that problem. (All other dL4 number types are already portable, except Base-10000 numbers.)

While using these functions, use the following symbolic constants when calculating offsets into exported data. They represent the size in bytes of portable values for program files:

Symbolic Constant	Defined as
SIZEOF_BYTE	1
SIZEOF_UINT16	2
SIZEOF_UINT32	4
SIZEOF_SIZE_T	SIZEOF_UINT32

RETURN VALUE

ExportXXXX()
(BYTE *)0 Error, illegal value
x Pointer to byte following value just stored
ImportXXXX()
X Value

SEE ALSO

Number Format Information

NAME

InitMem - Initialize memory management package

SYNOPSIS

```
#include <mem/memmgmt.h>

void InitMem()
```

DESCRIPTION

Initialize global variables for memory management package and perform assertions of package ssumptions. Must be called before calling all other functions in memory management package.

SEE ALSO

ShutdownMem()

NAME

IsMemLocked - Test if a memory region is locked

SYNOPSIS

```
#include <mem/memmgmt.h>

int IsMemLocked(MEMHND handle)
```

DESCRIPTION

IsMemLocked returns the lock count for a memory region, which will be non-zero if the region is locked.

RETURN VALUE

Int	Region is locked, return value is lock count.
0	Region is not locked.

SEE ALSO

LockMem(), **UnlockMem**()

NAME

IsMemZero - Zero memory test

SYNOPSIS

```
#include <mem/memmgmt.h>

int IsMemZero(const void *area, size_t size)
```

DESCRIPTION

IsMemZero()tests the given region of memory for zero byte values.

NOTES

IsMemZero() returns true (non-zero) if 'size' is zero.

RETURN VALUE

!0	All bytes are zero
0	One or more bytes are not zero

SEE ALSO

memcpy(), **memcmp**()

NAME

IsTableLocked - Test if a table is locked

SYNOPSIS

```
#include <mem/table.h>

int IsTableLocked(const TABLE *tbl)
```

DESCRIPTION

IsTableLocked returns the lock count for a table, which will be non-zero if the table is locked.

RETURN VALUE

Int	Table is locked, return value is lock count.
0	Table is not locked.

SEE ALSO

LockTable(), **UnlockTable**()

NAME

IsTableNull - Test if a table is allocated

SYNOPSIS

```
#include <mem/table.h>

int IsTableNull(const TABLE *tbl)
```

DESCRIPTION

IsTableNull tests whether a table is currently allocated.

RETURN VALUE

1	Table is not allocated.
0	Table is allocated.

SEE ALSO

AllocTable(), **FreeTable**()

NAME

LockMem - Lock a memory region for use

SYNOPSIS

```
#include <mem/memmgmt.h>

void *LockMem(MEMHND handle)
```

DESCRIPTION

LockMem returns a pointer to memory region '*handle*', and increments its lock count. A region which is locked is immune to movement and will remain locked until its lock count is decreased to zero.

RETURN VALUE

void * Pointer to locked memory region.

WARNINGS

A locked region cannot be deallocated with FreeMem().

SEE ALSO

AllocMem(), **UnlockMem**(), **IsMemLocked**()

NAME

LockTable - Lock a table for use

SYNOPSIS

```
#include <mem/table.h>

void *LockTable(TABLE *tbl)
```

DESCRIPTION

LockTable returns a pointer to table *'tbl'*, and increments its lock count. A table which is locked is immune to movement and will remain locked until its lock count is decreased to zero.

RETURN VALUE

void *	Pointer to first item in table.

WARNINGS

A locked table cannot be deallocated with **FreeTable**().

SEE ALSO

AllocTable(), **UnlockTable**()

NAME

MakeTable - Make a table

SYNOPSIS

```
#include <mem/table.h>

int MakeTable(TABLE *dtbl, TABLE *stbl)
```

DESCRIPTION

MakeTable creates a new table *'dtbl'* and copies the contents of table ‘*stbl’*.

RETURN VALUE

0	success
-1	error, check GetError() for actual error

SEE ALSO

AllocTable(), **CopyTable**()

NAME

MoveMem - Copy memory objects with possible overlap

SYNOPSIS

```
#include <mem/memmgmt.h>

void MoveMem(voidx *dest, voidx *src, size_t size)
```

DESCRIPTION

MoveMem() copies *size* bytes from *src* to *dest*. After the move is complete, the contents of the destination area will be identical to that of the pre-move source area regardless of any overlap between the two areas.

SEE ALSO

CopyMem(), **ConstMem**()

NAME

ReallocMem - Reallocate memory

SYNOPSIS

```
#include <mem/memmgmt.h>

int ReallocMem(MEMHND handle, size_t newsize)
```

DESCRIPTION

ReallocMem reallocates the size of a previously allocated memory region to *'newsize'* bytes. If reallocation fails, the region is left intact at the previous size. The contents will be the same up to the lesser of the new and old sizes.

RETURN VALUE

0	Reallocation succeeded.
-1	Reallocation failed.

RESTRICTIONS

ReallocMem cannot reallocate a region locked by **LockMem**().

SEE ALSO

AllocMem(), **FreeMem**(), **LockMem**(), **UnlockMem**(), **IsMemLocked**()

NAME

RegisterMemClass - Register memory class

SYNOPSIS

```
#include <mem/memmgmt.h>

unsigned int RegisterMemClass(const char *classname)
```

DESCRIPTION

Register *classname* in the MemClassTable and return its symbol number. This debug version is used if NDEBUG is defined.

RETURN VALUE

unsigned int lass number

RESTRICTIONS

classname must be a statically allocated constant string.

SEE ALSO

GetMemClassName()

NAME

ReverseMem - Reverse order of bytes in an area

SYNOPSIS

```
#include <mem/memmgmt.h>

void ReverseMem(void *area, size_t size)
```

DESCRIPTION

ReverseMem() reverses the order of the bytes within the specified area. For example, "1234567" would become "7654321".

SEE ALSO

ExchangeMem()

NAME

ShutdownMem - Shutdown memory management package

SYNOPSIS

```
#include <mem/memmgmt.h>

void ShutdownMem(void)
```

DESCRIPTION

Shutdown the memory management package and perform debugging checks.

SEE ALSO

InitMem()

NAME

UnlockMem - Unlock a memory region

SYNOPSIS

```
#include <mem/memmgmt.h>

int UnlockMem(MEMHND handle)
```

DESCRIPTION

UnlockMem decrements the lock count for the memory region *'handle'*. If the lock count is decreased to zero, the region is completely unlocked and subject to movement.

RETURN VALUE

int	New lock count.
0	Region is now unlocked.

RESTRICTIONS

Regions must be completely unlocked with **UnlockMem** in order to be changed with **ReallocMem**() or deallocated with **FreeMem**().

WARNINGS

Attempting to unlock a handle not already locked may corrupt the memory management package.

SEE ALSO

FreeMem(), **LockMem**(), **IsMemLocked**()

NAME

UnlockTable - Unlock a table

SYNOPSIS

```
#include <mem/memmgmt.h>

int UnlockTable(TABLE *tbl)
```

DESCRIPTION

UnlockTable decrements the lock count for table *'tbl'*. If the lock count is decreased to zero, the table is completely unlocked and subject to movement.

RETURN VALUE

int	New lock count.
0	Table is now unlocked.

RESTRICTIONS

Tables must be completely unlocked with **UnlockTable** in order to be changed with **AddTableItems**()/DelTableItems(), or deallocated with **FreeTable**().

WARNINGS

Attempting to unlock a table not already locked may corrupt the memory management package.

SEE ALSO

AddTableItems(), **DelTableItems**(), **FreeTable**(), **LockTable**(), **AddTableItems**()

Chapter 4 – Math Functions

The math functions are used to perform various operations on numbers in the dL4 numeric formats (see "Number Formats") or in the dL4 date formats (see "Date Format Codes"). Operations are never performed directly on any format other than **INFMT_ACCUM** and **IDFMT_ACCUM** which correspond to the **ACCUM** type definition. The non-ACCUM formats must always be loaded, via **LoadAccum**() or **LoadDate**(), into an **ACCUM** before performing any operation such as addition or multiplication. The final result is then stored from the **ACCUM** into specific format using **StoreAccum**() or **StoreDate**().

The internal components of numeric or date formats, including those of the **ACCUM** format, should never be manipulated directly. Instead, math functions should be used. For example, always use the **FixAccum**() function to convert an **ACCUM** into a long integer value. The only exception is that any value in a numeric format can be given a numeric value of zero by zeroing every byte in the numeric value. Any other direct manipulation may result in compatibility problems in future releases of dL4.

The only public structure definition in the math library is **DECDIG** which is used by the **DecDigToAccum**() and **AccumToDecDig**() functions. **DECDIG** has the following layout:

```
typedef struct {
    BYTE    Digits[MAXDECDIGITS];   /* base '\0' (not ASCII) and
normalized */
    short   Expon;/* no bias */
    BF16    Sign        : 1;        /* 0 = positive, 1 = negative */
    BF16    IsNAN       : 1;        /* 1 = NAN */
    BF16    IsInfinity  : 1;        /* 1 = infinity */
    BF16: 13;
} DECDIG;
```

Any member of **DECDIG** can be freely accessed and manipulated.

NAME

Number Formats

Many functions require a number format as an argument. The following is a list of the defined number formats found in “math/math.h":

Type	digits	Align	Class	Portable	Description
INFMT_ACCUM	*	2	0	no	Intermediate numeric result
INFMT_INT16	5	1	1	no	16-bit signed integer
INFMT_INT32	10	2	1	no	32-bit signed integer
INFMT_B10K3	12	2	2	no	UniBasic %3 base-10000 fp
INFMT_B10K4	16	2	2	no	UniBasic %4 base-10000 fp
INFMT_B10K2	6	2	2	no	UniBasic %5 base-10000 fp
INFMT_B10K6	20	2	2	no	UniBasic %6 base-10000 fp
INFMT_BCDINT1	4	1	1	yes	16-bit signed BCD integer (IRIS)
INFMT_IRISBCD2	6	1	2	yes	2-word BCD fp (IRIS)
INFMT_IRISBCD3	10	1	2	yes	3-word BCD fp (IRIS)
INFMT_IRISBCD4	14	1	2	yes	4-word BCD fp (IRIS)
INFMT_IRISBCD5	18	1	2	yes	5-word BCD fp (IRIS)
INFMT_BCDINT2	8	1	1	yes	32-bit signed BCD integer
INFMT_IEEEBCD2	6	1	2	yes	2-word BCD fp (IEEE)
INFMT_IEEEBCD3	10	1	2	yes	3-word BCD fp (IEEE)
INFMT_IEEEBCD4	14	1	2	yes	4-word BCD fp (IEEE)
INFMT_IEEEBCD5	18	1	2	yes	5-word BCD fp (IEEE)
INFMT_IEEEX1002	6	1	2	yes	2-word binary fp scaled X 100
INFMT_IEEEX1003	10	1	2	yes	3-word binary fp scaled X 100
INFMT_IEEEX1004	14	1	2	yes	4-word binary fp scaled X 100

* = MAXDECDIGITS is the maximum digits an ACCUM may hold. Currently, this is 20 digits.

+ = Number classes: 0=INCLASS_INVALID 1=INCLASS_INTEGER 2=INCLASS_FLOATING

INFMT_ACCUM should be used only for temporary storage and never saved to a file.

INFMT_INT16 and INFMT_INT32 are not normally portable as is. Use the Import*() and Export*() functions in the mem package if you need to save them to a file.

NAME

AbsAccum - Convert ACCUM *a to absolute value of ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void AbsAccum(ACCUM *a)
```

DESCRIPTION

Convert ACCUM *a to absolute value of ACCUM *a. If the value was originally a NAN, it will return a NAN.

SEE ALSO

NegAccum()

NAME

AccumToDecDig - Setup DECDIG structure from accumulator value

SYNOPSIS

```
#include <math/math.h>

void AccumToDecDig(DECDIG *d, ACCUM *a)
```

DESCRIPTION

Store ACCUM **a* in the structure DECDIG **d*.

SEE ALSO

DecDigToAccum()

NAME

AddAccum - Add ACCUM *b to ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void AddAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Add the ACCUM value from pointer *b* to the ACCUM value of pointer *a*. If overflow occurs, set ACCUM **a* to plus or minus infinity. If either of the parameter values is Not A Number (NAN), the result in ACCUM **a* will be a NAN.

***a*b**	**+In-range**	**-In-range**	**+0.0**	**-0.0**	**+Infinity**	**-Infinity**	**Zero**	**NAN**
+In-range	Add	Add	Add	Add	+Infinity	-Infinity	Add	NAN
-In-range	Add	Add	Add	Add	+Infinity	-Infinity	Add	NAN
+0.0	Add	Add	+0.0	+0.0	+Infinity	-Infinity	+0.0	NAN
-0.0	Add	Add	+0.0	-0.0	+Infinity	-Infinity	-0.0	NAN
+Infinity	+Infinity	+Infinity	+Infinity	+Infinity	+Infinity	NAN	+Infinity	NAN
-Infinity	-Infinity	-Infinity	-Infinity	-Infinity	NAN	-Infinity	-Infinity	NAN
Zero	Add	Add	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

SEE ALSO

SubAccum()

NAME

AscStringToNumber - Convert ASCII string to numeric ACCUM

SYNOPSIS

```
size_t AscStringToNumber(ACCUM *result, const char *str, size_t siz)
```

DESCRIPTION

AscStringToNumber() converts an ASCII character string representing a decimal number to an ACCUM. Conversion stops at the first illegal character, or after '*siz*' characters have been parsed.

RETURN VALUE

size_t Number of characters converted.

SEE ALSO

StringToNumber(), **DecDigToAccum**()

NAME

AtnAccum - Calculate arctangent of argument

SYNOPSIS

```
#include <math/math.h>

void AtnAccum(ACCUM *a)
```

DESCRIPTION

Calculate arctangent of ACCUM *a and return result in ACCUM *a. Result table for operand type:

+In-range	-In-Range	+0.0	-0.0	+Infinity	-Infinity	+Zero	NAN
ATN	ATN	+0.0	-0.0	+pi/2	-pi/2	Zero	NAN

WARNINGS

AtnAccum() alters the value of errno.

SEE ALSO

TanAccum()

NAME

CmpEQAccum - Compare ACCUM *a to ACCUM *b and return (a == b)

SYNOPSIS

```
#include <math/math.h>

void CmpEQAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Compare ACCUM *a to ACCUM *b and set ACCUM *a to one if they are equal, otherwise set ACCUM *a to zero. If either of the parameter values is a Not-A-Number (NAN), the result in ACCUM *a will be a NAN.

NOTES

CmpEQAccum() is identical to **CmpNEQAccum**() except the result is inverted. Exact zero, positive zero, and negative zero are all considered equal.

NAME

CmpGEAccum - Compare ACCUM *a to ACCUM *b and return (a >= b)

SYNOPSIS

```
#include <math/math.h>

void CmpGEAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Compare ACCUM *a to ACCUM *i and set ACCUM *a to one if ACCUM *a is greater than or equal to ACCUM *b, otherwise set ACCUM *a to zero. If either of the parameter values is a Not-A-Number (NAN), the result in ACCUM *a will be a NAN.

NOTES

CmpGEAccum() is identical to **CmpLTAccum**() except the result is inverted. Exact zero, positive zero, and negative zero are all considered equal.

NAME

CmpGTAccum - Compare ACCUM *a to ACCUM *b and return (a > b)

SYNOPSIS

```
#include <math/math.h>

void CmpGTAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Compare ACCUM *a to ACCUM *b and set ACCUM *a to one if ACCUM *a is greater than ACCUM *b, otherwise set ACCUM *a to zero. If either of the parameter values is a Not-A-Number (NAN), the result in ACCUM *a will be a NAN.

NOTES

CmpGTAccum() is identical to **CmpLEAccum**() except the result is inverted. Exact zero, positive zero, and negative zero are all considered equal.

NAME

CmpLEAccum - Compare ACCUM *a to ACCUM *b and return (a <= b)

SYNOPSIS

```
#include <math/math.h>

void CmpLEAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Compare ACCUM *a to ACCUM *b and set ACCUM *a to one if ACCUM *a is less than or equal to ACCUM *b, otherwise set ACCUM *a to zero. If either of the parameter values is a Not-A-Number (NAN), the result in ACCUM *a will be a NAN.

NOTES

CmpLEAccum() is identical to **CmpGTAccum**() except the result is inverted. Exact zero, positive zero, and negative zero are all considered equal.

NAME

CmpLTAccum - Compare ACCUM *a to ACCUM *b and return (a < b)

SYNOPSIS

```
#include <math/math.h>

void CmpLTAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Compare ACCUM *a to ACCUM *b and set ACCUM *a to one if ACCUM *a is less than ACCUM *b, otherwise set ACCUM *a to zero. If either of the parameter values is a Not-A-Number (NAN), the result in ACCUM *a will be a NAN.

NOTES

CmpLTAccum() is identical to **CmpGEAccum**() except the result is inverted. Exact zero, positive zero, and negative zero are all considered equal.

NAME

CmpNEQAccum - Compare ACCUM *a to ACCUM *b and return (a != b)

SYNOPSIS

```
#include <math/math.h>

void CmpNEQAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Compare ACCUM **a* to ACCUM **b* and set ACCUM **a* to one if they are not equal, otherwise set ACCUM **a* to zero. If either of the parameter values is a Not-A-Number (NAN), the result in ACCUM **a* will be a NAN.

NOTES

CmpNEQAccum() is identical to **CmpEQAccum**() except the result is inverted. Exact zero, positive zero, and negative zero are all considered equal.

NAME

ConvertNumbers - Convert and copy numbers

SYNOPSIS

```
#include <math/math.h>

int ConvertNumbers(void *dest, int dfmt, const void *src, int sfmt,
       size_t count)
```

DESCRIPTION

Converts ‘count’ numbers in ‘sfmt’ format starting at ‘src, to ‘dfmt’ format starting at ‘dest’. ‘src’ and ‘dest’ may be equal, provided that ‘sfmt’ and ‘dfmt’ use equal size.

RETURN VALUE

0	Numbers converted successfully.
-1	Numeric overflow occurred.

SEE ALSO

LoadAccum(), **StoreAccum**(), Number Format Information

NAME

CopyAccum - Copy one ACCUM to another

SYNOPSIS

```
#include <math/math.h>

void CopyAccum(ACCUM *dest, const ACCUM *src)
```

DESCRIPTION

Copies '*src*' ACCUM to the '*dest*' ACCUM.

SEE ALSO

LoadAccum(), **StoreAccum**()

NAME

CosAccum - Calculate cosine of argument

SYNOPSIS

```
#include <math/math.h>

void CosAccum(ACCUM *a)
```

DESCRIPTION

Calculate cosine of ACCUM **a* and return result in ACCUM**a*. Result table for operand type:

+In-range	-IN-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
COS	COS	+1.0	+1.0	NAN	NAN	+1.0	NAN

WARNINGS

CosAccum() alters the value of errno.

SEE ALSO

SinAccum()

NAME

DecDigToAccum - Convert value in DECDIG structure to ACCUM value

SYNOPSIS

```
#include <math/math.h>

void DecDigToAccum(ACCUM *a, DECDIG *d)
```

DESCRIPTION

Convert value in DECDIG structure to an ACCUM value. Returns infinity on overflow.

WARNINGS

The digits in DECDIG **d* must be left justified (no leading zeroes).

SEE ALSO

AccumToDecDig()

NAME

DivAccum - Divide ACCUM *a by ACCUM *b

SYNOPSIS

```
#include <math/math.h>

void DivAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Divide the ACCUM **a* by the ACCUM **b*. If overflow occurs, set ACCUM **a* to plus or minus infinity. If either of the parameter value is Not A Number (NAN), the result in **a* will be a NAN.

***a*b**	**+In-range**	**-In-range**	**+0.0**	**-0.0**	**+Infinity**	**-Infinity**	**Zero**	**NAN**
+In-range	Div	Div	+Infinity	-Infinity	+0.0	-0.0	NAN	NAN
-In-range	Div	Div	-Infinity	+Infinity	-0.0	+0.0	NAN	NAN
+0.0	+0.0	+0.0	NAN	NAN	+0.0	-0.0	NAN	NAN
-0.0	-0.0	-0.0	NAN	NAN	-0.0	+0.0	NAN	NAN
+Infinity	+Infinity	-Infinity	+Infinity	-Infinity	NAN	NAN	NAN	NAN
-Infinity	-Infinity	+Infinity	-Infinity	+Infinity	NAN	NAN	NAN	NAN
Zero	Zero	Zero	Zero	Zero	Zero	Zero	NAN	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

SEE ALSO

MulAccum()

NAME

EAccum - Return natural e

SYNOPSIS

```
#include <math/math.h>

void EAccum(ACCUM *a)
```

DESCRIPTION

Stores natural e in ACCUM **a*.

SEE ALSO

PiAccum()

NAME

ExpAccum - Calculate e to a specified power

SYNOPSIS

```
#include <math/math.h>

void ExpAccum(ACCUM *a)
```

DESCRIPTION

Calculate e raised to the power of ACCUM **a* and return result in ACCUM **a*. Result table for operand type:

+In-range	-In-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
EXP	EXP	+1.0	+1.0	+infinity	+0.0	+1.0	NAN

NOTES

If ACCUM *a is outside the range of the exp() function, its value will be reduced and the resulting exponent will be adjusted. While this extends the range of exp(), it results in limited accuracy and possibly odd behavior near the limits.

WARNINGS

ExpAccum() alters the value of errno.

SEE ALSO

LogAccum()

NAME

FixAccum - Return long integer value of ACCUM

SYNOPSIS

```
#include <math/math.h>

long FixAccum(ACCUM *a)
```

DESCRIPTION

FixAccum() returns the long value of ACCUM **a* truncated to an integer. If ACCUM **a* cannot be represented as a long, then LONG_MAX is returned and the **FixOverflowed**() function will return one. Otherwise, **FixOverflowed**() will return zero.

RETURN VALUE

X	long value of ACCUM *a

WARNINGS

FixAccum() assumes that LONG_MAX is less than 10000 cubed. If the value was originally a NAN, it will return a NAN.

SEE ALSO

FixOverflowed(), **IfIntAccum**(), **FloatAccum**(), **AddAccum**()

NAME

FixOverflowed - Return overflow status of last FixAccum()

SYNOPSIS

```
#include <math/math.h>

int FixOverflowed(void)
```

DESCRIPTION

FixOverflowed() returns 1 if the last use of the **FixAccum**() function caused an overflow. Otherwise, **FixOverflowed**() returns 0.

RETURN VALUE

1	Last **FixAccum**() caused an overflow.
0	Last **FixAccum**() did not cause an overflow.

SEE ALSO

FixAccum(), **IfIntAccum**(), **FloatAccum**(), **AddAccum**()

NAME

FloatAccum - Load long value into ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void FloatAccum(ACCUM *a, long lval)
```

DESCRIPTION

The long value *lval* is loaded in ACCUM **a* using the accumulator's long integer format.

SEE ALSO

FixAccum(), **AddAccum**()

NAME

FloatX100Accum - Load scaled long integer into ACCUM

SYNOPSIS

```
#include <math/math.h>

void FloatX100Accum(ACCUM *a, long scaledlval)
```

DESCRIPTION

The parameter *scaledlval* is a fixed point decimal value expressed as INT(number * 100) and contained in a long integer. The value thus has up to two decimal places and must be in the range of INT32_MAX/100 and INT32_MIN/100 inclusive.

NOTES

FloatX100Accum() assumes twos-complement arithmetic.

SEE ALSO

FloatAccum(), **AddAccum**()

NAME

FormatAccum - Convert ACCUM to a UNICODE string according to a mask
NativeFormatAccum - Convert ACCUM to a UNICODE string according to a mask using native locale information

SYNOPSIS

```
#include <math/math.h>

ssize_t FormatAccum(UNICODE *dest, size_t destsize, ACCUM *a,
const UNICODE *form, size_t formsize, UNICODE **start,
int europeanmasks)

ssize_t NativeFormatAccum(UNICODE *dest, size_t destsize, ACCUM *a,
const UNICODE *form, size_t formsize, UNICODE **start,
int europeanmasks)
```

DESCRIPTION

Format the given ACCUM according to a format string referenced by scanned from the given '*start*' position and this position is updated upon return. If the *europeanmasks* parameter is nonzero, then the meaning of comma and period in masks will be swapped. The *start* parameter may be NULL and, if so, the format string will be scanned from the beginning. **NativeFormatAccum**() uses native locale values for the currency symbol, decimal point, group separator, and digits.

NOTES

The format of '*form*' is identical to that of the PRINT USING statement implemented in dL4.

RETURN VALUE

-1	The accumulator was a Not-A-Number (NAN) or infinity or the mask was illegal.
X	Number of characters converted.

SEE ALSO

AccumToDecDig().

NAME

FraAccum - Discard integer portion portion of ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void FraAccum(ACCUM *a)
```

DESCRIPTION

Retain fractional portion of ACCUM **a* while discarding integer portion. If the value was originally a NAN, it will return a NAN.

SEE ALSO

IntAccum()

NAME

GetNumberClass -- Return the class of a number type

SYNOPSIS

```
#include <math/math.h>

int GetNumberClass(int fmt)
```

DESCRIPTION

Determine the class of a number and return it.

RETURN VALUE

0	Number has no format class or is not in class table.
X	Number class (INFMT_XXXX) of number format *fmt*

SEE ALSO

Number Format Information

NAME

InitMath - Initialize math package

SYNOPSIS

```
#include <math/math.h>

void InitMath()
```

DESCRIPTION

Initialize global variables for math package and perform assertions of math package assumptions.

SEE ALSO

ShutdownMath()

NAME

IntAccum - Truncate fractional portion of ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void IntAccum(ACCUM *a)
```

DESCRIPTION

Truncate ACCUM **a* to an integer value. If the value was originally a NAN, it will return a NAN.

SEE ALSO

FraAccum()

NAME

IsAccumInRange - Test for a number

SYNOPSIS

```
#include <math/math.h>

int IsAccumInRange(const ACCUM *a)
```

DESCRIPTION

Tests whether ACCUM *a is a valid number. That is the number is neither a NAN nor infinity.

RETURN VALUE

0	Number is not valid
1	Number is valid

NAME

IsAccumNAN - Test for NAN

SYNOPSIS

```
#include <math/math.h>

int IsAccumNAN(const ACCUM *a)
```

DESCRIPTION

Tests whether ACCUM *a is Not-A-Number (NAN).

RETURN VALUE

0	Number is number
1	Number is Not-A-Number (NAN)

NAME

IsAccumNeg - Test for negative number

SYNOPSIS

```
#include <math/math.h>

int IsAccumNeg(const ACCUM *a)
```

DESCRIPTION

Tests whether ACCUM *a is negative.

RETURN VALUE

0	Number is positive
1	Number is negative

WARNINGS

Does not test for Not-A-Number (NAN).

NAME

IsAccumOflw - Test for overflow

SYNOPSIS

```
#include <math/math.h>

int IsAccumOflw(const ACCUM *a)
```

DESCRIPTION

Tests whether ACCUM **a* is infinity (positive or negative).

RETURN VALUE

0	Number is not infinity
1	Number is infinity

NAME

IsAccumUflw - Test for underflow

SYNOPSIS

```
#include <math/math.h>

int IsAccumUflw(const ACCUM *a)
```

DESCRIPTION

Tests whether ACCUM *a underflowed. An underflow is either positive or negative zero, but not exactly zero.

RETURN VALUE

0	Number is not infinity
1	Number is infinity

NAME

IsAccumZero - Test for zero

SYNOPSIS

```
#include <math/math.h>

int IsAccumZero(const ACCUM *a)
```

DESCRIPTION

Tests whether ACCUM *a is exactly equal to zero. Positive and negative zero are not considered to be zero.

RETURN VALUE

0	Number is not an exact zero
1	Number is an exact zero

WARNINGS

Does not test for Not-A-Number (NAN).

NAME

IsNFormatPortable -- Check the portability of a number format

SYNOPSIS

```
#include <math/math.h>

int IsNFormatPortable(int fmt)
```

DESCRIPTION

Determine the portability of a number format between platforms.

RETURN VALUE

0	No
1	Yes

SEE ALSO

Number Format Information

NAME

IxrAccum - Raise 10 to the power ACCUM *a

SYNOPSIS

#include <math/math.h>

void **IxrAccum**(ACCUM **a*)

DESCRIPTION

Raise 10 to the power ACCUM **a* and return result in ACCUM **a*. If the value was originally a NAN, it will return a NAN.

SEE ALSO

IntAccum()

NAME

LoadAccum - Load value in specified format into ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void LoadAccum(ACCUM *a, const void *n, int fmt)
```

DESCRIPTION

Load value in *fmt* specified format from the pointer *n* into ACCUM **a*. If *fmt* is not supported, ACCUM **a* will be set to NAN (Not A Number).

