dL4

Language

Reference Guide

Revision 6.2

Dynamic Concepts Inc.

18-B Journey

Aliso Viejo, CA 92656

www.dynamic.com

Chapter 1 - Introduction

This version (6.2) of the dL4 Language Reference Guide is based on Version 6.2 of the dL4 product and covers all future releases, except for any new enhancements.

This guide is written for experienced BASIC programmers. It is a reference that describes the dL4 programming language. Information concerning statements, functions, and objects supported by the language can be found on these pages. This guide is divided into topical sections which describe the various components of the programming language.

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.
bin.expr	Binary expression	An expression yielding a binary string value.
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.
bin.var	Binary variable	A variable of binary data type. Refer to Chapters 2 and 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 - Data Types

Introduction

In dL4 there are four basic data types and two aggregate data types. Each type has its own rules of operation. The four basic types are Numeric, Character String, Date and Binary. The two aggregate, or derived, types are Array and Structure. The four basic data types are first described briefly below, then in more detail in the following paragraphs. Structures and arrays are described in Chapter 3 of this guide.

- **Numeric** data is made up of integers and floating-point numbers which can be manipulated by arithmetic operators.
- **Character** string data is comprised of Unicode characters. Although string data can contain numeric characters, there can be no direct arithmetic manipulation of string data without first converting the characters to numeric data.
- **Dates** are internal representations of specific points in real-time. Special functions are provided to manipulate and perform arithmetic-like operations on dates. Dates cannot be thought of as string or numeric data, but can be converted to or from character strings for input and display operations.
- **Binary** data is raw information which is not to be interpreted by dL4 as string, numeric, date, or any other type. It is often useful for the developer to manipulate data within a program while being guaranteed that the language does not translate.
- **Structures** aggregate data are programmer-defined sequence of individual named data items of the same or different data types, grouped together to form a single data item. Such a collection is most often used to describe a "record" of information, as in a data file.
- **Arrays** are ordered collections of the same data type where each individual item is referenced by subscripting. Multi-dimensional arrays are represented as arrays of arrays. The developer can also define arrays of structures, or structures containing arrays. The DIM statement reallocates arrays to the exact size specified, preserving only those array elements that remain within the new size of the array. An array can be enlarged to any size with new elements initialized to zero.

Numeric Data

Numeric data can be stored in a variety of internal formats, including Binary Integer, floating point Binary-Coded Decimal (BCD), etc. The particular format used for a variable is called its *precision*. The valid range for all numeric data is governed by the arithmetic library package used by dL4 and is approximately 10^{-507} through 10^{507} with 20-digit precision. All arithmetic calculations are performed to this degree of accuracy, although results can be truncated depending on the precision of variables used.

Numeric values supplied directly in statements are referred to as numeric constants. Very large or small constants can be expressed using floating-point E-notation (scientific notation).

Numeric Precision

Many numeric data precisions are supported, each with a different representation, accuracy and portability. Some precisions are included only for support of existing programs or data files. The following table of numeric precisions defines the storage requirements, significance and the approximate range of representation.

Table of Numeric Precisions

%	Parameters	Bytes	Decimal Digits	Range of values supported
1	16-bit signed integer	2	4+	-32768 to +32767
2	32-bit signed integer	4	9+	-2,147,483,648 to +2,147,483,647
3	3-word BITS Base 10000 floating[1]	6	9-12[2]	±.999999999999 E±63
4	4-word BITS Base 10000 floating	8	16	±.9999999999999999 E±63
5	2-word BITS Base 10000 floating	4	6	±.999999 E±63
6	6-word BITS Base 10000 floating	12	17-20	±.99999999999999999E±63
7	16-bit signed BCD integer	2	4	±7999
8	2-word IRIS BCD floating	4	6	±.999999 E±63
9	3-word IRIS BCD floating	6	10	±.9999999999 E± 63
10	4-word IRIS BCD floating	8	14	±.99999999999999 E±63
11	5-word IRIS BCD floating	10	18	±.999999999999999999 E±63
12	32-bit signed BCD integer	4	8	±79999999
13	2-word IEEE BCD floating	4	6	±.999999 E±63
14	3-word IEEE BCD floating	6	10	±.9999999999 E± 63
15	4-word IEEE BCD floating	8	14	±.99999999999999 E±63
16	5-word IEEE BCD floating	10	18	±.999999999999999999 E±63
17	2-word IEEE floating scaled X 100	4	7[3]	≈ ±99999.99
18	3-word IEEE floating scaled X 100	6	11	≈ ±999999999.99 E±35
19	4-word IEEE floating scaled X 100	8	‡	≈ ±999999999999.99 E±35

Programs declare precisions in either the form *%n* or *n%*. The former is used to specify an exact precision from the above table; the latter maps to a precision within a general type of representation.

The mapping of *n%* to a real precision is based upon the **Option Arithmetic** declaration within each program. Unless specified, the default is **Decimal** (alias **IEEE Decimal**).

Character String Data

A string is defined as a series of Unicode characters. Unicode is a character-encoding standard using a 16-bit character encoding scheme. It includes characters from the world's scripts, as well as technical symbols in common use. The ASCII character set is a sub-set of the UNICODE character set, mirroring the first 128 characters, i.e. ASCII values 0x00 - 0x7F are identical to UNICODE values.

String constants within programs are of two basic kinds: *quoted strings* (string literals) and *mnemonic strings*. String literals are enclosed by the quotation mark character and referred to as string literals. A zero byte is used internally to denote the logical end of a string. A string literal is governed by the following rules:

1. Must begin and end with a quotation mark character (").
2. Any character can be expressed by its octal or hexadecimal Unicode value enclosed within backslashes. For example, carriage return can be given as "\15\" or "\x0f\". Special characters that perform an action on input (commonly backspace, etc.) must be entered in this fashion to be accepted as data.

[1] Base 10000 representation is supported for older BITS and UniBasic files and is not portable across hardware platforms.

[2] The exact number of digits is based upon the decimal point alignment. Each byte-pair (word) holds 4 digits and a decimal point exists only on a word boundary. Therefore a 6-byte (3-word) value can represent 12 integer and no fractional digits, or respectively 8 and 4, 4 and 8 or 0 and 12. When a value has both integer and fractional components, and either component is less than 4-digits, you sacrifice the remaining digits in that word.

[3] Two fractional decimal digits are guaranteed to be accurate, if the value remains within the range given. Rounding errors may occur beyond two digits from the binary-decimal conversion.

3. All printable characters represent themselves except backslash (\) and quotation mark ("). Backslash is represented as "\\" (or "\134\"); quotation mark is represented by two consecutive apostrophes (single quotes) (' ').

Character mnemonic strings are helpful for referring to non-printable Unicode characters in a program. For example, the horizontal tabulation character is 11_8, or "\11\"; this can be more readably expressed with a mnemonic string as 'HT'. A mnemonic string is governed by the following rules:

1. Must begin and end with an apostrophe (single quote) character (').
2. Must contain one or more mnemonic codes separated by a space.
3. Each code can be optionally preceded by a list of one or more numeric constants, separated by commas, to be interpreted as "character parameters". Character parameters are themselves embedded as special characters preceding the main mnemonic code, and applying to it. The exact effect of any parameters is outside the scope of the language and determined by the I/O drivers. A single parameter value is often interpreted as a repetition count, such as '10GH' to output ten forms light horizontal characters.

The **PCHR$** function provides for the runtime construction of character parameters using expressions rather than constants. In addition, the special notation `@X,Y;` can be used as an abbreviation for `Pchr$(X,Y)+'MOVETO'`.

Dates

Dates serve as a standard storage method for date and time data, allowing date manipulation and culture-independent input and output of dates. Numerous functions are provided for the manipulation and conversion of dates. Dates are a distinct type of data different from string or numeric.

Table of Date Precisions

%	Description	Bytes	Minimum value	Maximum value
1	Days	2	2 Jan 1900 00:00:00 GMT	6 Jun 2079 00:00:00 GMT
2	Minutes	4	1 Jan 0001 00:01:00 GMT	16 Feb 8167 04:15:00 GMT
3	Milliseconds	6	1 Jan 0001 00:00:00.001 GMT	3 Aug 8920 05:31:50.655 GMT

Date arithmetic is always performed in terms of seconds, which can be fractional if a date variable has sufficient precision. The precision of date variables is determined exactly like numeric variables, with the *n%* or *%n* specification controlling the currently-selected precision. Unlike numeric precisions however, there is no mapping from *n%* to *%n* controllable by the **Option** statement; e.g., 1% always means %1, etc.

There is no default value assigned to a newly-allocated date variable. An uninitialized date variable uses a special value, indicating not a date. An error is generated if an attempt is made to access an uninitialized date variable. See Appendix B, Error Messages.

Please check the expression section in this manual for legal operations using date variables.

Binary

Binary data behaves the same as string data in some respects, except its contents are not translated. Binary strings give the developer a way to communicate "raw" data to/from a file or device and ensure that no translation or processing of any kind is performed.

Chapter 3 - Variables

Introduction

This chapter describes variable-naming conventions, subscripted variables (arrays), automatic dimensioning, re-dimensioning variables, structures , and structure variables. For a definition and basic discussion of variables, refer to *Introduction to dL4*.

Variable Names

A variable name consists of up to 32 characters which can be letters, digits or the underscore (_). The name cannot begin with a digit. Lower-case letters are equivalent to their upper-case counterparts.

Except for numeric variables, all variable names end with a type identifier character. This suffix is part of the name and must be specified in each reference to that variable within a program. String variables end with $; dates end with #; structures with .; and binary variables end with ?. Arrays end with the type of their base element. Variable names differing only in suffix refer to distinct variables, e.g., MyVar, MyVar$, and MyVar? are all separate variables.

Some examples of variable names include:

```
A
A$
payday#
SoundWave?
DATA_VALUE
PHONE_NUMBER$
```

Up to 4096 different variables can be used within a program. If this limit is exceeded, Error 8 is displayed:

```
Too many variable names
```

Subscripted Variables (Arrays)

References to array, character, and binary variables can include the specification of a subscript to identify a specific, or specific range of, data stored in them. A subscript is given in the form:

```
expr{, expr}...
```

Each *expr* is any numeric expression which, after evaluation, is truncated to an integer. The subscript(s) are then evaluated based upon the type of variable to which they are applied:

- When applied to a character string, up to two subscripts are used; these represent starting and ending character positions inclusive, with positions numbered from 1. If the second subscript is not given, the end of string is assumed.
- When applied to a binary string, up to two subscripts are used; these represent starting and ending byte positions inclusive, with positions numbered from 1. If the second subscript is not given, the end of string is assumed.

- When applied to an array, a single subscript is used; this represents the element number of the array, with elements numbered from 0. If an array is referenced without a subscript, element zero is assumed (except for MAT statements, which process entire arrays).

Multiple subscripts can be concatenated; each is evaluated in turn from left to right. This notation can be used to index into each successive level of a nested aggregate such as an array of strings or an array of arrays (i.e., multi-dimensional arrays). For example:

```
Print A[2][3]
```

prints the 4th element of the third array of A. For historical reasons, multiple subscripts can also be enclosed together with brackets, as in:

```
Print A[2,3]
```

String subscript values of zero are normally illegal and generate errors at runtime. If **OPTION STRING SUBSCRIPTS IRIS** is used, then zero subscripts will be normalized such that a starting subscript of 0 becomes 1 and an ending subscript of 0 is treated as if no ending subscript was specified.

Automatic Dimensioning

New local variables are normally allocated by a program using the **DIM** statement; numeric, date, and some array variables can be implicitly dimensioned by their initial usage, through a feature called Auto-Dimensioning. A simple reference to such a variable causes it to be allocated, if not already allocated. Auto-dimensioning occurs subject to the following rules:

- Auto-dimensioned numeric and date variables take on the current precision (i.e., last precision specified) of the running program-unit.
- Auto-dimensioned array variables take on a dimension of 10 with the current precision. Only arrays of numbers, dates, or further arrays of same can be auto-dimensioned. Therefore, even multi-dimensional arrays can be allocated in this way: `M[3][9] = 123.45`
- If **OPTION AUTO DIM OFF** is used, an error 25 (“variable not dimensioned”) will be generated wherever auto-dimensioning would be required.

Re-Dimensioning Variables

Once a variable is allocated, its *precision* cannot be changed with one exception: an array variable can be re-dimensioned to a different size or a different number of dimensions. A re-dimension remains in effect for the remainder of the program, or until changed again. A change in dimension does *not* affect the precision or value of the base array elements.

In addition, whenever a numeric array specified in a **MAT** statement is followed by subscripts, the subscript values are interpreted as a new dimension size for the selected array:

```
Mat X = Zer[32,5]
```

is identical to:

```
Dim X[32,5]
Mat X = Zer
```

Structures

A *structure* is a dL4 data type that groups several data elements or variables of identical or different data type. Each individual data element is called a structure *member*. Each member must be declared in advance of its use along with its data type.

The group of related members is combined and is collectively identified by a unique name known as the *structure tag name* or simply the *structure name*.

The structure data variable uses the structure name to associate itself with the group of members.

Structure variables provide numerous benefits to the application designer. For example:

- Defining a data record layout
- Operating on a large amount of organized data by referencing a single name
- Organizing related data into a form which simplifies programming and eliminates errors

Structure (.) Variables

Structure variables are indicated by a "." suffix and must be explicitly defined before use. To define a structure template, use one of the following general forms:

DEF STRUCT *struct.name name* {, ... }

DEF STRUCT *struct.name*
  MEMBER *name* {, ... }
  ...
END DEF

struct.name is a unique name tagged to this template. The name can be from one to thirty-two characters in length, and contain letters, digits, and underscores. **DEF STRUCT** does not actually allocate a structure using the supplied name; rather, it informs the compiler to define a unique structure template tagged with this name.

MEMBER *name* is any legal variable name, or precision declaration in the form: %p or p%. *name* can be any type of variable, string, numeric, date, binary or another structure. Any given member can also be an array. The syntax and function of **MEMBER** statements are nearly identical to that of **DIM**.

If the first general form is used, all **MEMBER** *names* must be contained on a single program line. The second general form can be used for readability, or when all of the members cannot be defined on a single line. The two general forms cannot be mixed within a single *struct.name* definition.

The **END DEF** statement defines the end of a structure definition.

Prior to using a structure, you must dimension one or more variables as a specific *struct.name*. The following general form is used to dimension a structure:

DIM *variable***.** { **[***expr* {, ... }**]** } **AS** *struct.name*

*variable***.** is an actual variable in the program which is to be referenced as a structure. The *variable* can include array subscript dimensions, if the *variable***.** is to be an array of structures.

As *struct.name* informs the compiler which compiled structure definition is to be used for *variable***.**

A structure definition itself can contain one or more structures, or arrays of structures. To define a structure which includes a structure, a **MEMBER** is expressed as follows:

MEMBER *name***.** { **[***expr* {, ... }**]** } **AS** *struct.name2*

*name***.** is the name within *struct.name* whose members are defined by the structure definition*struct.name2*. *struct.name2* must be an existing *struct.name* which has been previously defined.

The names of structure members are distinct from any other names outside the structure. For example, Data.Q$ is distinct from Q$ which is distinct from Data1.T.Q$.

The members of a structure are physically contiguous in memory, and are ordered in memory as defined by **DEF STRUCT**. Individual structure members cannot be re-dimensioned.

For syntactical reasons, a separator is needed between a structure variable and a member name; this is also represented by a ".". The separator becomes necessary for:

```
LET B.[3].S$="HELLO"
```

"B." is the variable name, [3] is the third array element and the second "." is the structure/member separator. In fact, a simple reference such as "A.Q$" is really "A..Q$" internally, but the second "." is assumed where it is redundant.

The order in which members of a structure are declared is important because this determines the order in which values are read from a DATA statement, or transferred to/from a file, etc. For example:

```
DEF STRUCT TEST=Q$[20],1%,R,S
DIM A. AS TEST
WRITE #1;A. ! This WRITE is exactly
WRITE #1;A.Q$,A.R,A.S ! like this one
```

Indeed, many older-style statements which operate upon a fixed number of parameters can now be supplied a structure instead. Supplying the structure is interpreted as if you supplied each member as a single variable, separated by comma. As discussed later, SEARCH is another statement where the Key, Record Variable and Status Variable can be passed within a structure.

Structures benefit from all the enhancements to arrays and strings (and follow the same rules), so:

```
DIM B.[10]
LET B.E=5 ! is equivalent to B.[0].E=5
DEF STRUCT TestInfo
      MEMBER StartTime$[25],StopTime$[25]
      MEMBER 4%,TotalSeconds,Seconds[128]
      MEMBER %1,MasterPort,FileClass
      MEMBER %1,NoOfTests,NoOfPorts,Iteration
      MEMBER %1,MinPorts,MaxPorts
      MEMBER %1,StepValue,SampleSize,1%,date#
      MEMBER %1,Timearray[5,5,5]
END DEF
```

Chapter 4 - Intrinsic Functions

Introduction

This chapter lists and briefly describes all dL4 intrinsic (pre-defined) functions.

Intrinsic Functions

All intrinsic (predefined) functions are documented below in alphabetical order.

Predefined Functions

Name	Parameters of Function
ABS(n)	Absolute value.
ASC(s$)	Unicode value of first character in string.
ATN(n)[4]	Arctangent.
BSTR$(n,b)	Returns the a string representation of the value n converted to the specified base b. The base must be 2, 8, or 16. Examples: BStr$(15,2) = "1111" ; BStr$(15,8) = "17" ; BStr$(15,16) = "F"
BVAL(n$,b)	Returns a numeric value for the string representation n$ of a number to the base b. The base must be 2, 8, or 16. Examples: BVal("1010",2) = 10 ; BVal("12",8) = 10 ; BVal("A",16) = 10
CHF(n)	Various numeric parameters of an open channel. The argument must be the channel number (0-99) of an open channel plus a constant which is a multiple of 100 to select mode. Interpretation of each mode is driver-dependent.
CHF(000 + c)	Driver dependent: typically number of records in the file open on channel *c*. This count will include any base record number such as used in Indexed-Contiguous files.
CHF(100 + c)	Driver dependent: typically current record number in the file open on channel *c*.
CHF(200 + c)	Driver dependent: typically current item number or offset in the file open on channel *c*.
CHF(300 + c)	Driver dependent: typically record length in words (16 bit) or bytes (if OPTION set) for the file open on channel *c*.
CHF(400 + c)	Driver dependent: typically file size in bytes for the file open on channel *c*.
CHF(500 + c)	Driver dependent: typically record length in bytes for the file open on channel *c*.
CHF(600 + c)	Driver dependent: typically file header length in bytes for the file open on channel *c*.
CHF(900 + c)	Driver dependent: typically file owner id number, if any, for the file open on channel *c*.
CHF(1000 + c)	Driver dependent: typically file group id number, if any, for the file open on channel *c*.
CHF(1100 + c)	Driver dependent: typically file permissions for the file open on channel *c*.
CHF(1200 + c)	Driver dependent: typically current column number for the file open on channel *c*.
CHF(1300 + c)	Driver dependent: typically current row number for the file open on channel *c*.
CHF(1400 + c)	Driver dependent: typically an operating system defined unique identifier for the file open on channel *c*.
CHF(1500 + c)	Driver dependent: if implemented, returns the number of characters read by the last input operation on the channel *c*. This function is normally used when performing binary input on a device or a network socket.
CHF#(n)	Various date/time parameters of an open channel. The argument must be the channel number (0-99) of an open channel plus a constant which is a multiple of 100 to select mode. Interpretation of each mode driver-dependent.
CHF#(100 + c)	Driver dependent: typically creation date/time for the file open on channel *c*. On systems, such as Unix, that do not support a creation date/time, the oldest available file date attribute will be returned.
CHF#(200 + c)	Driver dependent: typically last access date/time for the file open on channel *c*.
CHF#(300 + c)	Driver dependent: typically last modification date/time for the file open on channel *c*.
CHF$(n)	Various string parameters of an open channel. The argument must be the channel number (0-99) of an open channel plus a constant which is a multiple of 100 to select mode. Interpretation of each mode is driver-dependent.
CHF$(100 + c)	Open mode (“R”, “W”, “E”, and “L”) for the file open on channel *c*.
CHF$(600 + c)	Driver class name for the driver open on channel *c*.
CHF$(700 + c)	Driver name for the driver open on channel *c*.
CHF$(800 + c)	Filename (including relative or absolute path) or equivalent for the file open on channel *c*.
CHF$(900 + c)	Driver dependent: typically file owner name for the file open on channel *c*.
CHF$(1000 + c)	Driver dependent: typically file group name for the file open on channel *c*.
CHF$(1100 + c)	Driver dependent: typically file permissions for the file open on channel *c*.
CHF$(1200 + c)	Driver dependent: typically last input termination character for the file open on channel *c*.
CHF$(1300 + c)	Absolute path for the file open on channel *c*.
CHR(n)	Returns the decimal characteristic of the argument. This is an integer exponent X such that: $10^{X-1} <= n < 10^{X}$
CHR$(n)	Returns the Unicode character whose value is *n*. Note: when converting BITS programs, CHR() must be manually converted to CHR$().
CHR?(n)	Returns a one character binary string where the first character has the value *n*.
COS(n)[4]	Cosine.
DAT#(y,m,d)	Combines the given numeric year, month, and day values into a single date/time value.
DAT#(y,m,d,h,m,s)	As before but includes hour, minute, and second values.
DET(n)	Determinant of the last matrix inverted. See the **MAT INV** statement.
ERM$(n)	Supplies a descriptive text message for error number *n*..
ERR(n)	Various values pertaining to error, ESCAPE and interrupt branching.
ERR(0)	Number of last error.
ERR(1)	Line number of last error.

[4] Angles are interpreted as either radians or degrees depending on setting of the **OPTION ANGLE** statement.

ERR(2)	Line number of last ESCaped statement.
ERR(3)	Line number of last interrupted statement.
ERR(4)	Statement number on line of last error, ESCAPE, or interrupt.
ERR(5)	Statement number on line of last error.
ERR(6)	Statement number on line of last ESCaped statement.
ERR(7)	Statement number on line of last interrupted statement.
ERR(8)	-1
EXP(n)	Exponential, the constant *e* to the power given (e^n)
FRA(n)	Fractional portion. For example: FRA(4.5) yields 0.5.
GMT$(d#)[5]	Converts the given date/time value to an equivalent character string representation, using Greenwich Mean Time (i.e., Universal Time Coordinated) as the time zone.
GMT#(d$)[5]	Converts the given character string to an equivalent date/time value, using Greenwich Mean Time (i.e., Universal Time Coordinated) as the time zone.
HEX?(s$)	Returns a binary string containing the converted contents of *s$*, which is assumed to contain a hexadecimal representation of binary data.
HEX$(b?)	Returns a character string containing the hexadecimal representation of *b?*.
INT(n)	Returns the greatest integer less than or equal to *n*. For example: INT(4.5) yields 4, while INT(-4.5) yields -5.
INT(s$)	Returns the Unicode value of the first character in the string. This is functionally identical to the **ASC** function.
IXR(n)	Decimal radix 10 to the power of *n*. For example: IXR(3) returns 1000.
LBOUND(a,0)	Number of dimensions of array *a*. Trailing brackets ("[ ]") must follow array *a*.
LBOUND(a,n)	Lower subscript bound of dimension *n* of array *a*. Trailing brackets ("[ ]") must follow array *a*.
LCASE$(s$)	Converts all upper-case letters to lower-case.
LEN(s$)	Length of string in characters.
LOG(n)	Logarithm base *e* of *n*. Logarithm in any base B can be achieved using the theorem: $\log_B X = \log_e X / \log_e B$
LTRIM$(s$)	Removes leading white-space characters.
MAN(n)	Decimal mantissa of *n* in base 10.
MONTH(d#)	Numeric month value from *d#*; 1 - 12.
MONTH$(n)[5]	Name of month from n, 1 - 12.
MONTHDAY(d#)	Day number of month from *d#*; 1 - 31.
MSC	Miscellaneous numeric functions
MSC(0)	Current port number.
MSC(1)	Last logical input element accepted.
MSC(2)	-1 or the value of the SPC4 runtime parameter.
MSC(3)	Line number of last **GOSUB** executed. Value is returned and removed from the GOSUB stack.
MSC(4)	-1
MSC(5)	Current column counter on default output channel. When MSC(5) is used in a PRINT statement, the initial value of the column counter is returned.
MSC(6)	Returns current unused variable space as a large integer constant (INT_MAX), typically 2^{31}-1.
MSC(7)	Current user and/or group ID number.
MSC(8)	-1
MSC(9)	-1
MSC(10)	-1
MSC(11)	-1
MSC(12)	-1
MSC(13)	-1
MSC(14)	-1
MSC(15)	-1
MSC(16)	-1
MSC(17)	-1
MSC(18)	The constant π (3.141592653589793).
MSC(19)	The constant *e* (2.718281828459045).
MSC(20)	Maximum channels per user; returns 100.
MSC(21)	-1
MSC(22)	-1
MSC(23)	-1
MSC(24)	-1
MSC(25)	-1
MSC(26)	-1
MSC(27)	-1

[5] Exact character representation of date components depends on setting of the **OPTION DATE FORMAT** statement.