SEE ALSO

StoreAccum(), **LoadNItem**(), **Number Format Information**

NAME

LogAccum - Natural Logarithm function

SYNOPSIS

```
#include <math/math.h>

void LogAccum(ACCUM *a)
```

DESCRIPTION

Calculate natural logarithm of ACCUM **a* and return result in ACCUM **a*. Result table for operand type:

+In-range	-In-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
LOG	NAN	-Infinity	-Infinity	+Infinity	NAN	NAN	NAN

NOTES

If ACCUM **a* is outside the range of a double, the exponent will be reduced and handled separately. While this extends the range of log(), it results in limited accuracy and possibly odd behavior near the limits.

SEE ALSO

ExpAccum()

NAME

ManAccum - Transform ACCUM *a to mantissa of ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void ManAccum(ACCUM *a)
```

DESCRIPTION

Convert value of ACCUM **a* to mantissa of ACCUM **a*. If the value was originally a NAN, it will return a NAN.

SEE ALSO

ChrAccum()

NAME

ModAccum - Calculate ACCUM *a mod ACCUM *b

SYNOPSIS

```
#include <math/math.h>

void ModAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Calculate *a* modulo *b* using (*a* - INT(*a*/*b*)**b*). If overflow occurs, set ACCUM **a* to plus or minus infinity. If either of the parameter value is Not A Number (NAN), the result in ACCUM **a* will be a NAN. If the operation results in a No-Op, ACCUM **a* is unchanged.

*a*b	+In-range	-In-range	+ 0.0	-0.0	+Infinity	-Infinity	Zero	NAN
+In-range	Mod	Mod	NAN	NAN	No-Op	No-Op	NAN	NAN
-In-range	Mod	Mod	NAN	NAN	No-Op	No-Op	NAN	NAN
+0.0	+0.0	+0.0	NAN	NAN	+0.0	+0.0	NAN	NAN
-0.0	-0.0	-0.0	NAN	NAN	-0.0	-0.0	NAN	NAN
+Infinity	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN
-Infinity	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN
Zero	Zero	Zero	NAN	NAN	Zero	Zero	NAN	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

SEE ALSO

DivAccum()

NAME

MulAccum - Multiply ACCUM *a by ACCUM *b

SYNOPSIS

```
#include <math/math.h>

void MulAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Multiply the ACCUM **a* by ACCUM **b* returning the product ACCUM **a*. If overflow occurs, set *a* to plus or minus infinity. If either of the parameter value is Not A Number (NAN), the result in *a* will be a NAN.

a*b***	**+In-range**	**-In-range**	**+0.0**	**-0.0**	**+Infinity**	**-Infinity**	**Zero**	**NAN**
+In-range	Mul	Mul	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
-In-range	Mul	Mul	-0.0	+0.0	-Infinity	+Infinity	Zero	NAN
+0.0	+0.0	-0.0	+0.0	-0.0	NAN	NAN	Zero	NAN
-0.0	-0.0	+0.0	-0.0	+0.0	NAN	NAN	Zero	NAN
+Infinity	+Infinity	-Infinity	NAN	NAN	+Infinity	-Infinity	Zero	NAN
-Infinity	-Infinity	+Infinity	NAN	NAN	-Infinity	+Infinity	Zero	NAN
Zero	Zero	Zero	Zero	Zero	Zero	Zero	Zero	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

SEE ALSO

DivAccum()

NAME

NANAccum - Return Not-A-Number (NAN)

SYNOPSIS

```
#include <math/math.h>

void NANAccum(ACCUM *a)
```

DESCRIPTION

Stores Not-A-Number (NAN) in ACCUM *a.

NAME

NFormatAlign - Return format alignment

SYNOPSIS

```
#include <math/math.h>

int NFormatAlign(int fmt)
```

DESCRIPTION

Returns the format alignment of number format fmt as a function of log2(alignment in bytes) for all defined math formats.

RETURN VALUE

-1	Invalid format
1	Align to 2 bytes
2	Align to 4 bytes
3	Align to 8 bytes etc.

SEE ALSO

Number Format Information

NAME

NFormatMaxDigits - return max decimal digits of specified format

SYNOPSIS

```
#include <math/math.h>

int NFormatMaxDigits(int fmt)
```

DESCRIPTION

Returns the maximum number of digits allowed for a particular number format.

RETURN VALUE

-1	Invalid format
x	Maximum number of digits

SEE ALSO

Number Format Information

NAME

NFormatSize - Return format size

SYNOPSIS

```
#include <math/math.h>

size_t NFormatSize(int fmt)
```

DESCRIPTION

Returns the size in bytes of the format specified in *fmt*. This value may vary between platforms.

RETURN VALUE

0	Invalid format
>0	Size in bytes

NAME

NativeNumberToString - Convert ACCUM to a local language UNICODE string

SYNOPSIS

```
#include <math/math.h>

ssize_t NativeNumberToString(UNICODE *str, size_t size, ACCUM *accum)
```

DESCRIPTION

Convert numeric value of **accum* to a local language Unicode string and store it in **str* upto *size* Unicode characters.

RETURN VALUE

-1	The accumulator was a Not-A-Number (NAN) or infinity.
X	Number of characters converted.

SEE ALSO

NumberToString(), **AccumToDecDig**().

NAME

NegAccum - Negate ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void NegAccum(ACCUM *a)
```

DESCRIPTION

Negate ACCUM **a*. If the value was originally a NAN, it will return a NAN.

SEE ALSO

AbsAccum()

NAME

NegOflwAccum - Return negative infinity

SYNOPSIS

```
#include <math/math.h>

void NegOflwAccum(ACCUM *a)
```

DESCRIPTION

Store negative infinity in ACCUM **a*.

SEE ALSO

NegUflwAccum(), **PosOflwAccum**(), **PosUflwAccum**()

NAME

NegUflwAccum - Return negative zero

SYNOPSIS

```
#include <math/math.h>

void NegUflwAccum(ACCUM *a)
```

DESCRIPTION

Store negative zero in ACCUM **a*.

SEE ALSO

NegOflwAccum(), **PosOflwAccum**(), **PosUflwAccum**()

NAME

NumberToString - Convert ACCUM to a UNICODE string

SYNOPSIS

```
#include <math/math.h>

ssize_t NumberToString(UNICODE *str, size_t size, ACCUM *accum)
```

DESCRIPTION

Convert numeric value of accum to a Unicode string. Positive and negative zero is considered zero.

RETURN VALUE

-1	The accumulator was a Not-A-Number (NAN) or infinity.
X	Number of characters converted.

SEE ALSO

StringToNumber(), **AccumToDecDig**().

NAME

OneAccum - Return positive one

SYNOPSIS

```
#include <math/math.h>

void OneAccum(ACCUM *a)
```

DESCRIPTION

Store positive one in ACCUM **a*.

SEE ALSO

ZeroAccum()

NAME

PiAccum - Return PI

SYNOPSIS

```
#include <math/math.h>

void PiAccum(ACCUM *a)
```

DESCRIPTION

Store PI in ACCUM **a*.

SEE ALSO

EAccum()

NAME

PosOflwAccum - Return positive infinity

SYNOPSIS

```
#include <math/math.h>

void PosOflwAccum(ACCUM *a)
```

DESCRIPTION

Store positive infinity in ACCUM **a*.

SEE ALSO

NegOflwAccum(), **NegUflwAccum**(), **PosUflwAccum**()

NAME

PosUflwAccum - Return positive zero

SYNOPSIS

```
#include <math/math.h>

void PosUflwAccum(ACCUM *a)
```

DESCRIPTION

Store positive zero in ACCUM **a*.

SEE ALSO

NegOflwAccum(), **NegUflwAccum**(), **PosOflwAccum**()

NAME

PwrAccum - Calculate ACCUM *a raised to the power ACCUM *b

SYNOPSIS

```
#include <math/math.h>

void PwrAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Raise ACCUM **a* to the power ACCUM **b*. If overflow occurs, set ACCUM **a* to plus or minus infinity. If either of the parameter value is Not A Number (NAN), the result in ACCUM **a* will be a NAN.

***a*b**	**+In-range**	**-In-range**	**+0.0**	**-0.0**	**+Infinity**	**-Infinity**	**Zero**	**NAN**
+In-range	Power	Power	+1.0	+1.0	+Infinity	+0.0	+1.0	NAN
-In-range	Power	Power	+1.0	+1.0	-Infinity	-0.0	+1.0	NAN
+0.0	+0.0	+0.0	+1.0	+1.0	+0.0	+0.0	+1.0	NAN
-0.0	-0.0	-0.0	+1.0	+1.0	-0.0	-0.0	+1.0	NAN
+Infinity	+Infinity	+Infinity	+1.0	+1.0	NAN	NAN	+1.0	NAN
-Infinity	-Infinity	-Infinity	+1.0	+1.0	NAN	NAN	+1.0	NAN
Zero	Zero	Zero	+1.0	+1.0	Zero	Zero	+1.0	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

WARNINGS

PwrAccum() modifies errno.

SEE ALSO

SqrAccum()

NAME

RndAccum - Calculate pseudo-random number

SYNOPSIS

```
#include <math/math.h>

void RndAccum(ACCUM *a, RANDOMSEED *seed)
```

DESCRIPTION

Return in ACCUM '*a*' a pseudorandom number from the sequence determined by '*seed*' with a value between 0 and the initial value of ACCUM '*a*'. If ACCUM '*a*' is initially zero, a value between 0 and 1 will be returned. For example:

```
0  <=  RndAccum(5.5, seed)  <   5.5
-10 <   RndAccum(-10, seed)  <=  0
 0  <=  RndAccum(0, seed) <   1
```

Result table for operand type:

+In-range	-In-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
RND	RND	+0.0	-0.0	+Infinity	-Infinity	RND(1)	NAN

SEE ALSO

RndSeed()

NAME

RndSeed - Set seed for pseudorandom number generator

SYNOPSIS

```
#include <math/math.h>

void RndSeed(RANDOMSEED *seed, ACCUM *a)
```

DESCRIPTION

RndSeed() sets '*seed*', a 48-bit integer value used by the pseudorandom number generator (see **RndAccum**()), from the base 10000 digits of ACCUM **a* treated as unsigned 16-bit integers.

SEE ALSO

RndAccum()

NAME

RoundAccum - Round ACCUM *a to specified decimal place

SYNOPSIS

```
#include <math/math.h>

void RoundAccum(ACCUM *a, ACCUM *decimal)
```

DESCRIPTION

Round ACCUM **a* to the decimal place specified by ACCUM *decimal* For example, **RoundAccum**(123.456, 1) equals 123.5 and **RoundAccum**(123.456, -1) equals 120.0. Any fractional digits of *decimal* are ignored.

SEE ALSO

TruncAccum()

NAME

ShutdownMath - Shutdown math package

SYNOPSIS

```
#include <math/math.h>

void ShutdownMath(void)
```

DESCRIPTION

Shutdown math package.

SEE ALSO

InitMath()

NAME

SinAccum - Calculate sine of argument

SYNOPSIS

```
#include <math/math.h>

void SinAccum(ACCUM *a)
```

DESCRIPTION

Calculate sine of ACCUM **a* and return result in ACCUM **a*. Result table for operand type:

+In-range	-In-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
SIN	SIN	+0.0	-0.0	NAN	NAN	Zero	NAN

WARNINGS

SinAccum() alters the value of errno.

SEE ALSO

CosAccum()

NAME

SqrAccum - Square root function

SYNOPSIS

```
#include <math/math.h>

void SqrAccum(ACCUM *a)
```

DESCRIPTION

Calculate square root of ACCUM **a* and return result in ACCUM **a*. Result table for operand type:

+In-range	-In-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
SQR	NAN	+0.0	-0.0	+Infinity	NAN	Zero	NAN

SEE ALSO

PwrAccum()

NAME

StoreAccum - Store ACCUM *a in specified format

SYNOPSIS

```
#include <math/math.h>

int StoreAccum(void *n, ACCUM *a, int fmt)
```

DESCRIPTION

Store value of ACCUM **a* through pointer *n* according to the format *fmt*.

RETURN VALUE

0	Store succeeded
-1	Store failed due to illegal format or ACCUM **a* was Not-A-Number or infinity and the format cannot represent those values

NOTES

If rounding causes overflow, then the value will be stored without rounding

SEE ALSO

LoadAccum(), **Number Format Information**

NAME

StringToNumber - Convert UNICODE string to numeric ACCUM
NativeStringToNumber - Convert UNICODE string to numeric ACCUM

SYNOPSIS

```
size_t StringToNumber(ACCUM *result, const UNICODE *str, size_t siz)
size_t NativeStringToNumber(ACCUM *result, const UNICODE *str, size_t
siz)
```

DESCRIPTION

StringToNumber() converts a UNICODE character string representing a decimal number to an ACCUM. Conversion stops at the first illegal character, or after '*siz*' characters have been parsed. **NativeStringToNumber**() is similar, but uses the decimal point character defined for the native locale.

RETURN VALUE

size_t Number of characters converted.

SEE ALSO

DecDigToAccum(), **NumberToString**()

NAME

SubAccum - Subtract ACCUM *b from ACCUM *a

SYNOPSIS

```
#include <math/math.h>

void SubAccum(ACCUM *a, ACCUM *b)
```

DESCRIPTION

Subtract the ACCUM **b* from the ACCUM **a*. If overflow occurs, set ACCUM **a* to plus or minus infinity. If either of the parameter value is Not A Number (NAN), the result in a will be a NAN.

***a*b**	**+In-range**	**-In-range**	**+0.0**	**-0.0**	**+Infinity**	**-Infinity**	**Zero**	**NAN**
+In-range	Sub	Sub	Sub	Sub	-Infinity	+Infinity	Sub	NAN
-In-range	Sub	Sub	Sub	Sub	-Infinity	+Infinity	Sub	NAN
+0.0	Sub	Sub	+0.0	-0.0	-Infinity	+Infinity	+0.0	NAN
-0.0	Sub	Sub	-0.0	+0.0	-Infinity	+Infinity	-0.0	NAN
+Infinity	+Infinity	+Infinity	+Infinity	+Infinity	NAN	-Infinity	+Infinity	NAN
-Infinity	-Infinity	-Infinity	-Infinity	-Infinity	-Infinity	NAN	-Infinity	NAN
Zero	Sub	Sub	-0.0	+0.0	-Infinity	+Infinity	Zero	NAN
NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN	NAN

SEE ALSO

AddAccum()

NAME

TanAccum - Calculate tangent of argument

SYNOPSIS

```
#include <math/math.h>

void TanAccum(ACCUM *a)
```

DESCRIPTION

Calculate tangent of ACCUM *a and return result in ACCUM *a. Result table for operand type:

+In-range	-In-range	+0.0	-0.0	+Infinity	-Infinity	Zero	NAN
TAN	TAN	+0.0	-0.0	NAN	NAN	Zero	NAN

WARNINGS

TanAccum() alters the value of errno.

SEE ALSO

AtnAccum()

NAME

TruncAccum - Truncate ACCUM *a to specified decimal place

SYNOPSIS

```
#include <math/math.h>

void TruncAccum(ACCUM *a, ACCUM *decimal)
```

DESCRIPTION

Truncate ACCUM **a* to the decimal place specified by ACCUM *decimal*. For example, **TruncAccum**(123.456, 1) equals 123.4 and **TruncAccum**(123.456, -1) equals 120.0. Any fractional digits of *decimal* are ignored.

SEE ALSO

RoundAccum()

NAME

ZeroAccum - Return exact zero

SYNOPSIS

```
#include <math/math.h>

void ZeroAccum(ACCUM *a)
```

DESCRIPTION

Store exact zero in ACCUM **a*.

SEE ALSO

OneAccum()

Chapter 5 – Date Functions

The date functions are used to perform various operations on values in the dL4 date formats (see “Date Format Codes”). Operations are never performed directly on any format other than **IDFMT_ACCUM** which corresponds to the **ACCUM** type definition. The non-ACCUM formats must always be loaded via **LoadDate**() into an **ACCUM** before performing any operation such as addition or multiplication. The final result is usually stored from the **ACCUM** into a specific date format using **StoreDate**().

The internal components of date formats, including those of the **ACCUM** format, should never be manipulated directly. Instead, date functions should be used. For example, always use the **MakeLocalTime**() function to convert an **ACCUM** into a **DATE** value. Any direct manipulation may result in compatibility problems in future releases of dL4.

The only public structure definition in the date library is **DATE** which is used by various conversion functions. **DATE** has the following layout:

```
typedef struct {
    int     Year;       /* 1 - 8920, base year is "zero" */
    int     Month;      /* 0 - 11 */
    int     Yearday;    /* day of year, 0 - 365 */
    int     Monthday;   /* day of month, 1 - length of month */
    int     Weekday;    /* day of week, 0 - 6 */
    int     Hour, Minute, Second, Millisecond;
    long    ZoneOffset; /* time zone offset in seconds */
} DATE;
```

Any member of **DATE** can be freely accessed and manipulated.

NAME

Date Format Codes - Format codes used by various calls

DESCRIPTION

These are the valid date format codes used by various functions in this package:

Code	Description
IDFMT_ACCUM	Intermediate date result
IDFMT_DAYS	16-bit unsigned number of days 0000 = Null date 0001 = 02 Jan 1900 00:00:00 GMT ffff = 04 Jun 2179 00:00:00 GMT
IDFMT_MINUTES	32-bit unsigned number of minutes 00000000 = Null date 00000001 = 01 Jan 0001 00:01:00 GMT ffffffff = 14 Feb 8166 22:07:00 GMT
IDFMT_MILLISEC	48-bit unsigned number of milliseconds 000000000000 = Null date 000000000001 = 01 Jan 0001 00:00:00.001 GMT fffffffffff = 31 Jul 8919 08:57:02.654 GMT

These format codes are defined in "date/date.h"

Use functions **DFormatAlign**() and **DFormatSize**() if you need information about specific date formats.

NAME

CurrentDate - Load current date into ACCUM

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void CurrentDate(ACCUM *a)
```

DESCRIPTION

Load the current date and time into ACCUM **a*. The date in ACCUM **a* will be expressed in seconds since January 1, year 1, 00:00:00 GMT.

SEE ALSO

LoadDate()

NAME

DFormatAlign - Return log2(format alignment in bytes)

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

int DFormatAlign(int fmt)
```

DESCRIPTION

Return the date format alignment factor for the specified date format. This is a power of 2. The following values are returned:

Format code	Value
IDFMT_ACCUM	2
IDFMT_DAYS	1
IDFMT_MINUTES	1
DFMT_MILLISEC	1

RETURN VALUE

An invalid format code will return -1.

SEE ALSO

DFormatSize(), **Date format codes**

NAME

DFormatSize - Return size in bytes of specified date format

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

size_t DFormatSize(int fmt)
```

DESCRIPTION

Return the size in bytes of the specified date format.

The following values are returned:

Format code	Value
IDFMT_ACCUM	sizeof(ACCUM)
IDFMT_DAYS	sizeof(DATE_DAYS)
IDFMT_MINUTES	sizeof(DATE_MINUTES)
IDFMT_MILLISEC	sizeof(DATE_MILLISEC)

The actual sizes may vary from machine to machine.

RETURN VALUE

An invalid format code will return 0.

SEE ALSO

DFormatAlign(), **Date format codes**

NAME

GMTStringToDate - Convert GMT date and time from string to ACCUM
NativeGMTStringToDate - Convert GMT date and time from string to ACCUM
NativeStringToDate - Convert local date and time from string to ACCUM
StringToDate - Convert local date and time from string to ACCUM

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

size_t GMTStringToDate(ACCUM *a, const UNICODE *s, size_t len)
size_t NativeGMTStringToDate(ACCUM *a, const UNICODE *s, size_t len)
size_t NativeStringToDate(ACCUM *a, const UNICODE *s, size_t len)
size_t StringToDate(ACCUM *a, const UNICODE *s, size_t len)
```

DESCRIPTION

Convert the date and time in UNICODE string *s into a datein ACCUM *a. The date in ACCUM *a will be expressed in seconds since January 1, year 1, 00:00:00 GMT. The date and time are treated as being Greenwich Mean time or relative to the local time zone, depending on the function used. If the string contains an illegal date, ACCUM *a will be set to NAN (Not A Number).

The NativeXXXX() functions convert the date and time from a local language UNICODE string.

RETURN VALUE

0	Invalid date/time
x	Number of characters processed by function

SEE ALSO

LoadDate(), **Converting Dates to Strings**, **Date/time string Format**

NAME

DateOrdering - Return preferred year, month, day order

SYNOPSIS

```
#include <date/date.h>

DATEORDER DateOrdering(void)
DATEORDER NativeDateOrdering(void)
```

DESCRIPTION

Return the default or native date order as a DATEORDER value. DATEORDER is an enumerated type with the following values:

DATEORD_YMD	Year/Month/Day
DATEORD_YDM	Year/Day/Month
DATEORD_MDY	Month/Day/Year
DATEORD_MYD	Month/Year/Day
DATEORD_DMY	Day/Month/Year
DATEORD_DYM	Day/Year/Month

SEE ALSO

NativeMonthName(), **DateSeparator**(), **NativeDateSeparator**()

NAME

DateSeparator - Return default date separator string
NativeDateSeparator - Return native date separator string

SYNOPSIS

```
#include <date/date.h>

const UNICODE *DateSeparator(void)
const UNICODE *NativeDateSeparator(void)
```

DESCRIPTION

Return the default or native date separator string (such as "/" or ".").

SEE ALSO

NativeMonthName(), **DateOrdering**(), **NativeDateOrdering**()

NAME

DateToGMTString - Convert GMT date and time from ACCUM to string
DateToString - Convert local date and time from ACCUM to string
NativeDateToGMTString - Convert GMT date and time from ACCUM to string
NativeDateToString - Convert local date and time from ACCUM to string

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

ssize_t DateToGMTString(UNICODE *s, size_t size, ACCUM *a)
ssize_t DateToString(UNICODE *s, size_t size, ACCUM *a)
ssize_t NativeDateToGMTString(UNICODE *s, size_t size, ACCUM *a)
ssize_t NativeDateToString(UNICODE *s, size_t size, ACCUM *a)
```

DESCRIPTION

Convert the date and time in ACCUM ***a** to an UNICODE string. The date in ACCUM ***a** is expressed in seconds since January 1, year 1, 00:00:00 GMT. The string will use the “CALL $TIME” format (“JAN 19, 1992 14:10:15.015”) and use Greenwich Mean time or the local time zone, depending on the function used. If the receiving string is too small, the date string will be truncated.

For the NativeXXXX() functions, If the date/time is within the range of time_t (as defined in Posix), the local format as generated by the %c specifier (date and time formatted for the current locale) of Posix function strftime() or its equivalent will be used. Otherwise, the “CALL $TIME” format (“JAN 19, 1992 14:10:15.015”) will be used.

RETURN VALUE

-1	ACCUM **a* was zero or a NAN (Not a Number).
x	Number of characters in result string.

SEE ALSO

Converting Strings to Dates, **strftime**()

NAME

Date and Time String Format - Unicode text date format

DESCRIPTION

This is the UNICODE string format for specifying text dates as used by various functions in this package:

<dateandtime> ::= [<spaces>] <date> [<sep> <time>]

<date> ::= <month> <sep> <day> <sep> <year> |
<year> <sep> <month> <day> |
<day> <sep> <month> <sep> <year>

<month> ::= <month name> | <integer between 1 and 12>

<day> ::= <integer between 1 and 31>

<year> ::= <integer between 68 and 99 (assumed origin 1900)> |
<integer between 0 and 67 (assumed origin 2000)> |
<integer greater than 99 (assumed origin 0)>

<time> ::= <hour> [<sep> <minute> <sep> <seconds> [<sep> <millisecs>]]]["am" | "AM" | "pm" | "PM"]

<sep> ::= <a sequence of non-alphabetic and non-numeric characters>

NAME

GMTime() - Return Greenwich Mean year, month, day, time of date
LocalTime() - Return local year, month, day, time of specified date

SYNOPSIS

```
#include <date/date.h>

int GMTime(DATE *date, ACCUM *a)
int LocalTime(DATE *date, ACCUM *a)
```

DESCRIPTION

Return year, month, day, day of week, and time from date in ACCUM *a* in DATE structure date. These functions return -1 if ACCUM *a* is a NAN (Not A number), is zero, or if the date exceeds its range (01 Jan 0001 through 31 Jul 8919). **LocalTime**() is an extended range version of the standard **localtime**() function with different parameters. **GMTime**() is an extended range version of the standard **gmtime**() function with different parameters.

NOTES

Times outside of the range of localtime() are treated as using the same daylight savings rules as the nearest years within the range of localtime().

RETURN VALUE

0	Successful
-1	Failed

SEE ALSO

CurrentDate(), **YearDate**(), **MonthDate**(), **WeekdayDate**(), **NumbersToDate**()

NAME

InitDate - Initialize date package

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void InitDate(void)
```

DESCRIPTION

InitDate() is responsible for initializing the date package. This initialization includes setting up the month and day name tables.

SEE ALSO

ShutdownDate()

NAME

LoadDate - Load date in specified format into an ACCUM

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void LoadDate(ACCUM *a, const void *n, int fmt)
```

DESCRIPTION

Load a date in the specified format from pointer n into ACCUM **a*. The date in ACCUM **a* will be expressed in seconds since January 1, year 1, 00:00:00 GMT. Thus a date type IDFMT_MILLISEC (milliseconds since 01/01/0001) value of 123456 would be loaded into ACCUM **a* as 123.456. If the date is NULL (zero) or the format is not supported, ACCUM **a* will be set to NAN (Not A Number).

SEE ALSO

Date format codes, **StoreDate**(), **LoadDItem**()

NAME

MakeLocalTime - Calculate date from DATE structure of local time
MakeGMTime - Calculate date from DATE structure of GM time

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

int MakeLocalTime(ACCUM *a, DATE *date)
int MakeGMTime(ACCUM *a, DATE *date)
```

DESCRIPTION

MakeLocalTime() and **LocalTime**() calculate the date specified in the Year, Month, Monthday, Hour, Minute, Second, and Millisecond members of a DATE structure. The DATE structure is treated as a local time for **MakeLocalTime**() and a Greenwich Mean time for **MakeGMTime**(). The date in ACCUM *a* will be expressed in seconds since January 1, year 1, 00:00:00 GMT. If any of the members exceed their normal values (Hour > 24), then ACCUM *a* will be set to a NAN and minus one will be returned by the function.

RETURN VALUE

-1	Invalid date/time, ACCUM *a* set to NAN
0	Valid date/time return in ACCUM *a*.

NAME

MonthDate - Return month of specified date

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void MonthDate(ACCUM *a)
```

DESCRIPTION

Convert ACCUM **a* from a date in seconds to the month of that date (January equals one). If ACCUM **a* is a NAN (Not a Number) or a zero, then ACCUM **a* will be set to a NAN.

SEE ALSO

YearDate(), **WeekdayDate**()

NAME

MonthdayDate - Return day of month for specified date

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void MonthdayDate(ACCUM *a)
```

DESCRIPTION

Convert ACCUM **a* from a date in seconds to the day of the month of that date (origin one). If ACCUM **a* is a NAN (Not a Number) or a zero, then ACCUM **a* will be set to a NAN.

SEE ALSO

WeekdayDate(), **YearDate**(), **MonthDate**()

NAME

NumbersToDate() - Convert date components into true date

SYNOPSIS

```
#include <date/date.h>

void NumbersToDate(ACCUM *dateacc, ACCUM *datetime, size_t n)
```

DESCRIPTION

NumbersToDate() converts the local time expressed by the n elements of ACCUM array *datetime* into a date in ACCUM *dateacc*. The elements of *datetime* are interpreted as follows:

datetime[0] - year
datetime[1] - month, 1 to 12
datetime[2] - day, 1 - 31
datetime[3] - hour
datetime[4] - minute
datetime[5] - second

Except for the seconds value (*datetime*[5]), any fractional value will be ignored. If any illegal date value is specified (such as month 13), *dateacc* will be set to NAN (Not-A-Number). If only a date is specified (n < 3), then the time will be treated as Greenwich Mean time of noon so that the date remains constant in all timezones (almost).

RETURN VALUE

0	Date successfully converted
-1	Illegal date

SEE ALSO

StringToDate(), **Convert to DATE structure**

NAME

ShutdownDate - Shutdown date package

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void ShutdownDate(void)
```

DESCRIPTION

ShutdownDate() is responsible for terminating the date package. This termination includes releasing the month and day name tables.

SEE ALSO

InitDate()

NAME

StoreDate - Store date from an ACCUM in a specified date format

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

int StoreDate(void *n, ACCUM *a, int fmt)
```

DESCRIPTION

Store the date in ACCUM **a* through pointer *n* according to the specified date format.

RETURN VALUE

0	StoreDate() succeeded
-1	StoreDate() failed because the date in ACCUM **a* exceeded the range of the specified format or the specified format was illegal.

SEE ALSO

Date format codes, **LoadDate**()

NAME

WeekdayDate - Return week day of specified date

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void WeekdayDate(ACCUM *a)
```

DESCRIPTION

Convert ACCUM **a* from a date in seconds to the day of the week of that date (Sunday equals 1). If ACCUM **a* is a NAN (Not a Number) or a zero, then ACCUM **a* will be set to a NAN.

SEE ALSO

MonthdayDate(), **YearDate**(), **MonthDate**()

NAME

YearDate - Return year of specified date

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void YearDate(ACCUM *a)
```

DESCRIPTION

Convert ACCUM **a* from a date in seconds to the year of that date. If ACCUM **a* is a NAN (Not a Number) or a zero, then ACCUM **a* will be set to a NAN.

SEE ALSO

MonthDate(), **WeekdayDate**()

NAME

YeardayDate - Return day of year for specified date

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void YeardayDate(ACCUM *a)
```

DESCRIPTION

Convert ACCUM **a* from a date in seconds to the day of the year of that date (origin one). If ACCUM **a* is a NAN (Not a Number) or a zero, then ACCUM **a* will be set to a NAN.

SEE ALSO

MonthdayDate(), **WeekdayDate**(), **YearDate**(), **MonthDate**()

NAME

ZoneOffset - Return time zone offset from GMT for date/time

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

void ZoneOffset(ACCUM *a)
```

DESCRIPTION

Determine the offset in seconds from the local time zone to Greenwich Mean time for the date and time specified by the date in ACCUM *a*. This offset includes any known daylight savings time or other special time zone offsets. Return the offset in ACCUM *a*.

SEE ALSO

GMTStringToDate()

NAME

AbvDayName - Return abbreviated name of a day of the week
DayName - Return name of a day of the week
NativeAbvDayName - Return local abbreviated name of a day of the week
NativeDayName - Return local name of a day of the week

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

const UNICODE *AbvDayName(int day)
const UNICODE *DayName(int day)
const UNICODE *NativeAbvDayName(int day)
const UNICODE *NativeDayName(int day)
```

DESCRIPTION

Return a UNICODE pointer to statically allocated storage containing the null terminated name (abbreviated depending on the function) of the specified day (Sunday equals 1).

The NativeXXXX() functions return local day names.

Do not change the data located at the returned UNICODE pointer. Since it is static, any change will be permanent until the process ends. The day names are saved similar to the way environment variables are stored.

RETURN VALUE

(UNICODE *) 0	Illegal day specified
x	UNICODE pointer to static area containing the name

SEE ALSO

Month Names

NAME

DateToJulian - Convert a date value to a Julian day date
JulianToDate - Convert a Julian day date into a date value

SYNOPSIS

```
#include <date/date.h>

long DateToJulian(ACCUM *date)
int JulianToDate(ACCUM *date, long julian)
```

DESCRIPTION

DateToJulian() converts an ACCUM date to a Julian day date.

JulianToDate() converts a julian day date to an ACCUM GMT date at noon. Noon is used so that the local date will be constant whatever the time zone (almost).

RETURN VALUE

x	(DateToJulian()) Success, 'x' is the converted Julian day date
0	(JulianToDate()) Success, date value loaded into ACCUM
-1	Error, date out of range

NAME

AbvMonthName - Return abbreviated month name
MonthName - Return month name
NativeAbvMonthName - Return local abbreviated month name
NativeMonthName - Return local month name

SYNOPSIS

```
#include <math/math.h>
#include <date/date.h>

const UNICODE *AbvMonthName(int month)
const UNICODE *MonthName(int month)
const UNICODE *NativeAbvMonthName(int month)
const UNICODE *NativeMonthName(int month)
```

DESCRIPTION

Return a UNICODE pointer to a statically allocated area containing the null terminated name (abbreviated depending on the function used) of the specified month (January equals 1).

The NativeXXXX() functions return local month names.

Do not change the data located at the returned UNICODE pointer.

RETURN VALUE

(UNICODE *) 0	Illegal month specified
x	UNICODE pointer to static area containing name

SEE ALSO

Day Names

Chapter 6 – String Functions

The string functions are used to perform various operations on Unicode and ASCII strings. Most functions will treat a Unicode binary zero or ASCII binary zero (‘\0’) as a string terminator. The strings library as defined by the strings/strings.h include file does not contain any public structure definitions. All structure types defined by strings/strings.h should be manipulated only via dL4 runtime library functions and never directly accessed or modified. Any direct access or manipulation may cause compatibility problems in future releases of dL4.

NAME

AscCICompare - Case-insensitive string compare
AscCINCompare - Limited length case-insensitive string compare
AscCompare - Case-sensitive string compare
AscNCompare - Limited length case-sensitive string compare

SYNOPSIS

```
#include <strings/strings.h>

int AscCICompare(const char *s1, const char *s2)
int AscCINCompare(const char *s1, const char *s2, size_t n)
int AscCompare(const char *s1, const char *s2)
int AscNCompare(const char *s1, const char *s2, size_t n)
```

DESCRIPTION

These functions compare string s1 against string s2 stopping after the NULL or the first difference.

AscNCompare() and **AscCINCompare**() compare string *s1* against string *s2* stopping after the NULL, first difference, or n-characters. The strings are deemed equal if there is no difference in the first n-characters.The comparison is case-insensitive for the functions AscCICompare() and AscCINCompare(). All other comparisons are case-sensitive.

RESTRICTIONS

Assumes **s1* and **s2* are terminated by a NULL character.

RETURN VALUE

The result is < 0, == 0, or > 0 when *s1* is less than, equal to, or greater than *s2*.

SEE ALSO

Comparing Unicode Strings, **Comparing Unicode and ASCII Strings**

NAME

AscUcCopy - Copy ASCII string to Unicode string
AscUcNCopy - Copy ASCII string to Unicode string (length limited)

SYNOPSIS

```
#include <strings/strings.h>

size_t AscUcCopy(const UNICODE *s1, const char *s2)
size_t AscUcNCopy(const UNICODE *s1, const char *s2, size_t n)
```

DESCRIPTION

These functions copy 8-bit ASCII characters from **s2* to Unicode characters in **s1*, zero extending upper 8-bits, stopping after copying the NULL.

Function AscUcNCopy() copies at most *n* characters.

RESTRICTIONS

Assumes **s2* is terminated by an ASCII NULL character.

RETURN VALUE

size_t Number of characters copied, excluding the NULL.

SEE ALSO

ConvertFromUnicode(), **ConvertToUnicode**(), **Copying Unicode Characters**

NAME

ComputeCRC - Begin computing a new CRC32 value
ContinueCRC - Continue computing a CRC32 value

SYNOPSIS

```
CRC32 ComputeCRC(const void *ptr, size_t siz)
CRC32 ContinueCRC(CRC32 crc, const void *ptr, size_t siz)
```

DESCRIPTION

These functions compute a 32-bit CRC of a region(s) of memory. The CRC polynomial is TBD.

RETURN VALUE

X	The CRC32 value of the specified memory region

SEE ALSO

MD5

NAME

ConvertToLower -- Translate Unicode character string to lower-case
ConvertToUpper -- Translate Unicode character string to upper-case

SYNOPSIS

```
size_t ConvertToLower(UNICODE *dptr, size_t dlen, const UNICODE *sptr,
size_t slen, unsigned int flags)
size_t ConvertToUpper(UNICODE *dptr, size_t dlen, const UNICODE *sptr,
size_t slen, unsigned int flags)
```

DESCRIPTION

Copies the specified source string '*sptr*' to '*dptr*',converting any lower- or upper-case characters to lower- or upper-case. The '*flags*' argument may be set to CVT_NORMAL to stop conversion on a null character, or CVT_NULLS, to convert past nulls.

Note that '*dlen*' and '*slen*' represent string lengths in Unicode characters, which is not the same as byte length.

NOTES

For upper-case conversions, the character LATIN SMALL LETTER SHARP S converts to two 'S' characters. This must be accounted for in the size of the destination string.

RETURN VALUE

size_t Number of characters written to destination string.

SEE ALSO

ToAscLower(), **ToAscUpper**(), **Unicode character functions**

NAME

GetLocale - Return local language numeric formatting parameters
GetDefaultLocale - Return standard numeric formatting parameters

SYNOPSIS

```
#include <strings/strings.h>

const LOCALE *GetLocale(void)
const LOCALE *GetDefaultLocale(void)
```

DESCRIPTION

GetLocale() returns a pointer to a LOCALE structure that provides local language numeric formatting information such as the Unicode characters for the decimal point as well as the digits themselves. **GetDefaultLocale**() returns a pointer for dL4 standard numeric formatting (essentially ASCII in Unicode).

Structure LOCALE is defined in strings/strings.h

RETURN VALUE

a pointer to a statically allocated LOCALE structure

NAME

InitStrings - Initialize strings package

SYNOPSIS

```
#include <strings/strings.h>

void InitStrings(void)
```

DESCRIPTION

Initialize global variables for string handling package and perform assertions of package assumptions. This should be invoked before using any dL4 strings functions.

SEE ALSO

ShutdownStrings()

NAME

Unicode Character Functions

SYNOPSIS

```
#include <strings/strings.h>

int IsUcAsc(UNICODE c)
int IsUcAscAlnum(UNICODE c)
int IsUcAscAlpha(UNICODE c)
int IsUcAscDigit(UNICODE c)
int IsUcAscLower(UNICODE c)
int IsUcAscUpper(UNICODE c)
int IsUcDigit(UNICODE c)
int IsUcLower(UNICODE c)
int IsUcMinus(UNICODE c)
int IsUcPlus(UNICODE c)
int IsUcSpace(UNICODE c)
int IsUcUpper(UNICODE c)
int IsUcXDigit(UNICODE c)
int IsUcWhiteSpace(UNICODE c)
UNICODE ToUcLower(UNICODE c)
UNICODE ToUcUpper(UNICODE c)
char UcToAscLower(UNICODE c)
char UcToAscUpper(UNICODE c)
```

DESCRIPTION

IsUcAsc()/IsUcAscAlpha()/IsUcAscAlnum()/IsUcAscLower()/IsUcAscUpper() determines whether its argument is a valid ASCII character/letter/alpha-numeric/ASCII lower/upper case letter.

IsUcDigit() and IsUcXDigit() determine whether its argument is a valid Unicode decimal or hexadecimal digit and if so returns its integer value (base zero, not ASCII ‘0’).

IsUcMinus()/IsUcPlus()IsUcSpace() determines whether its argument is a valid Unicode minus/plus/space.

UcToAscLower()/UcToAscUpper() translates an Unicode upper/lower case letter (in the ASCII range) to ASCII lower/upper case. Any other character is unaffected. This is identical to the ToAscLower() and ToAscUpper() functions except that an Unicode argument is used instead of a char argument.

ToUcLower() and ToUcUpper() are identical to UcToAscLower() and UcToAscUpper() except that an Unicode value is returned instead of a char value.

IsUcAscLower() and IsUcLower() a are the same function.

IsUcAscLower() are IsUcUpper() are the same function.

IsUcWhiteSpace() determines whether its argument is a Unicode white-space character. White-space is defined asfollows:

Unicode value	Description
0x0009	Horizontal Tab
0x000a	Line Feed
0x000b	Vertical Tab
0x000c	Form Feed
0x000d	Carriage Return

RESTRICTIONS

IsUcAscLower(), IsUcAscUpper(), UcToAscLower(), and UcToAscUpper() are optimized to assume their arguments are in the ASCII range. Their behavior is undefined when passed non-ASCII characters.

RETURN VALUE

IsUcAsc(), **IsUcAscAlnum**(), **IsUcAscAlpha**(), **IsUcAscDigit**(), **IsUcAscLower**(), **IsUcAscUpper**(), **IsUcMinus**(), **IsUcPlus**(), **IsUcSpace**(), **IsUcWhiteSpace**()

0	False.
!0	True.

IsUcDigit(), **IsUcXDigit**()

-1	not a digit
>=0	Is a dedimal digit (0-9) or a hex digit (0-15)

UcToAscLower(), **UcToAscUpper**()

(altered character)

SEE ALSO

ToAscLower(), **ToAscUpper**(), **Case** Conversion, **ParseWSCItem**(), **SkipWSCItem**(), **SkipUcWhiteSpace**()

NAME

LongToUc - Convert long to Unicode string

SYNOPSIS

```
#include <strings/strings.h>

size_t LongToUc(UNICODE *s, size_t n, long v)
```

DESCRIPTION

LongToUc() converts the long value ‘v’ to a Unicode string beginning at performed in base 10. If the destination string is too small to hold the converted value, LongToUc() returns (size_t)0, with ‘s’ unchanged.

RETURN VALUE

(size_t)0	String *'s'* too small to convert *'v'*.
size_t	Number of Unicode characters output (excluding null).

SEE ALSO

UcToLong(), **UcToULong**(), **ULongToUc**(), **ULongToUcOctal**(), **ULongToUcHex**()

NAME

AddToNameTable - Add name to name table
CreateNameTable - Create name table
FreeNameTable - Free name table
SearchNameTable - Lookup name in name table

SYNOPSIS

```
#include <strings/strings.h>

int AddToNameTable(NAMETABLE *tbl, const void *name, size_t n,
const void *value)
int CreateNameTable(NAMETABLE *tbl, size_t valuesize, size_t increment)
void FreeNameTable(NAMETABLE *tbl)
int SearchNameTable(void *value, NAMETABLE *tbl, const void *name,
size_t n)
```

DESCRIPTION

The name table associates a name, which is a binary object, with a value, which is another binary object. Each value must be the same size: be of any size between 0 and USHRT_MAX inclusive. The size of each entry is rounded up to an integral number of unsigned shorts to align the name length.

Name Tables are similar to Association Tables except that Name Tables store more than just Unicode character translations. Also, Association Tables are faster when saving more than a few items and they should be used for Unicode translations.

CreateNameTable() initializes a Name Table by setting the value size and allocating the store table.

AddToNameTable() adds '*name'* of length *'n*' ' of leng units) and value

SearchNameTable() searches the name table for '*name*' of length '*n*' (sizeof() units). If found, copy the associated value to '**value*'.

FreeNameTable() deallocates all memory used in a nametable.

RETURN VALUE

0	Success.
-1	Failure. Name already exists (AddToNameTable()) or does not exist (SearchNameTable()). Use GetError() for error code.

SEE ALSO

Association Tables

NAME

ParseOctHexChar - Convert an octal or hex Unicode string into a Unicode character.

SYNOPSIS

```
#include <strings/strings.h>

UNICODE ParseOctHexChar(const UNICODE **ptr)
```

DESCRIPTION

This function converts a Unicode string into a Unicode character. The string must contain either an octal or hex number in Unicode string form and surrounded by a delimiting character. This delimiting character may be any Unicode character, but normally " is used. If there is an error in the number (i.e. a bad digit), the Unicode delimiter is returned.

The encoded octal/hex value must fit in a single Unicode character. Therefore, the maximum value that can be converted is 0xffff = 0177777 = 65535. The minimum is 0.

Hex numbers must begin with 'x'.

Examples:

hex: \0x1F\ zxe12az (=0xe12a) "0x123"

octal: \233\ z234z "14040"

On successful return, ptr points to the terminating delimiter.

If an error occurred, ptr is unchanged.

RETURN VALUE

x	Converted Unicode character, Unicode delimiter (ie: Unicode'\') if error

SEE ALSO

ParseSymbol(), **Unicode character functions**

NAME

ParseSymbol - Convert an Unicode symbol into ASCII
ParseNSymbol - Convert an Unicode symbol into ASCII

SYNOPSIS

#include <strings/strings.h>

size_t **ParseSymbol**(char **dest*, const UNICODE ***src*, size_t *maxlen*)

size_t **ParseNSymbol**(char **dest*, size_t *destsiz*, const UNICODE ***src*, size_t *srclen*)

DESCRIPTION

Convert the Unicode symbol at **src* to ASCII and store at **dest*. The symbol may begin with either an alphabetic character or (UNICODE)'_', but not a digit.

RETURN VALUE

x	0 if error, otherwise set to the length of the converted ASCII string (in *dest). The converted ASCII string is copied into here. *src is set to point to the next Unicode character after the symbol just parsed.

SEE ALSO

ParseOctHexChar(), **Unicode character functions**

NAME

ShutdownStrings - Shutdown strings package

SYNOPSIS

```
#include <strings/strings.h>

void ShutdownStrings()
```

DESCRIPTION

Shutdown string package.

SEE ALSO

InitStrings()

NAME

SkipUcWhiteSpace - Skip over UNICODE white space characters

SYNOPSIS

```
#include <strings/strings.h>

const UNICODE *SkipUcWhiteSpace(const UNICODE *s)

const UNICODE *SkipNUcWhiteSpace(const UNICODE *s,size_t n)
```

DESCRIPTION

Advance UNICODE pointer ‘s’ until a non-whitespace character is found.

RETURN VALUE

x UNICODE pointer to next non-white-space character or end of string

SEE ALSO

IsUcWhiteSpace()

NAME

ToAscLower - Translate an ASCII character to ASCII lower case
ToAscUpper - Translate an ASCII character to ASCII upper case

SYNOPSIS

```
#include <strings/strings.h>

char ToAscLower(char c)
char ToAscUpper(char c)
```

DESCRIPTION

ToAscLower()ToAscLower()/ translates an ASCII upper/lowercase letter to ASCII lower/upper case. Any other character is unaffected. This is identical to UcToAscLower() and is identical t except that a char argument is used instead of an Unicode argument.

SEE ALSO

Unicode character functions, **Case Conversion**

NAME

ULongToUc - Convert unsigned long to Unicode string
ULongToUcHex - Convert unsigned long to Unicode Hexadecimal string
ULongToUcOctal - Convert unsigned long to Unicode Octal string
ULongToUcBaseN - Convert unsigned long to Unicode base-N string
ULongToUcRJust - Convert unsigned long to right justified Unicode string

SYNOPSIS

```
#include <strings/strings.h>

size_t ULongToUc(UNICODE *s, size_t n, unsigned long v)
size_t ULongToUcHex(UNICODE *s, size_t n, unsigned long v)
size_t ULongToUcOctal(UNICODE *s, size_t n, unsigned long v)
size_t ULongToUcBaseN(UNICODE *s, size_t n, unsigned long v, int base)
size_t ULongToUcRJust(UNICODE *s, size_t n, unsigned long ul, size_t
      width)
```

DESCRIPTION

ULongToUc() converts the unsigned long value '*v*' to a Unicode string beginning at '*s*'. '*n*' gives the size of the destination string '*s*'. Conversion is performed in either decimal, octal, hex, or a specified base (2 - 36). If the destination string is too small to hold the converted value, **ULongToUc**() returns (size_t)0, with '*s*' unchanged.

ULongToUcRJust() right justifies the conversion within a field of '*width*' characters.

RETURN VALUE

(size_t)0	String 's' too small to convert '*v*'.
size_t	Number of Unicode characters output (excluding null).

SEE ALSO

LongToUc(), **UcToLong**(), **UcToULong**()

NAME

UcAscCICompare - Case-insensitive Unicode to ASCII string compare
UcAscCINCompare - Case-insensitive Unicode to ASCII string compare (length limited)
UcAscCompare - Compare Unicode string with ASCII string
UcAscNCompare - Compares two ASCII Unicode strings for equality (length limited)

SYNOPSIS

```
#include <strings/strings.h>

int UcAscCICompare(const UNICODE *s1, const char *s2)
int UcAscCINCompare(const UNICODE *s1, const char *s2, size_t n)
int UcAscCompare(const UNICODE *s1, const char *s2)
int UcAscNCompare(const UNICODE *s1, const char *s2, size_t n)
```

DESCRIPTION

These functions compare Unicode string *s1* against character string *s2* stopping after the NULL.

Functions **UcAscCINCompare**() and **UcAscNCompare**() stop after *n* Unicode characters if the strings are equal. (They stop before that if the strings are not equal.)

The comparison is case-insensitive for functions **UcAscCICompare**() and **UcAscCINCompare**().

RESTRICTIONS

Assumes **s1* and **s2* are terminated by a NULL character.

RETURN VALUE

The result is < 0, == 0, or > 0 when *s1* is less than, equal to, or greater than *s2*. Unicode characters outside the ASCII range are considered greater than.

SEE ALSO

Comparing ASCII Strings, **Comparing Unicode Strings**

NAME

UcCICompare - Case-insensitive Unicode string compare
UcCINCompare - Case-insensitive Unicode string compare (length limited)
UcCompare - Compares two Unicode strings for equality
UcNCompare - Compares two Unicode strings for equality (length limited)

SYNOPSIS

```
#include <strings/strings.h>

int UcCICompare(const UNICODE *s1, const UNICODE *s2)
int UcCINCompare(const UNICODE *s1, const UNICODE *s2, size_t n)
int UcCompare(const UNICODE *s1, const UNICODE *s2)
int UcNCompare(const UNICODE *s1, const UNICODE *s2, size_t n)
```

DESCRIPTION

These functions compare Unicode string **s1* against Unicode string **s2* stopping after the NULL.

Functions **UcCINCompare**() and **UcNCompare**() stop after *n* Unicode characters if the strings are equal. (They stop before that if the strings are not equal.)

The comparison is case-insensitive for functions **UcCICompare**() and **UcCINCompare**(). All other comparisons are case-sensitive.

RESTRICTIONS

Assumes **s1* and **s2* are terminated by a Unicode NULL character.

RETURN VALUE

The result is < 0, == 0, or > 0 when *s1* is less than, equal to, or greater than *s2*.

SEE ALSO

Comparing ASCII strings, **Comparing Unicode and ASCII strings**

NAME

UcCopy - Copy Unicode strings
UcNCopy - Copy Unicode strings (length limited)

SYNOPSIS

```
#include <strings/strings.h>

size_t UcCopy(const UNICODE *s1, const UNICODE *s2)
size_t UcNCopy(const UNICODE *s1, const UNICODE *s2, size_t n)
```

DESCRIPTION

UcCopy() copies Unicode characters from **s2* to **s1* stopping after copying the NULL.

UcNCopy() copies exactly *n* characters from **s2* to **s1*, truncating **s2* or adding null characters to **s1* if necessary. The result is not null-terminated if the length of **s2* is *n* or more.

RESTRICTIONS

Assumes **s2* is terminated by a Unicode NULL character.

RETURN VALUE

size_t Number of characters copied, excluding the NULL.

SEE ALSO

AscUcCopy(), **ConvertFromUnicode**(), **ConvertToUnicode**()

NAME

UcHexToBinary - Convert Unicode string of hexidecimal digits to binary

SYNOPSIS

```
#include <strings/strings.h>

size_t UcHexToBinary(BYTE *dest, size_t destsize, const UNICODE
      **termptr, const UNICODE *srcptr, size_t srcsize)
```

DESCRIPTION

UcHexToBinary() converts the Unicode string of hexadecimal digits '*srcptr*' of length '*srcsize*' to an array of binary bytes in '*dest*'. **UcHexToBinary**() returns the number of bytes stored in '*dest*' and sets '*termptr*', if not NULL, to point at the first character in '*srcptr*' that wasn't converted. The conversion stops at the first non-hexadecimal character pair in or at the end of '*dest*'.

RETURN VALUE

Returns the number of bytes stored in 'dest'

SEE ALSO

LongToUc(), **UcToULong**(), **ULongToUc**(), **ULongToUcOctal**(), **ULongToUcHex**()

NAME

UcLength - Length of a Unicode string
UcNLength - Length of a Unicode string (length limited)

SYNOPSIS

```
#include <strings/strings.h>

size_t UcLength(const UNICODE *s)
size_t UcNLength(const UNICODE *s, size_t n)
```

DESCRIPTION

These functions determine length of its argument, excluding the NULL, in Unicode characters.

UcNLength() determines length of its argument, excluding the NULL, in Unicode characters, up to *n* characters.

RESTRICTIONS

Assumes a the string is terminated by a Unicode NULL character.

RETURN VALUE

Length in Unicode characters.

NAME

UcNSearch - Length-limited UNICODE string search
UcCINSearch - Length-limited case-insensitive UNICODE string search

SYNOPSIS

```
#include <strings/strings.h>

const UNICODE *UcNSearch(const UNICODE *s, size_t ssize, const UNICODE
*t, size_t tsize)
const UNICODE *UcCINSearch(const UNICODE *s, size_t ssize, const UNICODE
*t, size_t tsize)
```

DESCRIPTION

UcNSearch() searches the Unicode string *s* of size *ssize* for a substring matching the Unicode target string *t* which is of size *tsize*. **UcCINSearch** performs a case-insensitive search. Both *s* and *t* may be terminated by a Unicode ‘ ‘ and thus they may be shorter than *ssize* or *tsize*.

RETURN VALUE

Returns a pointer to the matching substring in *s*. If a match is not found, NULL is returned.

SEE ALSO

UcNCompare(), **UcCINCompare**()

NAME

UcToBoolean - Convert Unicode string to integer boolean value

SYNOPSIS

```
#include <strings/strings.h>

int UcToBoolean(const UNICODE *s, const UNICODE **ptr, size_t n)
```

DESCRIPTION

UcToBoolean() returns the integer boolean value represented by the Unicode string '*s*'. '*s*' is considered to contain a boolean value if the leading characters of '*s*' match any of the following strings ignoring case:

"t", "true", "false", "f", "on", "off", "y", "yes", "no", "n", "0", "1"

It is the caller's responsibility to check if the trailing characters are legal (for example, "fine" will match "f", but the caller should reject "i" as a terminator). If the value of 'ptr' is not (UNICODE **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by '*ptr*'. If no boolean is recognized, that location is set to '*s*', and zero is returned.

RETURN VALUE

Returns 0 if a FALSE value is recognized and 1 if a TRUE value is recognized. If no value is recognized, a 0 is returned and the returned value in '*ptr*' will equal '*s*'.

SEE ALSO

LongToUc(), **UcToLong**(), **ULongToUc**(), **ULongToUcOctal**(), **ULongToUcHex**()

NAME

UcToLong - Unicode string to long

SYNOPSIS

```
#include <strings/strings.h>

long UcToLong(const UNICODE *s, const UNICODE **ptr, size_t n)
```

DESCRIPTION

UcToLong() returns the long integer value represented by the string of Unicode decimal digits *s*. If the value of *ptr* is not (UNICODE **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *s*, and zero is returned.

RETURN VALUE

Returns the decimal value represented by *s*. On overflow or underflow returns LONG_MAX or LONG_MIN.

SEE ALSO

LongToUc(), **UcToULong**(), **ULongToUcb ULongToUcOctal**(), **ULongToUcHex**()

NAME

UcToLongBaseN - Unicode string to long Base N

SYNOPSIS

```
long UcToLongBaseN(const UNICODE *s, const UNICODE **ptr, size_t n,
      int base)
```

DESCRIPTION

UcToLongBaseN() returns the long integer value represented by the string of Unicode decimal digits *s* using base, or if base is zero, the base specified by the string ("0xnn" for hexidecimal, "0nn" for octal, otherwise decimal). If the value of *ptr* is not (UNICODE **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by *ptr*. If no integer can be formed, that location is set to *s*, and zero is returned. If overflow occurs, parsing continues, but a value of LONG_MIN or LONG_MAX will be returned.

RETURN VALUE

Returns the value represented by *s*. If overflow occurs, LONG_MIN or LONG_MAX is returned to match the sign of the number.

NAME

UcToULong - Unicode string to unsigned long

SYNOPSIS

```
#include <strings/strings.h>

unsigned long UcToULong(const UNICODE *s, const UNICODE **ptr, size_t n)
```

DESCRIPTION

UcToULong() returns the unsigned long integer value represented by the string of Unicode decimal digits s. If the value of ptr is not (UNICODE **)NULL, a pointer to the character terminating the scan is returned in the location pointed to by ptr. If no integer can be formed, that location is set to str, and zero is returned.

RETURN VALUE

Returns the decimal value represented by *s*. On overflow returns ULONG_MAX.