MSC(28)	-1
MSC(29)	-1
MSC(30)	Current line number.
MSC(31)	Current statement number on line..
MSC(33)	Number of columns on the default I/O channel.
MSC(34)	Number of rows on the default I/O channel.
MSC(35)	Input buffer size in characters.
MSC(36)	-1
MSC(37)	Maximum number of ports supported.
MSC(38)	Total number of ports currently in-use.
MSC(39)	Current **OPTION DATE FORMAT** setting; 0 = Standard, 1 = Native.
MSC(40)	Number of columns for Dynamic Windows display device.
MSC(41)	Number of rows for Dynamic Windows display device.
MSC(42)	Window nesting level in Dynamic Windows.
MSC(43)	Current row counter on default output channel. When MSC(43) is used in a PRINT statement, the initial value of the row counter is returned.
MSC(44)	Dynamic Window system state. One if the window system is active, zero if it is not active.
MSC(45)	Element number of the GUI element ('WCxxxx') last read by an INPUT or READ statement
MSC(46)	Original line number of last error. If an error occurs in a subprogram or procedure and the error is not handled within that subprogram or procedure, the error will be reported to the caller and ERR(1) and SPC(10) will report the line number at which the subprogram or procedure was invoked. MSC(46) reports the line number within the original subprogram or procedure.
MSC$(n)	Miscellaneous string functions.
MSC$(-3)	dL4 revision string.
MSC$(-2)	dL4 revision formatted as RRLLBBSS.
MSC$(-1)	"" or the value of the SPC4 runtime parameter formatted as "RRLLBBSS".
MSC$(0)	System date and time in international format: `dd mon year  hh:mm:ss`
MSC$(1)	Current working directory path
MSC$(2)	Text description of last error.
MSC$(3)	System date and time in US format: `mon dd, year  hh:mm:ss`
MSC$(4)	Filename of the current program.
MSC$(5)	Filename of the parent program, when the current program was invoked by **SWAP**.
MSC$(6)	Return the current LIBSTRING value.
MSC$(7)	Return hot-key character used to invoke current swap program or " ".
MSC$(8)	Return operating system dependent directory separator string ("/" for Unix and "\" for Windows).
MSC$(9)	Absolute path of the directory containing the current program.
MSC$(264)	""
NOT(n)	Logical NOT. Returns 1 if *n* is zero, or zero if *n* is not zero.
NOT(s$)	String NOT. Returns 1 if *s$* is null (length 0), or zero if *s$* is not null.
PCHR$(n{,...})	Convert numeric or string value(s) to "character parameters", suitable for prefacing certain command characters.
POS(s$,op t${,s{,o}})	First position in *s$* where *op t$* is true. *s* is an optional position step value; *o* is an optional occurrence value (default 1). *op* can be any relational operator `<  <=  >  >=  =  <>` or a set operator **IS** or **EXCEPT**. The **IS** operator searches for the first character in *s$* that is in *t*$. The **EXCEPT** operator searches for the first character in *s$* that is not in *t$*. *s* can be negative to indicate backwards searching from the end of string.
REP$(s$,n)	Repeats *s$* *n* times.
RND(n)	A pseudo-random number X is generated in the range 0 < X < *n*.
ROUND(n,d)	Rounds *n* to *d* decimal places.
RTRIM$(s$)	Removes trailing white-space characters.
SGN(n)	Signum function. Returns the sign of *n*; -1 if *n* < 0, 0 if *n* = 0, or 1 if *n* > 0.
SIN(n)[4]	Sine.
SPC(n)	Special numeric functions.
SPC(0)	CPU time used in tenth-seconds.
SPC(1)	Connect time used in minutes.
SPC(2)	Hours since the system base date. This value is computed assuming all months have 31 days.
SPC(3)	Current tenth-second of the hour.
SPC(4)	-1 or the value of the SPC4 runtime parameter.
SPC(5)	Current user and/or group ID number.
SPC(6)	Current port number.
SPC(7)	User-defined.
SPC(8)	Last error number.
SPC(9)	Current line number.
SPC(10)	Line number of last error.

SPC(11)	Current directory name represented as a number, if possible.
SPC(12)	Directory of the current program represented as a number, if possible.
SPC(14)	Line number of last **GOSUB**. Value is returned and removed from the stack.
SPC(15)	Return and clear the last error number.
SPC(16)	Line number of last **GOSUB**. Value is returned and left on the stack.
SPC(17)	Length of last character-limited input.
SPC(18)	Constant base year; always returns 1980.
SPC(19)	The system license id in the form of a 32-bit unsigned integer.
SPC(20)	Current base year.
SPC(21)	Input buffer length.
SPC(22)	Returns available program space in words: a large integer constant (INT_MAX), typically 2^31-1.
SPC(23)	Current library directory from last **LIB** statement. -1 is returned if no current library or if it cannot be represented as a number.
SPC(24)	Line number of last **END**, **STOP** or **SUSPEND** statement.
SPC(264)	-1 or the value of the SPC264 runtime parameter.
SPC(272)	-1 or the value of the SPC272 runtime parameter.
SPC(n)	Return the numeric value of the environment variable “SPCn”. Environment variables do not override the standard SPC values and applications should use values of N greater than 99 to avoid possible conflicts.
SQR(n)	Square root.
STR$(n)	Convert the numeric value *n* into a character string. Unlike direct assignment, no white-space is included.
TAN(n)[4]	Tangent.
TIM(n)	Returns miscellaneous time-related numeric values.
TIM(0)	CPU time used in seconds.
TIM(1)	Connect time used in minutes.
TIM(2)	Hours since base date.
TIM(3)	Current tenth-second of the hour.
TIM(4)	Current date in the form: MMDDYY where MM is the month (1-12), DD is the day of the month (01-31) and YY is the year such as 89.
TIM(5)	Current date in the form YYDDD where DDD is the day of the year (1-366).
TIM(6)	Number of days since 0 January 1968.
TIM(7)	Current day of week (0=Sunday, 6=Saturday).
TIM(8)	Current year in the form YY, such as 89.
TIM(9)	Current month; 1=January, 12=December.
TIM(10)	Current day of the month; 1-31.
TIM(11)	Current hour of the day; 0-23.
TIM(12)	Current minute of the hour; 0-59.
TIM(13)	Current second of the minute; 0-59.9.
TIM(14)	Current date in the form: MMDDYYYY where MM is the month (1-12), DD is the day of the month (01-31) and YYYY is the year, such as 2001.
TIM(15)	Current date in the form YYYYDDD where DDD is the day of the year (1-366) and YYYY is the year, such as 2001.
TIM(16)	Current year in the form YYYY, such as 2001.
TIM#(n)	Returns miscellaneous date/time values.
TIM#(0)	Current real-time.
TIMEZONE(d#)	Local time-zone offset from GMT in seconds in effect as of *d#*.
TRUNCATE(n,d0	Truncates *n* to *d* decimal places.
UBOUND(a,0)	Number of dimensions of array *a*. Trailing brackets ("[ ]") must follow array *a*.
UBOUND(a,n)	Upper subscript bound of dimension *n* of array *a*. Trailing brackets ("[ ]") must follow array *a*.
UCASE$(s$)	Converts all lower-case letters to upper-case.
VAL(s$)	Convert the string value *s$* to a number.
WEEKDAY(d#)	Day of week number from *d#*; 1 = Sunday, 7 = Saturday.
WEEKDAY$(n)[5]	Day of week name for day *n*; 1 = Sunday, 7 = Saturday.
YEAR(d#)	Year number from *d#*.
YEARDAY(d#)	Day of year number from *d#*; 1 - 366.

Chapter 5 - Expressions

Introduction

This chapter describes dL4 operator precedence, by which dL4 evaluates expressions, and the operators themselves:

- Unary
- Arithmetic
- Concatenation
- Assignment
- Relational
- Boolean
- String Operator USING
- String Operator TO

In addition, Boolean Expressions, Channel Expressions, and String Assignment are described.

Operator Precedence

The operations within an expression are evaluated according to the precedence shown in the Operator Precedence Table below. Operators on the same level are evaluated from left to right in the expression. Parentheses can be used, however, to override this hierarchy. Predefined functions and procedures are evaluated before any operators are executed.

Operator Precedence Table

Operator(s)	Parameters	Evaluation Order
`+ -`	Unary + - (negation)	Right-to-Left
`^`	Exponentiation	Left-to-Right
`* / MOD`	Multiply, Divide, Modulo	Left-to-Right
`+ -`	Add, Subtract	Left-to-Right
`TO`	String searching: all characters of target string are significant	Left-to-Right
`USING`	Numeric formatting	Left-to-Right
`, +`	String concatenation	Left-to-Right
`< <= > >= <>`	Comparison	Left-to-Right
`AND`	Logical AND	Left-to-Right
`OR`	Logical OR	Left-to-Right
`:=`	Assignment	Right-to-Left

For example:

Expression	Evaluates as	Result
`3+4*5`	3+(4*5)	23
`(3+4)*5`	(3+4)*5	35
`14/7*10/2`	((14/7)*10)/2	10
`3^2*4`	(3^2)*4	36
`"3"+"B"`	"3" concatenate "B"	"3B"

Operators

The dL4 operators are described in the following paragraphs.

Unary Operators + -

The unary operators **(+ -)** are used to change the sign of an *argument*. They are evaluated from right-to-left and have the highest precedence. The **+** is a non-operation, and the **-** changes a negative value positive or a positive value negative.

Arithmetic Operators ^ * / % + -

Arithmetic operators follow unary operators in the precedence of an expression. The highest precedence is given to **(^)** invoking exponentiation, which is essentially repeated multiplication. A value y^x is read, "take the value *y* raised to the power *x*." In simpler terms, multiply *y* by itself *x* times. Exponentiation has the highest precedence of all of the arithmetic operators and is evaluated Left-to-Right.

Next, **(* / MOD)** which selects multiplication, division and modulo. The MOD operator returns the remainder of a division of the two operands. This is calculated as (x - INT(x/y)*y). 10%2 yields 0, 10%3 yields 1, etc. These operators are evaluated from left-to-right after exponentiation.

Finally, **(+ -)** addition and subtraction are the lowest precedence of the arithmetic operators. These are also evaluated from Left-to-Right.

Concatenation Operators + ,

Concatenation operators are used to link string expressions together. The result of concatenating two string expressions is the combination of both expressions into a single string expression. Each concatenated string is appended to the end of the result of the current expression. The concatenation of "This" +" That" results in the string: "This That", etc.

The **(+)** concatenation operator can be used in any expression involving strings; the **(,)** concatenation operator is equivalent but can only be used in **LET** and **IF** statements.

Assignment Operator: Colon Equal

The assignment operator, Colon Equal, with ":=" is different from "=" which is compare-for-equality. Compare-for-equality indicates that dL4 is attempting to determine if the values are equal. The word "assignment" comes from the way this operator assigns values to the variables. The following two statements are considered equivalent:

LET A = B
LET A:= B

But the next two statements are not considered equivalent:

LET A:= B:=C:=1
LET A=B=C=1

Regarding ":=", see the LET statement.

Relational Operators = <> > >= < <=

All relational operators are evaluated on an equal precedence and all group left-to-right. Their result is said to be true (one) if the relation is true, and false (zero) if the relation is false. Relational operators can be used in **IF** statements or as part of a boolean expression. The format is:

expression relation expression

where *relation* can be any of the following:

=	Equal
<>	Not Equal
>	Greater Than
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To

String data are compared using the Unicode value of each character, one character at a time. If the strings are not subscripted to control their length, then they are evaluated using the current logical length (from any optional starting position up to the first zero-byte terminator). Strings are equal only when they are exactly equal in length and contents. When a shorter string is compared to a longer one, and they are equal up to the length of the shorter string, the shorter string is said to be *less* than the longer string. If, during comparison, two characters do not match, the left string is said to be less than the right string if the Unicode value of the left character is less than the Unicode value of the right character.

Boolean Operators AND OR NOT

The Boolean operators are described in "Boolean Expressions and Operators", Chapter 5 of this guide.

String Operator USING

The **USING** operator groups from left-to-right and results in a formatted string result from a numeric *expression*. The format of this operator is:

numeric expression **USING** string expression.

The *numeric expression* is evaluated first. Next the *string expression* is evaluated and used to 'format' the *numeric expression* into a string result.

The format string is scanned, and any characters which are not *field descriptors* are copied to the destination until a *format* field is seen. Characters which can begin a format field are **$ # + - and ***. Other field descriptors are treated as text and are copied until a starting character is seen. After formatting a result, the remaining characters in the *format* string (up to the start of another format field) are copied to the destination.

Each *format* field is made up of certain characters describing the formatting to be done. These are called *field descriptors*. Numeric items are formatted according to the rules governing each descriptor. If an item cannot be formatted according to the field given, the field is output filled with asterisks (*). This generally occurs when a number is too large to be expressed with the number of digits available in the field.

Field Descriptors

Field descriptors for a *format* field fall into five categories:

1. Leading characters
2. Floating characters
3. Numeric Characters
4. Commas
5. Decimal Points

Leading Characters

A field can begin with one or two leading characters. The available leading characters are:

LEADING	**OUTPUT**
$	$ always
+	+ if item >= 0; - if item < 0
-	space if item >= 0; - if item < 0

The **$** can be combined with either **+** or **-** for a two-character leading group. Note that all three leading characters are also valid as floating characters. A group of two or more identical characters is considered a floating character designation.

Floating Characters

A field can contain groups of floating characters. This character "floats" and is eventually executed just before the first digit output. The available floating characters are the same as the leading characters ($, +, -) and are processed the same.

Numeric formatting outputs a sign (+ or -) only if one is specified within the format field. If none is given in the format, all items are output as positive, regardless of sign.

One extra floating character should be given in the format field in addition to the number given for the highest digit count desired. One space is required for the execution of the floating character itself. The remaining floating characters can be occupied by digits. For example, the format string "$$$$" can accommodate no number larger than 999, because one space is required for the dollar sign itself.

Numeric Characters

A field can contain groups of numeric characters. The available numeric characters are:

#	Digit or space if leading zero
&	Digit, leading zeroes not suppressed
*	Digit or "*" if leading zero

Every numeric character given in a format field can contain a digit. For example:

```
Format:     ####  &&&&  ***#  ***#
            17    0017  **17  **17
            247   0247  *247  *247
            6140  6140  6140  6140
            0     0000  ***0  ***0
```

Commas

A field can contain one or more commas which are output when significant. For example:

```
Format:     ##,###      #,###,###   &,&&&,&&&
               768            768   0,000,768
             2,147          2,147   0,002,147
            ******      1,034,957   1,034,957
```

The use of commas and decimal points in format masks is controlled by the **OPTION USING DECIMAL** and **OPTION NUMERIC FORMAT** statements.

OPTION USING DECIMAL IS COMMA effectively interchanges the meaning of periods and commas in format masks, not which character is output.

OPTION NUMERIC FORMAT NATIVE controls the output character.

Decimal Points

A field can contain a period for the fractional portion of an item. The fractional portion then follows and is truncated to the number of digits specified. Only numeric descriptors (#and*) can follow the period, and all are processed as a character. For example:

```
Format:     ##.###      ##.#  ##.&& **.**
            74.000      74.0  74.00 74.00
            16.408      16.4  16.40 16.40
```

The use of commas and decimal points in format masks is controlled by the **OPTION USING DECIMAL** and **OPTION NUMERIC FORMAT** statements.

OPTION USING DECIMAL IS COMMA effectively interchanges the meaning of periods and commas in format masks, not which character is output.

OPTION NUMERIC FORMAT NATIVE controls the output character.

String Operator TO

The **TO** operator is evaluated from left-to-right and is used to specify part of a string expression. The general form is:

string expression **TO** *string expression*

The *string expression* on the left is evaluated first and referred to as the *source*. Next the right *string expression* is evaluated and is referred to as the *pattern*. The resulting *string expression* is generated by copying all characters from the *source* up to and including the *pattern* string. If the *pattern* is not found within the *source*, then all characters of the *source* become the resulting *string expression*.

For example, if you have a large block of text and wish to copy the first sentence, you might use this operator to find the result of:

```
S$  TO ".  "            ! Locate first period followed by 2 spaces
```

Boolean Operators

The Boolean operators are **AND** and **OR**. Closely associated is the function **NOT**. They are used to convert normal expressions into Boolean operations. A Boolean operation yields a True/False condition.

- **NOT** reverses the condition; True becomes False and False becomes True.
- **AND** is used to compare the result of two expressions, yielding True only if both expressions are true.
- **OR** is used to compare the result of two expressions, yielding True if either of the expressions are true.

AND, **OR**, and **NOT** are processed left-to-right, and their precedence order is **NOT**, **AND**, **OR**. You may use parentheses to change precedence order.

The parameters of a boolean operator are evaluated as a boolean expression.

Boolean Expression

A boolean expression, or bool.expr, is a context dependent interpretation of an expression which is used by boolean operators, or in **IF**, **DO WHILE**, **DO UNTIL**, and **WHILE** statements. The interpretation of the expression produces a boolean, i.e. **TRUE/FALSE**, result according to the following rules:

Data Type	**TRUE (1)**	**FALSE(0)**
Numeric	non-zero	zero
String	non-zero length	zero length
Date	is a date	not a date
Binary	Not allowed	Not allowed

The following two sample programs illustrate usage of boolean expressions:

```
Rem this is a sample program
a = 5
While a + 5
   Print a
   a = a - 1
Wend
Rem end of sample program
```

```
Rem this is another sample program
a = 0
While a + 5
   Print a;
   If a > 0
        Print "is a positive value"
   Else If a < 0
        Print "is a negative value"
   Else
       Print "is a zero value"
   Endif
   a = a - 1
Wend
Rem end of sample program
```

Channel Expressions

Most Input/Output (I/O) statements in dL4 use a channel expression. A channel expression consists of a channel number followed by three optional numeric parameters. The three optional numeric parameters commonly indicate a record number, a field position, and a timeout value. However, it is possible for these parameters to indicate something else as the meaning of these parameters are driver-class dependent.

The generic format and specific examples of the channel expression follow:

```
#chan.no, {num.expr1{, num.expr2{, num.expr3 }}} ;
```

```
#9,5,2,1;
```

```
#9;
```

```
#9,record,byte_displ;
```

A channel expression begins with a #, and ends in a semicolon (;). The channel number follows "#", and must be in the range 0 to 99. Many statements will also accept channel number -3 or -4 which select the current standard input or standard output channels rescpectively. The final semicolon (;) indicates the end of a channel expression.

The parameters must be specified in its proper order. In other words, both the first and second parameters must also be specified in order to specify the third parameter. A value of negative one is used as a default parameter value. Thus, an expression requiring only the last parameter can be written as:

```
#9, -1, -1, 35;
```

Rules Governing String Processing

During the use of character strings within a program, the following rules are applied to operations:

- A string can contain any of the Unicode values from 0 to 65534. 65535 is explicitly not a Unicode character.

- A zero character is used to terminate any string segment.
- String variables can be subscripted to select a starting and ending character position within a string. A single subscript selects a starting point only. All strings terminate upon the occurrence of a zero terminator, the second subscript, or the physical dimension of the string.
- A *full string* is defined to be any reference to a string variable in which a single or no subscripts are supplied.
- A *sub-string* is defined to be any reference to a string variable using 2 *subscripts*.

String Assignment

When assigning data to a *full string*, the following rules are applied:

- The source is truncated to the size of the supplied destination.
- A zero terminator is inserted in the destination if the source is shorter than the destination.
- A zero terminator can be placed within a string by specifying a single subscript in the form: *S$[x] = ""*.

When you are assigning data to a *sub-string*, behavior of the sub-string is dependent on the setting of the **OPTION STRINGS** statement. If **OPTION STRINGS STANDARD** is set, the following rules apply:

- When the source is shorter than the destination, the remaining characters within the subscripts are deleted. Characters following the subscripted portion are shifted down to immediately follow the shorter source.
- When a zero terminator is overlaid in the destination, it is pushed forward to the first character position following the length of the source copied. This can cause a zero to be placed into the first character position beyond the second subscript if the source exactly fills or is larger than the destination.

If **OPTION STRINGS RAW** is set, the following rules apply:

- When the source is shorter than the destination, the second subscript is ignored. Only the number of characters supplied in the source are copied to the destination.
- When a zero terminator is overlaid in the destination, it is pushed forward to the first character position following the length of the source copied if and only if the source string does not completely fill the destination. No characters outside the supplied subscripts are altered.

Other special string functions are available to the application:

1. A string can be completely filled with a single character (or group of characters) except zero-byte terminators using the form:

   ```
   A$=" ",A$  ! to space fill A$
   ```

2. Characters beyond the zero terminator can be operated upon by specifying a starting subscript beyond the zero. Use the **LEN** function to determine the length of any string.
3. Numeric data can be converted to string and vice-versa using the **LET** Statement, or the functions **STR** and **VAL**.

Chapter 6 - Mnemonics

Introduction

This chapter describes dL4 mnemonics, listing:

- CRT mnemonics
- Graphic User Interface (GUI) mnemonics
- ASCII character mnemonic values
- General punctuation mnemonic values
- CJK symbols and punctuation
- Unclassified mnemonics
- Mnemonics for keyboard and auxiliary port
- Mnemonics to clear and reset the terminal
- Mnemonics applied to the cursor position
- Mnemonics to control attributes
- Mnemonics to control color
- Mnemonics to transmit data
- Miscellaneous mnemonics
- Special mnemonics for I/O control
- Table of extended graphics octal codes

Mnemonics

A mnemonic provides a way to specify special character values via a meaningful name instead of the exact octal or hexadecimal values. They are commonly used to control screen or printer attributes. The usage of mnemonics provides program portability.

Mnemonics can take one or more parameters as numeric integers preceding the mnemonic name. Most mnemonics take an optional parameter which signify a repeat count.

Many mnemonics take a 24-bit RGB color value as a parameter. The parameter value is formed as follows: RED * 65536 + GREEN * 256 + BLUE where RED, GREEN, and BLUE are color intensity values between 0 and 255. When used in dL4 for Windows or with dL4Term, the color value also has standard color values expressed as negative numbers. The standard values are:

-1	Dialog text color
-2	Dialog background color
-3	Window text color
-4	Window background color

-5 Highlighted text color

-6 Highlighted text background color

The support of a mnemonic is driver-class dependent. In the case of the terminal translation driver, it is also terminal description file dependent.

The following are some examples of mnemonics usage.

```
PRINT 'CS';        ! Clear screen
PRINT 'CS 10ML';   ! Clear and move left 10 positions.
PRINT @5,5;'CL';   ! Position to column 5, row 5 and clear line
PRINT @10,L;       ! Position cursor to column 10, row L.
```

Mnemonic Values

Mnemonics for Keyboard and Auxiliary Port

Mnemonic	Explanation
AE	Enable the Auxiliary port on the terminal. This mnemonic enables the Auxiliary Printer port until the **AD** mnemonic is sent.
AD	Disable the Auxiliary port on the back of the terminal.
BA	Begin Transparent output to Auxiliary printer port. Enabling Transparent output causes all output characters (and input echoing) to be directed to the Auxiliary Port of the terminal until the mnemonic EA is sent.
BO	Begin non-Transparent output to Auxiliary printer port. This mnemonic operates similarly to the 'BA' mnemonic except that data is transmitted to both the Auxiliary port and the screen until an EO mnemonic is sent.
CONTINUEAUX	Continue output to the auxiliary printer. This mnemonic is used with the SUSPENDAUX mnemonic to intersperse auxiliary output with normal output while maintaining the continuity of the auxiliary output.
EA	End Transparent output to Auxiliary port.
EO	End non-Transparent output to Auxiliary port.
EF	End Function Key Definition. This code terminates all characters being sent to down-load function keys using the mnemonics P1 through P8.
LK	Lock Keyboard. The keyboard is locked and no further characters are accepted from the terminal. All keys are locked out until the UK mnemonic is sent or until the terminal is reset.
P1	Begin Programming downloadable function key 1. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P2	Begin Programming downloadable function key 2. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P3	Begin Programming downloadable function key 3. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P4	Begin Programming downloadable function key 4. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P5	Begin Programming downloadable function key 5. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P6	Begin Programming downloadable function key 6. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P7	Begin Programming downloadable function key 7. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
P8	Begin Programming downloadable function key 8. All further characters are sent to the terminal's function key until the mnemonic EF is sent.
PGMFN	Program the function key specified by the numeric parameter 1 with the string specified by the string parameter 2. Example: Print PChr$(1,”Help\15\”);’PGMFN’

PGMHELPFN	Program the function key specified by the numeric parameter 1 with the string specified by the string parameter 2. When typed, the function key will send both the string and the action string of the current selected GUI ('WCxxxx') element. This mnemonic is normally used to support a context dependent help key in a GUI application. Example: Print PChr$(1,"Help\15\");'PGMHELPFN'
RF	Reset Function keys to their default values.
SUSPENDAUX	Suspend output to the auxiliary printer. This mnemonic is used with the CONTINUEAUX mnemonic to intersperse auxiliary output with normal output while maintaining the continuity of the auxiliary output.
UK	UnLock Keyboard. Characters and functions can now be entered from the keyboard.

Mnemonics to Clear and Reset the Terminal

Mnemonic	Explanation
CE	Clear from cursor to end of screen. All unprotected characters from the current cursor position up to the end of the screen are cleared.
CL	Clear from cursor to end of line. All unprotected characters from the current cursor up to the end of the line are cleared. Inside windows, CL/CE skips over protected fields.
CS	Clear the entire screen. All characters both protected and unprotected are cleared.
CT	Clear all TAB Stops set by the ST mnemonic.
CU	Clear all unprotected characters on the screen. This mnemonic is used to clear data from the screen while leaving any protected mask intact. Also, performs a Move Home (MH), if window tracking is on. The cursor is moved to position 0,0 of the current window.
ES	End Write Status Line. Characters output and echoed are no longer displayed in the status line of the terminal (See also: WS).
K0	CURSOR Set no cursor to be displayed on the terminal.
K1	CURSOR Set Blinking Block.
K2	CURSOR Set Steady Block.
K3	CURSOR Set Blinking Underline.
K4	CURSOR Set Steady Underline
NR	Narrow Character Display. Set wide display mode (commonly 132 columns) and display further output and echoed characters in narrow format.
NV	Normal video. Display reverse video as dark on lighted background.
RS	Reset Terminal. Send the commands to reset the terminal to its power-up parameters. This normally resets protocols, translations, function keys and clears the screen.
RV	Reverse video. Display reverse video as lighted characters on dark background.
SF	Status Line OFF. Turn off the optional status line at the bottom (or top) of the screen.
SO	Status Line ON. Turn on the optional status line at the bottom (or top) of the screen.
WD	Wide Character Display. Set the terminal into normal mode (commonly 80 columns) and display further output and echoed characters in normal format.
WS	Write Status Line. All further characters echoed or output are displayed in the terminal's status line until the ES mnemonic is sent.
XX	Initialize Terminal. This mnemonic can define a series of functions such as Clear screen, Clear Memory, Clear Status Line, etc. required to reset the terminal; See also: RS.

Mnemonics Applied to the Cursor Position

Mnemonic	Explanation
BK	Cursor Back. A carriage return without line-feed is sent to the screen moving the cursor to the beginning of the current line.
ALIGN	Move the cursor to the next character column which is a multiple of the parameter. For example, if printed at column 20, the mnemonic string ‘15ALIGN’ will move the cursor to column 30. This mnemonic is used by the comma operator of the PRINT statement.
CR	Perform a new-line operation. A carriage return and a line-feed are sent to the terminal. If the cursor is at the bottom of the window, the screen scrolls up one line. Some terminals do not scroll if the screen window contains protected fields. Hard-coded sequences of "\15\\12\" or 'CRLF' should be replaced with "\15\" or 'CR'.
DC	Delete Character. The character at the cursor is deleted and all remaining characters on the line are shifted left.
DL	Delete Line. The line containing the current cursor is deleted from the window and all remaining lines are moved up.
FF	Form Feed. Scroll to the next page. This mnemonic is used primarily for printers.
IC	Insert Character. A space is added at the current cursor position by shifting the character under the cursor (and all remaining characters on the line) right one position.
IL	Insert Line. A new line is added by shifting the line containing the cursor (and all following lines) down one line. Lines can disappear off the end of a window. The universal new line code is \15\. Inside windows, IL/DL moves to the beginning of the line.
LF	Perform a Line-Feed. This, in effect, is identical to a MD mnemonic. The cursor is moved down to the next line while staying at the same column.
MD	Move Down. The cursor is moved down to the next line while staying at the same column. Some terminals scroll if you are already on the last line of the screen. Inside windows, MD wraps on the last line if the window has WRAP style; otherwise it is non-operative.
MH	Move Home. The cursor is moved to position 0,0 of the current window.
ML	Move Left. The cursor is moved Left one character.
MOVETO	Move the cursor to the grid position specified by the parameters. If a single numeric parameter is given (‘10MOVETO’), then the cursor will be moved to the specified grid column on the current row. If two parameters are used (’10,20MOVETO’), then the cursor will set the cursor grid column to the first parameter (10) and the cursor grid row to the second parameter (20).
MP	Use Memory Pointer instead of cursor for next positioning command.
MR	Move Right. The cursor is moved Right one character. Inside windows, MR wraps on the last position if the window has WRAP style; otherwise it is non-operative.
MU	Move Up. The cursor is moved up to the previous line while staying at the same column.
TB	Tab Backward. The cursor is moved to the start of the previous TAB Stop as defined with the ST mnemonic.
TF	Tab Forward. The cursor is moved to the start of the next TAB Stop as defined with the ST mnemonic.
VT	Vertical Tab. Move the cursor Down in the window to the next preset Vertical Tab Stop. This mnemonic is normally used for printers using the supplied printer filter or when you direct data through the Auxiliary printer port.

Mnemonics to Control Attributes

Mnemonic	Explanation
BB	Begin Blink Mode. All further output and echoed characters blink until the EB mnemonic is sent.
BBOLD	Begin bold mode.
BC	Begin compressed mode.
BD	Begin Dimmed Intensity Mode. All further output and echoed characters are displayed in dimmed (half) intensity until the ED mnemonic is sent. Some terminals treat dimmed intensity data as protectable and use of the FM mnemonic causes dimmed fields to become protected. Inside windows, BP/EP implies dimmed and protected.
BG	Begin Graphics Mode. This is a legacy mnemonic that normally has no effect.
BI	Begin Italic mode.
BP	Begin Protectable Field. Further characters echoed or sent to the terminal are flagged as protectable and are usually displayed in half-intensity. Similarly, half-intensity data printed using the 'BD' mnemonic can also be protectable, depending upon your terminal. After you have painted your protectable fields on the terminal, you must issue the FM mnemonic to format and write-protect your protected field. Inside windows, BP does not imply FX.
BR	Begin Reversed Video . All further output and echoed characters are displayed in reverse video format. On most terminals, the background becomes lit and the characters are shown as black. Color monitors and other terminals can permit control of the display.
BSO	Begin strike-out mode.
BSUB	Begin subscript mode.
BSUP	Begin superscript mode.
BU	Begin Underline Mode. All further output and echoed characters are underlined until the EU mnemonic is sent.
BX	Begin Expanded Print. All further output and echoed characters are displayed in your pre-defined choice of double-high, double-wide or both.
CPI	Set the fontsize to produce the number of characters per inch specified by the numeric parameter ('10 CPI'). The mnemonic may also be used with two parameters, n and d, to set the number of characters per inch to the fraction n/d ('50 3 CPI' selects 16.66.. characters per inch).
EB	End Blink Mode. Characters output and echoed no longer blink.
EBOLD	End bold mode.
EC	End compressed mode.
ED	End Dimmed Mode. Characters output and echoed are no longer be in half-intensity.
EG	End Graphics Mode. This is a legacy mnemonic that normally has no effect.
EI	End italic mode.
EP	End Protectable Field. All further characters transmitted are not to be considered part of a protected field. Inside windows, EP does not imply FM.
ER	End Reversed Video. Characters output and echoed are no longer in reverse video format.
ESO	End strike-out mode.

ESUB	End subscript mode.
ESUP	End superscript mode.
EU	End Underline Mode. Characters output and echoed are no longer underlined.
EX	End Expanded Print. Characters output or echoed are no longer in expanded format.
FM	Enter Format Mode. Write protect is set on all characters previously sent using the BP mnemonic. The protectable fields are now protected preventing any overwriting of protected data. On some terminals, dimmed characters (BD) can also become protected.
FONTCELL	Set the font size to fit into a character cell whose height is the parameter times the current coordinate grid row height. The font width is set by the operating system to the preferred width for the specified font height and typeface. This mnemonic is used to precisely control the line height.
FONTFACE	Set the font typeface to the name supplied by the string parameter. For example, the statement 'PRINT PChr$("Helvetica");'FONTFACE'' would select Helvetica or an operating system chosen substitute as the current typeface.
FONTSIZE	Set the font size to the parameter times the current coordinate grid row height. The font width is set by the operating system to the preferred width for the specified font height and typeface.
FX	Exit Format Mode. All previously write-protected characters are now returned to their protectable state. Fields can be overwritten or changed until another FM is issued. Some terminals cannot overwrite protected characters once formatted by the FM mnemonic. A clear-screen (CS) is required to reset these fields.
LPI	Set font size to produce the number of lines per inch specified by the numeric parameter ('6 LPI').
RESETFONT	Reset font to default font and size.
ST	Set a TAB Stop at the cursor. To be used with the TF and TB mnemonics for presetting TAB stops on the screen.

Mnemonics to Control Color

Mnemonic	Explanation
RE	Color RED. All further output and echoed characters are displayed in Red.
GR	Color GREEN. All further output and echoed characters are displayed in Green.
YE	Color YELLOW. All further output and echoed characters are displayed in Yellow.
BL	Color BLUE. All further output and echoed characters are displayed in Blue.
BLACK	Color Black. All further output and echoed characters are displayed in Black.
MA	Color Magenta. All further output and echoed characters are displayed in Magenta.
CY	Color CYAN. All further output and echoed characters are displayed in Cyan.
WH	Color WHITE. All further output and echoed characters are displayed in White.
BACKCOLOR	Set background color to the RGB parameter. The parameter is a 24-bit integer RGB value in which the most significant 8-bits specify the red component, the middle 8-bits specify the green component, and the least significant 8-bits specify the blue component.
FONTCOLOR	Set text color to the RGB parameter. The parameter is a 24-bit integer RGB value in which the most significant 8-bits specify the red component, the middle 8-bits

	specify the green component, and the least significant 8-bits specify the blue component.
DEFAULTCOLOR	Set the default colors for the current session from the current text and background colors.
INVERT	Invert colors within a specified area. The mnemonic has 4 formats accepting 0, 1, 2, and 4 numeric parameters: 'INVERT' – invert colors from the cursor to the end of the line. 'n INVERT' – invert colors for 'n' columns from the cursor position. 'w,h INVERT' – invert colors in a rectangle of 'w' columns and 'h' rows' where the cursor is at the upper left corner of the rectangle. 'x1,y1,x2,y2 INVERT' – invert colors in a rectangle with the upper left corner at 'x1,y1' and the lower right corner at 'x2,y2'.
PENCOLOR	Set color used by BOX and LINE statements to the RGB parameter. The parameter is a 24-bit integer RGB value in which the most significant 8-bits specify the red component, the middle 8-bits specify the green component, and the least significant 8-bits specify the blue component.
RESETCOLOR	Reset the current foreground, pen, and background colors to the default values of the output window. Note that the 'CS' and 'XX' mnemonics differ in that 'CS' does not reset the current colors, but the 'XX' mnemonic does.

Mnemonics to Transmit Data

Mnemonic	Explanation
BT	Begin Transmission. Begin transmitting all characters from the terminal's memory. This function is highly terminal dependent.
ET	End Transmission. Disable transmission of characters from the terminal's memory.
LU	Send Line Unprotected. All non-protected characters from the current cursor through the end of the line are transmitted from the terminal.
PS	Print Screen. Send the contents of the current screen through the terminal's Auxiliary/Printer port.
PU	Send Page Unprotected. All unprotected characters on the screen are transmitted from the screen to the system.
SL	Send Line All. All characters (including protected fields) on the line containing the cursor are transmitted from the screen to the system.
SP	Send Page All. All characters (including protected fields) on the screen are transmitted to the system.
TL	Transmit Line unprotected. All non-protected characters from the current cursor through the end of the line are transmitted from the terminal.
TP	Transmit Line protected. All characters (including protected fields) on the screen from the current cursor to the end of the screen are transmitted to the system.
TR	Transmit Screen unprotected. All non-protected characters from the current cursor through the end of the screen are transmitted from the terminal.
TS	Transmit Screen protected. All characters from the current cursor through the end of the screen are transmitted from the terminal.

Mnemonics for Drawing

Mnemonic	**Explanation**
ELLIPSE	Draw an ellipse bounded by a rectangle using the first two parameters as one corner and the second two parameters as the opposite corner using the current pen color and pen weight. For example, the mnemonic string '10,15,30,50ELLIPSE' would draw an ellipse within the a rectangle with one corner at grid coordinates 10,15 and the opposite corner at coordinates 30,50. The interior of the ellipse is filled by the current brush (normally transparent). The current cursor position is not changed.
FILLIMAGE	Draw an image file (such as JPEG or BMP) filling the defined rectangle. PChr$(*filepath*,*x1*,*y1*,*x2*,*y2*);'FILLIMAGE' *filepath* Image file path. When using dL4Term, this must be a path on the client system. *x1* Grid column of the upper left rectangle corner *y1* Grid row of the upper left rectangle corner *x2* Grid column of the lower right left rectangle corner *y2* Grid row of the lower right rectangle corner
FITMAGE	Draw an image file (such as JPEG or BMP) inside the defined rectangle preserving the image aspect ratio. PChr$(*filepath*,*x1*,*y1*,*x2*,*y2*);'FITIMAGE' *filepath* Image file path. When using dL4Term, this must be a path on the client system. *x1* Grid column of the upper left rectangle corner *y1* Grid row of the upper left rectangle corner *x2* Grid column of the lower right left rectangle corner *y2* Grid row of the lower right rectangle corner
FRAME	Draw a frame around (outside) a rectangle using the first two parameters as one corner and the second two parameters as the opposite corner. The frame color is controlled by the overall color scheme and not by the color mnemonics. An optional fifth parameter, a single character string, specifies the frame style ("S" for sunken, "R" for raised, "E" for etched, and "B" for bump). The default frame style is the style used by a 'WCSTRING' input box.
LINETO	Draw line from the current cursor position to the specified coordinate grid and column ('10,15LINETO') which becomes the new current cursor position. The line is drawn using the current pen color and pen weight. This mnemonic is used by the LINE statement.
PENCOLOR	Set color used by BOX and LINE statements to the RGB parameter. The parameter is a 24-bit integer RGB value in which the most significant 8-bits specify the red component, the middle 8-bits specify the green component, and the least significant 8-bits specify the blue component
PENWEIGHT	Set the pen width to the parameter times the coordinate grid unit.
RECT	Draw a rectangle using the first two parameters as one corner and the second two parameters as the opposite corner using the current pen color and pen weight. For example, the mnemonic string '10,15,30,50RECT' would draw a rectangle with one corner at grid coordinates 10,15 and the opposite corner at coordinates 30,50. The interior of the rectangle is filled by the current brush (normally transparent). The current cursor position is not changed.

RECTTO	Draw a rectangle using the current cursor position as one corner and the two parameters as the opposite corner using the current pen color and pen weight. For example, the mnemonic string ‘30,50RECT’ would draw a rectangle with one corner at the current cursor position and the opposite corner at grid coordinates 30,50. The interior of the rectangle is filled by the current brush (normally transparent). The current cursor position is not changed. This mnemonic is used by the BOX statement.

Mnemonics to Define the Coordinate Grid

Mnemonic	**Explanation**
GRIDENGLISH	Set coordinate grid by English units. The coordinate grid is defined to be in thousandths of an inch times the parameter measured from the upper left corner of the printable area. For example, the mnemonic string ‘100gridenglish’ would set the grid to be in tenths of an inch and in that grid the statement “PRINT @15,23;” would position the cursor to a point 1.5 inches to the right and 2.3 inches down from the upper left corner of the printable area of the screen, window, or page. The mnemonic may also be used with two numeric parameters, ‘n,d GRIDENGLISH’, to set the grid size to the fraction n/d. Thus the mnemonic ‘1000,72 GRIDENGLISH’ would set the grid unit to (1000/72) thousandths of an inch which simplifies to 1/72 inch or a “point”.
GRIDMETRIC	Set coordinate grid by metric units. The coordinate grid is defined to be in hundredths of a millimeter times the parameter measured from the upper left corner of the printable area. For example, the mnemonic string ‘100gridmetric’ would set the grid to be in millimeters and in that grid the statement “PRINT @15,23;” would position the cursor to a point 15 millimeters to the right and 23 millimeters down from the upper left corner of the printable area of the screen, window, or page. The mnemonic may also be used with two numeric parameters, ‘n,d GRIDMETRIC’, to set the grid size to the fraction n/d.
GRIDFONT	Set coordinate grid by the current font size. The coordinate grid is defined to be in average character widths and heights divided by the parameter and measured from the upper left corner of the printable area. For example, the mnemonic string ‘1gridfont’ would set the grid to be in character columns and rows as defined by the average width and height of a character in the current font. This is the default coordinate grid. The column width and row height are determined by the font in use when the GRIDFONT mnemonic is processed and will not be changed if the font typeface, style, or size is changed until another GRIDFONT mnemonic is processed. The mnemonic may also be used with two numeric parameters, ‘n,d GRIDFONT’, to set the grid size to the fraction n/d.

Miscellaneous Mnemonics

Mnemonic	**Explanation**
BH	Box Horizontal character. This mnemonic is used to draw horizontal box characters using WINDOW. If undefined, the '_' character is printed.
BV	Box Vertical character. This mnemonic is used to draw vertical box characters using WINDOW. If undefined, the '|' character is printed.
LANDSCAPE	Set printer to landscape mode (‘1 LANDSCAPE’) or to portrait mode (‘0 LANDSCAPE’).

MARGIN	Set printer margins. The mnemonic has two forms: ‘w MARGIN’ which sets the left margin to “w” grid units and ‘w,h MARGIN’ which sets the left margin to ‘w’ grid units and the top/bottom margins to ‘h’ grid units.
RB	Ring BELL. Sends the sequence causing the terminal to beep.
TP	Toggle Page. Switches the display to another page of memory in the terminal.
RD	Read Cursor. The terminal transmits its current coordinate position to the program. This function is highly dependent upon the terminal.
PI	Position Indicator. This mnemonic is used by supplied utilities to display the requested number of input characters in a field. The form used by the program is usually 'nPInML' where n is the number of characters in the field. The default character for this mnemonic is _.
SA	User Defined mnemonic to contain any non-supported terminal function.
SB	User Defined mnemonic to contain any non-supported terminal function.
SC	User Defined mnemonic to contain any non-supported terminal function.
SD	User Defined mnemonic to contain any non-supported terminal function.
S1	User Defined mnemonic to contain any non-supported terminal function.
S2	User Defined mnemonic to contain any non-supported terminal function.
S3	User Defined mnemonic to contain any non-supported terminal function.
S4	User Defined mnemonic to contain any non-supported terminal function.

Special Mnemonics for I/O Control

Mnemonic	Explanation
BACTFN	Begin activate-on-function-character. INPUT terminates on receipt of any normal termination character (such as carriage return) or any mnemonic character that is defined as a data character (such as ‘F3’ or ‘NEXTPAGE’). The terminating character can be read using the KEY option of the INPUT statement.
EACTFN	Disable activate-on-function-character. Normal INPUT (default) is restored. Input is terminated by [EOL] (usually RETURN), length or time.
BCTRACK	Begin cursor tracking. If input is performed immediately after outputting a BCTRACK mnemonic, input edit keys will be treated as data and returned as mnemonic characters such as ‘ML’. Cursor tracking is terminated by outputting any character other than a BCTRACK mnemonic.
BEGIN	Sent to a GUI element or as part of preprogrammed typeahead to set the cursor to the start of the current value and visibly mark the current value for possible replacement or deletion. The mnemonic ‘n BEGIN’ performs these operations on GUI element “n” and sets the input focus to that element.
IOBC	Begin activate-on-control-character. The IOBC mnemonic enables XON/XOFF and CTRL Q/CTRL S are ignored. The terminating control character is placed into the last position of the INPUT string variable. INPUT continues to terminate on receipt of a control character until the mnemonic 'IOEC' is sent.
IOBD	Begin Destructive Backspace. When destructive backspace is enabled (default), pressing a BACKSPACE or CONTROL-H results in the sequence backspace, space, backspace being transmitted to the screen. Destructive backspace continues until the 'IOED' mnemonic is sent.

IOBE	Begin Input Echo. As characters are entered on the screen, they are displayed (normal default). Input echo continues until the IOEE mnemonic is sent. The SYSTEM statement provides an additional way to enable/disable echo. Any of the 3 methods can be used together or separately.
IOBF	Mnemonic accepted, but does not perform a function.
IOBI	Begin input transparency. The IOBI mnemonic enables Binary Input mode resulting in no input translation of characters received until the IOEI is sent. Nulls, [ESC]s, and control characters are placed into the string exactly as received with and without the high-order bit set. When Binary Input is enabled, your INPUT statements must specify a time limit or character count or input continues indefinitely. See also HALT Command to unlock a port, and SYSTEM Statement Binary Input Mode.
IOBO	Begin output transparency. The IOBO mnemonic enables Binary Output Mode resulting in no special output translation of characters.
IOBX	Begin XON/XOFF protocol. The IOBX mnemonic enables Unix sending XON/XOFF protocol when communicating with a Host computer until the IOEX mnemonic is sent. The system prevents overflow of the type-ahead buffer by sending an XOFF to a host when the buffer is full. This function is usually used when you activate a program on a port that is wired directly to another system. Normal keyboard XON/XOFF protocol is always enables.
IOB\	Begin sending the \ character to the screen whenever [ESC] is pressed. The default operation is to always send the \ character without [ESC] branching in effect. The \ is sent until the IOE\ mnemonic is sent.
IOCI	Clear the contents of the terminal's type-ahead buffer. Any input entered but not processed as INPUT is discarded.
IOEC	Disable activate-on-control-character. Normal INPUT (default) is restored, and XON/XOFF flow control are terminated. CTRL Q and CTRL S are recognized. Input is terminated by [EOL] (usually RETURN), length or time.
IOED	End Destructive Backspace. Stop sending backspace, space, backspace. Send only a single backspace and erase the input character from the input buffer.
IOEE	End Input Echo. Disable echo of input characters on the terminal. Identical to using SYSTEM Statement. Input characters are not displayed on the screen until echo is enabled by SYSTEM or an IOBE mnemonic is sent.
IOEF	Mnemonic accepted, but does not perform a function.
IOEI	End Input Transparency. Normal Input Mode is activated, and Binary Input is disabled. Special characters are processed and [EOL] (usually RETURN) terminates INPUT.
IOEX	End XON/XOFF Protocol. Normal overflow of the type-ahead buffer is allowed. This is the default condition whereby type-ahead buffer overflow outputs a bell to the terminal, and input is discarded.
IOE\	End sending the \ character to the screen whenever [ESC] is pressed. This function disables the IOB\ mnemonic and system default. The \ character is never sent to the terminal when [ESC]is pressed.
IOIH	Setup for special Input Handling. This mnemonic is followed by a byte defining the type of Input processing to be performed.
IORS	Reset the I/O parameters for this terminal. Echo is enabled as is the output of "\" on [ESC]. All other IO modes are turned off.

Mnemonics for Graphic User Interfaces

Mnemonic	**Explanation**	
ONCLOSE	Define action to perform when a user attempts to close a session. This mnemonic is used to prevent a user from improperly exiting an application. A user can close a session by disconnecting a telnet session, selecting a window exit button, or any external method of terminating the user interface. The mnemonic requires a numeric parameter (the action number) and a string parameter. Action 0 displays the string text in a message box and gives the user a choice of exiting dL4 or continuing the current application. Action 1 displays the string text in a message box and then continues the current application. Action 2 discards the user request to exit and sends the string text as input to the application. The ONCLOSE setting can be cleared by specifying action 0 with an empty string (“”). Usage:	
	PChr$(n,text);’ONCLOSE’	
	n	Action to perform.
	Text	String to display.
WCBUTTON	Create button. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});’WCBUTTON’	
	n	GUI element number
	x1	Grid column of upper left button corner
	y1	Grid row of upper left button corner
	x2	Grid column of lower right left button corner
	y2	Grid row of lower right button corner
	label	Title string displayed on button with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 32 = send input on loss of focus)
WCDEFAULTBTN	Create default button. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});’WCDEFAULTBTN’	
	n	GUI element number
	x1	Grid column of upper left button corner
	y1	Grid row of upper left button corner
	x2	Grid column of lower right left button corner
	y2	Grid row of lower right button corner
	label	Title string displayed on button with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 32 = send input on loss of focus)
WCPAD	Create transparent button. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*scale*}}});’WCPAD’	
	n	GUI element number
	x1	Grid column of upper left button corner

	y1	Grid row of upper left button corner
	x2	Grid column of lower right left button corner
	y2	Grid row of lower right button corner
	label	Not used.
	options	Numeric options (1 = disable, 2 = tab stop, 32 = send input on loss of focus)
	scale	Scaling value for the pointer coordinates returned by a WCQUERY of a WCPAD element.
WCCHECK	Create check box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});'WCCHECK'	
	n	GUI element number
	x1	Grid column of upper left check box corner
	y1	Grid row of upper left check box corner
	x2	Grid column of lower right left check box corner
	y2	Grid row of lower right check box corner
	label	Title string displayed in check box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on losss of focus)
WCRADIO	Create radio button. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});'WCRADIO'	
	n	GUI element number
	x1	Grid column of upper left radio button corner
	y1	Grid row of upper left radio button corner
	x2	Grid column of lower right left radio button corner
	y2	Grid row of lower right radio button corner
	label	Title string displayed in radio button rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)
WCNUMBER	Create numeric input box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*l*}}});'WCNUMBER'	
	n	GUI element number
	x1	Grid column of upper left edit box corner
	y1	Grid row of upper left edit box corner
	x2	Grid column of lower right left edit box corner
	y2	Grid row of lower right edit box corner
	label	Title string displayed in edit box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)

	l	Limit on number of characters accepted in edit box
WCSTRING	Create character input box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*l*}}});'WCSTRING'	
	n	GUI element number
	x1	Grid column of upper left edit box corner
	y1	Grid row of upper left edit box corner
	x2	Grid column of lower right left edit box corner
	y2	Grid row of lower right edit box corner
	label	Title string displayed in edit box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)
	l	Limit on number of characters accepted in edit box
WCPRIVATE	Create character hidden input box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*l*}}});'WCPRIVATE'	
	n	GUI element number
	x1	Grid column of upper left edit box corner
	y1	Grid row of upper left edit box corner
	x2	Grid column of lower right left edit box corner
	y2	Grid row of lower right edit box corner
	label	Title string displayed in edit box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)
	l	Limit on number of characters accepted in edit box
WCLABEL	Create a label for an input box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* ,*label*);'WCLABEL'	
	n	GUI element number
	x1	Grid column of upper left display box corner
	y1	Grid row of upper left display box corner
	x2	Grid column of lower right left display box corner
	y2	Grid row of lower right display box corner
	label	Title string displayed in display box rectangle with optional ampersand before selection key
WCTEXT	Create multi-line character display box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options* {,*width*}}});'WCTEXT'	
	n	GUI element number
	x1	Grid column of upper left text box corner
	y1	Grid row of upper left text box corner
	x2	Grid column of lower right left textt box corner

	y2	Grid row of lower right text box corner
	label	Title string displayed in text box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 32 = send input on loss of focus)
	width	Maximum line length in characters. Using this parameter also enables a horizontal scroll bar.
WCMEMO	Create multi-line character input box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*l*}}});'WCMEMO'	
	n	GUI element number
	x1	Grid column of upper left memo box corner
	y1	Grid row of upper left memo box corner
	x2	Grid column of lower right left memo box corner
	y2	Grid row of lower right memo box corner
	label	Title string displayed in memo box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)
	l	Limit on number of characters accepted in memo box
WCLIST	Create selection list box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});'WCLIST'	
	n	GUI element number
	x1	Grid column of upper left list box corner
	y1	Grid row of upper left list box corner
	x2	Grid column of lower right left list box corner
	y2	Grid row of lower right list box corner
	label	Title string displayed in list box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change,8 = allow multiple selection, 16 = first field invisible, 32 = send input on loss of focus)
WCSHOWLIST	Create read-only list box. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});'WCSHOWLIST'	
	n	GUI element number
	x1	Grid column of upper left list box corner
	y1	Grid row of upper left list box corner
	x2	Grid column of lower right left list box corner
	y2	Grid row of lower right list box corner
	label	Title string displayed in list box rectangle with optional ampersand before selection key
	options	Numeric options (1 = disable, 2 = tab stop, 16 = first field invisible, 32 = send input on loss of focus)