SEE ALSO

LongToUc(), **UcToLong**(), **ULongToUc**(), **ULongToUcOctal**(), **ULongToUcHex**()

NAME

AssociateUcChar - Associate a Unicode character to some data
CreateAssocationTable - Create a Unicode-to-??? association table
FreeAssociationTable - Release all memory used by an association table
GetUcCharAssociation - Retrieve a Unicode character association

SYNOPSIS

```
#include <strings/strings.h>

int AssociateUcChar(ASSOCTABLE *at, UNICODE uc, const void *data)
int CreateAssociationTable(ASSOCTABLE *at)
void FreeAssociationTable(ASSOCTABLE *at)
void *GetUcCharAssociation(ASSOCTABLE *at, UNICODE uc)
```

DESCRIPTION

An “association table” is conceptually a table of (void *) to some user-defined data, indexed by Unicode character values. An association table can be used to avoid a very sparse, statically-defined table which in the worst case could contain 64k pointers, most of them unused.

Association Tables are similar to Name Tables except that Name Tables store binary objects and are indexed by binary objects. Also, Association Tables are faster when saving more than a few items and they should be used for Unicode translations.

AssociateUcChar() modifies the association table ‘*at*’, such that Unicode value ‘*uc*’ is associated with the pointer ‘*data*’.

CreateAssociationTable() initializes the table pointed to by ‘*at*’ as a table of empty associations; i.e. each Unicode value is associated with (void *)0.

FreeAssociationTable() releases all memory used by an association table. The ‘*at*’ pointer is invalid after this function is invoked.

GetUcCharAssociation() returns the pointer associated with Unicode character '*uc*' from association table '*at*'.

See file strings/mnemonic.h for an example using Association Tables.

RETURN VALUE

GetUcCharAssociation()

void *	User-defined data pointer
(void *)0	No associated data pointer exists

AssociateUcChar() and **CreateAssociationTable**()

0	Operation was successful
-1	Memory allocation error occurred

SEE ALSO

Name Tables

NAME

InitCharSets - Initialize built in character sets
FreeCharSets - Release memory used by character sets
ConvertFromUnicode - Translate Unicode to single-byte character string
ConvertToUnicode - Translate single-byte character string to Unicode
ConvertWithCountsFromUnicode - Translate Unicode to byte character string
ConvertWithCountsToUnicode - Translate byte character string to Unicode
GetCharSetID - Return a character set id for later use
GetUcTranslationRule- Search table for character translation rule
RegisterCharSet - Register a new character set
ModifyCharSet - Modify an existing character set
RegisterCharSetName - Register a new character set name

SYNOPSIS

```
#include <strings/strings.h>

void InitCharSets(void)
void FreeCharSets(void)
ssize_t ConvertFromUnicode(BYTE *dptr, size_t dlen, const UNICODE *sptr,
      size_t slen, CSID csid, int flags)
ssize_t ConvertToUnicode(UNICODE *dptr, size_t dlen, const BYTE *sptr,
      size_t slen, CSID csid, int flags)
int ConvertWithCountsFromUnicode(size_t *dcnt, BYTE *dptr, size_t dlen,
      size_t *scnt, const UNICODE *sptr, size_t slen, CSID csid, int
      flags)
int ConvertWithCountsToUnicode(size_t *dcnt, UNICODE *dptr, size_t dlen,
      size_t *scnt, const BYTE *sptr, size_t slen, CSID csid, int flags)
CSID GetCharSetID(const char *csname)
const CSXLATERULE *GetUcTranslationRule(const CSXLATETBL *tbl, UNICODE
      uch)
CSID RegisterCharSet(CHARSET *newcs)
int ModifyCharSet(CSID csid, CHARSET *newcs)
int RegisterCharSetName(const char *name, CSID csid)
```

DESCRIPTION

InitCharSets() and **FreeCharSets**()

Initializes or releases the built-in character set conversion translation tables. These functions are not normally invoked by users since they are normally called by **InitStrings**() and **ShutdownStrings**().

ConvertFromUnicode() and **ConvertToUnicode**()

Translate characters from/to Unicode to/from a single-byte or multi-byte character set. '*csid*' identifies the character set. Note that *dlen* and *slen* represent string lengths in characters, which is not the same as byte length for the Unicode string. The '*flags*' argument may be set to CVT_NORMAL to stop conversion on a null character, CVT_NULLS, to convert past nulls, or CVT_TEST to translate without any actual output.

ConvertWithCountsFromUnicode() and **ConvertWithCountsToUnicode**()

Translate characters from/to Unicode to/from a single-byte or multi-byte character set. '*csid*' identifies the character set. Note that *dlen* and *slen* represent string lengths in characters, which is not the same as byte length for the Unicode string. The number of source characters translated

is returned in *scnt* and the number of destination characters written is returned in *dcnt*. The '*flags*' argument may be set to CVT_NORMAL to stop conversion on a null character, CVT_NULLS, to convert past nulls, or CVT_TEST to translate without any actual output.

GetCharSetID()

Scan the character set name table and return the character set ID number for the specified character set name. The comparison of character set names is case-insensitive.

GetUcTranslationRule()

Returns a pointer to the character translation rule in '*tbl*' which is associated with UNICODE character '*uch*'. If there is no associated rule, a null pointer is returned.

Restrictions: The rules in '*tbl*->Rules' must be sorted by Start character in ascending order with no overlapping ranges. That is:

rule[n].Start <= rule[n].End < rule[n+1].Start

'*tbl*->DimOfRules' must be the number of elements in '*tbl*->Rules'.

RegisterCharSet()

Saves a new user-defined character set for later use by **ConvertTo**/**FromUnicode**(). *newcs* must be already set up in your routine. It is added to TABLE **CharSets**.

Character set names are registered separately using function **RegisterCharSetName**().

The internal character set ID is returned by this function and may immediately be used in calls to **ConvertTo**/**FromUnicode**(), though you should register at least one name for the set as well.

ModifyCharSet()

Replaces the current definition of an existing character set with a new definition set for later use by ConvertTo/FromUnicode(). newcs must be already set up in your routine.

RegisterCharSetName()

Links a character set name with a character set. The character set must have been already registered using **RegisterCharSet**(). After a character set name is registered, that name may be used in subsequent calls to **GetCharSetID**().

The pre-defined (stock) character sets are:

Name	Char Set Id
"ASCII"	CSID_ASCII
"US-ASCII"	CSID_ASCII
"ISO 646"	CSID_ASCII
"ANSI"	CSID_ANSI1
"ANSI Latin 1"	CSID_ANSI1
"ISO 8859-1"	CSID_ANSI1
"IRIS"	CSID_IRIS
"IRIS-ASCII"	CSID_IRIS
"uniBasic"	CSID_UNIBASIC
"uniBasic-ASCII"	CSID_UNIBASIC
"IBM Code Page 437"	CSID_IBM437
"IBM Code Page 850"	CSID_IBM850
"Windows"	CSID_MS1252
"Windows Code Page 1252"	CSID_MS1252
"EBCDIC"	CSID_EBCDIC037
"EBCDIC 037"	CSID_EBCDIC037
"UTF-8"	CSID_UTF8

RETURN VALUE

ConvertFromUnicode() and **ConvertToUnicode**()

>= 0	Copy successful, x characters written to destination string.
-1	Error, unable to translate one or more characters.

ConvertWithCountsFromUnicode() and **ConvertWithCountsToUnicode**()

>= 0	Copy successful, *scnt* contains the number of source characters translated and dcnt contains the number of destination characters written.
-1	Error, unable to translate one or more characters. *scnt* contains the number of source characters successfully translated and *dcnt* contains the number of destination characters written.

GetCharSetID() and **RegisterCharSet**()

x	Character set id number
NO_CSID	Character set is not registered or memory allocation error

ModifyCharSet()

>=0	Character set successfully modified
-1	Unable to modify character set *csid*: character set is not registered

GetUcTranslationRule()

(CSXLATERULE *)0	No matching rule exists
(CSXLATERULE *)x	Pointer to the translation rule for the character

RegisterCharSetName()

>=0	Name successfully added
-1	Unable to save character set name: duplicate name or other error

SEE ALSO

Character Sets (Defining New Character Sets), **Comparing ASCII Strings**, **Comparing Unicode and ASCII StringsbComparing Unicode Strings**, **Copying ASCII Characters to Unicode, Copying Unicode Characters**, **Length of Unicode Strings**

NAME

Mnemonic Functions

SYNOPSIS

```
#include <strings/strings.h>
#include <strings/mnemonic.h>

int IsMnemonic(UNICODE c)

int IsMnemonicParm(UNICODE c)
BYTE GetMnemonicParmData(UNICODE c)
size_t LongToMnemonicParm(UNICODE *s, size_t n, long v)
UNICODE MakeMnemonicParm(BYTE c)
long MnemonicParmToLong(const UNICODE *s, const UNICODE **ptr, size_t n)

int IsContinuedParm(UNICODE c)
BYTE GetContinuedParmData(UNICODE c)
UNICODE MakeContinuedParm(BYTE c)

UNICODE SearchMnemonic(const UNICODE *src, size_t len)

void InitMnemonicAssociations(void)
const char *GetAssociatedMnemonic(UNICODE uc)
void FreeMnemonicAssociations(void)
```

DESCRIPTION

These are general functions for use with DCI Mnemonics:

IsMnemonic()
: Tests an Unicode character and returns true if it contains a DCI Mnemonic. Refer to strings/mnemonic.h for a list of DCI Mnemonics.

SearchMnemonic()
: Convert a Unicode text mnemonic (such as 'CS' or 'BP') into its single Unicode character representation. See **GetAssociatedMnemonic**()

These functions allow a program to convert from UNICODE characters to ASCII text:

InitMnemonicAssociations()
: The UNICODE-character-to-ASCII mnemonic descriptions are stored in an association table, which must be initialized before it can be used. Release the table using **FreeMnemonicAssociations**().

GetAssociatedMnemonic()
: This function complements **SearchMnemonic**() described above and takes an UNICODE character as an argument and translates it into a text mnemonic. (Any UNICODE character may be converted. If no specific mnemonic is defined, one will be created.)

FreeMnemonicAssociations()
: Releases the memory used by the UNICODE-character-to-ASCII association table. You should invoke this function before exiting a program (if **InitMnemonicAssociations**() was invoked).

These functions are meant to be used in conjunction with parameterized Mnemonics. They encode, decode, or test Unicode characters to see if they contain mnemonic information:

IsMnemonicParm() and **IsContinuedParm**()

Check if an Unicode character contains (or continues) mnemonic parameter data.

GetMnemonicParmData() and **GetContinuedParmData**()

Return the actual (or continued) mnemonic information. This is actually the low byte of the Unicode character.

MakeMnemonicParm() and **MakeContinuedParm**()

Convert byte values into the appropriate Unicode character. The data to convert should be >= 0 and <= 255 in order to fit into the low byte of the Unicode character, however this is not verified. (If you invoke these functions with a number outside that range, the resulting Unicode character will not be recognized properly.)

LongToMnemonicParm()

Converts the long value '*v*' to a mnemonic parameter string beginning at '*s*'. '*n*' gives the size of the destination string converted value, **LongToMnemonicParm**() returns (size_t)0, with Warning: The conversion process assumes twos-complement arithmetic.

MnemonicParmToLong()

Returns as a long the (signed 32-bit) integer value represented by the UNICODE parameter mnemonic string **s*. A pointer to the character terminating the scan is returned in the location pointed to by ptr. If no integer can be formed, that location is set to *s*, and zero is returned. Warning: the conversion process assumes twos-complement arithmetic.

Generally, the Mnemonic information is converted into a signed 32-bit value for further processing. Here is a code sample that converts a string of Mnemonic Unicode characters into usable form. The information is first decoded into an unsigned value in order to prevent an overflow condition:

```
INT32 mnemonicvalue; UINT32  parm; UNICODE uch, *s; int len;

len = ...;  /* length of decoded parameter */
parm = GetMnemonicParmData(uch);
for (; len && IsContinuedParm(*s); --len, ++s)
      parm = (parm << 8) | GetContinuedParmData(*s);
/*  protect ourselves on machines where UINT32 is actually > 32 bits. */
if (parm & ((UINT32)1 << 31))
      parm  |= ((INT32)-1 << 31);
mnemonicvalue = (INT32)parm;
```

RETURN VALUE

InitMnemonicAssociations(), **FreeMnemonicAssociations**()

void

GetAssociatedMnemonic()

(BYTE *)0	Error occurred
x	BYTE pointer to a static area containing the mnemonic

SearchMnemonic()

x	Unicode Mnemonic character
NO_UCCHAR	Unable to convert Unicode string to DCI Mnemonic

IsMnemonic(), **IsMnemonicParm**(), **IsContinuedParm**()

0	= False
!0	= True

GetMnemonicParmData(), **GetContinuedParmData**()

The parameter portion (ie: lower byte) of the Unicode character.

MakeMnemonicParm(), **MakeContinuedParm**()

The DCI Mnemonic parameter value as an Unicode character

LongToMnemonicParm()

(size_t)0	String '*s*' too small to convert '*v*' or '*v*' is outside the range of signed 32-bit numbers.
x	Number of Unicode characters output

MnemonicParmToLong()

X	Parameter value

Chapter 7 – Error Functions

The error functions are used to set and access the current dL4 error code and error message. The error library as defined by the error/errors.h include file does not contain any public structure definitions. All structure types defined by error/errors.h should be manipulated only via dL4 runtime library functions and never directly accessed or modified. Any direct access or manipulation may cause compatibility problems in future releases of dL4.

NAME

AddError - Append the BASIC error number
AddErrorLong - Append the BASIC error number and include a number
AddErrorASCII - Append the BASIC error number and include ASCII text
AddErrorUnicode - Append the BASIC error number and include Unicode text
AddErrorBinary - Append the BASIC error number and include binary data

SYNOPSIS

```
int AddError(ERRORNUM e)
int AddErrorLong(ERRORNUM e, long l)
int AddErrorASCII(ERRORNUM e, const char *s)
int AddErrorUnicode(ERRORNUM e, const UNICODE *s)
int AddErrorBinary(ERRORNUM e, const BYTE *p, size_t n)
```

DESCRIPTION

Append the global BASIC error number to the existing error information. Error numbers are defined in <error/errors.h> and are prefixed with BE_. **AddError**()'s return value is always -1, and can be used as the argument of a return() control structure. For example:

```
return(AddErrorASCII(BE_BAD, "Something bad happened!"));
```

RETURN VALUE

-1 Always returns -1; this does not mean **AddError**() failed.

SEE ALSO

ErrorMessage()

NAME

ErrorMessage - Return error message

SYNOPSIS

```
#include <error/errors.h>

const UNICODE *ErrorMessage(ERRORNUM errorcode)
```

DESCRIPTION

Return a UNICODE pointer to a statically allocated area containing the null terminated error message for the specified error.

RETURN VALUE

(UNICODE *) 0	Illegal error code specified
x	UNICODE pointer to static area containing message

NAME

FatalError - Flag a fatal error
FatalErrorLong - Flag a fatal error and include a number
FatalErrorASCII - Flag a fatal error and include ASCII text
FatalErrorUnicode - Flag a fatal error and include Unicode text
FatalErrorBinary - Flag a fatal error and include binary data

SYNOPSIS

```
#include <error/errors.h>

int FatalError(ERRORNUM e)
int FatalErrorLong(ERRORNUM e, long l)
int FatalErrorASCII(ERRORNUM e, const char *s)
int FatalErrorUnicode(ERRORNUM e, const UNICODE *i)
int FatalErrorBinary(ERRORNUM e, const BYTE *p, size_t n)
```

DESCRIPTION

Set the global BASIC error number to the specified error, to indicate an error has occurred. Error numbers are defined in <error/errors.h> and are prefixed with BE_. **FatalError** should be used to flag a totally unexpected event, such as falling into the default case of a switch block or getting an unexpected value from a system call. These functions replace Trap0InRun(), which is no longer supported. In addition, **FatalError***() also automatically saves the source filename and line number for later reference.

FatalError()'s return value is always -1, and can be used as the argument of a return() control structure. For
example:

```
return(FatalErrorLong(BE_SYSERRNO, (long)errno));
```

RETURN VALUE

-1 Always returns -1; this does not mean **FatalError**() failed.

SEE ALSO

ErrorMessage(), **SourceError**()

NAME

GetError - Get the last BASIC error number
GetErrorText - Get the current BASIC error text

SYNOPSIS

```
#include <error/errors.h>

ERRORNUM GetError()
MEMHND GetErrorText()
```

DESCRIPTION

Get the current BASIC error number or text set by **SetError**(), etc. The text referenced by **GetErrorText**() may include information on more than one error. Error numbers are defined in <error/errors.h> and are prefixed with BE_.

RETURN VALUE

GetError()

0	No BASIC error has occurred
!=0	The last BASIC error number

GetErrorText()

x	Memory handle for the actual error message text

SEE ALSO

ErrorMessage()

NAME

InitError - Initialize Error package

SYNOPSIS

```
#include <error/errors.h>

void InitError(void)
```

DESCRIPTION

Initialize global variables for the Error package and perform assertions of package assumptions.

SEE ALSO

ShutdownError()

NAME

OverrideError - Replace the BASIC error number
OverrideErrorLong - Replace the BASIC error number and include a number
OverrideErrorASCII - Replace the BASIC error number and include ASCII text
OverrideErrorUnicode - Replace the BASIC error number and include Unicode text
OverrideErrorBinary - Replace the BASIC error number and include binary data

SYNOPSIS

```
#include <error/errors.h>

int OverrideError(ERRORNUM e)
int OverrideErrorLong(ERRORNUM e, long l)
int OverrideErrorASCII(ERRORNUM e, const char *s)
int OverrideErrorUnicode(ERRORNUM e, const UNICODE *s)
int OverrideErrorBinary(ERRORNUM e, const BYTE *p, size_t n)
```

DESCRIPTION

OverrideErrorXXXX() is used after an error has occurred to change the error number to 'error'. Normally, it is used to change a generic error code to a more specific error code. **OverrideError**() will not change the error number if the existing error code is an exception code such as BE_INTERRUPT. Error numbers are defined in <error/errors.h> and are prefixed with BE_.

OverrideErrorXXXX()'s return value is always -1, and can be used as the argument of a return() control structure. For example:

```
return(OverrideError(BE_BADFILETYPE));
```

RESTRICTIONS

OverrideError() can only be called after an error has occurred and **SetError**() has been used to set the error code.

RETURN VALUE

-1 Always returns -1; this does not mean **OverrideError**() failed.

SEE ALSO

ErrorMessage()

NAME

PutErrorText - Display the current error message

SYNOPSIS

```
#include <error/errors.h>

void PutErrorText(void)
```

DESCRIPTION

Displays the current error message. his may actuallyinclude more than one error of AddError*() has been used.

SEE ALSO

ErrorMessage()

NAME

SetError - Set the BASIC error number
SetErrorLong - Set the BASIC error number and include a number
SetErrorASCII - Set the BASIC error number and include ASCII text
SetErrorUnicode - Set the BASIC error number and include Unicode text
SetErrorBinary - Set the BASIC error number and include binary data

SYNOPSIS

```
#include <error/errors.h>

int SetError(ERRORNUM e)
int SetErrorLong(ERRORNUM e, long l)
int SetErrorASCII(ERRORNUM e, const char *s)
int SetErrorUnicode(ERRORNUM e, const UNICODE *s)
int SetErrorBinary(ERRORNUM e, const BYTE *p, size_t n)
```

DESCRIPTION

Set the global BASIC error number to the specified error, to indicate an error has occurred. Error numbers are defined in <error/errors.h> and are prefixed with BE_. **SetError**()'s return value is always -1, and can be used as the argument of a return() control structure. For example:

```
return(SetErrorLong(BE_SYNTAX, lineno));
```

RETURN VALUE

-1 Always returns -1; this does not mean **SetError**() failed.

SEE ALSO

ErrorMessage()

NAME

ShutdownError - Shutdown Error package

SYNOPSIS

```
#include <error/errors.h>

void ShutdownError(void)
```

DESCRIPTION

Release global variables for the Error package.

SEE ALSO

InitError()

NAME

SourceError - Flag a source code error
SourceErrorLong - Flag a source code error and include a number
SourceErrorASCII - Flag a source code error and include ASCII text
SourceErrorUnicode - Flag a source code error and include Unicode text
SourceErrorBinary - Flag a source code error and include binary data

SYNOPSIS

```
#include <error/errors.h>

int SourceError(ERRORNUM e, const char *file, int line)
int SourceErrorLong(ERRORNUM e, const char *file, int line, long l)
int SourceErrorASCII(ERRORNUM e, const char *file, int line, const char
      *s)
int SourceErrorUnicode(ERRORNUM e, const char *file, int line, const
      UNICODE *i)
int SourceErrorBinary(ERRORNUM e, const char *file, int line, const BYTE
      *p, size_t i)
```

DESCRIPTION

Set the global BASIC error number to the specified error, to indicate an error has occurred. Error numbers are defined in <error/errors.h> and are prefixed with BE_. SourceError*() is not normally used directly. ed directly. calls these functions and should be used instead if possible.

SourceError()'s return value is always -1, and can be used as the argument of a return() control structure. For example:

```
return(SourceErrorLong(BE_SYSERRNO,"filename",123,(long)errno));
```

RETURN VALUE

-1 Always returns -1; this does not mean **SourceError**() failed.

SEE ALSO

ErrorMessage(), **FatalError**()

NAME

Trap0InRUN() - Flag a totally unexpected condition

SYNOPSIS

```
#include <error/errors.h>

int Trap0InRUN()
```

DESCRIPTION

Trap0InRun() is no longer supported.

This function is replaced by the FatalErrorXXXX() functions. You should use them instead of this function.

Trap0InRUN()'s return value is always -1, and can be used as the argument of a return() control structure. For example:

```
return(Trap0InRUN());
```

RETURN VALUE

-1 Always returns -1; this does not mean Trap0InRUN() failed.

SEE ALSO

ErrorMessage()

NAME

SaveErrorContext - Return memory handle containing current error number and text.
RestoreErrorContext - Restore error number and text from saved error context.

SYNOPSIS

```
#include <error/errors.h>

MEMHND SaveErrorContext(void)
int RestoreErrorContext(MEMHND errorhnd)
```

DESCRIPTION

SaveErrorContext() allocates and initializes a memory handle containing the current error context: the error number and error text. This context can be restored (and freed) by **RestoreErrorContext**(). If not restored, the error context must be freed by **FreeMem**().

SaveErrorContext() and **RestoreErrorContext**() are used to preserve the error number and error text during error cleanup operations which might otherwise change them.

WARNINGS

RestoreErrorContext() will NOT override a BE_INTERRUPT error; it will just release the memory handle.

RETURN VALUE

SaveErrorContext()

(MEMHND)0	Unable to allocate memory to save context
(MEMHND)h	Memory handle of saved context

RestoreErrorContext()

-1	Always returns -1; this does not mean **SetError**() failed.

SEE ALSO

ErrorMessage(), **GetError**()

Chapter 8 – Item Functions

The item functions are used to perform various operations on **ITEM** values. The items library as defined by the items/items.h include file does not contain any public structure definitions. All structure types defined by items/items.h should be manipulated only via dL4 runtime library functions and never directly accessed or modified. Any direct access or manipulation may cause compatibility problems in future releases of dL4. The one exception to this rule is that the "Data" member (variable.Data) of an **ITEM** value can be directly accessed to get a void pointer to the value of the **ITEM**.

An **ITEM** value is a general purpose descriptor which is used by user defined CALLS, functions, or drivers to access variables and values passed by a dL4 program. All **ITEM** values have one of the following types:

ITYP_NULL	No value
ITYP_NUMBER	Numeric value of INFMT_xxxx format
ITYP_CHARACTER	Unicode string of dimension DimOfCItem() and length LengthCItem()
ITYP_DATE	Date value of IDFMT_xxxx format
ITYP_UNKNOWN	Binary value of dimension DimOfUItem()
ITYP_STRUCTURE	Structure variable
ITYP_ARRAY	Array variable of dimension DimOfAItem()

The type of an ITEM can be determined by using the **TypeOfItem**(), **IsItemNull**(), **IsNItem**(), **IsCItem**(), **IsDItem**(), **IsUItem**(), **IsSItem**() or **IsAItem**() functions. Character **ITEM**s correspond to dL4 string variables ("S$") and unknown **ITEM**s correspond to dL4 binary variables ("B?").

Structure **ITEM**s describe an entire dL4 structure value. In most cases, a structure value should not be accessed directly and the **SetupMemberItem**() function will be needed to create a local **ITEM** that points to an individual structure member. Note that a member of a structure value may itself be a structure value. The **NumMembersSItem**() function can be used to determine how many members exist in a structure value.

Array **ITEM**s describe an entire dL4 array value. The following functions are used to access array **ITEM**s:
IndexElementItem(), **NumElementsAItem**(), **SetupElementItem**(), and **SetupBaseElementItem**(). A two dimensional matrix in dL4 is an array of arrays. If **SetupElementItem**() is used on a two dimensional matrix, the resulting **ITEM** will itself be an array **ITEM** describing the first row of the matrix. This concept extends to three dimensional and higher dimensional arrays.

NAME

CatString - Concatenate a character string item

SYNOPSIS

#include <items/items.h>

UNICODE ***CatString**(ITEM **pdi*, const ITEM **psi*, UNICODE **dp*)

DESCRIPTION

CatString() copies (concatenates) all characters from item '*psi*' to the selected position in item '*pdi*'. '*dp*' points to the desired starting position in the destination string '*pdi*'.

The rules for copying character strings are rather arcane, historically rooted in IRIS Business BASIC.

As each character is copied, any null occurring in '*pdi*' is successively "pushed-down", i.e. preserved as terminating the destination string. Such pushing can, at worst, place the null at the first position outside **DimOfCItem**(*pdi*). Any such outside-of-dimension position is normally not accessible, but at least one always physically exists (specifically for a null).

Also, if a null occurs in '*pdi*', '*pdi*->IsSubString' is forcibly cleared. Among other things, this serves to tell **CatWrapup**() to modify its wrapup rules accordingly.

RETURN VALUE

UNICODE *	Ending destination pointer (i.e. first not written), exclusive of any "pushed-down" null.

SEE ALSO

CatWrapup(), **MoveString**()

NAME

CatWrapup - Perform termination of string concatenation

SYNOPSIS

```
#include <items/items.h>

UNICODE *CatWrapup(ITEM *pdi, UNICODE *dp, int rawrules)
```

DESCRIPTION

CatWrapup() performs any necessary wrapup work related to string concatenation, presumably done by **CatString**(), and primarily consisting of simple placement of a null. '*dp*' points to the final (last unwritten) position in the destination string '*pdi*'; i.e. where a null would be placed.

The rules for copying character strings are rather arcane, historically rooted in IRIS Business BASIC and BITS BASIC.

If '*pdi*->IsSubString' is false, a null is output to the specified destination position; else the following extra processing is done:

> If '*dp*' is within the boundaries of '*pdi*' (indicating the destination string is not yet full), all characters past the end of '*pdi*' are moved down (left) to be flush with the data just copied (i.e. starting at '*dp*'). The '*rawrules*' flag, if true, inhibits this moving.

RETURN VALUE

UNICODE * Ending destination pointer (i.e. first not written), one beyond the terminating null.

SEE ALSO

CatString(), **MoveString**()

NAME

CopyAsciiValueCItem - Copy ASCII string to character ITEM

SYNOPSIS

#include items.h

int **CopyAsciiValueCItem**(ITEM ***dest*, const char ***ascii*)

DESCRIPTION

Convert and copy ASCII bytes from '*ascii'* to the character ITEM '*dest*'. is non-zero and '*dest*' is too small to contain all of '*ascii*', '*dest*' will be reallocated to the size of '*ascii*'.

RETURN VALUE

0	Successful
-1	Error, use **GetError**() for error code

SEE ALSO

CopyValueCItem(), **CopyValueUItem**(), **MakeCItemFromAscii**(), **MakeStringAscii**()

NAME

CopyItem - Copy an ITEM

SYNOPSIS

```
#include <items/items.h>

int CopyItem(ITEM *dest, const ITEM *src)
```

DESCRIPTION

CopyItem **src* to ITEM **dest*. The difference between this function and **DupItem**() is that is that merely copies the ITEM information while **CopyItem**() creates a completely new and separate ITEM with a copy of the original item's data. After returning from **CopyItem**(), both items will refer to different memory handles, which are locked.

When done with the ITEM **dest*, release it using **ReleaseItem**().