WCEDITLIST

Create editable selection list box. Usage:

PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*l*}}});'WCEDITLIST'

n	GUI element number
x1	Grid column of upper left edit list box corner
y1	Grid row of upper left edit list box corner
x2	Grid column of lower right left edit list box corner
y2	Grid row of lower right edit list box corner
label	Title string displayed in edit list box rectangle with optional ampersand before selection key
options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)
l	Limit on number of characters accepted in edit box

WCLISTDROP

Create drop down selection list. Usage:

PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*}});'WCLISTDROP'

n	GUI element number
x1	Grid column of upper left list box corner
y1	Grid row of upper left list box corner
x2	Grid column of lower right left list box corner
y2	Grid row of lower right list box corner
label	Title string displayed in list box rectangle with optional ampersand before selection key
options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 16 = first field invisible, 32 = send input on loss of focus)

WCEDITDROP

Create drop down editable list box. Usage:

PChr$(*n*,*x1*,*y1*,*x2*,*y2* {,*label* {,*options*{,*l*}}});'WCEDITDROP'

n	GUI element number
x1	Grid column of upper left edit box corner
y1	Grid row of upper left edit box corner
x2	Grid column of lower right left edit box corner
y2	Grid row of lower right edit box corner
label	Title string displayed in edit box rectangle with optional ampersand before selection key
options	Numeric options (1 = disable, 2 = tab stop, 4 = send input on change, 32 = send input on loss of focus)
l	Limit on number of characters accepted in edit box

WCMENU

Create menu. Usage:

PChr$(*n*,*label*,*shortcut*{,*options*});'WCMENU'

n	GUI element number
label	Menu title string with optional ampersand before selection key

	shortcut	Shortcut key string
	options	Numeric options (1 = disable)
WCSUBMENU	Create submenu. Usage:	
	PChr$(*n*,*label*,*shortcut*{,*options*});'WCSUBMENU'	
	n	GUI element number
	label	Menu title string with optional ampersand before selection key
	shortcut	Shortcut key string
	options	Numeric options (1 = disable)
WCMENUACTION	Create menu action item. Usage:	
	PChr$(*n*,*label*,*shortcut*{,*options*});'WCMENUACTION'	
	n	GUI element number
	label	Menu title string with optional ampersand before selection key
	shortcut	Shortcut key string
	options	Numeric options (1 = disable)
WCMENUCHECK	Create menu check box item. Usage:	
	PChr$(*n*,*label*,*shortcut*{,*options*});'WCMENUCHECK'	
	n	GUI element number
	label	Menu title string with optional ampersand before selection key
	shortcut	Shortcut key string
	options	Numeric options (1 = disable)
WCMENURADIO	Create menu radio button item. Usage:	
	PChr$(*n*,*label*,*shortcut*{,*options*});'WCMENURADIO'	
	n	GUI element number
	label	Menu title string with optional ampersand before selection key
	shortcut	Shortcut key string
	options	Numeric options (1 = disable)
WCMENUSEP	Create menu separator	
WCENDMENU	End menu or sub-menu definition	
WCGROUP	Group graphical elements. Usage:	
	PChr$(*n*,*x1*,*y1*,*x2*,*y2*,*label*);'WCGROUP'	
	n	GUI element number
	x1	Grid column of upper left group rectangle corner
	y1	Grid row of upper left group rectangle corner
	x2	Grid column of lower right left group rectangle corner
	y2	Grid row of lower right group rectangle corner
	label	Title string displayed in group rectangle outline
WCMSGASK	Display message dialog box and return as an input string the uppercase label of the button selected by the user. Usage:	

	PChr$(*nmsg*{,*title*{,*options*}});'WCMSGASK'	
	msg	Message string to be displayed in dialog box
	title	Optional title string for dialog box
	options	Optional string that controls the presence and labeling of buttons within the dialog box. The default value is "O". The first uppercase letter selects the default button. The supported values are:

"ARI"	"Abort", "Retry", "Ignore"
"O"	"Ok"
"OC"	"Ok", "Cancel"
"RC"	"Retry", "Cancel"
"YN"	"Yes", "No"
"YNC"	"Yes", "No", "Cancel"

WCMSGERROR	Display an error message dialog box and return as an input string the uppercase label of the button selected by the user. See 'WCMSGASK' for a description of the parameters.
WCMSGINFO	Display an information message dialog box and return as an input string the uppercase label of the button selected by the user. See 'WCMSGASK' for a description of the parameters
WCMSGWARN	Display a warning message dialog box and return as an input string the uppercase label of the button selected by the user. See 'WCMSGASK' for a description of the parameters
WCSELECT	Select parameter ('n WCSELECT') as current graphical element
WCENABLE	Enable user input/selection to/of a specified element('n WCENABLE') or a range of elements ('n,m WCENABLE').
WCDISABLE	Disable user input/selection to/of a specified element ('n WCDISABLE') or a range of elements ('n,mWCDISABLE').
WCQUERY	Request a single graphical element ('n WCQUERY') or a range of elements ('n,m WCQUERY') to send their current values.
WCASKCOLOR	Display color selection dialog using the parameter ('n WCASKCOLOR') as the default RGB color. The selected color, if any, will be sent as a decimal number followed by a carriage return.
WCDELETE	Delete specifed graphical elements ('n WCDELETE' or 'first,last WCDELETE')
WCACTION	Change action performed by input element. Usage:

PChr$(*n*,*action*,*label*);'WCACTION'

n	GUI element number
action	Action to be modifed.0 changes the text sent when the element is selected. 1 changes the text sent when the element value is changed.
label	String to be sent by the specified action.

WCEVENT	Enable or disable keyboard input deferral after a GUI event is reported. If input is deferred ('1 WCEVENT'), then keyboard input will be buffered and not processed after a GUI event is report until a 'WCFOCUS' or 'BEGIN'

mnemonic is printed. This allows the application to set the input focus to a desired input element and direct the keyboard input to that new element. Input deferral mode is disabled by the ‘0 WCEVENT’ mnemonic string or by an ‘XX’ mnemonic. A ‘2 WCEVENT’ mnemonic string clears and discards any deferred keyboard input.

WCEXTKEYS	Enable or disable extended keyboard behavior in graphical elements. Without any parameters or in the form ‘3 WCEXTKEYS’, the mnemonic enables treating the ENTER key as a tab between GUI elements and as a newline within WCMEMO input boxes. The form ‘1 WCEXTKEYS’ just enables treating the ENTER key as a tab between elements. The form ‘2 WCEXTKEYS’ just enables treating ENTER as a newline in WCMEMO boxes. The form ‘0 WCEXTKEYS’ disables both options.
AUTOCOMPLETE	Define autocompletion value for a WCSTRING or WCNUMBER box. PChr$(*n*,*value*);’AUTOCOMPLETE’ *n* — GUI element number *value* — Autocompletion string value. If the current value of the input box matches the leading characters of the string, the current value will be replaced by the string.
WCBQRYBUF	Enable separate buffering of data sent by ‘WCQUERY’. When enabled, ‘WCQUERY’ results are read from record 1 (“INPUT #3,1;S$”).
WCEQRYBUF	Disable separate buffering of data sent by ‘WCQUERY’
WCFOCUS	Set current focus to selected element (‘n WCFOCUS’)
WCMARK	Mark or select item in a list box (‘n WCMARK’)
WCUNMARK	Unmark or unselect item in a list box (‘n WCUNMARK’)
WCSETCOLOR	Set text and background colors for graphical elements. If sent to a window, it sets the defaults for all subsequently created elements. If sent to an element, it changes the colors of the element. The mnemonic can be used without parameters (‘WCSETCOLOR’) or with two RGB parameters (‘t b WCSETCOLOR’). If used without parameters, the mnemonic uses the current window colors.
WCRESETCOLOR	Reset text and background colors to the defaults for graphical elements. The mnemonic can be sent to a window or to an existing element.
WCMARKCOLOR	Set text and background colors for selected items in graphical element item lists. The mnemonic can be used with 0, 1, or 2 numeric parameters. With no parameters, the window text and background colors are used. A single parameter is treated as an RGB text color and the window background color is used for the background. Two RGB parameters (‘t b WCMARKCOLOR’) set the text and background colors explicitly.
WCSETFONT	Set font for controls
WCRESETFONT	Reset font for controls to the default font
WCWHERE	Request the graphical element that currently has the input focus to send the action string “n”as input (‘n WCWHERE’). If the window itself has the focus, a ‘CR’ will be be returned as input. Note that the user may move the focus after the ‘WCWHERE’ mnemonic has been processed.

Table of Extended Graphics Codes

Form and chart components:

Mnemonic	Hex Value	Meaning
G1	0x250c	FORMS LIGHT DOWN AND RIGHT
G2	0x2510	FORMS LIGHT DOWN AND LEFT
G3	0x2514	FORMS LIGHT UP AND RIGHT
G4	0x2518	FORMS LIGHT UP AND LEFT
GC	0x253c	FORMS LIGHT VERTICAL AND HORIZONTAL
GD	0x252c	FORMS LIGHT DOWN AND HORIZONTAL
GH	0x2500	FORMS LIGHT HORIZONTAL
GL	0x2524	FORMS LIGHT VERTICAL AND LEFT
GR	0x251c	FORMS LIGHT VERTICAL AND RIGHT
GU	0x2534	FORMS LIGHT UP AND HORIZONTAL
GV	0x2502	FORMS LIGHT VERTICAL

Table of Mnemonic Codes

Control Characters

Mnemonic	Hex Value	Meaning
NUL	0x0000	NULL
SOH	0x0001	START OF HEADING
STX	0x0002	START OF TEXT
ETX	0x0003	END OF TEXT
EOT	0x0004	END OF TRANSMISSION
ENQ	0x0005	ENQUIRY
ACK	0x0006	ACKNOWLEDGE
BEL	0x0007	BELL
BS	0x0008	BACKSPACE
HT	0x0009	HORIZONTAL TABULATION
LF	0x000a	LINE FEED
VT	0x000b	VERTICAL TABULATION
FF	0x000c	FORM FEED
CR	0x000d	CARRIAGE RETURN
SO	0x000e	SHIFT OUT (possibly "status line on")
SI	0x000f	SHIFT IN
DLE	0x0010	DATA LINK ESCAPE
DC1	0x0011	DEVICE CONTROL ONE

DC2	0x0012	DEVICE CONTROL TWO
DC3	0x0013	DEVICE CONTROL THREE
DC4	0x0014	DEVICE CONTROL FOUR
NAK	0x0015	NEGATIVE ACKNOWLEDGE
SYN	0x0016	SYNCHRONOUS IDLE
ETB	0x0017	END OF TRANSMISSION BLOCK
CAN	0x0018	CANCEL
EM	0x0019	END OF MEDIUM
SUB	0x001a	SUBSTITUTE
ESC	0x001b	ESCAPE
FS	0x001c	FILE SEPARATOR
GS	0x001d	GROUP SEPARATOR
RS	0x001e	RECORD SEPARATOR (or "reset terminal")
US	0x001f	UNIT SEPARATOR
0x0020 -	0x007e	Printable ASCII
DEL	0x007f	DELETE
PAD	0x0080	PADDING CHARACTER
HOP	0x0081	HIGH OCTET PRESET
BPH	0x0082	BREAK PERMITTED HERE
NBH	0x0083	NO BREAK HERE
IND	0x0084	INDEX
NEL	0x0085	NEXT LINE
SSA	0x0086	START OF SELECTED AREA
ESA	0x0087	END OF SELECTED AREA
HTS	0x0088	CHARACTER TABULATION SET
HTJ	0x0089	CHARACTER TABULATION WITH JUSTIFICATION
VTS	0x008a	LINE TABULATION SET
PLD	0x008b	PARTIAL LINE FORWARD
PLU	0x008c	PARTIAL LINE BACKWARD
RI	0x008d	REVERSE LINE FEED
SS2	0x008e	SINGLE-SHIFT TWO
SS3	0x008f	SINGLE-SHIFT THREE
DCS	0x0090	DEVICE CONTROL STRING
PU1	0x0091	PRIVATE USE ONE
PU2	0x0092	PRIVATE USE TWO
STS	0x0093	SET TRANSMIT STATE
CCH	0x0094	CANCEL CHARACTER
MW	0x0095	MESSAGE WAITING
SPA	0x0096	START OF GUARDED AREA
EPA	0x0097	END OF GUARDED AREA

SOS	0x0098	START OF STRING
SGCI	0x0099	SINGLE GRAPHIC CHARACTER INTRODUCER
SCI	0x009a	SINGLE CHARACTER INTRODUCER
CSI	0x009b	CONTROL SEQUENCE INTRODUCER
STRM	0x009c	STRING TERMINATOR
OSC	0x009d	OPERATING SYSTEM COMMAND
PM	0x009e	PRIVACY MESSAGE
APC	0x009f	APPLICATION PROGRAM COMMAND
NBSP	0x00a0	NON-BREAKING SPACE

General punctuation

Mnemonic	Hex Value	Meaning
ENQUAD	0x2000	EN QUAD
EMQUAD	0x2001	EM QUAD
ENSPACE	0x2002	EN SPACE
EMSPACE	0x2003	EM SPACE
THREEEMSP	0x2004	THREE-PER-EM SPACE
FOUREMSP	0x2005	FOUR-PER-EM SPACE
SIXEMSP	0x2006	SIX-PER-EM SPACE
FIGSP	0x2007	FIGURE SPACE
PUNCTSP	0x2008	PUNCTUATION SPACE
THINSP	0x2009	THIN SPACE
HAIRSP	0x200a	HAIR SPACE
ZWSP	0x200b	ZERO WIDTH SPACE
NONJOINER	0x200c	ZERO WIDTH NON-JOINER
JOINER	0x200d	ZERO WIDTH JOINER
LRMARK	0x200e	LEFT-TO-RIGHT MARK
RLMARK	0x200f	RIGHT-TO-LEFT MARK
LINESEP	0x2028	LINE SEPARATOR
PARASEP	0x2029	PARAGRAPH SEPARATOR
LRE	0x202a	LEFT-TO-RIGHT EMBEDDING
RLE	0x202b	RIGHT-TO-LEFT EMBEDDING
PDF	0x202c	POP DIRECTIONAL FORMATTING
LRO	0x202d	LEFT-TO-RIGHT OVERRIDE
RLO	0x202e	RIGHT-TO-LEFT OVERRIDE

CJK symbols and punctuation

Mnemonic	Hex Value	Meaning

IDEOSP	0x3000	IDEOGRAPHIC SPACE

UNCLASSIFIED

Mnemonic	Hex Value	Meaning
IOIHIR	0xf000	IO INPUT HANDLING IRIS
IOIHSM	0xf001	IO INPUT HANDLING SMBASIC INPUT
IOIHSR	0xf002	IO INPUT HANDLING SMBASIC READ RECORD
IOIHSI	0xf003	IO INPUT HANDLING SIMPLE
IOBE	0xf004	IO BEGIN INPUT ECHO
IOEE	0xf005	IO END INPUT ECHO
IOBI	0xf006	IO BEGIN TRANSPARENT INPUT
IOEI	0xf007	IO END TRANSPARENT INPUT
IOBO	0xf008	IO BEGIN TRANSPARENT OUTPUT
IOBD	0xf009	IO BEGIN DESTRUCTIVE BACKSPACE
IOED	0xf00a	IO END DESTRUCTIVE BACKSPACE
IOBS	0xf00b	IO BEGIN BACKSLASH ON ESCAPE
IOES	0xf00c	IO END BACKSLASH ON ESCAPE
IOCI	0xf00d	IO CLEAR INPUT BUFFER
IOBC	0xf00e	IO BEGIN ACTIVATE ON CONTROL CHARACTER
IOEC	0xf00f	IO END ACTIVATE ON CONTROL CHARACTER
IOBX	0xf010	IO BEGIN XON XOFF PROTOCOL
IOEX	0xf011	IO END XON XOFF PROTOCOL
IORS	0xf012	IO RESET ALL
IOBF	0xf013	IO BEGIN FUNCTION KEY INPUT TRANSLATION
IOEF	0xf014	IO END FUNCTION KEY INPUT TRANSLATION
IOTE	0xf015	IO TOGGLE INPUT ECHO
GRIDENGLISH	0xf020	SET COORDINATE GRID BY ENGLISH
GRIDMETRIC	0xf021	SET COORDINATE GRID BY METRIC
GRIDFONT	0xf022	SET COORDINATE GRID BY FONT
FONTFACE	0xf024	SET FONT TYPEFACE
FONTSIZE	0xf025	SET FONT SIZE
FONTWEIGHT	0xf026	SET FONT WEIGHT
FONTCOLOR	0xf027	SET FONT COLOR
PENSTYLE	0xf02c	SET PEN STYLE
PENWEIGHT	0xf02d	SET PEN WEIGHT
PENCOLOR	0xf02e	SET PEN COLOR
BRUSHCOLOR	0xf034	SET BRUSH COLOR
TALEFT	0xf038	SET TEXT ALIGNMENT LEFT
TACENTER	0xf039	SET TEXT ALIGNMENT CENTER

TARIGHT	0xf03a	SET TEXT ALIGNMENT RIGHT
TADECIMAL	0xf03b	SET TEXT ALIGNMENT DECIMAL
BACKCOLOR	0xf03c	SET BACKGROUND COLOR
LINETO	0xf03f	DRAW LINE TO
RECTTO	0xf03e	DRAW RECTANGLE TO
RECT	0xf040	DRAW RECTANGLE
ELLIPSE	0xf041	DRAW ELLIPSE
BCTRACK	0xf081	BEGIN CURSOR TRACKING
ET	0xf083	END TRANSMISSION
RB	0xf087	RING BELL
ML	0xf088	MOVE LEFT
TF	0xf089	TAB FORWARD
MH	0xf08f	MOVE HOME
CS	0xf090	CLEAR SCREEN
S1	0xf091	SPECIAL CODE 1
S2	0xf092	SPECIAL CODE 2
S3	0xf093	SPECIAL CODE 3
S4	0xf094	SPECIAL CODE 4
ES	0xf095	END WRITE STATUS LINE
SF	0xf097	STATUS LINE OFF
WS	0xf098	BEGIN WRITE STATUS LINE
K0	0xf099	SET CURSOR OFF
K1	0xf09a	SET CURSOR BLINKING BOX
K2	0xf09b	SET CURSOR STEADY BLOCK
K3	0xf09c	SET CURSOR BLINKING UNDERLINE
K4	0xf09d	SET CURSOR STEADY UNDERLINE
BG	0xf09e	BEGIN GRAPHICS MODE
EG	0xf09f	END GRAPHICS MODE
MR	0xf0a0	MOVE RIGHT
RD	0xf0a1	READ CURSOR POSITION
EF	0xf0a2	END PROGRAM FUNCTION KEY
CU	0xf0a3	CLEAR SCREEN UNPROTECTED
CL	0xf0a4	CLEAR TO END OF LINE
CE	0xf0a5	CLEAR TO END OF SCREEN
P1	0xf0a6	PROGRAM FUNCTION KEY 1
P2	0xf0a7	PROGRAM FUNCTION KEY 2
P3	0xf0a8	PROGRAM FUNCTION key 3
P4	0xf0a9	PROGRAM FUNCTION key 4
MD	0xf0aa	MOVE DOWN
MU	0xf0ab	MOVE UP

P5	0xf0ac	PROGRAM FUNCTION KEY 5
P6	0xf0ad	PROGRAM FUNCTION KEY 6
P7	0xf0ae	PROGRAM FUNCTION KEY 7
P8	0xf0af	PROGRAM FUNCTION KEY 8
BB	0xf0b0	BEGIN BLINK MODE
EB	0xf0b1	END BLINK MODE
BR	0xf0b2	BEGIN REVERSE VIDEO MODE
ER	0xf0b3	END REVERSE VIDEO MODE
BD	0xf0b4	BEGIN DIMMED INTENSITY MODE
ED	0xf0b5	END DIMMED INTENSITY MODE
BP	0xf0b6	BEGIN PROTECTED MODE
EP	0xf0b7	END PROTECTED MODE
BU	0xf0b8	BEGIN UNDERLINE MODE
EU	0xf0b9	END UNDERLINE MODE
BX	0xf0ba	BEGIN EXPANDED PRINT MODE
EX	0xf0bb	END EXPANDED PRINT MODE
FM	0xf0bc	BEGIN FORMAT MODE
FX	0xf0bd	END FORMAT MODE
LK	0xf0be	LOCK KEYBOARD
UK	0xf0bf	UNLOCK KEYBOARD
BT	0xf0c0	BEGIN TRANSMISSION FROM MEMORY
MP	0xf0c1	USE MEMORY POINTER FOR NEXT POSITION
IL	0xf0c2	INSERT LINE
DL	0xf0c3	DELETE LINE
IC	0xf0c4	INSERT CHARACTER
DC	0xf0c5	DELETE CHARACTER
CT	0xf0c6	CLEAR TABS
ST	0xf0c7	SET TAB
AE	0xf0c8	AUXILIARY PORT ENABLE
AD	0xf0c9	AUXILIARY PORT DISABLE
SL	0xf0ca	SEND LINE
LU	0xf0cb	SEND LINE UNPROTECTED
SP	0xf0cc	SEND PAGE
GN	0xf0cd	SET COLOR GREEN
TB	0xf0ce	TAB BACKWARD
PI	0xf0cf	INPUT POSITION INDICATOR
RE	0xf0d0	SET COLOR RED
PU	0xf0d1	SEND PAGE UNPROTECTED
YE	0xf0d2	SET COLOR YELLOW
BL	0xf0d3	SET COLOR BLUE

MA	0xf0d4	SET COLOR MAGENTA
CY	0xf0d5	SET COLOR CYAN
WH	0xf0d6	SET COLOR WHITE
XX	0xf0d7	RESET ALL
SA	0xf0d8	SPECIAL CODE A
SB	0xf0d9	SPECIAL CODE B
SC	0xf0da	SPECIAL CODE C
SD	0xf0db	SPECIAL CODE D
BV	0xf0dc	BOX VERTICAL LINE
BH	0xf0dd	BOX HORIZONTAL LINE
WD	0xf0e2	SET WIDE MODE
NR	0xf0e3	SET NARROW MODE
RF	0xf0e4	RESET FUNCTION KEYS
TL	0xf0e5	TRANSMIT LINE UNPROTECTED
TP	0xf0e6	TRANSMIT LINE PROTECTED
TR	0xf0e7	TRANSMIT SCREEN UNPROTECTED
TS	0xf0e8	TRANSMIT SCREEN PROTECTED
PS	0xf0e9	PRINT SCREEN
BA	0xf0eb	BEGIN TRANSPARENT PRINT MODE
EA	0xf0ec	END TRANSPARENT PRINT MODE
RV	0xf0ed	SET REVERSED VIDEO
NV	0xf0ee	SET NORMAL VIDEO
BO	0xf0ef	BEGIN VISIBLE PRINT MODE
EO	0xf0f0	END VISIBLE PRINT MODE
BK	0xf0f1	BACK TO BEGINNING OF LINE
BC	0xf0f2	BEGIN COMPRESSED MODE
EC	0xf0f3	END COMPRESSED MODE
BI	0xf0f4	BEGIN ITALIC MODE
EI	0xf0f5	END ITALIC MODE
BSO	0xf0f6	BEGIN STRIKE OUT MODE
ESO	0xf0f7	END STRIKE OUT MODE
BBOLD	0xf0f8	BEGIN BOLD MODE
EBOLD	0xf0f9	END BOLD MODE
BSUB	0xf0fa	BEGIN SUBSCRIPT MODE
ESUB	0xf0fb	END SUBSCRIPT MODE
BSUP	0xf0fc	BEGIN SUPERSCRIPT MODE
ESUP	0xf0fd	END SUPERSCRIPT MODE
ALIGN	0xf0fe	ALIGN TO NEXT HORIZONTAL BOUNDARY
MOVETO	0xf0ff	MOVE TO
ADD	0xf100	FUNCTION KEY ADD

BEGIN	0xf101	FUNCTION KEY BEGIN
CANCEL	0xf102	FUNCTION KEY CANCEL
CLEAR	0xf103	FUNCTION KEY CLEAR
CLOSE	0xf104	FUNCTION KEY CLOSE
COMMAND	0xf105	FUNCTION KEY COMMAND
COPY	0xf106	FUNCTION KEY COPY
CREATE	0xf107	FUNCTION KEY CREATE
CUT	0xf108	FUNCTION KEY CUT
DIVIDE	0xf109	FUNCTION KEY DIVIDE
END	0xf10a	FUNCTION KEY END
EXEC	0xf10b	FUNCTION KEY EXEC
EXIT	0xf10c	FUNCTION KEY EXIT
FIND	0xf10d	FUNCTION KEY FIND
HELP	0xf10e	FUNCTION KEY HELP
LOAD	0xf10f	FUNCTION KEY LOAD
MARK	0xf110	FUNCTION KEY MARK
MESSAGE	0xf111	FUNCTION KEY MESSAGE
MODIFY	0xf112	FUNCTION KEY MODIFY
MOVE	0xf113	FUNCTION KEY MOVE
MULTIPLY	0xf114	FUNCTION KEY MULTIPLY
NEXT	0xf115	FUNCTION KEY NEXT
NEXTPAGE	0xf116	FUNCTION KEY NEXTPAGE
NEW	0xf117	FUNCTION KEY NEW
OPEN	0xf118	FUNCTION KEY OPEN
OPTIONS	0xf119	FUNCTION KEY OPTIONS
PASTE	0xf11a	FUNCTION KEY PASTE
PAUSE	0xf11b	FUNCTION KEY PAUSE
PREV	0xf11c	FUNCTION KEY PREV
PREVPAGE	0xf11d	FUNCTION KEY PREVPAGE
PRINT	0xf11e	FUNCTION KEY PRINT
REDO	0xf11f	FUNCTION KEY REDO
REFRESH	0xf120	FUNCTION KEY REFRESH
RENAME	0xf121	FUNCTION KEY RENAME
REPLACE	0xf122	FUNCTION KEY REPLACE
RESTART	0xf123	FUNCTION KEY RESTART
RESTORE	0xf124	FUNCTION KEY RESTORE
RESUME	0xf125	FUNCTION KEY RESUME
RUN	0xf126	FUNCTION KEY RUN
SAVE	0xf127	FUNCTION KEY SAVE
SELECT	0xf128	FUNCTION KEY SELECT

SETTINGS	0xf129	FUNCTION KEY SETTINGS
SIZE	0xf12a	FUNCTION KEY SIZE
SORT	0xf12b	FUNCTION KEY SORT
START	0xf12c	FUNCTION KEY START
STOP	0xf12d	FUNCTION KEY STOP
SUBTRACT	0xf12e	FUNCTION KEY SUBTRACT
SUSPEND	0xf12f	FUNCTION KEY SUSPEND
UNDO	0xf130	FUNCTION KEY UNDO
F0	0xf140	FUNCTION KEY 0
F1	0xf141	FUNCTION KEY 1
F2	0xf142	FUNCTION KEY 2
.		
.		
.		
F63	0xf17f	FUNCTION KEY 63
BLACK	0xf180	SET COLOR BLACK
RESETCOLOR	0xf181	RESET FG/PEN/BG COLOR TO DEFAULT
WINDOW	0xf182	CREATE WINDOW
WMODAL	0xf183	CREATE MODAL WINDOW
WCHILD	0xf184	CREATE CHILD WINDOW
WDELETE	0xf185	CLOSE/DESTROY WINDOW
WHIDE	0xf186	MAKE WINDOW INVISIBLE
WTITLE	0xf187	CHANGE WINDOW TITLE
WSELECT	0xf188	SELECT CURRENT WINDOW
WRANK	0xf189	CHANGE WINDOW Z-ORDER
WCANVAS	0xf18a	CHANGE CANVAS SIZE
WOUTPUT	0xf18b	CHANGE OUTPUT REGION SIZE/POSITION
WVIEW	0xf18c	CHANGE DISPLAY WINDOW CANVAS SIZE/POSITION
WSCROLL	0xf18d	SCROLL WINDOW POSITION IN CANVAS
WMOVE	0xf18e	MOVE DISPLAY WINDOW ON SCREEN
WSHOW	0xf18f	MAKE WINDOW VISIBLE
WOUTPUTSIZE	0xf190	RESIZE OUTPUT REGION
WVIEWSIZE	0xf191	RESIZE DISPLAYED WINDOW IN CANVAS
WENABLE	0xf192	ENABLE WINDOW
WDISABLE	0xf193	DISABLE WINDOW
WCBUTTON	0xf194	CREATE BUTTON
WCCHECK	0xf195	CREATE CHECK BOX
WCRADIO	0xf196	CREATE RADIO BUTTON
WCNUMBER	0xf197	CREATE NUMERIC INPUT BOX
WCSTRING	0xf198	CREATE CHARACTER INPUT BOX

WCPRIVATE	0xf199	CREATE CHARACTER HIDDEN INPUT BOX
WCLABEL	0xf19a	CREATE A LABEL FOR AN INPUT BOX
WCTEXT	0xf19b	CREATE MULTI-LINE CHARACTER DISPLAY BOX
WCMEMO	0xf19c	CREATE MULTI-LINE CHARACTER INPUT BOX
WCLIST	0xf19d	CREATE SELECTION LIST BOX
WCEDITLIST	0xf19e	CREATE EDITABLE SELECTION LIST BOX
WCLISTDROP	0xf19f	CREATE DROP DOWN SELECTION LIST
WCEDITDROP	0xf1a0	CREATE DROP DOWN EDITABLE LIST BOX
WCMENU	0xf1a1	CREATE MENU
WCMENUACTION	0xf1a2	CREATE MENU ACTION ITEM
WCMENUCHECK	0xf1a3	CREATE MENU CHECK BOX ITEM
WCMENURADIO	0xf1a4	CREATE MENU RADIO BUTTON ITEM
WCMENUSEP	0xf1a5	CREATE MENU SEPARATOR
WCENDMENU	0xf1a6	END MENU OR SUB-MENU DEFINITION
WCGROUP	0xf1a7	GROUP GRAPHICAL ELEMENTS
WCSELECT	0xf1a8	SELECT CURRENT GRAPHICAL ELEMENT
WCENABLE	0xf1a9	ENABLE USER INPUT/SELECTION TO/OF ELEMENT
WCDISABLE	0xf1aa	DISABLE USER INPUT/SELECTION TO/OF ELEMENT
WCQUERY	0xf1ab	REQUEST GRAPHICAL ELEMENT TO SEND VALUE
WCDELETE	0xf1ac	DELETE A GRAPHICAL ELEMENT
WCACTION	0xf1ad	CHANGE ACTION PERFORMED BY INPUT ELEMENT
WCFOCUS	0xf1ae	SET CURRENT FOCUS TO SELECTED ELEMENT
WCMARK	0xf1af	MARK OR SELECT ITEM
WCUNMARK	0xf1b0	UNMARK OR UNSELECT ITEM
WCSUBMENU	0xf1b1	CREATE SUBMENU
WCSETFONT	0xf1b3	SET FONT FOR CONTROLS
INPUTSTART	0xf1b4	RECORD START OF INPUT
LITNUL	0xf1b5	LITERAL NULL (BINARY ZERO)
LITCR	0xf1b6	LITERAL CARRIAGE RETURN
RESETATTR	0xf1b7	CLEAR ALL ATTRIBUTES (BLINK, DIM, ..)
BACTFN	0xf1b8	BEGIN ACTIVATE ON MNEMONIC CHARS
EACTFN	0xf1b9	END ACTIVATE ON MNEMONIC CHARS
INVERT	0xf1ba	INVERT COLORS IN SPECIFIED AREA
PGMFN	0xf1bb	PROGRAM FUNCTION KEY
ONCLOSE	0xf1bc	WARN/PREVENT EXIT
LANDSCAPE	0xf1bd	ENABLE PRINTER LANDSCAPE/PORTRAIT
WCWHERE	0xf1be	RETURN CURRENT GRAPHICAL ELEMENT ACTION
LPI	0xf1bf	LINES PER INCH (PRINTERS)
CPI	0xf1c0	CHARACTERS PER INCH (PRINTERS)
FONTCELL	0xf1c1	FONT CHARACTER CELL SIZE

MARGIN	0xf1c2	SET MARGINS (PRINTERS)
WCDEFAULTBTN	0xf1c3	CREATE DEFAULT BUTTON
SUSPENDAUX	0xf1cd	SUSPEND AUX PRINTING
CONTINUEAUX	0xf1ce	CONTINUE AUX PRINTING
WCEVENT	0xf1cf	CONTROL TRANSMISSION OF GUI EVENTS
PGMHELPFN	0xf1d0	PROGRAM FUNCTION KEY TO RETURN FOCUS
WCBQRYBUF	0xf1d1	BEGIN WCQUERY INPUT BUFFERING
WCEQRYBUF	0xf1d2	END WCQUERY INPUT BUFFERING
WCRESETFONT	0xf1d3	RESET FONT FOR CONTROLS
RESETFONT	0xf1d4	RESET FONT FOR TERMINAL/WINDOW
DEFAULTCOLOR	0xf1d7	USE CURRENT COLORS AS SESSION DEFAULTS
WCSETCOLOR	0xf1d8	SET BG/FG COLORS FOR CONTROLS
WCRESETCOLOR	0xf1d9	RESET BG/FG COLORS FOR CONTROLS
WCASKCOLOR	0xf1da	ASK USER TO CHOOSE COLOR
WCMARKCOLOR	0xf1db	SET COLORS FOR SELECTED ITEMS
WCMSGWARN	0xf1dc	DISPLAY WARNING MESSAGE DIALOG
WCMSGINFO	0xf1dd	DISPLAY INFORMATION MESSAGE DIALOG
WCMSGASK	0xf1de	DISPLAY QUESTION MESSAGE DIALOG
WCMSGERROR	0xf1df	DISPLAY ERROR MESSAGE DIALOG
FITIMAGE	0xf1e3	DISPLAY IMAGE FILE PRESERVING ASPECT RATIO
FRAME	0xf1e4	DRAW EDGE AROUND RECTANGLE
FILLIMAGE	0xf1e6	DRAW IMAGE FILLING SPECIFIED RECTANGLE
WCEXTKEYS	0xf1e7	ENABLE OR DISABLE EXTENDED KEY BEHAVIOR
WCPAD	0xf1e8	CREATE TRANSPARENT BUTTON
WCSHOWLIST	0xf1e9	CREATE READ ONLY LIST BOX
AUTOCOMPLETE	0xf1ea	ENABLE AUTOCOMPLETION IN GRAPHICAL ELEMENTS

Chapter 7 - Statements

Introduction

This chapter describes dL4 BASIC statements that are used to create dL4 BASIC programs. A quick reference listing of these statements is available in Appendix D of this guide. The notations used to represent the syntax of statements is listed in "Syntax", Chapter 1 of this guide.

Statement Structure

A BASIC statement can optionally begin with either a line number or a label:

```
{stmt.no | label:} STATEMENT
```

dL4 BASIC statements are executed when a user executes the program. Debugging is facilitated through SCOPE, which is documented in the dL4 Command Reference Guide.

Certain statements may be executed immediately from the keyboard, i.e., they are executed as soon as the user finishes typing a statement. These statements are identified in this chapter by "Executable From Keyboard".

In this chapter, statements are listed alphabetically with the general forms given in terms of literal elements in upper case or variables in *italic* type. Upper case is used for all key words such as utilities, statements, functions, and environment variables. Key words are all cross-referenced in the Index at the back of this guide. Each statement begins on a separate page and conforms to the standard format.

NOTE: The syntax of every statement begins with:

{*stmt.no* | *label:*} STATEMENT {*parameters*}

as in:

{*stmt.no* | *label:*}ADD *chan. expr, arg {...};*

What this means is that some statements are executable from the keyboard, making statement numbers and labels unnecessary, while other statements are not executable from the keyboard. This guide clearly identifies whether each statement can or cannot be executed from the keyboard. To avoid repetition, this stmt.no/label argument is omitted from statement syntaxes, but it should be understood to exist in every case.

Statement Documentation Format

Each statement begins a new page in this guide, documented as follows:

STATEMENT

Synopsis

Summary of the functionality of the statement.

Syntax

STATEMENT syntax with its parameter lists.

Parameters

Description of each parameter.

Executable From Keyboard?

Yes or No

Remarks

Discussion of the usage of the statement in context.

Examples

```
Examples of the statement in context.
```

See also

Related statements.

Statements, Line Numbers and Labels

All program instructions are called *statements*. They have the general form:

```
{ line-no | label: } { statement { \ statement  } }
```

line no is the valid line number, 1 to 268369919.

label: is a valid statement label followed by a colon.

statement is any valid BASIC statement.

{\...} is the separator for multiple statements (also called sub-statements) appearing on the same statement line.

Line Identification

Each line begins with an optional line number, *line-no*, and ends with the **[EOL]** end of line character. If specified, *line-no* must be an integer in the range 1 through 268369919.

Following, or in place of, the *line-no* can be a statement *label*. The *label* can be from 1 to 32 characters in length consisting of letters, digits, and underscore. A *label* must begin with a letter or underscore and end with a colon.

Throughout this guide, *line-no* is used to indicate selection of either a line number or label. If a *label* is not explicitly defined for a statement, any supplied *line-no* is considered both the line number and label. If a statement has neither a *line-no* or *label*, it cannot be directly referenced by other program statements.

A *statement* is one instruction to be executed by the computer, such as printing a list of values. A program line is a line consisting of one or more *statements*.

Multiple-Statement Lines

Several *statements* can appear on a single line, separated by a backslash (****) . S*tatements* are numbered on each line from the left, starting with 1. For example:

```
PRINT TOTAL; J  \  IF J End
```

When utilizing multi-statement lines, you should note certain programming effects. Conditional branching (**GOTO, GOSUB**, **ON**) can only select the first *statement* of any line. Branching to statements (other than the first) is provided only by the **JUMP** statement.

ADD

Synopsis

Low-level statement to insert data or data definitions into a file.

Syntax

ADD *chan.expr* {*expr.list*} {;}

Parameters

chan.expr is a driver-class dependent channel expression.

expr.list is an arbitrary number of comma separated expressions or variables of any dL4 data types.

"**;**" unlocks the record after a successful **ADD**.

Executable From Keyboard?

Yes.

Remarks

The **ADD** statement is most commonly used to insert a new data record or to define a new index. **ADD** is a low-level statement intended for use in utilities and other programs that need to perform special file manipulations. Most applications should use the **ADD RECORD** or **ADD INDEX** statements rather than **ADD**. Refer to the dL4 Files and Devices reference manual for more information on using **ADD** with specific file types or drivers.

Examples

```
Add #1,0,0,-1;CustRec.Name$
```

```
Add #1,0,-1,-1;
```

See also

ADD RECORD, ADD INDEX, DEFINE RECORD

ADD INDEX

Synopsis

Add an index to a file.

Syntax

ADD INDEX *chan.no*, *index.no*; *struct.var*

Parameters

chan.no identifies a valid channel number.

index.no is a numeric expression whose integer value identifies an index to be created in the file.

struct.var is a variable of structure data type.

Executable From Keyboard?

Yes.

Remarks

In many drivers, indices may be added only before data has been written to the file. Indices should be created beginning with index 1 with consecutive index numbers.

Defining an index requires defining a structure where all members have 'fieldname' designations. This structure identifies the various parts of the key.

Options for the entire Key include: Unique, Duplicates and Packed.

Options for Key members include: Ascending, Descending, Uppercase.

```
Def Struct CustKey1   : Key "NameCtyBal" + Duplicates + Descending
  Member Name$[25]    : Key "Name" + Uppercase
  Member City$[25]    : Key "City" + Uppercase
  Member 3%,Balance   : Key "CurrBal"
End Def

Dim Key1. As CustKey1
Add Index #5,1;Key1.    ! Define index 1 as NameCtyBal directory
```

In this example, the structure CustKey1 is named "NameCtyBal" and represents an index of possibly duplicate keys which are to be collated in descending order.

The member Name$ is an 25-character string from the data field with the same name. It is to be uppercased. The field City$ is a 25-character string from the data filed with the same name. It is also to be uppercased. The last part of this key, Balance, is a 3% numeric field from the field named "CurrBal".

Once the structure is defined, a new index (directory) is added by the **ADD INDEX** statement and all active records are keyed immediately. If no errors result, the selected *index* was successfully added.

Examples

```
Add Index #1,1;Key1.

Add Index #1,2;Key2.  ! Indices must be added in order
```

See also

ADD

ADD RECORD

Synopsis

Add new record to file.

Syntax

ADD RECORD *chan.no*; *struct.var* {;}

Parameters

chan.no identifies a valid channel number.

struct.var is a variable of structure data type.

";" unlocks the record after a successful **ADD RECORD**.

Executable From Keyboard?

Yes.

Remarks

A new record is allocated, written and all keys associated with this record are inserted. When the add operation is complete, the new record becomes the current record.

If no errors result, the selected record was successfully added to the file.

Examples

```
Add Record #1;CustRec.
```

```
Add Record #chan;CustRec.;
```

See also

ADD

BOX

Synopsis

Draw a rectangular figure on display device.

Syntax

BOX {*chan.no*;} {*@x1,y1*;} [*TO @x2,y2*;] | [*SIZE w,h*]

Parameters

chan.no identifies a valid channel number.

x1,*y1* are the column, row coordinates of the upper left corner.

x2,*y2* are the lower right column, row coordinates.

w,*h* identify the width and height.

Executable From Keyboard?

Yes.

Remarks

Box drawing is a function of the window and printer drivers, and uses the **#,#RECTTO** and **#,#RECT** mnemonics.

If @x1,y1 is not specified, the current cursor position is used as the upper left corner.

Examples

```
Box @7,2; To @70,10;
Box @7,2; Size 70,19
Box To @70,10;
```

See Also

LINE, **SIZE**

BUILD

Synopsis

Create and open a file.

Syntax1

BUILD chan.no, file.spec.str {**AS** driver-class **|** driver-name } {, {chan.no,} file.spec.str {**AS** driver-class **|** driver-name}} ...

Syntax2

BUILD chan.no, file.spec.items **AS** driver-class **|** driver-name {, {chan.no,} file.spec.items **AS** driver-class **|** driver-name} ...

Syntax3

BUILD *chan.no*, + *file.spec.str* {, {*chan.no*,} +*file.spec.str*} ...

Parameters

chan.no identifies a valid channel number, which the program uses for subsequent references to the file.

file.spec.str, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to build and open a file.

driver-class specifies the driver-class, instead of using a default driver-class derived from the file.spec.

driver-name specifies the driver-name, instead of using a default driver-class derived from the file.spec.

file.spec.items, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to build and open a file.

\+ *file.spec.str* identifies a valid dL4 file specification used to create and open a text file.

Executable From Keyboard?

Yes.

Remarks

Each *file.spec.str,* which is described in detail in Chapter 9 of this guide, contains the file's *attributes* and *filename* to be created. Multiple strings may be specified to create several files and they will be opened on successive *channel* numbers. Any new *channel* number (**#***channel*) in the filename list will cause assignment of channels to continue from that number.

The *attributes* are optional and may consist of several items, selecting the type, structure, and protection of the file.

The *filename* is any legal *filename*. If the *filename* is to replace an existing file on the system, the name must be terminated with an exclamation point (**!**).

Unless as **AS** clause is used, the file type to be built will be determined by the *file.spec.str* or *file.spec.items*.

If the file is to be created as a Contiguous data file, the initial *Record Count* and *Record Length* must be specified in the form "**[**count**:**length**]**". The *Record Count* is the initial number of records to be allocated to the file. *Record length* is specified in words.

If no record count/length is specified, the file is created as a Formatted Item file. The Record Length and format is defined by the program when Record 0 is written.

If the *str.expr* defining the filename is preceded by a + sign (note: the + character is not within the *str.expr*), the file is created as a text file.

The **AS** clause can be used to override the default driver selection:

```
Build #c; <filename> As "CLASS NAME"
Build #c; <filename> As "DRIVER NAME"
```

class might be "Full-ISAM" for any available full ISAM driver, or a specific full ISAM driver.

Older-style **BUILD** statements such as:

```
Build #1,+"MYFILE!"
```

can be made more readable as:

```
Build #1,"MYFILE" As "TEXT"
```

Examples

```
Build #1,"cust.masterfi!" As "Full-ISAM"
Build #0,"2/ABC" , + "/usr/ub/3/textfile!"
Build #C,"<644> [1000:256] PAYROLL/CFILE!"
```

See also

OPEN, EOPEN, ROPEN, WOPEN

CALL (BASIC Program)

Synopsis

Call a BASIC program.

Syntax

CALL *filename* {, *parm.list*}

Parameters

filename is a string literal or expression containing a dL4 BASIC program filename which is optionally preceded by a relative or absolute directory pathname.

parm.list is a comma separated list of expressions or variables of any data types to be passed to the calling program.

Executable From Keyboard?

No.

Remarks

BASIC programs called as subroutines are referred to as *subprograms*. A subprogram accepts a list of argument variables passed by the calling program by use of the **ENTER** statement. The number and type of arguments in the **CALL** statement must match those in the **ENTER** statements of the called program. The maximum number of arguments is limited only by the maximum statement length.

A subprogram accepts and returns values through the passed list of arguments which may be any combination of: *variables, constants, or expressions.* The argument name in the subprogram does not need to (and generally won't) be identical to the name of the passed variable in the calling program. For example, if the calling program passes A$ and T, the subprogram may **ENTER** with DATA$ and VALUE. The variable names specified by **ENTER** are mapped to reference the data space of the variable names passed in the **CALL**. All other variables in a subprogram are considered local to the subprogram.

Subprograms can be nested indefinitely, limited only by the maximum process size of the Operating System.

The *parm.lis*t may be defined as any combination of *str.vars, num.vars, mat.vars, str.exprs, num.exprs, array.vars* or *str.lit*, depending on the requirements of the subroutine being called. A *mat.var* or *array.var* in **CALL** or **ENTER** must be specified with empty subscripts; e.g. A3[ ]. Otherwise, only the first array element will be passed as an argument. The subroutine may use these items for input and output of data. A variable (not an expression) must be specified in positions of the *parm.list* which return information to the program.

Examples

```
Call "pgm",A$,B[],C[2],Input$
```

See also

CALL (Procedure), **ENTER**

CALL (Procedure)

Synopsis

Call a procedure.

Syntax

CALL *proc.name* ({*parm.list*})

Parameters

proc.name is the name of a valid existing procedure.

parm.list is a comma separated list of expressions or variables of any data types to be passed to the calling procedure.

Executable From Keyboard?

No.

Remarks

Whenever a *proc.name* is to be used before its definition within the current program unit or program, or physically resides in another program, a **DECLARE** statement must occur before its first use.

An error is generated before program execution starts, if any **EXTERNAL** *proc.name* references are unresolved.

Optionally *parameters* may be passed to the procedure in the *param.list*. The *parameters* may be any type of data, including a *structure*. When passing a *structure*, the procedure must also include its own structure definition of an identical structure and supply the structures designation.

Variables are passed to procedures by reference, not by name. Expressions are passed to procedures by value. When variables are passed by reference to a procedure, that procedure actually points its referenced variables to the caller's supplied variables data space. Any changes to the variable are affected in the caller's program. If a procedure updates, or returns a value in, a referenced variable, that operation will be lost if the caller passed an expression. Normally, procedures need not concern themselves with what was passed, however the caller should be aware of the appropriate calling sequence.

When a caller invokes a procedure which accepts a specific list of arguments, the interpreter verifies that the parameter types being passed are of the correct type. If the procedure calls for a string, the interpreter will verify that the argument is string.

An error is not generated should a caller pass an expression when the procedure assumes a variable reference. The caller simply elects not to care about any result returned in that variable reference.

Examples

```
! This is an example of the CALL statement (calling a procedure)
External Sub Printit(S$)
      If Not(S$) Exit Sub           ! nothing to print, exit
      Print S$
End Sub

Call Printit("Call a procedure")
Call Printit("")
```

See also

END SUB, SUB, DECLARE, EXTERNAL SUB, CALL (BASIC Program)

CASE

Synopsis

Control complex conditional and branching operations.

Syntax1

CASE [*num.lit* **|** [*num.lit* **TO** *num.lit*] **|** [**IS** *rel.op num.lit*]] {, [*num.lit* **|** [*num.lit* **TO** *num.lit*] **|** [**IS** *rel.op num.lit*]]} ...

Syntax2

CASE [*str.lit* **|** [*str.lit* **TO** *str.lit*] **|** [**IS** *rel.op str.lit*]] {, [*str.lit* **|** [*str.lit* **TO** *str.lit*] **|** [**IS** *rel.op str.lit*]]} ...

Syntax3

CASE ELSE

Parameters

num.lit is a numeric literal.

rel.op is a relational operator.

str.lit is a string literal.

Executable From Keyboard?

No.

Remarks

The **CASE** statement specifies the conditions for which its associated statements are executed. Multiple conditions, separated by comma may be specified.

CASE ELSE is optional and the associated statements are executed when no other **CASE** *expression* matched the value of the primary *expr*. If present, **CASE ELSE** must be the last **CASE** in the block.

Examples

```
! This is an example of the Case statement
Dim %1, Choice
Print 'CS'
Choice = 1
Do Until Choice = 6
      Select Case Choice
      Case 1
            Print @15,Choice + 15;"This is case 1"
      Case 2 To 3
            Print @15,Choice + 15;"This is case 2 or 3"
      Case IS > 3
            Print @15,Choice + 15;"This is case greater than 3"
      Case Else
            Print @15,Choice + 15;"This is default case"
      End Select
      Choice = Choice + 1
Loop
```

See also

SELECT CASE, ELSE, END SELECT

CHAIN

Synopsis

Transfer control to another program.

Syntax

CHAIN *filename* {, *num.expr* {, *num.var*}}

Parameters

filename is a string literal or expression containing a dL4 BASIC program filename which is optionally preceded by a relative or absolute directory pathname.

num.expr is an expression yielding a starting *stmt.no* in the new program to begin execution.

num.var is a variable of numeric type which is set to the *stmt.no* following the **CHAIN** in the current program.

Executable From Keyboard?

Yes.

Remarks

CHAINing to a null string terminates the current program. If the program was executed under **SCOPE**, the user will return to *command mode*. If the program was executed under **RUN**, then **RUN** will exit.

There are two types of **CHAIN** operations; *short* and *long*.

A *short* **CHAIN** transfers control from one BASIC program to another. All files remain open and common variables are passed using **COM** or **CHAIN READ** / **CHAIN WRITE**. A *short* **CHAIN** is performed if the *filename* is the name of an existing BASIC program, or begins with the string 'RUN' or 'run'.

A *long* **CHAIN** appends the supplied *filename* to the type-ahead buffer, exits the program to *command mode*, and processes type-ahead as though the command was entered from the keyboard.

Several commands may be within a *long* **CHAIN**, and they are executed in sequence. A *long* **CHAIN** is performed for dL4 programs whenever a *short* **CHAIN** fails. If *filename* begins with the character "\010\", "\031\", "\032\", or "\177\" a long chain will be performed after deleting that character.

Each command should be terminated with an **[EOL]** terminator. The number of characters that can be passed in this fashion is limited to the size of the user's input buffer.