RETURN VALUE

0	Success
-1	Error occurred. Use **GetError**() for actual error.

SEE ALSO

DupItem(), **Releasing Items**

NAME

CopyTextLine - Translate carriage returns in text

SYNOPSIS

```
#include <items/items.h>

ssize_t CopyTextLine(ITEM *dest, ITEM *src)
```

DESCRIPTION

CopyTextLine copies a line of Unicode text from the item referenced by '*src*', into the item referenced by '*dest*', with carriage returns, if repeated, expanded to multiple characters. The resulting string is better suited to conversion to an 8-bit character set for output to an external character stream or text file. Processed characters are trimmed from '*src*' (see **LTrimCItem**()).

RETURN VALUE

<0	Error, use **GetError**() for error code
n	Number of characters stored in '*dest*'

SEE ALSO

TranslateTextLine()

NAME

CopyValueCItem - Copy Unicode string to character ITEM
CopyValueUItem - Copy BYTE array to binary ITEM

SYNOPSIS

```
#include items.h

int CopyValueCItem(ITEM *dest, const void *src, size_t len)
int CopyValueUItem(ITEM *dest, const void *src, size_t len)
```

DESCRIPTION

Copy '*src*' into the ITEM '*dest*'. If '*dest*' is smaller than than '*len*' units of '*src*' and *dest*->IsTemporary is set, reallocate '*dest*' to the required size.

RETURN VALUE

0	Successful
-1	Error, use **GetError**() for error code

SEE ALSO

CopyAsciiValueCItem(), **MakeCItemFromAscii**(), **MakeStringAscii**(), **SetupCItem**(), **SetupUItem**()

NAME

DimOfAItem - Return dimension of an array ITEM
DimOfCItem - Return dimension of a character ITEM
DimOfUItem - Return dimension of an unknown ITEM

SYNOPSIS

```
size_t DimOfAItem(const ITEM *pi)
size_t DimOfCItem(const ITEM *pi)
size_t DimOfUItem(const ITEM *pi)
```

DESCRIPTION

Return the size in elements (array elements, bytes, or UNICODE characters) of ITEM **pi*.

RETURN VALUE

size_t	Dimension of item

SEE ALSO

LengthCItem()

NAME

DupItem - Duplicate an ITEM

SYNOPSIS

```
#include <items/items.h>

void DupItem(ITEM *dest, const ITEM *src)
```

DESCRIPTION

Duplicate ITEM **src* to ITEM **dest*. The difference between this function and **CopyItem**() is that **CopyItem**() creates a completely new ITEM with a copy of the item's data while this function merely copies the ITEM information. After returning from **DupItem**(), both items will refer to the same memory handle, which is locked.

For example, to create a sub-string of a Unicode character ITEM without modifying the original ITEM, duplicate the ITEM using **DupItem**(), then convert it to a sub-string using **SubstringCItem**(). Any changes to the sub-string **dest* will reflect in the **src*, because they both refer to the same data area.

When done with the ITEM **dest*, release it using **ReleaseItem**().

SEE ALSO

CopyItem(), **Releasing Items**, **Substrings**

NAME

FindNextWS - Find the next white-space in a Unicode string

SYNOPSIS

```
#include <items/items.h>

size_t FindNextWS(const ITEM *pi)
```

DESCRIPTION

FindNextWS() scans a Unicode string and returns the position of the first white-space character or EOS if none is found. EOS is defined as either the dimension of the item (**DimOfCItem**()) or when a Unicode null is reached, whichever comes first.

RETURN VALUE

size_t The position of the first white-space character or EOS (origin 0).

SEE ALSO

SkipWSCItem(), **ParseWSCItem**()

NAME

FixNItem - Convert a numeric ITEM to a long

SYNOPSIS

```
#include <items/items.h>

long FixNItem(const ITEM *item)
```

DESCRIPTION

Return the value of the numeric ITEM **item* as a long.

RETURN VALUE

Long Integer value of ITEM **item*

SEE ALSO

FloatNItem(), **LoadNItem**(), **StoreNItem**()

NAME

FloatNItem - Convert a long to a numeric ITEM

SYNOPSIS

```
#include <items/items.h>

int FloatNItem(ITEM *dest, long src)
```

DESCRIPTION

Convert the specified long *src* to the numeric type specified by ITEM **dest* and store it in **dest*. ITEM **dest* must already have data allocated to it.

RETURN VALUE

0	Store succeeded
-1	Store failed due to illegal format or overflow, use **GetError**() for error code

SEE ALSO

FixNItem(), **StoreDItem**(), **StoreNItem**()

NAME

GetCharSetIDCItem - Lookup a character set id

SYNOPSIS

```
#include <items/items.h>

CSID GetCharSetIDCItem(ITEM *charsetname)
```

DESCRIPTION

GetCharSetIDCItem() looks up the character set named by the character ITEM '*charsetname*' and returns its character set id, if it exists. The search of character set names is case-insensitive.

RETURN VALUE

Csid	Success, 'csid' is the character set id number of the character set.
NO_CSID	Error, character set not found, use **GetError**() for error code.

SEE ALSO

GetCharSetID()

NAME

InitItems - Initialize Items package

SYNOPSIS

```
#include <items/items.h>

void InitItems(void)
```

DESCRIPTION

Initialize global variables for the Items package and perform assertions of package assumptions.

SEE ALSO

ShutdownItems()

NAME

IsItemNull - Test for null ITEM

SYNOPSIS

```
#include <items/items.h>

int IsItemNull(ITEM *item)
```

DESCRIPTION

Returns true if the type of **item* is ITYP_NULL. Note that this is different than if an item simply has no data.

RETURN VALUE

0	Item is not null
1	Item is null

SEE ALSO

MakeCItem(), **MakeNItem**(), **NullItem**(), **Releasing Items**, **CopyItem**(), **DupItem**(), **Item Types**

NAME

ChangeAItemFormat - Convert numeric ITEM array to new format
ChangeNItemFormat - Convert numeric ITEM to new format
SetAItemFormat - Set numeric array ITEM format
SetNItemFormat - Set numeric ITEM format

SYNOPSIS

```
#include <items/items.h>
#include <items/math.h>

int ChangeAItemFormat(ITEM *item, int fmt)
int ChangeNItemFormat(ITEM *item, int fmt)
int SetAItemFormat(ITEM *item, int fmt)
int SetNItemFormat(ITEM *item, int fmt)
```

DESCRIPTION

ChangeNItemFormat() and **ChangeAItemFormat**() change the numeric format of (the array) **item* to *fmt* and convert the numbers referred to by it.

SetNItemFormat() and **SetAItemFormat**() set the numeric format of (the array) **item* to *fmt*. No data is converted and **item* should be an ITYP_NULL or ITYP_ARRAY item on entry, depending on the function.

The **item*(s) must be an existing ITEM with allocated numeric data not in use. The following conditions must be true for any of these functions to succeed:

1. *Item* must not be an array element;
2. *Item* must have memory allocated to it;
3. The item's data must not be in use anywhere else;
4. The item's data size must match the size of *fmt*;
5. The old and new data format must align the same way;
6. The old and new format must belong to the same number class.

RETURN VALUE

0	Format was set successfully
-1	Error, use GetError() for error code: BE_BADPREC - Format sizes differ or data not aligned BE_VARINUSE - Data is either in use, locked, or not allocated BE_OVERFLOW - Overflow while converting to new format

SEE ALSO

"**Number Format Types**" in the "dL4 Math Package", **ConvertNumbers**(), **NFormatSize**(), **FormatAlign**(), **GetNumberClass**()

NAME

LengthCItem - Return the length of a Unicode string ITEM

SYNOPSIS

```
#include <items/items.h>

size_t LengthCItem(const ITEM *item)
```

DESCRIPTION

Return the length of the specified Unicode character string ITEM **item*. The length is the lesser of either the dimensioned size or the number of characters preceding the terminating null character.

RETURN VALUE

size_t Length in Unicode characters

SEE ALSO

Comparing Items, **Searching Items**, **SkipWSCItem**(), **Substrings**

NAME

LoadDItem - Load a date ITEM into an ACCUM
LoadNItem - Load a numeric ITEM into an ACCUM

SYNOPSIS

```
#include items.h

void LoadDItem(ACCUM *dest, const ITEM *src)
void LoadNItem(ACCUM *dest, const ITEM *src)
```

DESCRIPTION

Load the (data) value of the date or numeric ITEM **src* into the ACCUM **dest*.

SEE ALSO

StoreDItem(), **StoreNItem**(), **FixNItem**(), **LoadDate**(), **LoadAccum**()

NAME

MakeCItem - Make a new Unicode character ITEM
MakeDItem - Make a new date ITEM
MakeNItem - Make a new numeric ITEM
MakeUItem - Make a new binary string ITEM

SYNOPSIS

```
#include <items/items.h>

int MakeCItem(ITEM *item, const void *src, size_t len)
int MakeDItem(ITEM *item, const void *src, int fmt)
int MakeNItem(ITEM *item, const void *src, int fmt)
int MakeUItem(ITEM *item, const void *src, size_t len)
```

DESCRIPTION

MakeCItem() and **MakeUItem**()

Make ITEM **item* from the Unicode or binary character string **src* of length *len*. A new memory area of *len* + 1 (for the terminating null) Unicode or binary characters is allocated, locked, and associated with **item*. If *src* != null, the string **src* is copied to the new memory area.

MakeNItem() and **MakeNItem**()

Make ITEM *i from the number **src* of type *fmt*. Valid identifiers for *fmt* are described in sections "**Date Format Codes**" of the "dL4 Date Package" and "**Number Format Types**" of the "dL4 Math Package". A new memory area is allocated, locked, and associated with **item*. If *src* != null, the value at **src* is copied to the new memory area.

When done with the ITEM **item*, release it using **ReleaseItem**().

The difference between these and the **SetupXXXX**() functions is that the **SetupXXXX**() functions create ITEMs that point to existing data while these functions actually copy the data to a new memory area.

RETURN VALUE

0	Successful operation
-1	Error, use **GetError**() for error code: BE_MEMOFLW - Failed to allocate memory

SEE ALSO

Setting Up New Items, **Releasing Items**, "**Date Format Codes**" of the "dL4 Date Package", "**Number Format Types**" in the "dL4 Math Package"

NAME

MakeCItemFromAscii - Make a character ITEM from an ASCII string

SYNOPSIS

```
#include items.h

int MakeCItemFromAscii(ITEM *item, const char *ascii)
```

DESCRIPTION

Allocate and return a character ITEM containing the ASCII string. When done with the ITEM **item*, release it using **ReleaseItem()**.

RETURN VALUE

0	Successful
-1	Error, use **GetError**() for error code

SEE ALSO

MakeStringAscii(), **CopyAsciiValueCItem**(), **CopyValueCItem**(), **CopyValueUItem**()

NAME

MakeStringAscii - Convert character ITEM to binary ASCII ITEM

SYNOPSIS

```
#include <items/items.h>

int MakeStringAscii(ITEM *item)
```

DESCRIPTION

MakeStringAscii() replaces the zero-terminated character ITEM **item* with an ITYP_UNKNOWN ITEM containing the ASCII equivalents of each character in the original ITEM. The function will fail and return -1 if any characters cannot be converted to ASCII or if the new ITEM cannot be allocated. The new ITEM will always have a zero terminator.

NOTES

The generated ASCII ITEM will lose any sub-string characteristic. When done with the ITEM **itemt*, release it using **ReleaseItem**().

RETURN VALUE

0	ITEM converted to ITYP_UNKNOWN with ASCII values
-1	Error, use **GetError**() for error code: BE_BADCHAR - Character could not be converted to ASCII BE_MEMOFLW - Failed to allocate memory

SEE ALSO

MakeCItemFromAscii()

NAME

MoveString - Copies Unicode character string ITEMs

SYNOPSIS

```
#include <items/items.h>

UNICODE *MoveString(ITEM *dest, const ITEM *src, int rawrules)
```

DESCRIPTION

MoveString is simply a combination of **CatString** and **CatWrapup** (as in **CatWrapup**(*dest*, **CatString**(*dest*, *src*, *dest*->Date), *rawrules*);). Refer to those functions for more details.

RETURN VALUE

UNICODE * Ending destination pointer (i.e. first not written), one beyond the terminating null.

SEE ALSO

CatString(), **CatWrapup**(), **Substrings**

NAME

NullItem - Null an ITEM
NullItems - Null several ITEMs

SYNOPSIS

```
#include <items/items.h>

void NullItem(ITEM *item)
void NullItems(ITEM *item, int n)
```

DESCRIPTION

Create an ITYP_NULL item. Disassociate any existing data from ITEM **item*. The disassociated data is not freed.

NullItems() nulls *n* items where **items* points to the first element.

SEE ALSO

IsItemNull(), **Releasing Items**

NAME

NumMembersSItem - Return number of members in a structure ITEM

SYNOPSIS

```
#include <items/items.h>

size_t NumMembersSItem(const ITEM *structitem)
```

DESCRIPTION

Return the number of members in an ITEM that references an entire structure (**IsSItem**() is true).

RETURN VALUE

size_t Number of members in the structure

SEE ALSO

SetupMemberItem(), **SetupElementItem**()

NAME

ParseChar - Parse left most Unicode character from an ITEM

SYNOPSIS

```
#include <items/items.h>

UNICODE ParseChar(ITEM *src)
```

DESCRIPTION

Parse the left most Unicode character from ITEM **src* and return it. The resulting item is now a sub-string with its left most character missing and the remaining characters shifted left once.

RETURN VALUE

UNICODE The left most Unicode character

SEE ALSO

Removing Leading or Trailing Characters

NAME

ParseFilespec - Parse file specifications for OPEN, BUILD, etc.
ParseFilespecOptions - Parse file specification for MODIFY

SYNOPSIS

```
#include <items/items.h>

int ParseFilespec(ITEM *dest, ITEM *src)
int ParseFilespecOptions(ITEM *dest, ITEM *src)
```

DESCRIPTION

ParseFilespec() and **ParseFilespecOptions**() parse various sub-string items from a “file specification” string, returning with ITEM **src* advanced past all parsed items. ITEM **dest* must point to an array of MAXFILESPECITEMS items, which are setup to reference the following:

Item	Type	Refers to
dest[FILESPEC_PATH]	ITYP_CHARACTER	Pathname
dest[FILESPEC_OPTIONS]	ITYP_CHARACTER	Options from "(options)"
dest[FILESPEC_PROTECTION]	ITYP_CHARACTER	Protection from "<protection>"
dest[FILESPEC_COST]	ITYP_NUMBER	Cost from "$c.cc"
dest[FILESPEC_NUMRECORDS]	ITYP_NUMBER	Records from "[R:L]"
dest[FILESPEC_RECORDLEN]	ITYP_NUMBER	Record length from "[R:L]"
dest[FILESPEC_ARGS]	ITYP_CHARACTER	Arguments past the pathname

This list constitutes parameters suitable for passing to **ChannelOpen**(), C, etc. in the “dL4 Channel Package”. For a detailed description of these parameters, see “Filenames and Pathnames” in the “dL4 Language Reference Manual”. When done with the ITEMs **dest*, release it using **ReleaseItems**().

Parsing stops after the pathname, which must exist. All other items may exist at most once, in any order, and return null if not found. Any data remaining after the pathname is copied to *dest*[FILESPEC_ARGS] and is also left unchanged in **src*.

At least the pathname must be specified in **src* in order for **ParseFilespec**() to succeed. **ParseFilespecOptions**() requires only that at least one field is specified.

MAXFILESPECITEMS, ITYP_CHARACTER, and ITYP_NUMBER are defined in “items/items.h”.

RESTRICTIONS

**dest* and **src* must refer to ITYP_CHARACTER items, else results are undefined.

RETURN VALUE

Int	Number of items returned (some may be null items)
-1	Error, use **GetError**() for error code:
	BE_BADFILENAME - Pathname not found or contains invalid characters
	BE_FILESYNTAX - Error in parsing items 1 - 5
	BE_MEMOFLW - Out of memory

SEE ALSO

ParsePathname(), **ChannelOpen**() and related functions in the "dL4 Channel Package".

NAME

ParseNumber - Parse a Unicode string ITEM for a decimal number

SYNOPSIS

```
#include <items/items.h>
#include <math/math.h>

size_t ParseNumber(ACCUM *dest, ITEM *src)
```

DESCRIPTION

Parses a decimal number from a Unicode character string ITEM **src*, skipping any leading white space, and stores the numeric value in ACCUM **dest*. The resulting ITEM **src* becomes a sub-string starting at the first character past the parsed number.

RESTRICTIONS

**dest* and **src* must refer to an ITYP_CHARACTER item, else results are undefined.

RETURN VALUE

>0	Number of parsed Unicode characters
0	Not a valid number

SEE ALSO

ParseChar(), **StringToNumber**()

NAME

ParsePathname - Parse portable pathname from a character ITEM

SYNOPSIS

```
#include <items/items.h>

int ParsePathname(ITEM *dest, ITEM *src)
```

DESCRIPTION

ParsePathname() parses a pathname (actually the first field) from the string item '*src*' returning with '*src*' advanced past the pathname. The pathname is delimited by either white space or EOS. If '*dest*' is not a null pointer, it is setup to reference the parsed pathname. When done with the ITEM **dest*, release it using **ReleaseItem**().

This function is intended to parse the first field of a character item. (Fields are separated by white space.) This first field does not need to be a pathname.

If there are no fields in **src* (**src* == "" or all white space), **dest* is set to "". (That is, null data, not null item.)

RESTRICTIONS

**src* must refer to an ITYP_CHARACTER item, else results are undefined. **dest* is initialized to an ITYP_CHARACTER item.

RETURN VALUE

0	Successful operation
-1	Error, use **GetError**() for error code:

SEE ALSO

ParseFilespec()

NAME

ParseWSCItem - Parse white space from a Unicode string ITEM

SYNOPSIS

```
#include <items/items.h>

size_t ParseWSCItem(ITEM *dest, ITEM *src)
```

DESCRIPTION

ParseWSCItem() parses white space from the string ITEM **src* returning with **src* advanced past the white space. If ITEM **dest* is not a null pointer, it is setup to reference the parsed white space. When done with the ITEM **dest*, release it using **ReleaseItem**().

RESTRICTIONS

**src* must refer to an ITYP_CHARACTER item, else results are undefined. *dest* may be null. If *dest* is not null, it must point to an ITYP_CHARACTER item.

RETURN VALUE

>0	Number of parsed Unicode characters
0	No leading white space

SEE ALSO

SkipWSCItem(), **FindNextWS**(), **IsUcWhiteSpace**()

NAME

ParseWSDelimitedCItem - Parse whitespace delimited field

SYNOPSIS

```
#include <items/items.h>

size_t ParseWSDelimitedCItem(ITEM *dest, ITEM *src)
```

DESCRIPTION

ParseWSDelimitedCItem() parses the first field from the string item '*src*' returning with '*src*' advanced past the field. The field is delimited by either white space or EOS. If '*dest*' is not a null pointer, it is setup to reference the parsed field. When done with the ITEM **dest*, release it using **ReleaseItem**().

This function is intended to parse the first field of a character item. (Fields are separated by white space.)

If there are no fields in **src* (**src* == "" or all white space), **dest* is set to "". (That is, null data, not null item.)

RESTRICTIONS

**src* must refer to an ITYP_CHARACTER item, else results are undefined. **dest* is initialized to an ITYP_CHARACTER item.

RETURN VALUE

N	Number of characters in field

SEE ALSO

ParseChar(), **ParseNumber**()

NAME

ReallocCItem - Reallocate a Unicode string ITEM
ReallocUItem - Reallocate a binary string ITEM

SYNOPSIS

```
#include <items/items.h>

int ReallocCItem(ITEM *item, size_t newlen)
int ReallocUItem(ITEM *item, size_t newlen)
```

DESCRIPTION

Reallocate the data area of the Unicode character or binary string ITEM **item* for *newlen* characters plus one for the null (for ReallocCItem()) or exactly *newlen* bytes (for ReallocUItem()). The contents will be the same up to the lesser of the new and old sizes.

RESTRICTIONS

**item* must refer to an ITYP_CHARACTER or ITYP_UNKNOWN item (depending on the function), else results are undefined.

RETURN VALUE

0	Reallocation successful
-1	Error, use **GetError**() for error code: BE_MEMOFLW - Failed to allocate memory BE_VARINUSE - Data is either in use, locked, or not allocated

SEE ALSO

Creating New Items

NAME

ReleaseItem - Release the memory for an ITEM
ReleaseItems - Release the memory for several ITEMs

SYNOPSIS

```
#include <items/items.h>

void ReleaseItem(ITEM *item)
void ReleaseItems(ITEM *items, int num)
```

DESCRIPTION

ReleaseItem() unlocks any memory associated with ITEM **item*, unlocks it, and if its temporary and has no other locks, frees it. The resulting **item* becomes a ITYP_NULL item.

ReleaseItems() releases *num* items where **items* points to the first element.

SEE ALSO

NullItem(), **NullItems**()

NAME

SetupConstCItem - Setup an Unicode character string ITEM
SetupConstUItem - Setup a binary ITEM
SetupCItem - Setup an Unicode character string ITEM
SetupDItem - Setup a date ITEM
SetupNItem - Setup a numeric ITEM
SetupUItem - Setup a binary ITEM

SYNOPSIS

```
#include <items/items.h>

void SetupConstCItem(ITEM *item, UNICODE *src, size_t len)
void SetupConstUItem(ITEM *item, void *src, size_t len)
void SetupCItem(ITEM *item, UNICODE *src, size_t len)
void SetupDItem(ITEM *item, void *src, int fmt)
void SetupNItem(ITEM *item, void *src, int fmt)
void SetupUItem(ITEM *item, void *src, size_t len)
```

DESCRIPTION

Setup the character/date/numeric/binary ITEM **item* to describe the specified Unicode string/date/number/binary object **src*.

For **SetupCItem**()
The actual size of the item must be *len* plus one and the zero terminator must be present, though this is not verified.

For **SetupDItem**() and **SetupNItem**()
Valid identifiers for *fmt* are described in sections "**Date Format Codes**" of the "dL4 Date Package" and "**Number Format Types**" of the "dL4 Math Package".

For **SetupUItem**()
The item size is set to exactly *len*.

The difference between these and the **MakeXXXX**() functions is that the **MakeXXXX**() functions actually copy the data to a new memory area while these functions merely create an ITEM wrapper around an existing data object.

SetupConstCItem() and **SetupConstUCItem**() are the same as **SetupCItem**() and **SetupUItem**() except in the casting of **src* to avoid a 'const discarded' warning from some compilers.

SEE ALSO

Creating New Items, "**Date Format Codes**" of the "dL4 Date Package", "**Number Format Types**" in the "dL4 Math Package"

NAME

SetupMemberItem - Setup an ITEM to select a member of a structure

SYNOPSIS

```
#include <items/items.h>

void SetupMemberItem(ITEM *member, ITEM *structure, size_t index)
```

DESCRIPTION

Setup ITEM *member* to reference the Nth member of the structure ITEM *structure*. The ITEM *structure* must be of type ITYP_STRUCTURE and have a valid non-null Def value. If the selected member of the structure is itself a structure, then member will be of type ITYP_STRUCTURE and **SetupMemberItem**() can be used again to access its members. When done with the ITEM **member*, release it using **ReleaseItem**().

Example: Given that s is an ITEM selecting a structure defined by "Def Struct Rec=2%,a,b$(20),c", then **SetupMemberItem**(&m, &s, 1) would set ITEM m to be a character ITEM referencing b$.

SEE ALSO

SetupElementItem(), **NumMembersSItem**()

NAME

ShutdownItems - Shutdown Items package

SYNOPSIS

```
#include <items/items.h>

void ShutdownItems(void)
```

DESCRIPTION

Release global variables for the Items package.

SEE ALSO

InitItems()

NAME

SkipWSCItem - Trim leading white space from a Unicode string ITEM

SYNOPSIS

```
#include <items/items.h>

size_t SkipWSCItem(ITEM *src)
```

DESCRIPTION

SkipWSCItem() trims leading white space from the string ITEM **src* by advancing the ITEM's data pointer.

RESTRICTIONS

**src* must refer to an ITYP_CHARACTER item, else results are undefined.

RETURN VALUE

>0	Number of leading Unicode characters removed
0	No leading white space

SEE ALSO

ParseWSCItem(), **FindNextWS**(), **IsUcWhiteSpace**()

NAME

StoreDItem - Store an ACCUM into a date ITEM
StoreNItem - Store an ACCUM into a numeric ITEM

SYNOPSIS

```
#include items.h

int StoreDItem(ITEM *dest, ACCUM *src)
int StoreNItem(ITEM *dest, ACCUM *src)
```

DESCRIPTION

Convert the specified ACCUM **src* to the date or numeric type specified by ITEM **dest* and store it in **dest*. ITEM **dest* must already have data allocated to it.

RETURN VALUE

0	Store succeeded
-1	Store failed, use **GetError**() for error code: BE_OVERFLOW - Invalid format or number too big.

SEE ALSO

LoadDItem(), **LoadNItem**(), **FloatNItem**(), **StoreAccum**(), **StoreDate**()

NAME

SubstringCItem - Convert character ITEM to sub-string of itself
SubstringUItem - Convert binary ITEM to sub-string of itself

SYNOPSIS

```
#include <items/items.h>

int SubstringCItem(ITEM *item, size_t offset, size_t len)
int SubstringUItem(ITEM *item, size_t offset, size_t len)
```

DESCRIPTION

These functions are used to modify a Unicode character string or binary ITEM **item* to select a sub-string of the original item where *offset* is the base zero index of the start of sub-string and *len* is its length (which may be zero).

If you want to create a substring and preserve the original string, use **DupItem**() to create a duplicate item and invoke **SubstringCItem**() or **SubstringUItem**() on the duplicated item.

RETURN VALUE

0	Successful operation
-1	Error, use **GetError**() for error code: BE_PARMRANGE - *offset* and *len* exceeds string size

SEE ALSO

DupItem(), **Removing Leading or Trailing Characters**

NAME

TranslateTextLine - Translate mnemonics in text

SYNOPSIS

```
#include <items/items.h>

ssize_t TranslateTextLine(ITEM *dest, ITEM *src, unsigned int *column)
```

DESCRIPTION

TranslateTextLine copies a line of Unicode text from the item referenced by '*src*', into the item referenced by '*dest*', with certain special characters translated and possibly expanded to multiple characters. The resulting string is better suited to conversion to an 8-bit character set for output to an external character stream or text file. Processed characters are trimmed from '*src*' (see **LTrimCItem**()).

'*column*' points to an unsigned int which constitutes the current "column counter" for the output stream, meaning columnar position within the current text line. The column counter is used and updated by **TranslateTextLine**() based on the characters copied.

RETURN VALUE

<0	Error, use **GetError**() for error code
n	Number of characters stored in '*dest*'

SEE ALSO

CopyTextLine()

NAME

TypeOfItem - Return an ITEM's type

SYNOPSIS

```
#include <items/items.h>

unsigned int TypeOfItem(const ITEM *item)
```

DESCRIPTION

Return the type of the ITEM *item as one of the following identifiers defined in "items/items.h":

ITYP_ARRAY
ITYP_CHARACTER
ITYP_DATE
ITYP_NUMBER
ITYP_STRUCTURE
ITYP_UNKNOWN
ITYP_NULL

RETURN VALUE

Int	Type of ITEM

SEE ALSO

"Item Type"

NAME

IndexElementItem - Adjust an ITEM to select another array element

LBoundAItem - Return the index of the first element of an array ITEM
UBoundAItem - Return the index of the last element of an array ITEM

NumElementsAItem - Return the number of elements in an array ITEM

SetupBaseElementItem - Setup an ITEM to the first base array element

SetupElementItem - Setup an ITEM to select the first element of an array

SYNOPSIS

```
#include <items/items.h>

void IndexElementItem(ITEM *element, int i)

int LBoundAItem(const ITEM *arrayitem)
int UBoundAItem(const ITEM *arrayitem)

size_t NumElementsAItem(const ITEM *arrayitem)

size_t SetupBaseElementItem(ITEM *element, ITEM *array)

void SetupElementItem(ITEM *element, ITEM *array)
```

DESCRIPTION

IndexElementItem()

Adjust an ITEM that currently references an array element to select the *i*'th element relative to the current element. The operation is analogous to the C statement "(void) p += i" where p is any pointer and *i* is the index value. Negative index values can be used to move backwards through an array.

LBoundAItem() and **UBoundAItem**()

Returns the lower/upper bound of the index range of an array ITEM. Currently, the lower bound is always zero. The upper bound is index of the last element in the array.

NumElementsAItem()

Returns the number of elements in an ITEM that references an entire array (IsAItem() is true).

SetupBaseElementItem()

Setup ITEM element to reference the first base element of the array ITEM array. The ITEM *array* must be of type ITYP_ARRAY and have a valid non-null Def value. If the array is an array of arrays (a multi- dimensional array), then **SetupBaseElementItem**() will descend to the first non-array level of the ITEM. If the array is an array of structures, then element will have the type ITYP_STRUCTURE. When done with the ITEM **element*, release it using **ReleaseItem**().

A base element is the first data element of an array or matrix, as opposed to a row of a matrix (as would be selected by SetupElementItem()).

SetupElementItem()

Setup ITEM *element* to reference the first element of the array ITEM array. The ITEM *array* must be of type ITYP_ARRAY and have a valid non-null Def value. If the array is an array of arrays (a matrix), then element will have the type ITYP_ARRAY and will reference the first row of the matrix. If the array is an array of structures, then element will have the type ITYP_STRUCTURE. When done with the ITEM **element*, release it using **ReleaseItem**().

RESTRICTIONS

IndexElementItem()

'*element*' must point at an array element with a valid Def pointer.

RETURN VALUE

LBoundAItem(), **UBoundAItem**(), **NumElementsAItem**()

x Index of first or last element of the array

SetupBaseElementItem()

x Number of elements in all levels of the array.

SEE ALSO

SetupMemberItem()

NAME

CompareCItem - Compare Unicode string ITEMs
CompareUItem - Compare binary string ITEMs
NativeCompareCItem - Compare Unicode string ITEMs
CompareCICItem - Compare Unicode string ITEMs ignoring case
CompareCIAsciiCItem - Compare Unicode string ITEM to ASCII ignoring case

SYNOPSIS

```
#include <items/items.h>

int CompareCItem(const ITEM *a, const ITEM *b)
int CompareUItem(const ITEM *a, const ITEM *b)
int NativeCompareCItem(const ITEM *a, const ITEM *b)
int CompareCICItem(const ITEM *a, const ITEM *b)
int CompareCIAsciiCItem(const ITEM *a, const char *ascii)
```

DESCRIPTION

Any comparison checks at most “x” elements, where “x” is the lesser of the dimensioned length of **a* or **b*. In addition, when comparing character items the comparison stops at the first UNICODE null character (if it hasn’t finished before that).

CompareCItem() and **NativeCompareString**() are identical at this time.

CompareCICItem() performs a case-insensitive comparison.

CompareCIAsciiCItem() performs a case-insensitive comparison against an ASCII string.

RESTRICTIONS

CompareCItem(), **NativeCompareCItem**(), **CompareCICItem**(), and **CompareCIAsciiCItem**()
**a* and **b* must refer to an ITYP_CHARACTER item, else results are undefined.

CompareUItem()
**a* and **b* must refer to an ITYP_UNKNOWN item, else results are undefined.

RETURN VALUE

=0	Strings are identical
>0	**a* is greater than **b*
<0	**a* is less than **b*

SEE ALSO

Searching Items

NAME

SearchCItem - Search for a Unicode string ITEM within a string
SearchUItem - Search for a binary string ITEM within a string

SYNOPSIS

```
#include <items/items.h>

ssize_t SearchCItem(ITEM *si, ITEM *ti)
ssize_t SearchUItem(ITEM *si, ITEM *ti)
```

DESCRIPTION

Search the Unicode character or binary string ITEM **si* for the first occurrence of Unicode character or binary string ITEM **ti*.

RESTRICTIONS

**si* and **ti* must refer to an ITYP_CHARACTER or ITYP_UNKNOWN item (depending on the function), else results are undefined.

RETURN VALUE

Int	First character position within **si* matching **ti*
-1	Failed search

SEE ALSO

Comparing Items

NAME

LTrimCItem - Modify a character ITEM to delete leading characters
LTrimUItem - Modify a binary ITEM to delete leading elements
RTrimCItem - Modify a character ITEM to delete trailing characters
RTrimUItem - Modify a binary ITEM to delete trailing elements

SYNOPSIS

```
#include <items/items.h>

int LTrimCItem(ITEM *item, size_t n)
int LTrimUItem(ITEM *item, size_t n)
int RTrimCItem(ITEM *item, size_t n)
int RTrimUItem(ITEM *item, size_t n)
```

DESCRIPTION

These functions are used to modify the Unicode character or binary ITEM **item* to select a sub-string of the original ITEM such that the first or last *n* bytes or Unicode characters are deleted. The resulting sub-string may be of length zero.

RETURN VALUE

0	Successful operation
-1	Error, use **GetError**() for error code(BE_PARMRANGE, n exceeds string size)

NAME

Item Types
IsAItem - Test an ITEM for type array
IsCItem - Test an ITEM for type character
IsDItem - Test an ITEM for type date
IsNItem - Test an ITEM for type number
IsSItem - Test an ITEM for type structure
IsUItem - Test an ITEM for type binary (unknown)

SYNOPSIS

```
#include <items/items.h>

int IsAItem(ITEM *item)
int IsCItem(ITEM *item)
int IsDItem(ITEM *item)
int IsNItem(ITEM *item)
int IsSItem(ITEM *item)
int IsUItem(ITEM *item)
```

DESCRIPTION

Test the specified ITEM **item* for the type associated with the function used:

IsAItem()	Test ITEM **item* for type ITYP_ARRAY
IsCItem()	Test ITEM **item* for type ITYP_CHARACTER
IsDItem()	Test ITEM **item* for type ITYP_DATE
IsNItem()	Test ITEM **item* for type ITYP_NUMBER
IsSItem()	Test ITEM **item* for type ITYP_STRUCTURE
IsUItem()	Test ITEM **item* for type ITYP_UNKNOWN

RETURN VALUE

0	ITEM is not of specific type
1	ITEM is of specific type

SEE ALSO

TypeOfItem(), **IsItemNull**()

Chapter 9 – Channel Functions

The channel functions are used to perform channel operations such as opening and reading files. In addition, channel functions provide access to dL4 communications (“SIGNAL”) and system information. Only the channel functions and definitions documented in this chapter should be used; all other functions and definitions are internal to dL4 and are not supported for general use.

NAME

ChClose - Perform channel closing operations

SYNOPSIS

```
int ChClose(DRVRCMNDCLOSE cmd, CHANNUM chan)
```

DESCRIPTION

Allow the program to invoke the various file closing operation directly. These operations are normally handled by wrapper functions.

The valid values for 'cmd' are:

Symbol	Function	Description
DCL_CLEAR	(Closing Channels)	Clear and close a channel
DCL_CLOSE	(Closing Channels)	Close a channel

RETURN VALUE

0 Success.
-1 Failure. Use **GetError**() for error code.

SEE ALSO

ChHndClose(), **ChCtrl**(), **ChDirect**(), **ChOpen**(), **Closing channels**

NAME

ChCtrl - Perform general file functions

SYNOPSIS

```
int ChCtrl(DRVRCMNDCTRL cmd, CHANEXPR *ce, ITEM *items, size_t nitems)
```

DESCRIPTION

Allow the program to invoke the various general file operations directly. These operations are normally handled by wrapper functions.

The valid values for '*cmd*' are:

Symbol	Function	Description
DCC_GETPARM	ChannelGet()	Get a driver parameter
DCC_SETPARM	ChannelSet()	Set a driver parameter
DCC_SETPOSITION	ChannelSetFP()	Set the file pointer
DCC_NCHF	ChannelFunctN()	CHF(x)
DCC_CCHF	ChannelFunctC()	CHF$(x)
DCC_QUERYCHAR	ChannelQueryChar()	Test the displayability of a char
DCC_RELINQUISH		Implementation-specific
DCC_REACQUIRE		Implementation-specific
DCC_SIZE	ChannelSize()	Resize an object
DCC_MOVE	ChannelMove()	Move an object
DCC_SHOWWINDOW	ChannelShow()	Display an object
DCC_HIDEWINDOW	ChannelHide()	Un-display an object
DCC_RANKWINDOW	ChannelRank()	Rank an object
DCC_HSCROLL	ChannelHScroll()	Scroll an object horizontally
DCC_VSCROLL	ChannelVScroll()	Scroll an object vertically
DCC_ADD	ChannelAdd()	Add stored information to a channel
DCC_DELETE	ChannelDelete()	Delete data from a channel
DCC_SEARCH	ChannelSearch()	Search for data in a channel
DCC_ERASE	ChannelErase()	Erase data in a channel
DCC_SYNC	ChannelSync()	Flush the data in a channel
DCC_SHAREDLOCK		Create a shared lock on a channel
DCC_EXCLUSIVELOCK		Create an exclusive lock on a chnl
DCC_UNLOCK	ChannelUnlock()	Remove an existing lock on a chnl
DCC_MAPITEM	ChannelMapItem()	Map field numbers to field names
DCC_READITEM	ChannelReadItem()	Read data from a channel
DCC_WRITEITEM	ChannelWriteItem()	Write data to a channel
DCC_MATREADITEM	ChannelMatReadItem()	Read data from a channel
DCC_MATWRITEITEM	ChannelMatWriteItem()	Write data to a channel
DCC_RAWREADITEM	ChannelRawReadItem()	BITS READ
DCC_RAWWRITEITEM	ChannelRawReadItem()	BITS READ
DCC_RAWMATREADITEM	ChannelRawMatReadItem()	BITS MAT READ
DCC_RAWMATWRITEITEM.	ChannelRawMatReadItem()	BITS MAT READ
DCC_UNREADITEM	ChannelUnreadItem()	Push chars as next channel input

Most channel information is passed in a CHANEXPR structure, with the format:

```
typedef struct {
      long Parms[3];    /* Parms[0] = record */
                        /* Parms[1] = item/byte */
                        /* Parms[2] = time-out */
      CHANNUM     Channel;
} CHANEXPR;
```

The Parm[123] fields may have different meanings depending on which function is called. See the individual function documentation for more information.

RETURN VALUE

0	Success.
-1	Failure. Use **GetError**() for error code.

SEE ALSO

ChClose(), **ChDirect**(), **ChOpen**()

NAME

ChDirect - Perform direct file operations without a channel

SYNOPSIS

```
int ChDirect(DRVRCMNDDIREC cmd, ITEM *ASname, ITEM *items, size_t
      nitems)
```

DESCRIPTION

Allow the program to invoke the various file operations directly. These operations are normally handled by wrapper functions.

The valid values for 'cmd' are:

Symbol	Function	Description
DCD_DUPLICATE	FileDuplicate()	Copy a file
DCD_KILL	FileKill()	Delete a file
DCD_MODIFY	FileModify()	Change a file's attributes

RETURN VALUE

0	Success.
-1	Failure. Use **GetError**() for error code.

SEE ALSO

ChClose(), **ChCtrl**(), **ChOpen**()

NAME

ChOpen - Open and build files

SYNOPSIS

```
int ChOpen(DRVRCMNDOPEN cmd, OPENEXPR *oe, ITEM *items, size_t nitems)
```

DESCRIPTION

Allow the program to invoke the various file open operations directly. These operations are normally handled by wrapper functions.

The valid values for 'cmd' are:

Symbol	Function	Description
DCO_BUILD	ChannelBuild()	Create a new file
DCO_OPEN	(Opening channels)	Open an existing file

Some file open information is passed through the OPENEXPR structure, formatted as follows:

```
typedef struct {
      ITEM     Driver;  /* character ITEM containing Class or Title */
                        /* of the desired driver. Null if not */
                        /* specified. */
      CHANNUM Channel;  /* desired channel number */
} OPENEXPR;
```

RETURN VALUE

0	Success.
-1	Failure. Use **GetError**() for error code.

SEE ALSO

ChClose(), **ChCtrl**(), **ChDirect**()

NAME

ChannelAdd - Add stored information to a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelAdd(CHANEXPR *i)
```

DESCRIPTION

This function adds data currently in a channel to a file. The net effect is that a record is added to a file, such as adding a full-isam record.

This function is also used to attach phantom ports.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelBuild(), **ChannelBuildString**(), **ChannelDelete**(), **ChannelErase**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **IsChannelOpen**(), **Opening channels**, **Closing channels**

NAME

ChannelBuild - Build a file from an ITEM list
ChannelBuildString - Build a file from a filename specification

SYNOPSIS

```
#include <channel/channel.h>

int ChannelBuild(OPENEXPR *oe, ITEM *FileInfo, size_t NumItems)
int ChannelBuildString(OPENEXPR *oe, ITEM *Filename)
```

DESCRIPTION

Attempt to create a file. On successful completion, the file is open for read/write.

Refer to **ChOpen**() for a description of OPENEXPR.

Refer to the documentation of function ParseFilespec() in the Items package documentation for a description of **FileInfo*.

NumItems is the number of item elements in **FileInfo*.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChOpen(), **ChannelErase**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **IsChannelOpen**(), **Opening channels**, **Closing channels**, **ParseFilespec**()

NAME

ChannelDelete - Delete the data on a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelDelete(CHANEXPR *ce)
```

DESCRIPTION

This function purges and data currently in a channel. The net effect is that a file is returned to its initial state.

This function is also used to evict ports.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelAdd(), **ChannelBuild**(), **ChannelBuildString**(), **ChannelErase**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **IsChannelOpen**(), **Opening channels**, **Closing channels**

NAME

ChannelErase - Erase the data in a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelErase(CHANEXPR *ce)
```

DESCRIPTION

This function erases the contents of a channel. The effect is driver dependent.

This function is also used to clear the terminal screen.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelAdd(), **ChannelBuild**(), **ChannelBuildString**(), **ChannelDelete**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **IsChannelOpen**(), **Opening channels**, **Closing channels**

NAME

ChannelFunctC - Process dL4 CHF$(x) function requests
ChannelFunctN - Process dL4 CHF(x) function requests
ChannelFunctD - Process dL4 CHF#(x) function requests
ChannelFunctU - Process dL4 CHF?(x) function requests

SYNOPSIS

```
#include <channel/channel.h>
#include <channel/classes.h>

int ChannelFunctC(CHANEXPR *ce, ITEM *CHFInfo)
int ChannelFunctN(CHANEXPR *ce, ITEM *CHFInfo)
int ChannelFunctD(CHANEXPR *ce, ITEM *CHFInfo)
int ChannelFunctU(CHANEXPR *ce, ITEM *CHFInfo)
```

DESCRIPTION

The **ChannelFunctXXXX**() functions process dL4 CHFx() function requests for the character, numeric, date, and binary data types.

Actual information requests are passed in *ce*->Parms[0] and the results are stored in **CHFInfo*. Any prior contents of **CHFInfo* are lost.

The valid information requests for ChannelFunctN() are:

Request	Description
DP0_FILE_NCHFNRECS	Return file size in records
DP0_FILE_NCHFRECNUM	Return current record number
DP0_FILE_NCHFRECDISP	Return current record displacement
DP0_FILE_NCHFRECLEN	Return record length in bytes
DP0_FILE_NCHFNBYTES	Return file size in bytes
DP0_FILE_NCHFRECLENB	Return record length in bytes

The valid information requests for ChannelFunctC() are:

Request	Description
DP0_FILE_CCHFOPENMODE	Get open mode of channel
DP0_FILE_CCHFCLASS	Get driver class of file open on channel
DP0_FILE_CCHFTITLE	Get driver title of file open on channel
DP0_FILE_CCHFFILENAME	Get filename of file open on channel

Remember that not all drivers support all of these operations. You need to refer to the individual drivers to discover which requests are supported.

RETURN VALUE

>=0	Success. Result stored in **CHFInfo*
-1	Error, use **GetError**() for error code.

SEE ALSO

Individual driver documentation

NAME

ChannelGet - Get driver-specific information
ChannelSet - Set driver-specific information

SYNOPSIS

```
#include <channel/channel.h>
#include <channel/classes.h>

int ChannelGet(CHANEXPR *ce, ITEM *DriverInfo, size_t NumItems)
int ChannelSet(CHANEXPR *ce, ITEM *DriverInfo, size_t NumItems)
```

DESCRIPTION

These functions get/set driver-specific information from/into **DriverInfo*. Below are the parameters that you may Get/Set. Refer to the actual driver for the functions that are implemented.

When making requests to the system channel, set CHANEXPR.Channel to SystemChan. Similarly, when sending requests to the communications driver, set CHANEXPR.Channel to CommChan. Requests to Get or Set information are passed using the Parms[] array in CHANEXPR. The actual request is passed in Parms[0] with supplemental data passed in Parms[1] and Parms[2] as needed.

Below is a list of valid values for Parms[0], see channel/classes.h for the most up to date list:

Symbol (Parms[0] value)	Description
DP0_SYS_GETINFO	Get misc. system info selected by Parms[1]
DP0_COMM_GETINFO	Get misc. comm info selected by Parms[1]
DP0_RAW_GETOSFHND	Get OS file handle of file in binary ITEM
DP0_INDEX_SETCHARSET	Set character set
DP0_CONTIG_GETHEADER	Read and return file header
DP0_WIN_GETCOL	Get current column number (0 - x)
DP0_WIN_GETNCOL	Get number of terminal columns
DP0_WIN_GETNROW	Get number of terminal rows
DP0_WIN_GETTERMTYPE	Return terminal type number
DP0_WIN_GETNWCOL	Get number of columns in display window
DP0_WIN_GETNWROW	Get number of rows in display window
DP0_WIN_GETNCCOL	Get number of columns in window canvas
DP0_WIN_GETNCROW	Get number of rows in window canvas
DP0_WIN_GETNRCOL	Get number of columns in window region
DP0_WIN_GETNRROW	Get number of rows in window region
DP0_WIN_GETWCOL	Return column of window
DP0_WIN_GETWROW	Return row of window
DP0_WIN_GETCWCOL	Return column of window in canvas
DP0_WIN_GETCWROW	Return row of window in canvas
DP0_WIN_GETCRCOL	Return column of output region in canvas
DP0_WIN_GETCRROW	Return row of output region in canvas
DP0_WIN_GETIOOPT	Get mnemonic string to restore current PUA_IOxx options
DP0_WIN_GETWHND	Get window handle of window. Not for general use.
DP0_WIN_GETUSEWDW	Get "open using window" setting. Not for general use.
DP0_WIN_SETRESTORE	Restore standard I/O options
DP0_WIN_SETECHO	Set echo enabled on/off by Parms[1]
DP0_WIN_SETINPUTPEND	Set input pending on/off by Parms[1]
DP0_WIN_SETACTCTLINPUT	Set activate-on-ctl-ch on/off by Parms[1]
DP0_WIN_SETBINARYINPUT	Set binary input on/off by Parms[1]
DP0_WIN_SETBINARYOUTPUT	Set binary output on/off by Parms[1]
DP0_WIN_SETTITLE	Set window title

When invoking DCC_GETPARM and Parms[0] = DP0_COMM_GETINFO for Comm class drivers, set Parms[1] to one of the following values. These particular requests have wrapper functions, which you should use instead:

Symbol (Parms[1] value)	Function	Value returned
DP1_COMM_GETOWNPORT	GetPortNumber()	Own port number
DP1_COMM_GETAVPORT	GetAvailablePortNumber()	Next available port number
DP1_COMM_GETMAXPORTS	GetMaxNumberOfPorts()	Maximum number of ports
DP1_COMM_GETINUSECNT	GetNumberOfPortsInUse()	Number of ports current in-use

When invoking DCC_GETPARM or DCC_SETPARM with Comm class drivers, Parms[0] indicates a port number if >= 0. In that case, Parms[1] should be to one of the following functions:

Symbol (Parms[1] value)	Description
DP1_COMM_STATWORD	Get/Set status word (PORT stmt mode 3)
DP1_COMM_CMNDSTR	Get/Set command string (PORT stmt mode 2)
DP1_COMM_TRXCOSTR	Get/Set command string (CALL $TRXCO)

When invoking DCC_GETPARM and Parms[0] = DP0_SYS_GETINFO for System class drivers, set Parms[1] to one of the following values. These requests should be made using GetSysInfo() instead of calling GetSysInfo(directly:

Symbol (Parms[1] value)	Description (and returned item type)
DP1_SYS_GETCD	Get current directory path (character)
DP1_SYS_GETCDLEN	Get length of current directory path
DP1_SYS_GETCURRLU	Get current logical unit number
DP1_SYS_GETSWITCHES	Get cpu switch info (SPC(7) - numeric)
DP1_SYS_GETENVVAR	Get environment var (the character item)
DP1_SYS_GETCPUTIME	Get CPU time (seconds - numeric)
DP1_SYS_GETCONNECT	Get connect time (seconds - numeric)
DP1_SYS_GETLICENSE	Get software license number
DP1_SYS_GETGRPUSER	Get group/used is for MSC(7) (numeric)
DP1_SYS_GETPRVGRPUSER	Get group/used is for SPC(5) (numeric)
DP1_SYS_GETPOSIXCD	Get POSIX current directory path (unknown)

When invoking DCC_SETPARM and Parms[0] = DP0_SYS_SETINFO for System class drivers, set Parms[1] to one of the following
values:

Symbol (Parms[1] value)	Description (and returned item type)
DP1_SYS_SETCD	Get current directory path (character)

When invoking DCC_SETPARM with Full-ISAM class drivers, set Parms[0] >= 0 (record/key definition), Parms[1] = field/keypart number, and Parms[2] = flag bit definitions (or them together). Refer to the specific driver for more information:

Symbol (Parms[2] value)	Description
DP2_ISAM_DESCENDING	Descending sequence
DP2_ISAM_UPPERCASE	Case-insensitive comparison
DP2_ISAM_UNIQUE	Ensure uniqueness
DP2_ISAM_PACKED	Compress for space (if possible)

DP2_ISAM_DECIMALMSK	Decimal places (-1 to 30)

RETURN VALUE

>=0	Success. Result stored in **CHFInfo*
-1	Error, use **GetError**() for error code.

SEE ALSO

Individual driver documentation

NAME

ChannelMapItem - Map FullISAM field numbers to Full-ISAM field names

SYNOPSIS

```
#include <channel/channel.h>

int ChannelMapItem(CHANEXPR *ce, ITEM *FieldInfo, size_t NumItems)
```

DESCRIPTION

This function sets an equivalence between Full-ISAM field numbers and Full-ISAM field names for use by subsequent operations. This equivalence remains until changed by another call to **ChannelMapItem**() or the channel is closed.

Refer to the Full-ISAM driver documentation for a description of field numbers and field names.

RETURN VALUE

>=0	Success
-1	Error, use **GetError**() for error code.

NAME

ChannelQueryChar - Check if an unicode character can be represented visually on a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelQueryChar(CHANEXPR *ce, ITEM *Char2Check)
```

DESCRIPTION

This function checks an unicode char and returns >=0 if that character can be represented as a glyph by the driver open on CHANEXPR. Otherwise, -1 is returned. See below. For example, the 'copyright' character can be displayed on the ANSI terminal but not an ASCII terminal. **ChannelQueryChar**() would return 0 for the ANSI terminal and –1 for the ASCII terminal. '*Char2Check*' should be a single character ITEM if only the first character is to be tested. Many drivers only check the first character of '*Char2Check*' and programs should not assume that only the first or all of the characters will be tested unless a specific driver will be used.

This function is used mainly when LISTing and DUMPing programs and when outputing to a device such as a printer.

RETURN VALUE

0	Character can be represented on CHANEXPR.
-1	Character cannot be represented on CHANEXPR. Use **GetError**() for error code.

NAME

ChannelSearch - Perform key-related functions on indexed or FullISAM files

SYNOPSIS

```
#include <channel/channel.h>
#include <channel/classes.h>

int ChannelSearch(CHANEXPR *ChanExpr, ITEM *SearchInfo, size_t NumItems)
```

DESCRIPTION

This is the interface for key-related functions on indexed and Full-ISAM files.

Before calling this function, the Parms[] fields of *ChanExpr* must be set as follows:

Parms[] Value
[0] Search mode (see below)
[1] Index number
[2] (unused)

Refer to the individual driver documentation for descriptions of the information to pass in **SearchInfo*[].

The valid search modes are:

Mode	Description
0	Configure indexes. uniBasic indexed files only. See **ChannelSet**() for information about defining Full-ISAM indexes. Parm2 = index number: 0=freeze structure, >0=define index (*SearchInfo*[1] contains the key length.)
1	Get index info. uniBasic indexed files only. See **ChannelGet**() for information about Full-ISAM files. See below for descriptions of the available information.
2	Set next key EQUAL to key in **SearchInfo*.
3	Set next key GREATER THAN key in **SearchInfo*.
4	Insert a new key. uniBasic indexed files only.
5	Delete a key from an index. uniBasic indexed files only.
6	Check for next key less than key in **SearchInfo*.
7	(ignored)
8	Change key insertion strategy. uniBasic indexed files only. (see the ctree manual for the allowable insertion strategies.)
9	(same as mode 6)
10	Set next key GREATER THAN OR EQUAL TO key in **SearchInfo*. (Full-ISAM files only.)
11	Set next key LESS THAN OR EQUAL TO key in **SearchInfo*. (Full-ISAM files only.)

When the search mode is one and the index is zero, *SearchInfo*[2] contains the request. Data is returned/taken from in *SearchInfo*[1]. The valid requests are:

Request	Description
0	Return the first real data record
1	Return available record count
2	Allocate a record
3	Deallocate a record
4	Return the number of records
5	Return the number of records
6	Set first real data record
7	Return number of active records

These are the errors returned in *SearchInfo*[2].data for uniBasic indexed files:

Error value	Num value	Description
SE_OK	0	Operation completed
SE_FAILED	1	Failed, probably key not found
SE_ENDOFINDEX	2	Begining or end of index reached
SE_ENDOFDATA	3	Unable to allocate record
SE_NOINDICES	4	No indices defined in file
SE_PGMORFILE	5	Key too small, bad index, or bad file
SE_INDEXSEQ	6	Index number out of sequence
SE_NOTCONTIG	7	File isn't contiguous
SE_ISINDEXED	8	Indices are already defined
SE_BADRECNUM	9	Record number is illegal
SE_BADINDEX	10	Index number is illegal
SE_NOINDEX	13	No such index defined

Full-ISAM files generate dL4 errors instead of returning an error status.

RETURN VALUE

>=0 Success.

-1 Error, use **GetError**() for error code.

NAME

ChannelSetFP - Set the file position to a specific record

SYNOPSIS

```
#include <channel/channel.h>

int ChannelSetFP(CHANEXPR *ce)
```

DESCRIPTION

Set the file pointer to a specific record. The Parms[] fields of CHANEXPR must be set as follows before calling this function:

Parms[]	Value
[0]	Record number. >=0 = record number. -1 = next record. -2 = current record. -3 = previous record. Any other negative number will return an error.
[1]	Item number or offset. -1 is mapped to 0. Any other negative number will return an error.
[2]	(unused)

This function is also used to clear communication messages of type parms[1].

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

NAME

ChannelUnlock - Release pending locks on a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelUnlock(CHANEXPR *ce, ITEM *ItemList, size_t NumItems)
```

DESCRIPTION

Release a pending record lock on a channel starting at the position indicated in **ce* for the number of bytes equal to the length of *ItemList*[0]. If *NumItems*=0 the remainder of the file is unlocked.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

Locking data, **Reading channels**, **Writing channels**

NAME

ChannelUnreadItem - Force data back onto a channel's input stream

SYNOPSIS

```
#include <channel/channel.h>

int ChannelUnreadItem(CHANEXPR *ce, ITEM *data)
```

DESCRIPTION

This function behaves similarly to ungetc(). It forces **data* back onto the input stream for CHANEXPR. Subsequent reads will then retrieve the data that was unread here.

This function is intended for terminal i/o operations only.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

Reading channels, **Writing channels**

NAME

CheckChannelNumber - Validate a channel number

SYNOPSIS

```
#include <channel/channel.h>

int CheckChannelNumber(CHANNUM chan)
```

DESCRIPTION

This function verifies that *chan* is a valid channel number.

RETURN VALUE

0	*chan* contains a valid channel number
-1	*chan* does not contain a valid channel number.

SEE ALSO

IsChannelFree(), **IsChannelOpen**()

NAME

ConvertToOSPathname - Convert a pathname into an OS-openable form

SYNOPSIS

```
int ConvertToOSPathname(ITEM *dest, ITEM *src, int options, PATHINFO
      *misc)
```

DESCRIPTION

ConvertToOSPathname() parses a pathname from the string item '*src*' optionally returning with '*src*' advanced past the pathname. **dest* is assumed to be a null item on entry and any old data in it will be overwritten by this function. If '*src*' refers to a binary (unknown) item, it is simply copied to *dest* without any translation. This facilitates successive calls to this function from other routines. If '*src*' refers to a character item, the pathname is delimited by either a space or the end of string. (On return, '*src*' points either to EOS or the first non-WS after the delimiting WS.) Any white-space before the pathname is discarded.

**dest* will be returned as a binary item containing the translated pathname. This translated pathname will have the current directory name prepended to it if it is not already an absolute path. If misc is not null, leading '$'s and a single trailing '!' will be counted and stripped from the resulting OS pathname. If *misc* is null, these characters are copied to **dest* and no special processing will occur.

The following translations are performed on the pathname copied to **dest*. This results in a pathname that may be directly opened by the OS. Note that some of these may be platform-specific:

1. Convert from Unicode to ASCII (no translation on Windows).
2. Convert to lower case (Unix) or upper case (Windows). (Only if not an absolute path.)
3. Replace ':' with '/' (Unix) or '\' (Windows). (Only if not an absolute path.)
4. If *misc* is not null, count the number of leading '$'s. Save the count in PATHINFO.LeadingDollars. Leading '$'s are not copied to **dest* if *misc* is not null.
5. If i is not null and a trailing '!' is found, PATHINFO.TrailingBang is set to one. A trailing '!' is not copied to **dest* if *misc* is not null.
6. For directory names of the form ([0-9])+[:/] strip any leading zeros. Examples: 003/ becomes 3/ and 040: becomes 40/ but 0/ stays 0/
7. Prepend the current directory to the result. (Only if not an absolute path.)

This results in an absolute path that may be directly opened by the OS.

Value	Description
CTOP_ADV	Advance 'src' past the parsed pathname and subsequent white-space.
CTOP_NOADV	Do not advance 'src' past the parsed pathname.

The PATHINFO structure is used to keep track of leading $'s and a single trailing '!'. If this is specified, these characters are also stripped from the destination. There are no wrapper functions for this struct and you may access its members directly.