Any *long* **CHAIN** which enters or passes input to *command mode* first closes all channels.

Any **CHAIN** terminates the current program.

The **CHAIN** statement is illegal in a procedure.

Examples

```
Chain "3/FILENAME"
```

```
Chain Q$,4000,B
```

See also

COM, CHAIN READ, CHAIN WRITE

CHAIN READ

Synopsis

Read variables from a previous program.

Syntax

CHAIN READ [*var.list* | = | *]

Parameters

var.list is a list of comma separated variables of any dL4 data types passed to this program.

Executable From Keyboard?

No.

Remarks

CHAIN READ specifies common variables passed to this program via **CHAIN WRITE** statements in a preceding program. Multiple **CHAIN READ** statements may be used, and they may be placed anywhere within a program. Variables listed in a **CHAIN READ** may not be dimensioned by a **DIM** statement. If a specified variable was not passed by a **CHAIN WRITE** statement, an error is generated.

CHAIN READ = causes all variables passed as common to be read into the program. All such variables must appear in the program at least once (even if not used).

CHAIN READ * functions like **CHAIN READ =** except that variables passed to, but not appearing in this program are ignored.

The **CHAIN READ** statement is ignored if executed. When a program passes data to another using **CHAIN WRITE**, the new program's **CHAIN READ** statements are executed during the **CHAIN** operation.

The actual **CHAIN READ** statements may be placed anywhere in a program, however the best method is to group them together at the beginning of a program near your **DIM** statements.

CHAIN READ statements may not be used together with **COM**.

The **CHAIN READ** statement is illegal in a procedure.

Examples

```
Chain Read A,B,C,X$
```

```
Chain Read *
```

See also

CHAIN READ IF, CHAIN WRITE, COM

CHAIN READ IF

Synopsis

Conditionally read variables from a previous program.

Syntax

CHAIN READ IF [*var.list* **|** = **|** *]

Parameters

var.list is a list of comma separated variables of any dL4 data types passed to this program.

Executable From Keyboard?

No.

Remarks

CHAIN READ IF specifies common variables passed to this program via **CHAIN WRITE** statements in a preceding program. Multiple **CHAIN READ IF** statements may be used, and they may be placed anywhere within a program. Variables listed in a **CHAIN READ IF** may not be dimensioned by a **DIM** statement. If a specified variable was not passed by a **CHAIN WRITE** statement, no error is generated.

CHAIN READ IF = causes all variables passed as common to be read into the program. All such variables must appear in the program at least once (even if not used).

CHAIN READ IF * functions like **CHAIN READ IF =** except that variables passed to, but not appearing in this program are ignored.

The **CHAIN READ IF** *s*tatement is ignored if executed. When a program passes data to another using **CHAIN WRITE**, the new program's **CHAIN READ IF** statements are executed during the **CHAIN** operation.

The actual **CHAIN READ IF** statements may be placed anywhere in a program, however the best method is to group them together at the beginning of a program near your **DIM** statements.

CHAIN READ IF statements may not be used together with **COM**.

The **CHAIN READ IF** statement is illegal in a procedure.

Examples

```
Chain Read If A,B,C,X$
```

```
Chain Read If *
```

See also

CHAIN READ, CHAIN WRITE, COM

CHAIN WRITE

Synopsis

Write variables to the program selected by the preceding **CHAIN** statement.

Syntax

CHAIN WRITE [*var.list* | *]

Parameters

var.list is a list of comma separated variables of any dL4 data types to be passed to the chained program.

Executable From Keyboard?

No.

Remarks

CHAIN WRITE statements specify variables to be passed as common to the next program. All variables specified must be dimensioned or otherwise have a value assigned to them in order to be passed. It is the responsibility of the receiving program to contain the necessary **CHAIN READ** statements to accept the data.

All variables are passed complete to their dimensioned length, such that strings with embedded nulls are passed in their entirety.

A **CHAIN WRITE** must not be directly executed. Multiple **CHAIN WRITE** statements may be used, and should only be placed as a group after a **CHAIN** or **SWAP** statement.

CHAIN WRITE * passes all variables in the program as common. It cannot be used with any other **CHAIN WRITE** statements.

CHAIN WRITE statements may not be used together with **COM**.

The **CHAIN WRITE** statement is illegal in a procedure.

Examples

```
Chain Write A,B,C,X$
```

```
Chain Write *
```

See also

CHAIN READ, COM

CHANNEL

Synopsis

Low-level statement to perform a driver-specific command.

Syntax

CHANNEL *chan.cmd, chan.expr {expr.list}*

Parameters

chan.cmd is an integer value indicating a driver-class dependent action.

chan.expr is a driver-class dependent channel expression.

expr.list is an arbitrary number of comma separated expressions or variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

Refer to the dL4 Files and Devices reference manual for information on channel commands supported by specific drivers.

Examples

```
Channel 38, #1, 1; Creationdate#
Channel 38, #1, 2; LastAccessdate#
Channel 38, #1, 3; Modificationdate#
```

See also

CHDIR

Synopsis

Change default directory to a specified path.

Syntax

CHDIR *str.expr*

Parameters

str.expr is an expression yielding a string value.

Executable From Keyboard?

Yes.

Remarks

The *str.expr* must be a legal filename of a directory.

Examples

```
Chdir C$
```

```
Chdir "../menu"
```

See also

CLEAR

Synopsis

Clear channels or initialize variables.

Syntax1

CLEAR {*chan.no* {**,** *chan.no*}...}

Syntax2

CLEAR *var.list*

Parameters

chan.no is a valid channel number.

var.list is an arbitrary number of comma separated variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

The *chan.no* expression is evaluated, truncated to an integer and used to select the *channel number* (0 to 99) to clear. Multiple channels, separated by comma may be cleared. If no *chan.no* is given, all opened files (Channels 0 to 99) are cleared. Record locks on the file are removed, the file header may be updated and the system file descriptor is released. A cleared channel is available for re-use for another file.

CLEAR differs from **CLOSE** in that it will always succeed: any I/O errors that occur while clearing the channel will be ignored. Additionally, if the channel was opened with **BUILD**, the file will be deleted. Refer to the dL4 Files and Devices reference manual for the file type or driver specific effects of **CLEAR**.

Clearing a variable initializes its value as if the variable had just been **DIM**ed. Numeric and binary values are zeroed. String values are set to nulls. Date values are set to a special value that indicates that it isn't a valid date.

dL4 programs generate an error when a specified *chan.no* is not currently open.

Examples

```
Clear #5,#8,#X+2
```

```
Clear
```

See also

CHANNEL, CLOSE

CLOSE

Synopsis

Close specified or all channels.

Syntax

CLOSE {*chan.no* {**,** *chan.no*}...}

Parameters

chan.no identifies a valid channel number.

Executable From Keyboard?

Yes.

Remarks

The *chan.no* expression is evaluated, truncated to an integer and used to select the *channel number* (0 to 99) to close. Multiple channels, separated by comma may be closed. If no *chan.no* is given, all opened files (Channels 0 to 99) are closed. Record locks on the file are removed, the file header may be updated and the system file descriptor is released. A cleared channel is available for re-use for another file.

Refer to the dL4 Files and Devices reference manual for file type or device specific effects of **CLOSE**.

dL4 programs generate an error when a specified *chan.no* is not currently open.

Examples

```
Close #1
```

```
Close #5,#8,#X+2
```

```
Close
```

See also

BUILD, CHANNEL, CLEAR, EOPEN, OPEN, ROPEN, WOPEN

COM

Synopsis

Specify common variables.

Syntax

COM {[%*prec* **|** *prec*%] ,} *var.list* { , [%*prec* **|** *prec*%], *var.list* } ...

Parameters

prec indicates the precision number defined for the variable.

var.list is an arbitrary number of comma separated variables of any dL4 data types.

Executable From Keyboard?

No.

Remarks

The **COM** statement allocates space and defines precision for variables which can be passed between programs. The form is identical to the **DIM** statement, except that all variables defined by **COM** are flagged as common and eligible to be passed during **CHAIN** or **SWAP**.

Precisions can be defined for the variables in the *var.list* by including the optional **%prec** or **prec%** precision. All further variables in the *var.list* will be at the last specified precision. The last supplied precision in a **COM** or **DIM** statement is used as the default for all automatically assigned variables.

All **COM** statements in a program must be executed before any statement which allocates or defines a new variable (**LET, DIM, IF,** etc.). Statements such as **REM, ESCSET, GOTO,** etc. which use no variables may precede **COM**. An error is generated if a **COM** statement is executed out of order.

Variables to be passed must be defined in a **COM** statement by each program that is to use them. Generally, two or more programs using a set of common variables will contain identical **COM** statements in order to pass the entire set between them. A program **CHAIN** may exclude certain variables in its common set, and these variables become unassigned. Similarly, the program may add variables to the set, and they will be allocated and initialized as done by a **DIM**. Numeric precision may not be changed between programs, but strings and arrays may be re-dimensioned to smaller sizes using **COM**.

CHAIN READ and **CHAIN WRITE** statements may not be used together with **COM**.

The **COM** statement is illegal in a procedure.

Examples

```
Com A$[19],B$[1],T4$[132]

Com C$[1762]

Com A[5],T$[120],D[23,14],%3,X[17]

Com 1%,A,B,%2,C,D,%3,E,F,4%G
```

See also

CHAIN READ, CHAIN WRITE, DIM

CONV

Synopsis

Convert binary data to decimal, or convert decimal data to binary.

Syntax1

CONV 0, *expr*, *num.var*

Syntax2

CONV 1, *var*, *num.expr*

Parameters

expr is an expression of string or binary data type.

num.var is a variable of numeric type.

var is a variable of string or binary data type.

num.expr is an expression yielding a numeric value to be converted.

Executable From Keyboard?

Yes.

Remarks

The **CONV** mode 0 statement extracts binary information from a *var* or *expr* and returns the value in decimal into a *num.var*. Additionally, using **CONV** mode 1, numeric information in a *num.expr* can be converted to binary and placed into a *var* or *expr*.

The *var* or *expr* specifies the binary string and must define a string of one to four characters. The *num.var* is the decimal numeric variable. When converting from or to a string, each character will be treated as an 8-bit byte and the upper 8-bits of the Unicode character will be treated as zeroes.

The valid numeric ranges, as well as the internal storage format, are determined by the length of the *var* or *expr* given. This variable would usually be subscripted to select the desired length, otherwise the dimensioned length of the string would be assumed. The following table compares the string length with the range of values that can be stored.

str.var	SIZE	DECIMAL
`B$[x,x]`	`1 byte`	`0 to 255`
`B$[x,x+1]`	`2 bytes`	`0 to 65535`
`B$[x,x+2]`	`3 bytes`	`0 to 16777215`
`B$[x,x+3]`	`4 bytes`	`-2,147,483,648 to 2,147,483,647`

The conversion process allows positive integers only to be represented in 1, 2, or 3 byte lengths. A negative value must be converted to a 4 byte length to retain its negative sign. Converting a negative value to a shorter length and back would result in a truncated positive integer different from the original value.

The 4 byte length described here is identical to the internal format of a double-precision integer numeric variable written to a file, and such a value could be read as a string and converted to numeric. The 2 byte length, however, is NOT compatible with the %1 format because it is unsigned. Signed values could be converted using 1, 2, or 3 byte lengths provided the program performs an adjustment for 16-bit two's complement notation.

Examples

```
100 Rem Convert binary to decimal D
110 Conv 0,A$[1,n],D
120 If D>R Then Let D=D-A
200 Rem Convert decimal D to binary
210 If D<0 Then Let D=D+A
220 Conv 1,A$[1,n],D
```

Size (n)	**Range (R)**	**Adjust by (A)**
1 byte	-128 to 127	256 (2^{8})
2 bytes	-32768 to 32767	65536 (2^{16})
3 byte	-8388608 to 8388607	16777216 (2^{24})

This method causes the upper bit of each string to be considered a sign bit, just as is done by **CONV** with the 4 byte length. In the case of 2 bytes, for example, the values 0 thru 32767 represent themselves, while 65535 thru 32768 represent -1 thru -32768.

See also

PRECISIONS, STRINGS

DATA

Synopsis

Define internal program data.

Syntax

DATA *num.lit* **|** *str.lit* {, *num.lit* **|** *str.lit*}...

Parameters

num.lit is a numeric literal value.

str.lit is a quoted sequence of characters.

Executable From Keyboard?

No.

Remarks

Each *num.lit* or *str.lit* is stored within the program as a numeric or string constant according to its type. Character strings must be quoted.

No other statement may follow **DATA** on the same program line. All text up to the end of the line is considered part of the DATA statement.

DATA statements may appear anywhere within a program and are ignored if executed, that is, they are treated like **REM** comments.

Each **DATA** statement may contain as many values as can be entered, up to the size of the input buffer.

Numeric data items must be separated by comma, but can be in decimal and E-notation. A comma cannot be part of a numeric item that will be read into a *num.var*.

For IRIS compatibility, a **%prec** declaration may be included before numeric values, but it will be ignored and discarded.

Examples

```
Data 200,300,400,500,600,700.25,800,23.45
```

```
Data "quoted string, has comma", "\015\\015\"
```

See also

READ, RESTORE

DECLARE

Synopsis

Declare a non-local procedure or provide a forward definition.

Syntax1

DECLARE { **EXTERNAL | INTRINSIC** } **SUB** *proc.name* {, ...}

Syntax2

DECLARE { **EXTERNAL | INTRINSIC** } **FUNCTION** *func.name* {, ...}

Parameters

proc.name is a valid procedure name.

func.name is a valid function name.

Executable From Keyboard:

No.

Remarks

EXTERNAL identifies the procedure as a separate secondary program unit with its own set of variables and program options.

INTRINSIC identifies the procedure as an internal language function, added by a developer and linked into the runtime. These functions are written in **C** and include some of the familiar IRIS calls, such as **$TRXCO**.

If the procedure is an internal procedure within the program unit, neither **EXTERNAL** nor **INTRINSIC** is declared. Internal procedures share everything with the surrounding program unit.

If any of the declared procedures are **EXTERNAL** and outside of the program, they must be in one of a declared list of library files. At runtime, those libraries declared with the **EXTERNAL LIB** statement are opened and the required procedures are dynamically linked into the calling program.

Examples

```
Declare Intrinsic Function FmtOf

Declare External Function IsPrime

Declare Function IsPrime

Declare External Sub VerifyDate(D$, ...)
```

See also

END FUNCTION, END SUB, SUB, EXTERNAL LIB, EXTERNAL SUB, FUNCTION

DEF FN

Synopsis

Define user function.

Syntax

DEF *func.name* ({*parm.list*}) = *expr*

Parameters

func.name is a valid function name.

parm.list is a comma separated list of expressions or variables of any data types to be passed to the calling function.

expr is an expression of the same type as the *func.name*.

Executable From Keyboard

No.

Remarks

Each user function must have a **DEF** statement executed before it can be used. User functions cannot be redefined using subsequent **DEF** statements within the same program unit.

The parenthesized *parm.list* is considered a dummy argument. The *expr* is the expression to be evaluated whenever the function is called. When this occurs, the actual argument supplied will be substituted for every occurrence of the dummy argument in the given expression. Any variable currently in use with the same name as the dummy argument is not affected by the function call.

A user function may call another user function in its definition, provided the called function has already been defined. User functions may be nested in this manner up to a maximum of 500 levels.

Examples

```
Def FNA(X)=(X^3)*(X^2)*X
Def DoIt(V)=(V^4)*FNA(V)  ! Nested FNA
Def Round(X)=SGN(X)*ABS(100*INT(X)+.5)/100
```

See also

EXTERNAL FUNCTION, FUNCTIONS, DECLARE

DEFINE RECORD

Synopsis

Define the record format for a file.

Syntax

DEFINE RECORD *chan.no*; *struct.var*

Parameters

chan.no is a valid channel number.

struct.var is a variable of structure data type.

Executable From Keyboard

Yes.

Remarks

The **DEFINE RECORD** statement is used to establish the record definition and data dictionary of a newly built Full-ISAM database file.

structvar is the name of a structure variable including **ITEM** "Fieldname" specifications for each member of the structure template. Refer to the dL4 Files and Devices reference manual for details on character and length requirements for field names.

The record layout of the file is structured according to the members of the given structure, i.e. types, sizes, and fieldnames.

No data records are written to the file by the **DEFINE RECORD** operation.

For example, given the following structure template:

```
Def Struct Customer      ! Define using 'fieldnames'
  Member Name$[25]    : Item "Name" ! supply database fieldnames.
  Member Address$[25] : Item "Addr"
  Member City$[25]    : Item "City"
  Member State$[2]    : Item "State"
  Member Zip$[10]     : Item "PostCode"
  Member 3%,Balance   : Item "CurrBal" : Decimals 2
End Def
```

and the following dim and build statements:

```
Dim Cust. As Customer
Build #5, "Customers" As "Full-ISAM"
```

the structure is mapped to the record layout of the file.

```
Define Record #5; Cust.
```

If no errors result, the record definition was accepted and written to the file.

Examples

```
Define Record #1;CustRec.
```

See also

ADD RECORD, SET

DEF STRUCT

Synopsis

Define a structure.

Syntax1

DEF STRUCT *struct.name*= {%*prec* **|** *prec*% ,} *var.list* {, { %*prec* **|** *prec*% ,} *var.list*} ...

Syntax2

DEF STRUCT *struct.name*

MEMBER {%*prec* **|** *prec*% ,} *var.list* {, { %*prec* **|** *prec*% ,} *var.list*} ...

.

.

.

END DEF

Syntax3

DEF STRUCT *struct.name* {: **ITEM** *id* } {:**RAW**}

MEMBER {%*prec* **|** *prec*% ,} *var.name* [: **ITEM** *id*] { **DECIMALS** *digits*}

.

.

.

END DEF

Syntax4

DEF STRUCT *struct.name* {: **KEY** *id option.list* }

MEMBER {%*prec* **|** *prec*%,} *var.name* [: **KEY** *id option.list*] { **DECIMALS** *digits*}

.

.

.

END DEF

Parameters

struct.name is a structure identifier.

prec indicates the precision number defined for the variable.

var.list is a list of comma separated variable names of any dL4 data types.

id is a string or a numeric literal identifying a fieldname or an item number.

var.name is a variable name.

digits is a numeric literal identifying the number of decimal digits.

option.list is a list of **UPPERCASE**, **DESCENDING**, **UNIQUE**, **VARLEN**, and/or **PACKED** key options, each preceded by a plus sign ("+").

Executable From Keyboard

No.

Remarks

DEF STRUCT is the start of the template for the definition of a complex data type. *struct.name* is a unique name tagged to this template. The name may be from one to thirty-two characters in length, and contain letters, digits, and underscores. **DEF STRUCT** does not actually allocate a structure using the supplied name, rather it informs the compiler to define a unique structure template tagged with this name.

var.name may be any type of variable declaration: string, numeric, date, binary, array or another structure. The syntax and function of **MEMBER** statements are nearly identical to that of **DIM**. Any **MEMBER** statement declaring a numeric or date member must specify the precision (**%prec** or **prec%**). Any **MEMBER** statement declaring an array is expressed as follows:

`Member` *`var.name`* **`[`***`num.expr`* `{, ...}`**`]`**

The subscript dimensions of the array may be given with [*num.expr* {, ...}]. Any **MEMBER** statement declaring a structure as a member is expressed as follows:

`Member` *`var.name`*`. {`**`[`***`num.expr`* `{, ...}`**`]`** `}` **`As`** *`struct.name2`*

*var.name***.** is the name of a structure whose members are defined by the structure definition *struct.name2*. *struct.name2* must be an existing *struct.name* which has been previously defined. The *var.name***.** may include array subscript dimensions as in [*num.expr* {, ...}], if *var.name***.** is to be an array of structures.

If Syntax1 is used, all **MEMBER** *var.list* names must be contained on a single program line. Syntax2, Syntax3, or Syntax4 may be used for readability, or when all of the members cannot be defined on a single line.

The **END DEF** statement defines the end of a structure definition.

Prior to using a structure, you must dimension one or more variables as a specific *struct.name*. The following general form is used to dimension a structure:

`Dim` *`variable`*`. { `**`[`***`expr`* `{, ... }`**`]`** `}` **`As`** *`struct.name`*

*variable***.** is an actual variable in the program which is to be referenced as a structure. The *variable* may include array subscript dimensions, if the *variable***.** is to be an array of structures.

As *struct.name* informs the compiler which compiled structure definition is to be used for *variable***.**

A structure definition itself may contain one or more structures, or arrays of structures. To define a structure which includes a structure, a **MEMBER** is expressed as follows:

`Member` *`name.`* `{ `**`[`***`expr`* `{, ... }`**`]`** `}` **`As`** *`struct.name2`*

*name***.** is the name within *struct.name2* whose members are defined by the structure definition *struct.name2*. *struct.name2* must be an existing *structname* which has been previously defined.

The names of structure members are distinct from any other names outside the structure; e.g. Data.Q$ is distinct from Q$ which is distinct from Data1.T.Q$.

The members of a structure are physically contiguous in memory, and are ordered in memory as defined by **DEF STRUCT**. Individual structure members cannot be re-dimensioned.

The **RAW** option enables special file access behavior similar to **OPTION FILE ACCESS RAW** but applied only to the members of the structure when used in an **ADD RECORD**, **READ RECORD**, or **WRITE RECORD** statement.

Examples

```
Def Struct Stat = %4,Population,City$[40]
Def Struct StatMem
  Member %4, Population
  Member City$[40]
End Def
```

See also

END DEF, MEMBER

DELETE INDEX

Synopsis

Delete an index in a file.

Syntax

DELETE INDEX *chan.no*, *index.no*

Parameters

chan.no is a valid channel number.

index.no is a numeric expression whose integer value identifies an index to be deleted in the file.

Executable From Keyboard?

Yes.

Remarks

When an index is no longer required, it may be deleted. It is driver dependent whether deleting an index is supported or results in savings of disk space. In most cases, it is assumed that the file structure will reuse the empty portion of the file.

If no errors result, the selected *index* was successfully deleted.

DELETE INDEX is not supported by any driver in dL4 revision 3.1 or earlier. For later revisions of dL4, refer to the dL4 Files and Devices reference manual to determine whether a particular driver supports **DELETE INDEX**.

Example

```
Delete Index #1,2
```

See also

ADD INDEX

DELETE RECORD

Synopsis

Delete current locked record from a file.

Syntax

DELETE RECORD *chan.no*

Parameters

chan.no is a valid channel number.

Executable From Keyboard?

Yes.

Remarks

The current record is deallocated, and all keys associated with this record are removed. The current record must be locked in order to be deleted.

If no errors result, the current record was successfully deleted.

Examples

```
Delete Record #2
```

See also

DIM

Synopsis

Allocate space for variables.

Syntax1

DIM {[%*prec* **|** *prec*%] ,} *var.list* { , [%*prec* **|** *prec*%], *var.list* } ...

Syntax2

DIM *var.list* **AS** *struct.name*

Parameters

prec indicates the precision number defined for the variable.

var.list is a list of comma separated variables of any dL4 data types. See Chapter 3 for information on variable types and subscripting variables.

struct.name is a structure identifier.

Executable From Keyboard?

Yes.

Remarks

The **DIM** statement allocates space and defines precision for variables which are considered local to the current program. The form is identical to the **COM** statement, except that all variables defined by **DIM** are not automatically passed during **CHAIN** statements unless specified using **CHAIN WRITE** and **CHAIN READ**.

Precisions can be defined for the variables in the *var.list* by including the optional **%prec** or **prec%** precision. All further variables in the *var.list* will be at the last specified precision. The last supplied precision in a **COM** or **DIM** statement is used as the default for all automatically assigned variables.

If the *var.list* contains an *str.var*, in the form *str.var$*[*num.expr]*, the *num.expr* within subscripts is evaluated, truncated to an integer, and used as the maximum size of the string variable in characters. Any attempt to store data beyond this maximum results in data truncation. String variables <u>must</u> appear in a **DIM** or **COM** statement before use by any other statement. They cannot be re-dimensioned unless the variable is deallocated (see the **FREE** statement).

If the *var.list* contains an *binary.var,* in the form *binary.var?*[*num.expr*], the *num.expr* within subscripts is evaluated, truncated to an integer, and used as the maximum size of the binary variable in 8-bit bytes. Any attempt to store data beyond this maximum results in data truncation. Binary variables <u>must</u> appear in a **DIM** or **COM** statement before use by any other statement. They cannot be re-dimensioned unless the variable is deallocated (see the **FREE** statement).

If the *var.list* contains a variable in the form *struct.var.* then **Syntax2** is used to dimension the variable as a structure of type *struct.name*. The variable may include array subscript dimensions, if it is to be an array of structures. The **AS** *struct.name* informs the compiler which compiled structure definition is to be used for *struct.var.* (see the **DEF STRUCT** statement).

If the *var.list* contains a *num.var* or *date.var* without subscripts, it is allocated at the current default precision as a simple numeric or date variable.

If the *var.list* contains a variable in the form *var.name*[*num.expr*], or *var.name*[*num.expr1,num.expr2*], it is allocated at the current default precision as a one or two dimensional array. An array can have up to 16 dimensions. The expression within subscripts are evaluated, truncated to integers, and used to select the size (number of elements) of the array. Variables specifying one expression result in a one-dimensional array (vector or list). Two expressions separated by a comma result in a two-dimensional array (matrix). Any array used in a program without specifically being mentioned in a **DIM** or **COM** statement is automatically dimensioned to [10] for each dimension.

It is considered good programming practice to define all variables (other than temporaries and variables to use the default precision) in a **DIM** or **COM** statement.

The final **%prec** or **prec%** executed in your program selects the default for any run-time variable assignments.

Examples

```
Dim Alpha$[26],Byte?[80],DayOfMonth#[31]
```

```
Dim CustInfo.[1000] As Customer
```

```
Dim State$[50,2],%3,X[17]
```

```
Dim %1,A,B,2%,C,D,3%,E,F,%4
```

See also

DEF STRUCT, COM

DO

Synopsis

Begin a program loop.