```
typedef struct {
      BF16 LeadingDollars : 7;      /* number of leading dollar */
                                    /* signs seen */
      BF16 TrailingBang : 1;        /* exclamation point given */
                                    /* as terminator */
} PATHINFO;
```

RESTRICTIONS

'*src*' must refer to either an ITYP_CHARACTER or ITYP_UNKNOWN item, else results are undefined. '*src*' must be null-terminated.

RETURN VALUE

>=0	Length of converted pathname
-1	Error, use **GetError**() for actual error

SEE ALSO

ConvertToUserPathname(), **GetDirectoryOfOSPathname**()

NAME

ConvertToUserPathname - Convert an OS (ASCII) pathname to Unicode

SYNOPSIS

```
void ConvertToUserPathname(ITEM *dest, ITEM *src)
```

DESCRIPTION

This function converts an OS (ASCII for Unix, Unicode for Windows) pathname (from **src* created by **ConvertToOSPathname**()) to Unicode so it can be re-opened or displayed. The idea is to take the output from this function and be able to use it to re-open the file on another channel.

**src* is compared against the current working directory. If all of the current directory matches the first part of **src* and there are no upper-case characters after that, then **src* is converted to a relative pathname and translated to upper case. Otherwise, **src* is simply converted to Unicode and returned in **dest* without modification (as an absolute path).

**dest* will be returned as an ITYP_CHARACTER item.

RESTRICTIONS

'*src*' must refer to an ITYP_UNKNOWN item, else results are undefined. '*dest*' must refer to an ITYP_CHARACTER or Null item else results are undefined.

NOTES

This function assumes that the underlying file system is in ASCII for Unix and Unicode for Windows and the data in **src* was created to reflect this.

RETURN VALUE

>=0	Length of converted pathname
-1	Error, use **GetError**() for actual error

SEE ALSO

ConvertToOSPathname(), **GetDirectoryOfOSPathname**()

NAME

CtrlChannels - Perform control command on all channels in range
CtrlAllChannels - Perform control command on all channels

SYNOPSIS

```
void CtrlChannels(DRVRCMNDCTRL cmd, CHANNUM firstchan, CHANNUM lastchan)
void CtrlAllChannels(DRVRCMNDCTRL cmd)
```

DESCRIPTION

CtrlAllChannels() performs control 'cmd' on all channels while CtrlChannels() performs control 'cmd' on channels firstchan through lastchan.

SEE ALSO

ChCtrl()

NAME

ExecuteOSCommand - Execute an operating system command

SYNOPSIS

`int` **`ExecuteOSCommand`**`(ITEM *`*`command`*`)`

DESCRIPTION

Create a new process and run a shell command.

**command* must refer to an ITYP_CHARACTER item.

This function is a wrapper for the DC_WRITEITEM operation of the system driver.

RETURN VALUE

0	Successful
-1	Error occurred, use **GetError**() for error code: BE_?? Errors possible from **ChannelWriteItem**().

SEE ALSO

ChannelWriteItem(), **System driver**

NAME

FileDuplicate - Duplicate (copy) a file
FileKill - Delete a file
FileModify - Modify a file's attributes

SYNOPSIS

```
#include <channel/channel.h>

int FileDuplicate(ITEM *ASName, ITEM *items, size_t nitems)
int FileKill(ITEM *ASName, ITEM *items, size_t nitems)
int FileModify(ITEM *ASName, ITEM *items)
```

DESCRIPTION

FileDuplicate()

Copy a file. *nitems* must be >= 2. *items*[0] = source filename. *items*[1] = destination filename.

FileKill()

Delete a file. *nitems* must be >= 1. Deletes the file in *items*[0].

FileModify()

Change a file's attributes. Pass an item list in the same order as the one returned by **ParseFilespec**(). Change the fields you want. The file options, number of records, and record length may not be changed.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelBuild(), **ChannelBuildString**(), **ChannelErase**(), **IsChannelOpen**(), **Opening channels**, **Closing channels**, **ParseFilespec**()

NAME

FilespecChOpen - Open a file using a file specification

SYNOPSIS

```
int FilespecChOpen(DRVRCMNDOPEN cmd, OPENEXPR *oe, ITEM *filespec)
```

DESCRIPTION

Open a file using a file specification. Normally, you must pass an array of file specification items (generated by **ParseFilespec**()) to **ChOpen**. This function simplifies the file-open process by allowing the programmer to pass the original file spec as a single character item.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChOpen(), **ParseFilespec**()

NAME

ChannelHScroll - Scroll a window horizontally
ChannelVScroll - Scroll a window vertically
ChannelSize - Alter the size of an object
ChannelMove - Move an object
ChannelShow - Display an object
ChannelHide - Un-display an object
ChannelRank - Change the ordering of an object

SYNOPSIS

```
#include <channel/channel.h>
#include <channel/classes.h>

int ChannelHScroll(CHANEXPR *ce, ITEM *Xdelta)
int ChannelVScroll(CHANEXPR *ce, ITEM *Ydelta)
int ChannelSize(CHANEXPR *ce, ITEM *items, size_t NumItems)
int ChannelMove(CHANEXPR *ce, ITEM *items, size_t NumItems)
int ChannelShow(CHANEXPR *ce)
int ChannelHide(CHANEXPR *ce)
int ChannelRank(CHANEXPR *ce)
```

DESCRIPTION

A window actually consists of three rectangles: a display window (normally referred to as the window), a canvas, and an output region. The user program draws text and graphics through the output region onto the canvas. The position and size of the output region selects what portion of the canvas can be written. All style-related actions, including wrapping and scrolling, are performed within the output region; the canvas outside the region is not changed. The display window controls what portion of the canvas will be visible and where it will be shown on the screen. The output region and display window do not necessarily have to be the same area of the canvas. Any window border or title is shown outside of the display window.

ChannelHScroll() and **ChannelVScroll**()

Scroll the current window [XY]delta columns/rows relative to its current position across the canvas. [*XY*]*delta* are numeric items and may be positive or negative. If the new position is outside the screen size (ie: < 0 or >NumColumns or >NumRows), error BE_PARMRANGE will be returned.

ChannelSize()

Adjust a window's or canvas' size. Set *items*[0] to the new width and *items*[1] to the new height. Three sizing operations are possible by setting *ce*->Parms[0]:

Name	Description
DP0_WIN_SIZEWINDOW	Set display window size
DP0_WIN_SIZECANVAS	Set window canvas size
DP0_WIN_SIZEREGION	Set output region size

ChannelMove()

Move the display window or output region of the window 'window' in the window canvas or the screen and update the screen. Although an item list may passed to this function, only the first item, *items*[0], is used. It must be a character item and contain coordinate mnemonics terminated with PUA_MOVETO. Set Parms[0] to one of the following requests:

Name	Description
DP0_WIN_MOVEWINDOW	Move a window within a screen
DP0_WIN_MOVECANVAS	Move a window within a canvas
DP0_WIN_MOVEREGION	Move a window within an output region

ChannelShow() and **ChannelHide**()

Set hidden status of the current window and, if changed, update the screen. ‘window’ must not be a physical window.

ChannelRank()

Raise or lower the position of ‘window’ in the Z-order and update the screen. Set Parms[0] to one of the following requests:

Name	Description
DP0_WIN_RANKRAISE	Position ‘window’ at the top of the Z order and, if ‘refresh’ is non-zero, update the screen. ‘window’ must not be a physical window.
DP0_WIN_RANKLOWER	Position ‘window’ at the bottom of the Z order and, if ‘refresh’ is non-zero, update the screen. ‘window’ must not be a physical window.
DP0_WIN_RANKRAISEFAMILY	Position ‘window’ and its descendants at the top of the Z order without changing their relative order and update the screen. ‘window’ must not be a physical window.
DP0_WIN_RANKLOWERFAMILY	Position ‘window’ and its descendants at the bottom of the Z order without changing their relative order and update the screen. ‘window’ must not be a physical window.

A physical window is the actual display screen used in the “Window Off” mode. The only exception is if a program starts within a window then that first window is considered a physical window.

RETURN VALUE

>=0 Success.
-1 Error, use **GetError**() for error code.

NAME

GetDirectoryOfOSPathname - Return parent directory of a OS pathname

SYNOPSIS

```
ssize_t GetDirectoryOfOSPathname(ITEM *directory, ITEM *ospathname)
```

DESCRIPTION

GetDirectoryOfOSPathname() allocates a character item into '*directory*' and returns the parent directory of the OS pathname.

This directory is derived by removing the filename portion of a pathname created using **ConvertToOSPathname**().

WARNING

If 'ospathname' refers to a file in the current working directory, then the returned directory will be "", length zero.

NOTES

This function is OS dependent.

RETURN VALUE

-1	Error, use **GetError**() for error code
x	Success. The directory, if any, is in character item

SEE ALSO

ConvertToUserPathname(), **ConvertToOSPathname**(), **Directory Functions**

NAME

GetMiscCommInfo - Get information from communications channel

SYNOPSIS

```
#include <channel/channel.h>
#include <channel/classes.h>

int GetMiscCommInfo(COMMGETIPARM1 dp1)
```

DESCRIPTION

GetMiscCommInfo() returns the information selected by dp1. The valid dp1 value's are:

Name	Description
DP1_COMM_GETOWNPORT	Get own port number
DP1_COMM_GETAVPORT	Get an available port number
DP1_COMM_GETMAXPORTS	Get maximum number of allowed ports
DP1_COMM_GETINUSECNT	Get number of ports current in-use

You should use the wrapper functions defined in Port-related Functions instead of using these directly, if possible.

RETURN VALUE

0	Successful
-1	Error occurred, use GetError() for error code: BE_PARMRANGE - Illegal argument value BE_?? - Errors possible from ChannelGet().

SEE ALSO

ChannelGet(), **Port-related Functions**, **System V Communications** Driver

NAME

GetSysInfo - Get information from system channel

SYNOPSIS

```
#include <channel/channel.h>
#include <channel/classes.h>

int GetSysInfo(ITEM *item, SYSGETIPARM1 itemnumber)
```

DESCRIPTION

GetSysInfo() reads the information selected by itemnumber into item. The valid itemnumber's are:

Name	Description (item type returned)
DP1_SYS_GETCD	Get current directory path (character)
DP1_SYS_GETCDLEN	Get length of current directory path (numeric)
DP1_SYS_GETCURRLU	Get current logical unit number (numeric)
DP1_SYS_GETSWITCHES	Get cpu switch info for SPC(7) (numeric)
DP1_SYS_GETENVVAR	Get environment var (the character item
DP1_SYS_GETCPUTIME	Get CPU time (seconds, numeric)
DP1_SYS_GETCONNECT	Get connect time (seconds, numeric)
DP1_SYS_GETLICENSE	Get software license number (numeric)
DP1_SYS_GETGRPUSER	Get group/used id for MSC(7) (numeric)
DP1_SYS_GETPRVGRPUSER	Get group/used id for SPC(5) (numeric)
DP1_SYS_GETPOSIXCD	Get POSIX current directory path (unknown)

RETURN VALUE

0	Successful
-1	Error occurred, use **GetError**() for error code: BE_PARMRANGE - Illegal argument value BE_?? - Errors possible from **ChannelGet**().

SEE ALSO

ChannelGet(), **Directory Functions**, **System driver**

NAME

InitChannels - Initialize Driver Channels package

SYNOPSIS

```
#include <channel/channel.h>

void InitChannels()
```

DESCRIPTION

Initialize global variables for the driver channel package and perform assertions of package assumptions.

SEE ALSO

ShutdownChannels()

NAME

IsChannelFree - Test if a channel is free
IsChannelOpen - Test if a channel is open

SYNOPSIS

```
#include <channel/channel.h>

int IsChannelFree(CHANNUM chnl)
int IsChannelOpen(CHANNUM chnl)
```

DESCRIPTION

IsChannelFree()
Check if a channel is available.

IsChannelOpen()
Check if a channel is open. If the channel is reserved, 0 will be returned.

RETURN VALUE

0	Channel is NOT free/open or the channel number is invalid
1	Channel is free/open

SEE ALSO

ChannelBuild(), **ChannelBuildString**(), **ChannelErase**(), **CheckChannelNumber**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **GetFreeHiddenChannel**(), **GetFreeUserChannel**(), **Opening channels**, **Closing channels**

NAME

IsPortablePath - Determine if a user path is portable
IsRelativePortablePath - Determine if a user path is relative

SYNOPSIS

```
int IsPortablePath(ITEM *path)
int IsRelativePortablePath(ITEM *path)
```

DESCRIPTION

IsPortablePath() returns one if the character ITEM *'path'* is a portable path, otherwise it returns zero.

IsRelativePortablePath() returns one if the character ITEM *'path'* is a relative portable path, otherwise it returns zero.

For both functions, a 'TRUE' result indicates that the path can be treated as a portable path, but does NOT guarantee that the path is a legal portable path.

RETURN VALUE

0	*path* is not portable/relative-portable
1	*path* is portable/relative-portable

SEE ALSO

ConvertToUserPathname()

NAME

MakeStringWritable - Prepare an UNICODE string for output to a device

SYNOPSIS

```
#include <channel/chanlib.h>

int MakeStringWritable(ITEM *item, CHANNUM channel, int hexmode)
```

DESCRIPTION

Converts the character ITEM **item* for output to the device/file open on channel. If a character cannot be displayed on channel, as determined by **ChannelQueryChar**(), it is replaced with a " and its hex (if hexmode is true) or octal (if hexmode is not true) value.

The original **item* is released and replaced with the converted item.

RETURN VALUE

0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelQueryChar()

NAME

PutTypeAheadString - Place data in a channel's buffer

SYNOPSIS

```
#include <channel/channel.h>

int PutTypeAheadString(CHANNUM channel, ITEM *item)
```

DESCRIPTION

This is a wrapper function that calls a driver using **ChannelUnreadItem**(). It is up to the actual driver to process **item*.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

Posix tty driver

NAME

SendSignal - Send numeric signal to port

SYNOPSIS

```
int SendSignal(int port, MSGTYPE msgtype, long value1, long value2)
```

DESCRIPTION

Send a signal (msgtype = MSG_SEND) or system signal (msgtype = MSG_SYST) consisting of two numbers, *'value1'* and *'value2'* to 'port'.

RETURN VALUE

-1	Error while sending signal, use **GetError**() for error code
0	Success

SEE ALSO

ChannelWriteItem()

NAME

ShutdownChannels - Shutdown Driver Channels package

SYNOPSIS

```
#include <channel/channel.h>

void ShutdownChannels()
```

DESCRIPTION

Shuts down the channel package by clearing all channels and performing a shutdown of all drivers.

SEE ALSO

InitChannels()

NAME

CreateChunk - Create a chunk object using data from a binary item
DeleteEmbeddedChunk - Remove an embedded chunk from another chunk
EmbedChunk - Enbed one chunk into another chunk
ExtractEmbeddedChunk - Copy embedded chunk data to another chunk
MakeUItemFromChunk - Copy data from a chunk into an ITEM
ReadChunk - Read a chunk from a file
WriteChunk - Write a chunk to a file
ReleaseChunk - Free the memory associated with a chunk

SYNOPSIS

```
int CreateChunk(CHUNK *chunk, ITEM *data, unsigned short ctype)
int DeleteEmbeddedChunk(CHUNK *chunk, unsigned short ctype)
int EmbedChunk(CHUNK *dest, CHUNK *src)
int ExtractEmbeddedChunk(CHUNK *dest, CHUNK *src, unsigned short ctype)
int MakeUItemFromChunk(ITEM *dest, CHUNK *chunk)
ssize_t ReadChunk(CHUNK *chunk, CHANHND ch, long *parms, int
      expectedtype)
ssize_t WriteChunk(CHUNK *chunk, CHANHND ch, long *parms)
void ReleaseChunk(CHUNK *chunk)
```

DESCRIPTION

A "chunk" is an object containing miscellaneous data that is treated as a single unit. It is maintained within binary items. The length of the chunk data is the length of the binary item in which it resides.

CreateChunk()

Create a new chunk from the data contained in binary item **data*. **chunk* is initialized in this function, so any data it previously contained will be lost. If **chunk* already contained chunk data, a memory leak will result (similar to re-using ITEMs without releasing them first).

DeleteEmbeddedChunk()

Removes the embedded chunk of type ctype from **chunk*. The size of **chunk* is adjusted to reflect the deletion.

EmbedChunk()

Embeds chunk **src* int chunk **dest*. **dest* expands to contain all of **src*, including its chunk header. **src* is not changed. **dest* and **src* must contain valid chunk data on entry.

ExtractEmbeddedChunk()

ExtractEmbeddedChunk() copies chunk data of type *ctype* from **src* to **dest*. **dest* is created and should not be initialized prior to calling this function, else a memory leak will occur.

The requested chunk is not removed from **src*.

MakeUItemFromChunk()

Create a binary item and copy chunk data into it. The chunk data copied does not include the chunk header. **dest* should not contain data on entry else a memory leak will result.

ReadChunk()

Read the chunk pointed to by **parms* from file *ch*. **dest* is created and should not be initialized prior to calling this function, else a memory leak will occur.

The type of chunk read must match the expected type otherwise -1 will be returned and the error code will be set to BE_RECNOTFND.

WriteChunk()

Writes **chunk* to file ch at location **parms*. **chunk* must be a valid chunk or the write may fail (depending on what garbage was in **chunk*).

ReleaseChunk()

Unlocks memory associated with **chunk* and frees the memory if there are no outstanding locks on it. **chunk* is undefined after this function completes.

RETURN VALUE

CreateChunk(), **DeleteEmbeddedChunk**(), **EmbedChunk**(), **ExtractEmbeddedChunk**()

0	Success
-1	Failure. Use **GetError**() for error code.

MakeUItemFromChunk()

x	Type of chunk data that was extracted.
-1	Failure. Use **GetError**() for error code.

ReadChunk(), **WriteChunk**()

>=0	Success. The length in bytes of the chunk, including the chunk header, is returned.
-1	Failure. Use **GetError**() for error code.

SEE ALSO

driver/pformat.c and driver/pcontig.c for examples

NAME

ChannelClear - Abort any pending file operation and close the channel
ChannelClose - Complete pending operations and close a channel
ClearAllChannels - Clear all open channels
ClearChannels - Clear all channels in range

SYNOPSIS

```
#include <channel/channel.h>

int ChannelClear(CHANNUM chnl)
int ChannelClose(CHANNUM chnl)
void ClearAllChannels(void)
void ClearChannels(CHANNUM firstchan, CHANNUM lastchan)
```

DESCRIPTION

ChannelClose() performs an orderly shutdown of operations on *chnl* and closes the file channel.

ChannelClear() closes *chnl* immediately without any housekeeping. This means, for example, that not all information may be written or a format structure may not be frozen, and the file format may be incomplete or unusable.

The operation of **ChannelClose**() and **ChannelClear**() depend on how a driver was written. For some file types, such as text file, these operations are identical. For others, such as uniBasic Formatted files and uniBasic indexed files, a **ChannelClear**() will not perform internal unlocking and structure freezing.

ClearAllChannels() clears all open channels including hidden and user channels. **ClearChannels**() is similar to **ClearAllChannels** except that it allows the programmer to specify a range of channels to clear.

All of these functions eventually call **ChClose**(), which actually closes the channels. The only value '*cmd*'s for this function are DCL_CLOSE and DCL_CLEAR with an ascending channel range. Use of this function is discouraged in favor of the other clear/close functions described here.

RETURN VALUE

ChannelClose(), **ChannelClear**()

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelBuild(), ChanbnelBuildString(), **ChannelErase**(), **ChClose**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **GetFreeHiddenChannel**(), **GetFreeUserChannel**(), **IsChannelOpen**(), **Opening channels**

NAME

GetCurrentSysDirectory - Get current directory
SetCurrentSysDirectory - Set current directory

SYNOPSIS

```
#include <channel/channel.h>

int GetCurrentSysDirectory(ITEM *item)
int SetCurrentSysDirectory(ITEM *item)
```

DESCRIPTION

GetCurrentSysDirectory() allocates and returns a character ITEM containing the current working directory.

SetCurrentSysDirectory() sets the current working directory to the path specified by the character ITEM *item*.

These are wrapper functions for the system driver. Refer to the System driver documentation for more information.

RETURN VALUE

0	Successful
-1	Error occurred, use **GetError**() for error code: BE_PARMRANGE - Illegal argument value BE_?? - Errors possible from **ChannelGet**().

SEE ALSO

GetDirectoryOfOSPathname(), **System driver**, **GetSysInfo**()

NAME

AbortiveEventsPending - The number of abortive queue events pending
ClearChanEvents - Clear all pending channel events
DontQueueEveryEvent - Reset a flag so that each CEType event queue request is combined with existing CEType requests
GetChanEvent - Dequeue a channel event
IsEventPending - Return the number of outstanding events for a specific event type
QueueChanEvent - Enqueue a channel event
QueueEveryEvent - Set a flag so that each CEType event queue request results in a new queue entry
QueuePriorityChanEvent - Enqueue a channel event at a specific priority
SetAbortiveChanEvent - Change an event's type to 'abortive'
SetNonAbortiveChanEvent - Change an event's type to 'non-abortive'

SYNOPSIS

```
#include <channel/channel.h>

BYTE AbortiveEventsPending(void)
void ClearChanEvents(void)
void DontQueueEveryEvent(int CEType)
int GetChanEvent(CHANEVENT *ce)
int IsEventPending(CHANEVENTTYPE cet)
int QueueChanEvent(CHANEVENTTYPE cet)
void QueueEveryEvent(int CEType)
int QueuePriorityChanEvent(CHANEVENTTYPE cet, int priority)
void SetAbortiveChanEvent(int CEType)
void SetNonAbortiveChanEvent(int CEType)
```

DESCRIPTION

Asynchronous events are those that occur at any time, such as a user pressing escape. Each event has a priority >= 0. Higher values equate to higher priority, and higher priority events get processed first. These events are processed by the dL4 front end (i.e. scope).

Normally, there is at most one queue entry for each event type. However, if **QueueEveryEvent**() is set for this event type, each new request is enqueued (if the queue is not full).

QueuePriorityChanEvent() enqueues a channel event at a specific priority while **QueueChanEvent**() enqueues a channel event with a priority of zero. If the event is abortive, the number of abortive queued events is incremented.

GetChanEvent() returns the first entry of the Event Queue into **ce*. The entry is removed from the queue. If the event is abortive, the number of abortive queued events is decremented. No priority is specifically given to Abortive events.

Normally, there is only one event queue entry for each pending *CEType*, and each additional request simply increments a counter. Function **QueueEveryEvent**() sets a flag so that every subsequent enqueue request results in a new (separate) queue entry, not simply an increment of the outstanding requests for a given *CEType* as is normal. Reset this flag with **DontQueueEveryEvent**().

SetAbortiveChanEvent() changes an event type from non-abortive to abortive. The number of pending events of this type are added to the number of outstanding abortive events. If the event was already abortive, no action is taken. The event type is reset to non-abortive using function **SetNonAbortiveChanEvent**(), which also reduces the count of outstanding abortive events.

AbortiveEventsPending() returns a count of pending abortive events.

ClearChanEvents() cancels all pending channel events regardless of priority. The **'QueueEveryEvent'** flag for each event is unaffected by this operation.

These are the valid CHANEVENTTYPE values:

Event type	Description
CET_NOEVENT	Special catch-all event type
CET_SHUTDOWN	Shutdown the port ASAP
CET_ESCAPE	User-requested escape
CET_ABORT	User-requested abort
CET_ALTESCAPE	Alternate user-requested escape
CET_MESSAGE	Communication message received
CET_SIGNALSELF	User-requested message loopback
CET_SWAP	User-requested swap to pre-defined program
CET_PORTCMND	Port mode 2 style command string waiting
CET_TRXCO	CALL 98 style command string waiting (unused)

These are used in the CHANEVENT structure, defined as follows:

```
typedef struct {
      CHANEVENTTYPE Type;     /* event type  */
      int Priority;           /* event priority (larger values */
                              /* are higher) */
} CHANEVENT;
```

RETURN VALUE

AbortiveEventsPending()

The actual count of abortive events. 0 if no abortive events are queued.

IsEventPending()

The actual count of pending events. 0 if no events are pending.

GetChanEvent(), **QueueChanEvent**()

0	Event queue operation was successful
-1	Event queue is empty

There are two global channels that are available to any function that needs them:

Channel	Description
CommChan	System V message queue channel
SystemChan	Channel for system-related functions

In addition, CommChan and SystemChan have driver-specific functions that may be used in addition to the other functions in this chapter.

Refer to the appropriate Driver manual pages for more information.

NAME

ChannelExclusiveLock - Create a Exclusive over a file region
ChannelSharedLock - Create a Shared over a file region

SYNOPSIS

```
#include <channel/channel.h>

int ChannelExclusiveLock(CHANEXPR *ce, ITEM *ItemList, size_t NumItems)
int ChannelSharedLock(CHANEXPR *ce, ITEM *ItemList, size_t NumItems)
```

DESCRIPTION

Create an Exclusive or Shared lock over a region of the file starting at the position described in **ce* for the number of bytes equal to the length of *ItemList*[0]. If *NumItems*=0 the remainder of the file is locked.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

Locking data, **Reading channels**, **Writing channels**

NAME

ChannelOpen - Open a file for read/write from an ITEM list
ChannelOpenString - Open a file for read/write from a

SYNOPSIS

```
#include <channel/channel.h>

int ChannelOpen(OPENEXPR *ce, ITEM *FileInfo, size_t NumItems)
int ChannelOpenString(OPENEXPR *ce, ITEM *Filename)
```

DESCRIPTION

Attempt to Open a file.

Normally, files are open for read/write. Other open modes may be specified through the FILESPEC_PROTECTION field in **FileInfo* or as part of **Filename*. The possible values are:

Values	Meaning
E	Exclusively open the file (EOPEN)
L	Don't use or obey record locks
R	Open the file with read protection
W	Open the file with write protection

Example: "<WL>" = ROPEN = Open the file with write-protection and ignore record locks.

Refer to the documentation of function **ParseFilespec**() in the Items package documentation for a description of **FileInfo*.

In OPENEXPR, set the Driver field to contain the class or title of the desired driver. OPENEXPR.Channel may be set the any of these values:

Value	Description
CHOPEN_FINDUSERCHAN	Find an unused hidden channel number.
CHOPEN_FINDHIDDENCHAN	Find an unused hidden channel number.
x	Any other number to be taken literally as the channel number.

NumItems is the number if item elements in **FileInfo*.

ParseFilespec() (see Items documentation) is used to decompose the passed **Filename*, which is then passed to **ChannelOpen**() to open the actual file.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelBuild(), **ChannelBuildString**(), **ChannelErase**(), **FileDuplicate**(), **FileKill**(), **FileModify**(), **IsChannelFree**(), **IsChannelOpen**(), **Closing channels**, **ParseFilespec**(), Individual driver documentation

NAME

GetAvailablePortNumber - Return the next available port number
GetPortNumber - Return your port number
GetMaxNumberOfPorts - Return the Maximum port number
GetNumberOfPortsInUse - Return the number of dL4 users

SYNOPSIS

```
int GetAvailablePortNumber(int beg, int end)
int GetPortNumber(void)
int GetMaxNumberOfPorts(void)
int GetNumberOfPortsInUse(void)
```

DESCRIPTION

GetAvailablePortNumber() returns the next available port number in the range *beg*..*end*. The search is ascending if *end*>*beg* and descending if *beg*>*end*.

GetPortNumber() returns your port number. Your port number is determined at startup using the following method:

1) Check the environment for "PORT=%d".
2) Find a port number by examining the ttyname for clues.
3) Find a port number by brute force search starting at the maximum port and working downward.

GetMaxNumberOfPorts() returns the Maximum port number, defined as 4096.

GetNumberOfPortsInUse() returns the number of dL4 users.

These are wrapper functions for ths System V Communications driver.

RETURN VALUE

x	Normal return value of function
-1	Error. See **GetError**() for error code

SEE ALSO

GetMiscCommInfo(), **System V Communications driver**

NAME

ReadProfile - Read next value pair from profile
SearchProfile - Search profile for section name
SearchProfileAscii - Search profile for ASCII section name

SYNOPSIS

```
int ReadProfile(CHANEXPR *ce, UNICODE *left, size_t leftsize, UNICODE
      *right, size_t rightsize)
int SearchProfile(CHANEXPR *ce, ITEM *sectionname)
int SearchProfileAscii(CHANEXPR *ce, const char *sectionname)
```

DESCRIPTION

ReadProfile()

Read next value pair from the profile file open in **ce* into *'left'* and *'right'*. Both values will be zero terminated. At the end of the current section, a BE_RECNOTFND error will be returned. At exit, the channel expression *'ce'* is set to read the next pair.

SearchProfile() and **SearchProfileAscii**()

Search profile channel specified by **ce* for the section specified by the section.

WARNINGS

ReadProfile()

'leftsize' and *'rightsize'* must be large enough to contain both the value string and the zero terminator or the value will be truncated.

SearchProfile() and **SearchProfileAscii**()

If a section is not found, the name of the section will be added to the error text. This is useful in most applications, but could be a security hole if the section name is secret.

RETURN VALUE

0	Successful
-1	Error, use **GetError**() for error code

NAME

ChannelMatReadItem - IRIS-style read a whole item from a channel
ChannelRawMatReadItem - BITS-style read a whole item from a channel
ChannelRawReadItem - BITS-style read an item from a channel
ChannelReadItem - IRIS-style read an item from a channel
ChannelReadItems - IRIS-style read >= 1 items from a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelMatReadItem(CHANEXPR *ce, ITEM *RdInfo)
int ChannelRawMatReadItem(CHANEXPR *ce, ITEM *RdInfo)
int ChannelRawReadItem(CHANEXPR *ce, ITEM *RdInfo)
int ChannelReadItem(CHANEXPR *ce, ITEM *RdInfo)
int ChannelReadItems(CHANEXPR *ce, ITEM *RdInfo, size_t nitems)
```

DESCRIPTION

These functions read data from an open channel, one ITEM at a time.

On exit, the requested record is locked. The record is unlocked if these functions exit abnormally.

Whole arrays may only be MatRead, not Read. An error will result if you try **ChannelReadItem**() or **ChannelRawReadItem** on an array.

Structure items may not be assed to these routines. You need to pass each structure member one at a time.

The MatRead functions try to fill as much of the destination as possible. They do not stop at the first null byte (for strings) or element (arrays).

The Parms[] fields in CHANEXPR must be set as follows before calling this function:

Parms[x]	Value
Parms[0]	Record number to read (-1=current record, -2=next record)
Parms[1]	Offset or item number in record
Parms[2]	Delay (timeout) in tenth of seconds

On function exit, the Parms[] fields of CHANEXPR are modified as follows:

Parms[x]	Value
Parms[0]	-1 or -2 depending on the driver. (Sets up the next read to read the current or next record)
Parms[1]	Next offset or item number to read. This is derived by adding the size of the item you wanted to read (from the ITEM structure) to the offset you passed to thefunction.
Parms[2]	unchanged

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelUnlock(), **ChannelUnreadItem**(), **Opening channels**, **Closing channels**, **Writing channels**, Individual driver documentation

NAME

ChannelMatWriteItem - IRIS-style write a whole item from a channel
ChannelRawMatWriteItem - BITS-style write a whole item from a channel
ChannelRawWriteItem - BITS-style write an item from a channel
ChannelWriteItem - IRIS-style write an item from a channel
ChannelWriteItems - IRIS-style write >= 1 items from a channel

SYNOPSIS

```
#include <channel/channel.h>

int ChannelMatWriteItem(CHANEXPR *ce, ITEM *WrInfo)
int ChannelRawMatWriteItem(CHANEXPR *ce, ITEM *WrInfo)
int ChannelRawWriteItem(CHANEXPR *ce, ITEM *WrInfo)
int ChannelWriteItem(CHANEXPR *ce, ITEM *WrInfo)
int ChannelWriteItems(CHANEXPR *ce, ITEM *WrInfo, size_t nitems)
```

DESCRIPTION

These functions write data to an open channel, one ITEM at a time.

On exit, the specified record is locked. The record is unlocked if these functions exit abnormally.

Whole arrays may only be MatWrite, not Write. An error will result if you try **ChannelWriteItem**() on an array.

Structure items may not be passed to these routines. You need to pass each structure member one at a time.

The MatWrite functions try to fill as much of the destination as possible. They do not stop at the first null byte (for strings) or element (arrays).

The Parms[] fields in CHANEXPR must be set as follows before calling this function:

Parms[x]	Value
Parms[0]	Record number to read (-1=current record, -2=next record)
Parms[1]	Offset or item number in record
Parms[2]	Delay (timeout) in tenth of seconds

On function exit, the Parms[] fields of CHANEXPR are modified as follows:

Parms[x]	Value
Parms[0]	-1 or -2 depending on the driver. (Sets up the next write to write the current or next record)
Parms[1]	Next offset or item number to write. This is derived by adding the size of the item you wanted to write (from the ITEM structure) to the offset you passed to the function.
Parms[2]	unchanged

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChannelUnlock(), **ChannelUnreadItem**(), **Opening channels**, **Closing channels**, **Reading channels**, Individual driver documentation

Chapter 10 – Driver Functions

The driver functions are used only by dL4 drivers and must not be used by user defined CALLs, by user defined functions, or anywhere outside of a dL4 driver. Only the driver functions and definitions documented in this chapter should be used; all other functions and definitions are internal to dL4 and are not supported for general use.

NAME

ChDirectRawFile - Perform DIRECT command using the raw file driver

SYNOPSIS

```
#include <channel/driver.h>

int ChDirectRawFile(DRVRCMNDDIREC cmd, ITEM *items, size_t nitems)
```

DESCRIPTION

Perform a DIRECT command (delete, copy, or modify a file) using the raw file driver.

RETURN VALUE

0	Successful
-1	Error, use **GetError**() for error code

NAME

ChHndClose - Perform channel closing operations using a channel handle

SYNOPSIS

```
#include <channel/driver.h>

int ChHndClose(DRVRCMNDCLOSE cmd, CHANHND chanhnd)
```

DESCRIPTION

Release storage used by the channel handle table and call the close function of the driver associated with chanhnd.

Normally, you should use ChClose() instead of this function since ChHndClose() won't actually close the channel if it is in use elsewhere. ChClose() also manages channel links for you.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChClose(), **ChHndCtrl**(), **ChHndOpen**(), **ChHndDirect**()

NAME

ChHndCtrl - Perform general file functions using a channel handle

SYNOPSIS

```
#include <channel/driver.h>

int ChHndCtrl(DRVRCMNDCTRL cmd, CHANHND chanhnd, long *parms, ITEM
      *items, size_t nitems)
```

DESCRIPTION

Map the function ‘cmd’ into the driver specified by ‘*chanhnd*’. This function is similar to **ChCtrl**() except that the channel handle is used instead of a CHANEXPR.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChCtrl(), **ChHndClose**(), **ChHndOpen**(), **ChHndDirect**()

NAME

ChHndOpen - Perform channel opening operations using a channel handle

SYNOPSIS

```
#include <channel/driver.h>

CHANHND ChHndOpen(DRVRCMNDOPEN cmd, ITEM *ASname, ITEM *items,
      size_t nitems)
```

DESCRIPTION

Map the function '*cmd*' into the driver specified by '*chanhnd*'. This function is similar to **ChOpen**() except that the channel handle is used instead of a CHANEXPR.

RETURN VALUE

>=0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

ChOpen(), **ChHndCtrl**(), **ChHndClose**(), **ChHndDirect**(), **ChHndOpenRawFile**(), **ChHndOpenByteFile**(), **Open Modes**

NAME

ChHndOpenRawFile - Open a channel handle using the raw file driver
ChHndOpenHugeRecordFile - Open a channel handle using the raw file driver with a record size of one gigabyte

SYNOPSIS

```
#include <channel/driver.h>

CHANHND ChHndOpenRawFile(DRVRCMNDOPEN cmd, ITEM *items, size_t nitems)
CHANHND ChHndOpenHugeRecordFile(DRVRCMNDOPEN cmd, ITEM *items, size_t
nitems)
```

DESCRIPTION

ChHndOpenRawFile() opens a channel handle to a file using the raw file driver.
ChHndOpenHugeRecordFile() is similar, but ignores all parameters except the path and protection and guarantees that the record size is HUGEFILE_RECORDLENGTH (one gigabyte)

RETURN VALUE

(CHANHND)h	Successful
(CHANHND)0	Error, use **GetError**() for error code

SEE ALSO

ChHndOpen()

NAME

ChannelOSPathname - Get an open file's native absolute pathname

SYNOPSIS

```
#include <channel/driver.h>

int ChannelGetOSPathname(CHANEXPR *ce, ITEM *id)
```

DESCRIPTION

Returns the actual absolute pathname opened on channel *_ce_. *_id_ is an item of type unknown or is made if passed as a null item. On Unix or POSIX systems, *_id_ will contain the path in ASCII bytes. On Windows systems, *_id_ will contain the path in Unicode characters.

RETURN VALUE

>=0	Success. Result stored in *_id_.
-1	Error, use **GetError**() for error code.

SEE ALSO

Individual driver documentation

NAME

InitContigMetrics - Initialize a CTGMETRICS structure
ExportContigMetrics - Export CTGMETRICS data
ImportContigMetrics - Import CTGMETRICS data

SYNOPSIS

```
#include <channel/driver.h>

void InitContigMetrics(CTGMETRICS *rm)
int ExportContigMetrics(CHUNK *data, CTGMETRICS *rm)
int ImportContigMetrics(CTGMETRICS *rm, CHUNK *data)
```

DESCRIPTION

Contiguous file metrics are used to save file information in a chunk, which is then saved in a file. These functions provide a portable way to save file header information. Functions may freely access the fields within the CTGMETRICS structure:

```
typedef struct {
      unsigned long NumRecordsTotal;      /* number of total records */
      unsigned long NumRecordsInUse;      /* number of in-use records */
      unsigned long RecordNumberBias;     /* record numbering bias */
      BYTE IsIndexed;                     /* set if file is indexed */
} CTGMETRICS;
```

New fields may be added to this structure at any time (Any new fields will be added to the end of the structure to provide upward compatability).

InitContigMetrics()
Save default information to a CTGMETRICS structure. Any existing data is lost.

ExportContigMetrics()
Create a new chunk of type CTYP_CTGMETRICS using data from the CTGMETRICS structure.

ImportContigMetrics()
Exract data from a chunk of type CTYP_CTGMETRICS into a CTGMETRICS structure.

RETURN VALUE

ExportContigMetrics() and **ImportContigMetrics**()

0	Success.
-1	Failure. Use **GetError**() for error code.

SEE ALSO

chunk

NAME

GetChannelHnd - Retrieve the handle of a channel

SYNOPSIS

```
#include <driver/chanlib.h>

CHANHND GetChannelHnd(CHANNUM chan)
```

DESCRIPTION

GetChannelHnd() returns the handle associated with the channel 'chan'. If the 'chan' is invalid or represents a reserved channel, (CHANHND)0 is returned.

RETURN VALUE

(CHANHND)h	Success.
(CHANHND)0	Error, use **GetError**() for error code

NAME

GetEnvLong - Translate the value of an environment variable to a long

SYNOPSIS

```
#include <driver/drvrlib.h>

long GetEnvLong(const char *varname, long defaultvalue)
```

DESCRIPTION

Translate an environment variable to a long value. The environment variable must be translatable into a long value. That is, it must be a string of digits.

If the environment variable is not found or its value translates to zero, defaultvalue is returned.

RETURN VALUE

Long value containing either the translated environment value or the default value passed.

NAME

PairedFileDirect - Perform non-channel functions on paired files

SYNOPSIS

```
#include <channel/driver.h>

int PairedFileDirect(DRVRCMNDDIREC cmd, ITEM *items, size_t nitems, int
      (*MakeOtherFilename)(ITEM *dest, const ITEM *src))
```

DESCRIPTION

Perform non-channel operations on paired files, i.e. files which exist as two files on the host O/S filesystem, such as Indexed Contiguous files.

The caller-supplied function **MakeOtherFilename**() is used to generate the second filename of the pair based on the first. Therefore, in order to use **PairedFileDirect**(), the name of the companion file must be a derivative of themain filename. **MakeOtherFilename**() should accept an O/S pathname in *'src'* and create an O/S pathname for the companion file in *'dest'*.

RESTRICTIONS

Only the following operations are currently supported: DCD_DUPLICATE DCD_MODIFY DCD_KILL

All others cause a BE_BADDRVOP error.

RETURN VALUE

>=0	Successful
-1	Error, use **GetError**() for error code

NAME

ParseProtection - Parse protection field for Posix

SYNOPSIS

```
#include <channel/driver.h>

int ParseProtection(mode_t *perm, ITEM *protection)
```

DESCRIPTION

Parse the Unicode string in 'protection' and extract the file access permissions and into 'perm'. The protection string consists attribute letters optionally followed by a two or three octal digit access bits.

Letters:

"A"	Allow reading by any member of the group
"B"	Allow writing by any member of the group
"D"	Prohibit deletion of the file (n/a to POSIX)
"P"	Allow reading and writing by all
"R"	Prohibit reading by anyone except the file owner
"W"	Prohibit writing by anyone except the file owner

Two digit octal number: The number is interpreted as IRIS access bits with each bit having the following meaning:

40	Prohibit reading by other groups
20	Prohibit writing by other groups
10	Prohibit copying by other groups (n/a to POSIX)
04	Prohibit reading by the same group
02	Prohibit writing by the same group
01	Prohibit copying by the same group (n/a to POSIX)

Three digit octal number: The number is interpreted as traditional Unix access bits where the first digit controls access by the file owner, the second controls access by other members of the owner's group, and the third controls access by members of other groups. The bits of each digit have the following meaning:

4	Allow reading
2	Allow writing
1	Allow execution

If protection is a null ITEM or the access permissions are not specified, '*perm' will be set to a suitable default value.

RETURN VALUE

-1	Error, use **GetError**() for error code
0	Successful evaluation

NAME

InitRecordMetrics - Initialize a RECMETRICS structure
ExportRecordMetrics - Export RECMETRICS data
ImportRecordMetrics - Import RECMETRICS data

SYNOPSIS

```
#include <channel/driver.h>

void InitRecordMetrics(RECMETRICS *rm)
int ExportRecordMetrics(CHUNK *data, RECMETRICS *rm)
int ImportRecordMetrics(RECMETRICS *rm, CHUNK *data)
```

DESCRIPTION

Record metrics are used to save record information in a chunk, which is then saved in a file. These functions provide a portable way to save a file's record length andcharacter set. Functions may freely access the fields within the RECMETRICS structure:

```
typedef struct {
      unsigned long RecordLength;   /* record length in bytes */
      char CharSet[32];             /* character set for string data */
} RECMETRICS;
```

New fields may be added to this structure at any time (Any new fields will be added to the end of the structure to provide upward compatability).

InitRecordMetrics()
Save default information to a RECMETRICS structure. Any existing data is lost.

ExportRecordMetrics()
Create a new chunk of type CTYP_RECMETRICS using data from the RECMETRICS structure.

ImportRecordMetrics()
Exract data from a chunk of type CTYP_RECMETRICS into a RECMETRICS structure.

RETURN VALUE

ExportRecordMetrics() and **ImportRecordMetrics**()

0	Success.
-1	Failure. Use **GetError**() for error code.

SEE ALSO

chunk

NAME

SetFilePos - Convert logical record and item numbers to real values

SYNOPSIS

```
#include <channel/driver.h>

int SetFilePos(CHANINFO *ci, long *parms)
```

DESCRIPTION

This function converts logical record (including record -1 and -2) and item numbers to real numbers usable by the actual read and write routines.

Pass the record and item numbers in *parms*[0] and *parms*[1].

RETURN VALUE

0	Success.
-1	Error, use **GetError**() for error code.

SEE ALSO

portable and uniBasic formatted and indexed file drivers, CHANINFO definition in channel/driver.h

NAME

DefaultDriverInit - Perform defaults for driver initialization commands
DefaultDriverCtrl - Perform defaults for driver control commands
DefaultDriverClose - Perform defaults for driver close commands
DefaultDriverDirect - Perform defaults for driver direct commands
DefaultDriverOpen - Perform defaults for driver open commands

SYNOPSIS

```
#include <channel/driver.h>

int DefaultDriverInit(DRVRCMNDINIT cmd)
int DefaultDriverCtrl(DRVRCMNDCTRL cmd, CHANINFO *chaninfo, long *parms,
      ITEM *items, size_t nitems)
int DefaultDriverClose(DRVRCMNDCLOSE cmd, CHANINFO *chaninfo)
int DefaultDriverDirect(DRVRCMNDDIREC cmd, ITEM *items, size_t nitems)
CHANHND DefaultDriverOpen(DRVRCMNDOPEN cmd, ITEM *items, size_t nitems)
```

DESCRIPTION

Typically, these routines are called when your driver gets an operation it doesn't support. Refer to existing drivers for examples.

DefaultDriverInit() performs the default actions for each of the driver initialization commands listed below:

DCI_INIT - do nothing and return 0
DCI_SHUTDOWN - do nothing and return 0
DCI_IDLESHUTDOWN - return BE_BADRVOP error to indicate that the driver did not shutdown

DefaultDriverCtrl() performs default actions for each of the following driver control commands:

DCC_UNLOCK - do nothing and return 0
DCC_NCHF - copy requested information from chaninfo
DCC_CCHF - copy requested information from chaninfo

DefaultDriverClose() performs the default actions for each of the driver close commands: it does nothing and then returns zero to indicate a successful close.

DefaultDriverDirect() performs default actions for each of the following driver direct commands:

DCD_DUPLICATE - Pass command to raw file driver
DCD_MODIFY - Pass command to raw file driver
DCD_KILL - Pass command to raw file driver

DefaultDriverOpen() performs the default actions for each of the driver open commands: it returns a BE_BADDRVOP error.

RETURN VALUE

DefaultDriverOpen()

(CHANHND)0	Error, use **GetError**() for error code
(CHANHND)x	Success.

All others:

-1	Error, use **GetError**() for error code
0	Success.

NAME

LoadDriver - Load driver, if necessary, and return driver index
LockDriver - Lock driver into active driver table
ReleaseDriver - Release and (if necessary) shut down a driver
ShutdownAllDrivers - Issue shutdown commands to all active drivers

SYNOPSIS

```
#include <channel/chanlib.h>

DRVRIDX LoadDriver(ITEM *ASname)
int LockDriver(DRVRIDX dridx)
void ReleaseDriver(DRVRIDX dridx)
void ShutdownAllDrivers(void)
```

DESCRIPTION

LoadDriver() returns the driver index of the driver specified by the class or title name in the character ITEM *ASname*. If the driver does not exist or cannot be loaded/initialized, NO_DRIVER will be returned.

LockDriver() locks the specified driver in the active driver table and, if necessary, initializes the driver.

ReleaseDriver() decrements the driver usage count and, if the count is zero, attempts to shutdown the driver.

ShutdownAllDrivers() issues shutdown (DCI_SHUTDOWN) commands to all active drivers.

NOTES

LoadDriver() and **LockDriver** support recursive usage of **LoadDriver**(), **LockDriver**(), and **ReleaseDriver**() by driver init routines **.**

WARNINGS

While **ReleaseDriver**() does support recursion, it does NOT support use of LoadDriver() by driver shutdown functions.

ShutdownAllDrivers() does not support loading of other drivers by driver shutdown function.

RETURN VALUE

LoadDriver()

(DRVRIDX)d	Success.
NO_DRIVER	Error, use **GetError**() for error code

LockDriver()

0	Success.
-1	Error, use **GetError**() for error code

NAME

LinkChannel - Link a channel number to a channel handle
UnlinkChannel - Release a channel number to channel handle link

SYNOPSIS

```
void LinkChannel(CHANNUM chan, CHANHND chanhnd)
CHANHND UnlinkChannel(CHANNUM chan)
```

DESCRIPTION

Link and unlink channel numbers with channel handles. These are low-level functions that you will probably not need. Use **ChOpen**() and channel oriented functions if possible.

WARNINGS

No error checking is performed in these routines. If you pass invalid data, memory may be corrupted. This, in turn, may cause dL4 to fail ungracefully.

RETURN VALUE

UnlinkChannel()

(CHANHND)x Handle of the channel that was just unlinked.

SEE ALSO

ChOpen(), **ChClose**()

NAME

ChHndGetOpenMode - Get open access mode from channel handle
IsExclusiveOpenSpecified - Check if Exclusive Open specified
IsReadProtectionSpecified - Check if Read Protection specified
IsWriteProtectionSpecified - Check if Write Protection specified
ParseOpenMode - Parse open access mode from ITEM

SYNOPSIS

```
#include <channel/driver.h>

int ChHndGetOpenMode(OPENMODE *mode, CHANHND chanhnd)
int IsExclusiveOpenSpecified(ITEM *prot)
int IsReadProtectionSpecified(ITEM *prot)
int IsWriteProtectionSpecified(ITEM *prot)
int ParseOpenMode(OPENMODE *mode, ITEM *item)
```

DESCRIPTION

ChHndGetOpenMode() sets '*mode*' according to the access mode of the channel handle '*chanhnd*'.

IsExclusiveOpenSpecified(), **IsReadProtectionSpecified**(), and **IsWriteProtectionSpecified**() scan **prot* to determine if Exclusive Open/ Read Protection/Write Protection was specified. **prot* must contain the FILESPEC_PROTECTION item returned by **ParseFileSpec**(). If **prot* is not a character item, 0 is always returned.

ParseOpenMode() initializes the open access mode '*mode*' according to the ITEM '*item*'. If '*item*' is a null ITEM, then '*mode*' is simply cleared. Otherwise, '*mode*' is cleared and then set according to the characters found within the character '*item*' as follows:

If any other characters are found, a BE_FILESYNTAX error will be returned. Duplicated mode letters are legal. '*item*' must be a character or null ITEM and typically contains the FILESPEC_PROTECTION item returned by **ParseFileSpec**().

RETURN VALUE

ChHndGetOpenMode(), **ParseOpenMode**()

-1	Error, use **GetError**() for error code
0	Success.

IsXXXXSpecified()

0	Open Mode is not specified
1	Open Mode is specified

SEE ALSO

ChOpen(), **ParseFileSpec**()

Appendix A - Glossary

This glossary defines terms in the context of dL4. For the concepts behind many of these terms, refer to *Introduction to dL4:*

absolute pathname	the full pathname, starting at the root.
BASIC object code	SEE object code.
block	one or more statements treated as though they were a single statement.
channel	a communication method between an application and a dL4 driver for requesting specific file operations.
character	a letter, number, or other special data representation.
character code	a numeric value that represents a particular character in a set, such as the ASCII character set.
character data type	a representation of a letter, number, or other special data representation.
character set	a mapping of characters to their identifying numeric values.
context	SEE runtime context.
driver	a dL4 driver acts as a translator converting a generic file operation request from an application program into a specific command that carries out the requested operation.
executable	a program that is ready for execution.
file	a collection of records.
index	a mechanism of locating data.
infinite loop	the never-ending repetition of a block of dL4 statements.
interface	SEE port.
ISAM files	ISAM (Indexed Sequential Access Method) is a storage and retrieval system that allows efficient access to data records using key values.
key values	identifying values used in a file to describe and locate a desired record.
keyword	a reserved word used as part of dL4 syntax.
loop	the repeated circular execution of one or more statements.
member	each individual data type in a structure data type. See structure data type.
nested loop	a loop within a loop.
object code	a translation, not readable to the user, of a program source code that can be directly executed by the computer.
operand	a piece of data upon which an operation is performed.
phantom port	a port that does not have access to its display device. Typically it runs in background.
portable	capable of being ported to different systems.
position parameter	A position parameter is used by some BASIC/Debugger commands to specify a line in a dL4 program. Refer to *dL4 Command Reference Guide*, Appendix C for description of position parameter.

program	a set of executable instructions.
relative pathname	a partial pathname relative to your current working directory.
record	a set of related fields.
reserved word	in dL4, a word that has a fixed function and cannot be used for any other purpose. Same as keyword.
root	the root directory, which is the main directory that contains everything on the disk.
run time	related to the events that occur while a program is being executed.
runtime context	a machine state when a dL4 program is executed.
SCCS	Source Code Control System (SCCS) is a Unix utility that allows source code level revision control for a project.
source code	a user-readable text file containing dL4 BASIC language statements.
step into	trace inside a function.
step through	execute a function but do not trace inside a function. Trace resumes outside the function.
string	a sequence of alphanumeric characters. dL4 converts all strings to Unicode characters.
structure data type	a data type that organizes different data types so that they can be referenced as a single unit. Typically, used to define a record in a data file.
subscript	a number inside brackets that differentiates one element of an array from another.
Unicode	a 16-bit character set capable of encoding all known characters and used as a worldwide character-encoding standard.

Appendix B - Unicode Character Set

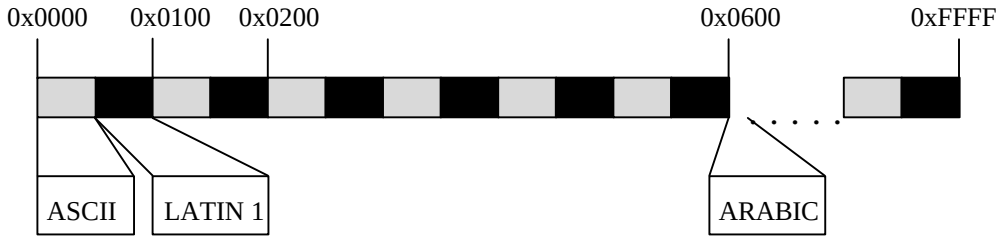
Introduction

Unicode is a 16-bit, fixed-width, uniform text and character encoding scheme. It includes most of world's written scripts, publishing characters, mathematical and technical symbols, geometric shapes, basic dingbats and punctuation marks. In addition to modern languages such as Arabic, Bengali and Thai, it also includes such classical languages as Greek, Hebrew, Pali and Sanskrit.

The Unicode set can represent more than 65,000 characters and includes many of the traditional character sets. The first 128 characters, i.e. 0x00 - 0x7F, are identical to the ASCII character set. The first 256 characters, i.e. 0x00 - 0xFF, represent the ISO 8859-1, or Latin1 character set (note that ASCII is a subset of ISO 8859-1). Unicode values 0x2500 - 0x257F and 0x2580 - 0x27BF, represent forms and charts, and special graphics characters, respectively. Unicode values 0xE000 – 0xF7FF are defined as “private use” or implementation defined characters; dL4 uses these characters for mnemonics and mnemonic parameters.

One of the advantages of the Unicode character set over other character sets is that it allows data representation from anywhere in the world in a uniform, plaintext format. In other words, Unicode simplifies software internationalization.

The following illustrates the Unicode encoding layout.



Unicode is used internally for all text processing in dL4. Externally, the various drivers at the I/O level perform any necessary translation to the appropriate character set for a given file or device. Obviously, not all hardware devices are capable of displaying or printing the full complement of Unicode characters. The techniques used to handle the Unicode character set are driver-class dependent.

A full definition of the Unicode character set can be found in The Unicode Standard, Worldwide Character Encoding, Volumes I and II, published by Addison -Wesley.

Index