Syntax

DO

Parameters

None.

Executable From Keyboard?

No.

Remarks

Program loops may be established using the **DO** and **LOOP** statements as a means of blocking a set of repeated statements. These statements provide greater flexibility and looping control than **FOR / NEXT**.

The bare **DO** loop must have a specific termination statement such as **IF *condition* EXIT DO** as one of the blocked statements or an infinite loop will result.

Execution resumes at the statement following the **DO** and continues normally. Upon execution of the **LOOP** statement, execution resumes at the statement following the corresponding **DO**.

Unlike **FOR**, **DO** loops may nest indefinitely. In addition, each **DO** loop must contain exactly one matching **LOOP** statement. The compiler ensures that all loops are properly matched. Although not recommended, branching from outside to inside a **DO** loop will not cause an error, rather the program will remain in the loop until it terminates. The **DO** statement itself need not be executed to commence looping.

Examples

```
Do
      done = 1
      Print done
      If done Exit Do
Loop
```

See also

DO UNTIL, DO WHILE, EXIT DO, LOOP

DO UNTIL

Synopsis

Begin a loop to be performed as long as the expression is false.

Syntax

DO UNTIL *bool.expr*

Parameters

bool.expr is an expression evaluated to produce a boolean value.

Executable From Keyboard?

No.

Remarks

Program loops may be established using the **DO** and **LOOP** statements as a means of blocking a set of repeated statements. These statements provide greater flexibility and looping control than **FOR / NEXT**.

The **UNTIL** *expression* provides the loop with a specific termination condition. **UNTIL** provides for looping as long as the *expression* remains false - that is until it becomes true.

The optional **UNTIL** clause may be placed on either the line containing the **DO** or **LOOP** statement, depending upon when *expression* is to be tested. By placing the clause with **LOOP**, the developer ensures that at least one iteration is performed.

Execution resumes at the statement following the **DO** and continues normally. Upon execution of the **LOOP** statement, execution resumes at the statement following the corresponding **DO**.

Unlike **FOR**, **DO** loops may nest indefinitely. In addition, each **DO** loop must contain exactly one matching **LOOP** statement. The compiler ensures that all loops are properly matched. Although not recommended, branching from outside to inside a **DO** loop will not cause an error, rather the program will remain in the loop until it terminates. The **DO** statement itself need not be executed to commence looping.

Examples

```
Choice = 1
Do Until Choice = 4
      Print Choice
      Choice = Choice + 1
Loop
```

See also

DO, DO WHILE, LOOP, EXIT DO

DO WHILE

Synopsis

Begin a loop to be performed as long as the expression is true.

Syntax

DO WHILE *bool.expr*

Parameters

bool.expr is an expression evaluated to produce a boolean value.

Executable From Keyboard

No.

Remarks

Program loops may be established using the **DO** and **LOOP** statements as a means of blocking a set of repeated statements. These statements provide greater flexibility and looping control than **FOR / NEXT**.

The **WHILE** *expression* provides the loop with a specific termination condition. **WHILE** provides for looping as long as the *expression* remains true.

The optional **WHILE** clause may be placed on either the line containing the **DO** or **LOOP** statement, depending upon when *expression* is to be tested. By placing the clause with **LOOP**, the developer ensures that at least one iteration is performed.

Execution resumes at the statement following the **DO** and continues normally. Upon execution of the **LOOP** statement, execution resumes at the statement following the corresponding **DO**.

Unlike **FOR**, **DO** loops may nest indefinitely. In addition, each **DO** loop must contain exactly one matching **LOOP** statement. The compiler ensures that all loops are properly matched. Although not recommended, branching from outside to inside a **DO** loop will not cause an error, rather the program will remain in the loop until it terminates. The **DO** statement itself need not be executed to commence looping.

Examples

```
Choice = 1
Do While Choice < 4
      Print Choice
      Choice = Choice + 1
Loop
```

See also

DO, DO UNTIL, LOOP, EXIT DO

DUPLICATE

Synopsis

Copy a file.

Syntax

DUPLICATE *str.expr* {**AS** *driver-class* **|** *driver-name* }

Parameters

str.expr is a string literal or expression containing a source filename followed by a destination filename (space separated) each of which is optionally preceded by a relative or absolute directory pathname.

driver-class specifies the driver-class.

driver-name specifies the driver-name.

Executable From Keyboard?

Yes.

Remarks

If the destination file already exists, an exclamation point ("**!**") must be appended to the destination filename to overwrite the existing file.

If the file consists of two or more subfiles, each file will be copied. For example, an Indexed Contiguous file might consist of a data file ("source") and an index file ("source.idx"). These files would be copied to the destination filename ("destination" and "destination.idx"). Refer to the dL4 Files and Devices reference manual for more information on specific file types.

Examples

```
Duplicate "PAYROLL PAY1QTRBKUP"
Duplicate "/usr/ub/23/file /u/u1/23/file"
```

See also

EDIT

Synopsis

Format numeric and string expressions.

Syntax

EDIT *str.expr*, *str.var*; *expr.list*

Parameters

str.expr is an expression yielding a string value.

str.var is any destination string variable used to receive the formatted result.

expr.list is an arbitrary number of comma separated expressions or variables of string or numeric data types.

Executable From Keyboard?

Yes.

Remarks

The *str.expr* defines the format string to apply to the list of variables in the *expr.list*. Output is formatted according to the rules for the String Operator: **USING**.

Only numeric data is formatted, string data is copied exactly to the destination.

The **EDIT** statement is used to format string and numeric output. **EDIT** operates similar to **LET USING**; formatting output and storing the result in a string variable. Unlike **LET USING**, **EDIT** allows a list of arguments for the formatted result.

Examples

```
Edit "$#,##&.##",D$;T,E,F,"TAXES",T9
```

```
Edit A$,B$;"TOTAL DUE",Z,"BALANCE",Q,R$,T9
```

See also

LET USING

ELSE

Synopsis

Control conditional branching.

Syntax

ELSE {**IF** *bool.expr*}

Parameters

bool.expr is a expression evaluated to produce a boolean value.

Executable From Keyboard?

No.

Remarks

Inclusion of an **ELSE** or **ELSE IF** block is optional. **ELSE** must be the only statement on the line (except that it may be followed by a trailing **!** comment).

Statements to be executed on the *bool.expr* being true follow the **ELSE IF** on subsequent lines. All statements up to the associated **ELSE** or **ENDIF** are part of the true condition.

ELSE defines an optional block of *stmts* to execute when the corresponding Blocked-**IF** was false.

Examples

```
If (A=100 And B=200)
      Print A,B
Else If A=100
      Print B
Else
      Print A
End If
```

See also

IF, THEN, END IF

END

Synopsis

Terminate the program.

Syntax

END

Parameters

None.

Executable From Keyboard?

Yes.

Remarks

If the program was executed from the SCOPE Interactive Development Environment (IDE), an **END** statement causes program execution to cease and the user is returned to the SCOPE IDE following the prompt:

```
Ready
```

If the program was executed from another environment, such as the Operating System prompt, via the applicable **RUN** *filename* command, the user is returned to that environment.

Other statements may follow an **END**, and inclusion of an **END** is optional. If a program reaches its physical end of the program and no **END** statement exists, an implied **END** is performed.

END leaves the current program (with all variables) in the user's partition. All channels are closed automatically.

The **END** statement is illegal in a procedure.

Examples

```
End
```

See also

STOP, SUSPEND

END DEF

Synopsis

End a structure definition.

Syntax

END DEF

Parameters

None.

Executable From Keyboard?

No.

Remarks

The **END DEF** statement defines the end of a structure definition.

Examples

```
Def Struct StatMem
      Member %4, Population
      Member City$[40]
End Def
```

See also

DEF STRUCT

END FUNCTION

Synopsis

End a FUNCTION definition.

Syntax

END FUNCTION *return.expr*

Parameters

return.expr yields the value to be returned, which must match the data type of the function.

Executable From Keyboard?

No.

Remarks

END FUNCTION is used to mark the end of the definition of a multi-line function and provide the return value for the function.

The **EXIT FUNCTION** statement can be used to return from a function before reaching the **END FUNCTION** statement.

Examples

```
External Function IsPrime(N)
      Dim %2,I
      If N = 1 Exit Function 0                  ! not a prime number
      For I=2 To Sqr(N)
            If Not(N Mod I) Exit Function 0     ! not prime
      Next I
End Function 1                                  ! prime
```

See also

EXIT FUNCTION, EXTERNAL FUNCTION, FUNCTION

END IF

Synopsis

End conditional branch.

Syntax

END IF

Parameters

None

Executable From Keyboard?

No.

Remarks

END IF must be the only statements on the line (except that it may be followed by a trailing **!** comment).

END IF defines the end of a blocked **IF**.

An **ELSE IF** does not need an **END IF**.

Examples

```
If A=100
      Print A
      If J
            Write #3,R;A$
      Else
            Read #3,R;A$
      End If
End If
```

See also

IF, ELSE, THEN

END SELECT

Synopsis

End complex conditional branch

Syntax

END SELECT

Parameters

None.

Executable From Keyboard?

No.

Remarks

The compiler ensures that each **END SELECT** statement has a previous matching **SELECT CASE** statement.

Examples

```
Random (0)
Choice = INT(RND(4))
Select Case Choice
      Case 1
            Print "This is case 1"
      Case 2
            Print "This is case 2"
      Case Else
            Print "This is default case"
End Select
```

See also

CASE, SELECT CASE

END SUB

Synopsis

End a procedure definition.

Syntax

END SUB

Parameters

None.

Executable From Keyboard?

Yes.

Remarks

END SUB is used to mark the end of the definition of a procedure.

The **EXIT SUB** statement can be used to return from a procedure before reaching the **END SUB** statement.

Examples

```
External Sub DoIt(D$)
      Print D$
End Sub
```

See also

SUB, EXTERNAL SUB, EXIT SUB

END TRY

Synopsis

End a **TRY** block.

Syntax

END TRY

Parameters

None.

Executable From Keyboard?

No.

Remarks

END TRY is used to mark the end of a **TRY** block. Error branching is restored at the upon the completion of the block.

Examples

```
Dim %1, Chan
Chan = 2
Try
      Open #Chan,"cust.master"
      Print "Opened cust.master on channel "; Chan
Else If Spc(8) = 42          ! file not found
      Call "fm.cust", Chan
      Print "Attempting to open cust.master file again"
      Retry
Else
      Print "Unexpected Error: ";Spc(8); " at line ";Spc(10)
End Try
Print "Terminating program"
Close
```

See also

TRY

ENTER

Synopsis

Accept arguments into a procedure.

Syntax

ENTER *parm.list*

Parameters

parm.list is a list of variables associated with parameters passed, optionally followed by three dots ("...").

Executable From Keyboard?

No.

Remarks

The **ENTER** statement accepts argument variables from a **CALL** by *filename* to a saved BASIC program (subprogram) or can be used to process variable length parameter lists in a procedure.

The **ENTER** statement can be located on any line of the subprogram, but the variables cannot be used until the **ENTER** statement has been executed. This means that the **ENTER** statement should be at the beginning of the program in most cases.

The number and types of variables in the **ENTER** statement must match the **CALL** statement or function invocation exactly or an error message is displayed.

The *parm.lis*t may be defined as any combination of *variables*, depending on the requirements of the subprogram. The subprogram can only return data within arguments that are passed as variables, subscripted numeric variables, or matrix variables. A matrix variable in a **CALL**, a function reference, or an **ENTER** is given as a variable with empty subscripts; e.g. A3[].

If a subprogram is called with arguments, but no **ENTER** statement is executed, no error will occur and the arguments will not be changed. If a subprogram has no parameters, an **ENTER** statement with no parameters can be used to detect unnecessary arguments on the invoking **CALL** statement.

Subprograms called by *filename* and procedures may also accept a variable list of parameters. The compiler performs no type or parameter checking for subprograms and procedures defined with a variable list of parameters. Procedures with a variable list of parameters are defined in the following manner:

Sub *name* (*fixed.parms*, ...)	**Function** *name* (*fixed.parms*, ...)
Sub *name* (**...**)	**Function** *name* (...)

Checking is only performed during the runtime processing of any **ENTER** statement within the called subprogram or procedure. It is the sole the responsibility of the subprogram or procedure to check the passed parameters.

A caller's list of arguments is placed into a list to be processed by the actual subprogram or procedure. The general form of the **ENTER** statement when used for this purpose is:

Enter *expected.parameter* { **, ...** }

expected.parameter specifies the type of parameter expected by the procedure. If the next parameter in the list matches the supplied *expected.parameter*, it is extracted from the list and passed to the procedure. If not, an error is generated to the procedure which may decide to alter its course of action.

If additional parameters might follow, the **ENTER** statement <u>must</u> end with **...** This preserves any remaining arguments in the list passed by the caller. If the subprogram or procedure is certain that additional parameters are not in the list, or that an error should result if there are, do not terminate the **ENTER** statement with **...**

Examples

```
Call $PGM,B$,A,D$[4,7]        (from master program)
Enter B$,J,F$                 (from called subprogram)

! This is an example of the Enter Statement with
! a variable length parameter list
External Sub VerifyDate(D$, ...)
      Option Date Format Native
      Dim 2%, D#
      Dim %1, NoStatVar

      Try Enter R$, ... Else Dim R$[6]
      Dim %1
      Try Enter S Else S = 0; NoStatVar = 1

      Try
            Let D# = D$
            R$ = (Year(D#) Mod 100) * 10000 + Month(D#) * 100 +
                  MonthDay(D#) Using "&&&&&&"
      Else
            S = 1
      End Try
      If S And NoStatVar Error 38
End Sub

Call VerifyDate("06/05/97", S)
If S
      Print "Not a valid date"
Else
      Print "Valid date"
End If
```

See also

CALL, LIB, END, SUB, EXTERNAL SUB, FUNCTION, EXTERNAL FUNCTION

EOFCLR

Synopsis

Clear end-of-file branching.

Syntax

EOFCLR

Parameters

None.

Executable From Keyboard?

No.

Remarks

EOFCLR clears any special end-of-file branching in effect. Normal error processing is resumed. If an error branch is in effect from an **ERRSET**, **ERRSTM**, or **IF ERR**, it will be in control of further end-of-file errors.

Examples

```
Eofclr
```

See also

IF ERR, ERRSET, ERRSTM, EOFSET

EOFSET

Synopsis

Specify end-of-file error branching.

Syntax

EOFSET *label*: **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

EOFSET traps any further occurrence of error 52, "Record not written". If such an error occurs on any channel, the program will branch to the *label:* or *stmt.no* given in the **EOFSET** statement. **EOFSET** affects only this single error. Other errors are processed in the current error handling mode.

IF ERR, **ERRSET** and **ERRSTM** statements are used to trap all errors, including end-of-file. The **EOFSET** statement is used to override normal error branching for this special error.

EOFSET branching remains in effect until specifically cleared by **EOFCLR**. Other error branching disable functions do not clear this special branch.

Examples

```
Eofset 1050
```

```
Eofset NoData
```

See also

IF ERR, ERRSET, ERRCLR, ERRSTM, EOFCLR

EOPEN

Synopsis

Exclusively **OPEN** an existing file.

Syntax1

EOPEN *chan.no*, *file.spec.str* {**AS** *driver-class* **|** *driver-name* } {, {*chan.no*,} *file.spec.str* {**AS** *driver-class* **|** *driver-name*}} ...

Syntax2

EOPEN *chan.no*, *file.spec.items* **AS** *driver-class* **|** *driver-name* {, {*chan.no*,} *file.spec.items* **AS** *driver-class* **|** *driver-name*} ...

Parameters

chan.no identifies a valid channel number, which the program uses for subsequent references to the file.

file.spec.str, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

driver-class specifies the driver-class, instead of using a default driver-class derived from the file.spec.

driver-name specifies the driver-name, instead of using a default driver-class derived from the file.spec.

file.spec.items, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

Executable From Keyboard?

Yes.

Remarks

The **EOPEN** statement exclusively links a selected file to a channel.

EOPEN differs from **OPEN** in that the request will exclusively lock the file to the program. **EOPEN**, **OPEN**, **ROPEN** or **WOPEN** requests by other programs will not be allowed until the file is closed.

The operation of **EOPEN** is driver and operating system dependent. Refer to the dL4 Files and Devices reference manual to determine if and how **EOPEN** is supported for specific file types.

Examples

```
Eopen #1,"23/MMFILE", C$
Eopen #1,"23/MMFILE" As "Full-ISAM"
Eopen #2,"FILE1","FILE2",#10,"FILE4"
```

See also

BUILD, OPEN, ROPEN, WOPEN

ERASE

Synopsis

Perform driver-class dependent erase function.

Syntax

ERASE *chan.no*

Parameters

chan.no is a valid channel number.

Executable From Keyboard?

Yes.

Remarks

Refer to the dL4 Files and Devices reference manual for information on a specific driver.

Examples

```
! This is an example of the Erase statement
Dim s$[1]
Print 'CS'
W = 38 \ H = 12
Open #1,{" Windows ","TITL",W,H} As "Window"
Print #1; "Enter any character to Erase (Clear) Window ";
Read #1;S$
Erase #1
```

See also

CHANNEL

ERRCLR

Synopsis

Clear error branching.

Syntax

ERRCLR

Parameters

None.

Executable From Keyboard?

No.

Remarks

ERRCLR clears any error-branching in effect and returns normal error processing to the application. Normal error processing is to abort the current running program and output the error message text:

`Error in statement` *`stn`*`;`*`sub-stn`* `/ Text description of error`

Special end-of-file branching in effect from the **EOFSET** statement is not cleared by **ERRCLR**.

ERRCLR is used to clear automatic branch-on-error conditions previously set using **ERRSET**, **ERRSTM** and **IF ERR**.

Normal error termination does not close all opened data files.

Examples

```
Errclr
```

See also

EOFSET, ERRCLR, IF ERR, ERRSTM, ERRSET

ERROR

Synopsis

Generate a dL4 BASIC error.

Syntax

ERROR *num.expr*

Parameters

num.expr is an expression yielding an error number.

Executable From Keyboard?

No.

Remarks

The **ERROR** statement generates a dL4 error to the current running program. The specified error number is returned by **SPC(8)**, and forces an error event within a **TRY** block, procedure, or to any other error handler. The statement is helpful when writing procedures or user calls to provide a meaningful exit to the caller.

num.expr is any expression which, following evaluation, is truncated to an integer and returned to the application as an error number (event).

Application defined error numbers should have values >= 10,000.

Examples

```
Error E+10000
```

See also

ERRSET

Synopsis

Specify error branching.

Syntax

ERRSET *label*: **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

ERRSET is used to specify a label: or *stmt.no* to receive program control upon the occurrence of any BASIC error.

Error branching remains in effect until an **ERRCLR** is executed.

When the **ERRSET** statement is executed, any existing error branching from an **IF ERR**, or **ERRSTM** is reset to branch to the selected *stmt.no* upon occurrence of any error.

ERRSET does not affect the state of the special **EOFSET** branch on end-of-file error.

Examples

```
Errset 8000
```

```
Errset ItDied
```

See also

EOFSET, ERRCLR, IF ERR, ERRSTM

ERRSTM

Synopsis

Specify statement(s) to execute on an error.

Syntax

ERRSTM *stmt* { \ *stmt* } ...

Parameters

stmt is any valid dL4 BASIC statement.

Executable From Keyboard?

No.

Remarks

The **ERRSTM** statement specifies a line of statements to be executed upon the occurrence of any error.

Error statement processing remains in effect until an **ERRCLR** statement is executed.

When the **ERRSTM** statement is executed, any existing error branching from an **IF ERR**, or **ERRSET** is reset to perform the *stmts* following **ERRSTM** upon the occurrence of any error. Normal execution resumes at the next BASIC line, reserving all *stmts* following **ERRSTM** for when an error occurs.

ERRSTM must be the last statement of a multi-statement line.

ERRSTM has no effect on any special **EOFSET** end-of-file branch in effect.

Examples

```
Errstm Print "ERROR OCCURRED AT LINE:";Spc 10
Errstm Close \ Stop
Errstm If Spc 8 = 42 Stop Else !Success
```

See also

EOFSET, ERRCLR, IF ERR, ERRSET

ESCCLR

Synopsis

Clear any **ESCAPE** branching in effect.

Syntax

ESCCLR

Parameters

None.

Executable From Keyboard?

No.

Remarks

ESCCLR removes any special **ESC**ape branching or disabling in effect.

Previous **ESC**ape branching or disable set by **ESCSET**, **ESCSTM** or **ESCDIS** statements is disabled, and normal **ESC**ape termination of a program is resumed.

The **[ABORT]** character may be used to override and abort any program that has **ESC**ape disabled, or an **ESC**ape branch in effect.

Examples

```
Escclr
```

See also

ESCSET, ESCDIS, ESCSTM, IF ERR

ESCDIS

Synopsis

Disable escape events.

Syntax

ESCDIS

Parameters

None.

Executable From Keyboard?

No.

Remarks

The **ESCDIS** statement prevents unauthorized **ESC**ape termination of any BASIC program. Any pressing of the **ESC**ape key by the user is ignored.

ESCDIS remains in effect until an **ESCSET**, **ESCSTM** or **ESCCLR** is executed.

When the **ESCDIS** statement is executed, any existing **ESC**ape branching is reset to ignore further **ESC**ape characters.

The **[ABORT]** character may be used to override and abort any program that has **ESC**ape processing.

Examples

```
Escdis
```

See also

ESCSET

Synopsis

Enable branch to statement on escape events.

Syntax

ESCSET *label*: **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

ESCSET specifies a *label:* or *stmt.no* to receive program control upon pressing of the **ESC**ape key.

Escape branching remains in effect until an **ESCCLR** is executed.

The **[ABORT]** character may be used to override and abort any program that has **ESC**ape processing.

When the **ESCSET** statement is executed, any existing **ESC**ape branching from the **ESCSTM** or **ESCDIS** is reset to branch to the **ESCSTM** *stmt.no* upon the occurrence of an **ESC**ape.

ESCCLR is used to clear automatic branch-on-**ESC**ape and resume normal **ESC**ape processing. Normal **ESC**ape processing terminates the running BASIC program and produces a **STOP at** prompt on the screen:

```
Stop at statement xx;yy in program name
```

Normal **ESC**ape termination does not close all opened data files.

Note that **ESC**ape's function may be assigned to keys other than **ESC**ape itself, just as the **ESC**ape key may be assigned to perform some other function. The **ESC**ape statements described above will act upon any key currently defined as an **[ESCAPE]**.

Examples

```
Escset 8000
```

```
Escset ItDied
```

See also

ESCDIS, ESCCLR, ERRSET, IF ERR

ESCSTM

Synopsis

Specify statement(s) to execute on escape events.

Syntax

ESCSTM *stmt* { \ *stmt* } ...

Parameters

stmt is any valid dL4 BASIC statement.

Executable From Keyboard?

No.

Remarks

The **ESCSTM** statement specifies a line of statements to be executed upon the pressing of an **ESC**ape key.

ESCape statement processing remains in effect until an **ESCCLR** statement is executed.

The **[ABORT]** character may be used to override and abort any program that has **ESC**ape processing.

When the **ESCSTM** statement is executed, any existing **ESC**ape branching from the **ESCSET** or **ESCDIS** is reset to perform the *stmts* following **ESCSTM** upon the occurrence of any error. Normal execution resumes at the next BASIC line, reserving all *stmts* following **ESCSTM** for an **ESC**ape.

ESCSTM must be the last statement of a multi-statement line.

Note that **ESC**ape's function may be assigned to keys other than **ESC**ape itself, just as the **ESC**ape key may be assigned to perform some other function. The **ESC**ape statements described above will act upon any key currently defined as an **[ESCAPE]**.

Examples

```
Escstm Print "ESCAPE PRESSED AT LINE";Err(2)
Escstm Close \ Stop
Escstm Close \ Chain "MAINMENU"
```

See also

ERRSTM, ESCSET, ESCCLR

EXIT DO

Synopsis

Exit a **DO** loop.

Syntax

EXIT DO

Parameters

None.

Executable From Keyboard?

No.

Remarks

The **EXIT DO** statement gracefully exits a **DO** loop.

EXIT DO is the preferable method to terminate a **DO** loop when writing portable code. Branching out of a loop is never recommended.

Examples

```
Do
      done = 1
      Print done
      If done
            Exit Do
      End If
Loop
```

See also

DO, DO UNTIL, DO WHILE, LOOP, EXIT FOR

EXIT FOR

Synopsis

Exit a **FOR/NEXT** loop.

Syntax

EXIT FOR

Parameters

None.

Executable From Keyboard?

No.

Remarks

The **EXIT FOR** statement gracefully exits a **FOR** loop.

EXIT FOR is the preferable method to terminate a **FOR** loop when writing portable code. Branching out of a loop is never recommended, and may lead to stack overflows.

Examples

```
For I = 1 To 10
      If I > 5
            Exit For
      End If
      Print "i = ";I
Next I
```

See also

FOR, NEXT

EXIT FUNCTION

Synopsis

Exit a function.

Syntax

EXIT FUNCTION *return.expr*

Parameters

return.expr yields the value to be returned, which must match the data type of the function.

Executable From Keyboard?

No.

Remarks

EXIT FUNCTION provides an alternate means other than **END FUNCTION** to return to the routine that called the function. It is generally used in the body of the function upon meeting some condition.

Examples

```
External Function IsPrime(N)
      Dim %2,I

      If N = 1 Exit Function 0    ! not a prime number
      For I=2 To Sqr(N)
            If Not(Fra(N / I)
                  Exit Function 0   ! not prime
            End If
      Next I
End Function 1                      ! prime
```

See also

END FUNCTION, EXTERNAL FUNCTION, FUNCTION

EXIT SUB

Synopsis

Exit a subroutine.

Syntax

EXIT SUB

Parameters

None.

Executable From Keyboard?

No.

Remarks

EXIT SUB provides an alternate means other than **END SUB** to return to the calling program. It is generally used in the body of the subroutine upon meeting some condition.

Examples

```
External Sub DoIt(D$)
      If D$ = "" Then Print "Nothing to print." \ Exit Sub
      Print D$
End Sub

Call DoIt('CS')
Call DoIt("Print this.")
Call DoIt("")
```

See also

END SUB, SUB, EXTERNAL SUB

EXTERNAL FUNCTION

Synopsis

Define a function.

Syntax

EXTERNAL FUNCTION *func.name* ({*parm.list* })

Parameters

func.name is the function name.

parm.list is a list of variables associated with parameters passed, optionally followed by three dots ("...").

Executable From Keyboard?

No.

Remarks

EXTERNAL identifies the function as a separate secondary program unit which shares nothing with its surrounding program and any main program unit, except channels. It is an independent program unit within a program and visible to other program units both inside and outside of the program. Regardless of its physical location, it has its own set of variables, Lib directory, **DATA** statements, current precision, stacks, **OPTIONS**, etc.

The developer declares a function **EXTERNAL** whenever:

- The function is to share only variables and data passed by reference with the caller. It declares its own data, precisions and local variables which are independent of any surrounding program unit.
- The function sets its own parameters independent of the caller.
- The function shares nothing with the caller except parameters and channels.
- Other programs need to call the function.

A group of External functions (and subroutines) may be saved in a single program, called a library file. A program which has both an executable main program unit as well as External functions may also be referenced as a library by other programs. However, it is advisable to segregate shared External functions into library files which do not include a main program unit to ensure that they remain constant and available to other program units. An exception for compatibility purposes might be a function which is called by *filename* and therefore exists as a main program unit of the library file.

A function exits and returns a value to the caller when an **EXIT FUNCTION** or **END FUNCTION** statement is executed.

A *func.name* may be from one-to-thirty-two characters in length and must end with the type designation matching the data type returned from the function. Numeric data has no suffix, strings end with $, dates with # and binary variables end with ?. Structures may be passed and operated upon, but a function cannot return a structure.

Whenever a function is to be used before its definition within the current program unit or program, or physically resides in another program, a **DECLARE** statement must occur before its first use.

Functions may be written to allow the caller to pass other than a fixed list of parameters. Parameter types and number are not checked by the compiler or interpreter. Rather, it is left to the function to process each of the arguments passed by a caller. To define a function of this type, the following general forms are supported:

```
Function name (...)
```

The definition of the function itself specifies '**...**' informing the compiler and interpreter to leave the parameter type and number checking to the function.

It is also permitted to define a function which has a known (required) list of parameters, followed by additional optional parameters. Optional parameters must be the last parameters in the function definition. The following example requires a numeric parameter and a string parameter, followed by an optional number of parameters.

```
Function func.name (parameter1, parameter2$, ... }
```

Functions of this type utilize the **ENTER** statement to accept optional parameters.

The **EXTERNAL FUNCTION** statement is illegal in a procedure.

Examples

```
External Function IsPrime(N)
      Dim %2,I
      If N = 1 Exit Function 0    ! not a prime number
      For I=2 To Sqr(N)
            If Not(Fra(N / I))
                  Exit Function 0   ! not prime
            End If
      Next I
End Function 1                      ! prime
```

See also

FUNCTION, SUB, EXTERNAL SUB, END FUNCTION, EXIT FUNCTION, DECLARE

EXTERNAL LIB

Synopsis

Declare library file(s).

Syntax

EXTERNAL LIB *filename* {, *filename* } ...

Parameters

filename is a string literal or expression containing a dL4 BASIC program filename which is optionally preceded by a relative or absolute directory pathname.

Executable From Keyboard?

No.

Remarks

If any of the declared procedures are **EXTERNAL** <u>and outside of the program</u>, they must be in one of a declared list of library files. At runtime, those libraries are opened and the required procedures are dynamically linked into the calling program. The linking process consists of scanning the lists of **EXTERNAL LIB** *filenames* loading and linking any required secondary program units until all **EXTERNAL** references are resolved. **EXTERNAL LIB** declarations may be placed anywhere within a program, and they affect the entire program.

filename is the name of a *saved* program which is to be opened during the dynamic linking phase when the current program is first executed. Whenever a program is loaded, via **CHAIN**, **RUN**, **CALL "*filename*"** or **SWAP**, all references to **EXTERNAL** procedures <u>must be resolved</u> prior to execution. An error is generated if any **EXTERNAL** procedure references are unresolved.

Examples

```
External Lib "OldCalls"
```

```
External Lib "OldCalls",L$
```

See also

EXTERNAL SUB, EXTERNAL FUNCTION, DECLARE

EXTERNAL SUB

Synopsis

Define a subroutine.

Syntax

EXTERNAL SUB *proc.name* (*parm.list*)

Parameters

proc.name is the procedure name.

parm.list is a list of variables associated with parameters passed, optionally followed by three dots ("...").

Executable From Keyboard?

No.

Remarks

EXTERNAL identifies the subroutine as a separate secondary program unit which shares nothing with its surrounding program and any main program unit, except channels. It is an independent program unit within a program and visible to other program units both inside and outside of the program. Regardless of its physical location, it has its own set of variables, Lib directory, **DATA** statements, current precision, stacks, **OPTIONS**, etc.

Variables are passed to procedures by reference, not by name. Expressions are passed to procedures by value. Normally, procedures need not concern themselves with what was passed, however the caller should be aware of the appropriate calling sequence. If a procedure updates, or returns a value in, a referenced variable, that operation will be lost if the caller passed an expression.

Sometimes the caller may intentionally wish to pass an expression to prevent the update of a local variable passed by reference. This may be accomplished by converting the variable into an expression. For example, the variable 'numeric' can be made an expression in the *parm.list* by denoting it as (numeric + 0) and 'string$' can be denoted as (string$ + "").

The developer declares a subroutine **EXTERNAL** whenever:

- The subroutine is to share only variables and data passed by reference with the caller. It declares its own data, precisions and local variables which are independent of any surrounding program unit.
- The subroutine sets its own parameters independent of the caller.
- The subroutine shares nothing with the caller, except parameters and channels.
- Other programs need to call the subroutine.

A group of External subroutines (and functions) may be saved in a single program, called a library file. A program which has both an executable main program unit as well as External subroutines may also be referenced as a library by other programs. However, it is advisable to segregate shared External subroutines into library files which do not include a main program unit to ensure that they remain constant and available to other program units. An exception for compatibility purposes might be a subroutine which is called by *filename* and therefore exists as a main program unit of the library file.

It is also permitted to define a subroutine which has a known (required) list of parameters, followed by additional optional parameters. Optional parameters must be the last parameters in the subroutine definition. The following example requires a numeric parameter and a string parameter, followed by an optional number of parameters.

```
External Sub proc.name (parameter1, parameter2$, ... }
```

Subroutines of this type utilize the **ENTER** statement to accept optional parameters.

The **EXTERNAL SUB** statement is illegal in a procedure.

Examples

```
External Sub DoIt(D$)
      Print D$
End Sub
```

See also

DECLARE, SUB, EXTERNAL FUNCTION, FUNCTION

FOR

Synopsis

Loop while incrementing or decrementing a numeric variable through an interval.

Syntax

FOR *num.var* = *num.expr1* **TO** *num.expr2* {**STEP** *num.expr3*}

Parameters

num.var is a variable of numeric data type.

num.expr1 is an expression yielding a numeric value, which is assigned as the initial value of *num.var*.

num.expr2 is an expression yielding a numeric value, which is used as the limit value for *num.var*.

num.expr3 is an expression yielding a numeric value, which determines the amount that the *num.var* is increased or decreased during each iteration of **NEXT**.

Executable From Keyboard?

No.

Remarks

The **FOR** statement is used in conjunction with the **NEXT** statement for repetitive statement execution. Statements between the **FOR/NEXT** may be re-executed a given number of iterations. This repetitive execution is known as a *loop*.

The *num.var* is termed the *index* variable and is used to control the *loop*.

Looping is initiated by setting the *index* variable equal to the *initial* value. At this point, a preliminary check is made to see if the *loop* should be executed at all. If: *initial* > *final* AND *step* > 0, or *initial* < *final* AND *step* < 0, then the *loop* statements are not executed and the program resumes following the associated **NEXT** statement (**NEXT** with same *index* variable). If not, execution continues with the statement following the **FOR**.

Upon execution of the associated **NEXT** statement, the *step* value is added to the *index*. If the new *index* will exceed the *final* value, normal program execution resumes at the statement following the **NEXT** with the *index* variable set to the terminating value; e.g. if the *step* value is such that the *index* will eventually equal the *final* value, the loop terminates with *index* = *final*+*step*. Otherwise, *index* is set to the first value causing the loop to terminate.

A step value of zero will produce an infinite loop.

FOR/NEXT loops may be nested if certain precautions are taken. The following is an example of valid nesting:

```
10    For A=1 To 10
20          For B=1 To 5
30                For C=B+1 To 4*A
40                      ! Statements
50                Next C
60          Next B
70    Next A
```

The range of **FOR/NEXT** loops may not overlap. The following is an example of invalid nesting:

```
10    For I=1 To 10
20          For J=I+1 To 20
30                ! Statements
40          Next I
50    Next J
```

Example

```
For I=1 To 3
      ! Statements
Next I
```

Initially, I is set to 1, *final* is set to 3 and *step* defaults to 1. Each execution of the **NEXT** first checks if (I+1)>3. When (I+1)>3, execution resumes following the **NEXT** with I=4.

```
10 For I=10 To 1 Step -2
20    ! Statements
30 Next I
```

Initially, I is set to 10, *final* is set to 1, and *step* is set to -2. Each execution of the **NEXT** first checks if (I-2)<1. When (I-2)<1, the loop terminates, in this example with I=0. The loop is performed 5 times for I = 10, 8, 6, 4, and 2.

See also

DO, EXIT FOR, NEXT

FREE

Synopsis

Deallocate (undimension) variable(s).

Syntax1

FREE *var.list1*

Syntax2

FREE ALL {**EXCEPT** *var.list2*}

Parameters

var.list1 is an arbitrary number of comma separated variables of any dL4 data types.

var.list2 is an arbitrary number of comma separated variables of any dL4 data types, which are not freed.

Executable From Keyboard?

Yes.

Remarks

A freed string variable should not be referenced.

Freeing a numeric variable causes the next reference to re**Dim** it to the last precision level.

Examples

```
Free N
Free N,P$,D#
Free All Except N,P$
```

See also

DIM

FUNCTION

Synopsis

Define a multi-line procedure which returns a value.

Syntax

FUNCTION *func.name* ({*parm.list* })

Parameters

func.name is the function name.

parm.list is a list of variables associated with parameters passed, optionally followed by three dots ("...").

Executable From Keyboard?

No.

Remarks

FUNCTION declares a function which operates as a separate program block within a program unit which returns a value to the caller. A Function may also operate upon, and return values through, supplied parameters passed by reference.

A function exits and returns a value to the caller when an **EXIT FUNCTION** or **END FUNCTION** statement is executed.

A *func.name* may be from one-to-thirty-two characters in length and must end with the type designation matching the data type returned from the function. Numeric data has no suffix, strings end with $, dates with # and binary variables end with ?. Structures may be passed and operated upon, but a function cannot return a structure.

Whenever a function is to be used before its definition within the current program unit or program, or physically resides in another program, a **DECLARE** statement must occur before its first use.

Functions may be written to allow the caller to pass other than a fixed list of parameters. Parameter types and number are not checked by the compiler or interpreter. Rather, it is left to the function to process each of the arguments passed by a caller.

To define a function of this type, the following general forms are supported:

```
Function name (...)
```

The definition of the function itself specifies '**...**' informing the compiler and interpreter to leave the parameter type and number checking to the function.

It is also permitted to define a function which has a known (required) list of parameters, followed by additional optional parameters. Optional parameters must be the last parameters in the function definition. The following example requires a numeric parameter and a string parameter, followed by an optional number of parameters.

```
Function func.name (parameter1, parameter2$, ... }
```

Functions of this type utilize the **ENTER** statement to accept optional parameters.

Examples

```
Function IsPrime(N)
      If N = 1 Exit Function 0      ! not a prime number
      For I=2 To Sqr(N)
            If Not(Fra(N / I))
                  Exit Function 0   ! not prime
            End If
      Next I
End Function 1                      ! prime
```

See also

END FUNCTION, EXIT FUNCTION, EXTERNAL FUNCTION, EXTERNAL SUB, SUB

GET

Synopsis

Obtain driver-class dependent information from a channel.

Syntax

GET *chan.expr var.list*

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is an arbitrary number of comma separated variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

Refer to the dL4 Files and Devices reference manual for information on using **GET** with a specific driver.

Examples

```
Get #2,1,-1;Opt,name$
```

See also

SET

GOSUB

Synopsis

Unconditionally branch to a subroutine

Syntax

GOSUB *label*: **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

The **GOSUB** statement is used in conjunction with the **RETURN** statement to provide traditional BASIC subroutines. New programs should use the **CALL** and **SUB** statements which support named subroutines with parameters.

GOSUB, like **GOTO**, performs an unconditional branch to the specified line number. Unlike **GOTO**, however, the statement number performing the **GOSUB** is saved. Upon the execution of a **RETURN** statement, normal execution would resume at the statement following the **GOSUB**. **GOSUB** and **RETURN** are not paired as are **FOR/NEXT**; i.e. any **RETURN** will return to the last **GOSUB** issued.

Subroutines may be nested to eight levels or the number of levels defined by the program **OPTION** statements before a **RETURN** must be executed.

Failure to return from all nested levels can cause an error.

See the **RETURN** statement for variations on returning from subroutines.

Examples

```
Gosub 1000
```

```
Gosub Start_Input:
```

See also

CALL, GOTO, OPTION GOSUB NESTING, RETURN, SUB

GOTO

Synopsis

Unconditionally branch to a statement.

Syntax

GOTO *label*: **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

The **GOTO** statement is used to unconditionally branch to another statement within a program and resume normal execution there.

GOTO always transfers control to the first sub-statement on the specified line, and the line must exist. For transfer to any sub-statement on a line, see the **JUMP** statement.

The verb **GOTO** may also be entered as **GO TO** .

A statement that performs a **GOTO** itself may cause an infinite loop terminated only by **ESC**ape, or **ESC**ape Override **[ABORT]**.

Examples

```
Goto 1000
```

```
Goto BEGIN:
```

See also

JUMP, GOSUB

IF

Synopsis

Control conditional branching.

Syntax1

IF *bool.expr* {**THEN}** *stmt* {**ELSE** *stmt*}

Syntax2

IF *bool.expr* {**THEN**}

{*stmt*}...

ENDIF

Syntax3

IF *bool.expr*

{**THEN** }

{ *stmt* } ...

{ **ELSE IF** *bool.expr*

{ *stmt* } ... }

{ **ELSE**

{ *stmt* } ... }

ENDIF

Parameters

bool.expr is an expression evaluated to produce a boolean value.

stmt is any valid dL4 BASIC statement.

Executable From Keyboard?

No.

Remarks

The **IF** statement tests a *boolean expression* and conditionally performs statements based on the *expression* being true or false. See "Boolean Expression" in chapter 5 for a description of boolean expressions.

The **IF** statement will test the given *expression* for validity and execute the *stmt* following **THEN** if and only if the *expression* proves true. If the *expression* is not true, the statement is checked for the **ELSE** operator. If found, the *stmt* following the **ELSE** will be executed; otherwise, the program continues normally.

Entry of the **THEN** operator is generally optional.

The *stmt* following **THEN** and/or **ELSE** may be any BASIC statement or a *stmt.no* alone implying a **GOTO** *stmt.no*. The verb **GOTO** can also be specifically entered, with the same result. Either **THEN** or **GOTO** must be supplied in order to perform a **GOTO**.

A false **IF** condition continues execution with the next statement line, instead of with the next *sub-stmt.no*. When an **IF** is true, all remaining statements on the line are executed. An **ELSE** can be used to override this feature. Both of the following examples perform the same function. In the first example, both statements are executed if the expression A=100 is true. If false, execution resumes on the next line of statements.

The second example performs a **GOTO** the next statement if the reverse expression is true, otherwise the **ELSE** is executed following with the remaining statements on the line:

```
If A=100 Gosub 1000 \ Goto 1000
If A<>100 Goto 120 Else Gosub 1000 \ Goto 1000
```

The **OPTION** statement **OPTION IF BY STATEMENTS** can be used to force execution of only one statement for each non-blocked **IF** statement without an **ELSE**. In the first example above, the statement "GOTO 1000" is executed for any condition. With the default of **OPTION IF BY LINES** in effect, the statement "GOTO 1000" is executed only for the true condition.

A blocked-**IF** structure provides a more convenient method of executing several statements for both the true and false conditions for applications.

Blocked-**IF** statements are assumed whenever an **IF** or **ELSE IF** statement ends following an expression. No *stmts* may follow the expression excepting an optional **REM**.

Inclusion of an **ELSE** or **ELSE IF** block is optional. The **THEN** statement is completely ignored and can be omitted, if desired. **THEN**, **ELSE**, and **ENDIF** must be the only statements on their line (except that they may be followed by a trailing **REM** comment).

Statements to be executed on the expression being true follow the **IF** (or **THEN**) on subsequent lines. All statements up to the associated **ELSE** or **ENDIF** are part of the true condition.

ELSE defines an optional block of *stmts* to execute when the corresponding Blocked-**IF** was false.

ENDIF defines the end of a blocked **IF**.

Blocked-**IF**s can be nested to any level, and are indented like **FOR-NEXT** loops for readability. There must be an **ENDIF** for every blocked-**IF** in the program. The integrity of the blocked-**IF**s is checked by the **RUN**, **CHAIN**, **SAVE**, **VERIFY** and **CHECK** commands. Once checked, a program is flagged OK eliminating further verification until a statement is changed within a program.

Examples

```
If A*5 > B*10 Then Call PrintReport
If Len(A Using A$ TO ".") >132 Print #3;
If A-5 Then 340 Else If J=100 Gosub 100 Else Stop
If C$[1,1]<=Z$[10,10] And C$<>"X" Then 280
If (J=10 Or C=20) And (T=10 OR F=12) Stop
```

```
Blocked-IF:
```

```
If A=100 And B=200
      Print A,B
Else If A=100
            Print B
      Else
            Print A
End If
```

See also

ELSE, THEN, END IF, GOTO, JUMP, OPTION IF, SELECT CASE

IF ERR 0 | 1

Synopsis

Specify a statement to execute when an error occurs.

Syntax

IF ERR 0 **|** 1 {*stmt*}

Parameters

stmt is any valid dL4 BASIC statement.

Executable From Keyboard?

No.

Remarks

IF ERR 0 is used to specify a line of statements to be executed upon the occurrence of any error.

IF ERR 1 may also be used to specify an error branch, however a separate error number is not reserved for **[INTERRUPT]**.

When an **IF ERR 0** statement is executed, any existing error branching from a previous **IF ERR 0** , **ERRSET**, or **ERRSTM** is reset to the *stmts* following the **IF ERR 0**. Normal execution resumes at the next BASIC line, reserving all *stmts* following **IF ERR 0** for error processing.

ESCape is also trapped generating a special Error code to the application.

ESCSTM, **ESCSET**, **EOFSET**, and **ESCDIS** statements can be used in addition to **IF ERR**.

Error statement processing remains in effect until an **ERRCLR** or **IF ERR 0** statement is executed without any trailing *stmt*.

IF ERR statements must be the last statement of a multi-statement line.

IF ERR statements are illegal in a procedure.

Examples

```
If ERR 0 Gosub 1000
```

```
If ERR 0
```

See also

EOFSET, ERR, ERRSET, ERRSTM, ERRCLR, JUMP

INPUT

Synopsis

Retrieve keyboard or channel input.

Syntax1

INPUT [{**LEN** *num.expr1*;} {**TIM** *num.expr2*;} {**KEY** *str.var*; } { (*num.expr3*, *num.var*)} {*crt.expr*;} {*str.lit*} *var.list*] ...

Syntax2

INPUT *chan.expr* [{**LEN** *num.expr1*;} {**TIM** *num.expr2*;} {**KEY** *str.var*; } { (*num.expr3*, *num.var*)} *var.list*] ...

Parameters

num.expr1 is an expression yielding the maximum number of characters to read.

num.expr2 is an expression yielding the tenth-seconds time limit.

str.var receives the input terminating character, if any.

num.expr3 is an expression yielding an input mode.

num.var is a variable of numeric data type.

crt.expr indicates a CRT expression used to position the cursor.

str.lit is a literal text prompt message.

var.list indicates a list of variables of any dL4 data types, excluding structures, binary, and array data types, to receive input.

chan.expr is a driver-class dependent channel expression.

Executable From Keyboard?

Yes.

Remarks

If a *chan.expr* is specified, the input for this statement will be satisfied by the selected *channel*. If the *chan.expr* is not specified (or the selected *channel* is not open), input will be taken from the standard input channel, usually the keyboard. The standard input channel can also be specified by using channel -3. When requesting input from a *chan.expr*, the *crt.expr*, *num.expr1*, *num.expr2*, and *str.lit* options should not be used.

If a *crt.expr* is specified, it is evaluated and output. Typically, a *crt.expr* is used to position the cursor on the screen and/or clear lines, etc. prior to the request for input. Use of a *crt.expr* will suppress the normal prompt unless a specific *str.lit* is specified.

If a *str.lit* is specified, the default prompt-message **?** is replaced by the literal text within quotes. A null prompt "" suppresses the output of the prompt-message as does the inclusion of any *crt.expr*.

If a **LEN** *num.expr1;* is specified, the *num.expr1* is evaluated, truncated to an integer and set as the maximum number of characters to be accepted for input. Unless a special input mode (such as binary input) is in effect, the **[ENTER]** character may be used to terminate a character limited input prior to exhausting the specified character count. If *num.expr1* is greater than 16384, then input can be terminated only by the **[ENTER]** character and at most (*num.expr1* - 16384) characters will be accepted.

If a **TIM** *num.expr2;* is specified, the *num.expr2* is evaluated, truncated to an integer and set as the number of tenth-seconds to wait for input. If no input is seen within the specified interval, a system **SIGNAL** is sent to the program with the actual number of characters entered. A **SIGNAL 5** statement should immediately follow to prevent overflowing the communication buffer. If timeout signals have been disabled by an "**OPTION INPUT TIMEOUT SIGNAL OFF**" statement, a timeout will cause an error. If *num.expr2* equals -1, the input will timeout immediately.

Both a **TIM** *num.expr1;* and **LEN** *num.expr2;* can be specified on the same **INPUT** statement.

Length or time limits may also be specified using *num.expr2*. A special *num.expr3* value is provided to read the contents of the terminal's input buffer and is used by programs to read parameters entered on a command line. Two different mechanisms exist to invoke control features.

```
(num.expr3, num.var)    control  with a returned response
```

The num*.expr3* is evaluated and truncated to an integer. The second parameter must be a *num.var* and will be set following the **INPUT** as the response.

If the *num.expr3* evaluates to zero, the entire contents of the input buffer is selected as the input. The *num.var* is not set to any value in this mode. Typically, this mode is used within a program that can accept its input from a command line. To read the last command line, the input must be performed prior to any other **INPUT** or **PRINT** statements which corrupt the input buffer.

If the *num.expr3* evaluates to a positive value, the program is suspended for that number of tenth-seconds or until the **[ENTER]** character is entered terminating the input. The actual number of tenth-seconds that were spent waiting for **INPUT** is returned as a positive value in *num.var*. If no **[ENTER]** character (return) is received within the specified interval, the *num.var* is set to the negative of the specified tenth-second wait interval and any input characters are passed to the **INPUT** *var.list*.

If the *num.expr3* evaluates to a negative value, the value is converted to a positive number selecting the maximum number of characters to be accepted for input. -5 causes the system to wait for the input of 5 characters. The actual number of input characters is returned in the *num.var*. The **[ENTER]** character may be used to terminate a character limited input prior to exhausting the specified character count.

GENERAL OPERATION OF DATA INPUT

Following the parsing of the optional parameters, the program is suspended while data is read from the standard input; usually the terminal. Characters previously entered (and buffered) are processed first.

Characters are echoed (for keyboard input) unless echo is disabled by the previous entry of the **[TOGGLEECHO]** character (normally **CTRL E**), the '**IOEE**' mnemonic, or a **SYSTEM 9** statement.

If the **INPUT** is not satisfied, the program is suspended until the **[ENTER]** character (return) is entered, the specified character limit is reached, or a time-out occurs on timed input. When any of these conditions occurs, the program resumes operation and begins processing input into the variables defined in the *var.list*. The **[ESCAPE]** or **[ABORT]** characters will terminate input and abort the statement.

SYSTEM 26 and **27** alter the operation of character limited input. Normal operation is to automatically resume execution of the program when the limiting number of characters have been processed. Executing a **SYSTEM 27** forces character limited **INPUT** to require entry of the **[ENTER]** character (return). When the limit is reached, the terminal's bell is sounded and extra characters (except for edit keys) are ignored. **SYSTEM 26** resets character limited input to operate normally, that is, resume execution when the limiting number of characters have been processed.

No special processing is performed on the characters received. Data is passed to the program exactly as received from the driver (see the dL4 Files and Devices reference manual)..

When *binary input* **IOBI** (or **SYSTEM 14**) is enabled, all characters are passed directly to the program. All character input processing for **[ENTER]**, **[ESCAPE]**, **[BACKSPACE]**, etc. is suspended and the program must process all input data.

WARNING: When using Binary Input, it is possible to lock the terminal if your program does not provide a way to terminate itself. If you lock a terminal, use another port to **HALT** or otherwise terminate the locked program.

Cursor tracking can be enabled by printing a 'BCTRACK' mnemonic as the final character of *str.lit* or in a preceding **PRINT** statement (assuming there is no *str.lit* string).

When a *str.var* is specified in the *var.list*, all characters are copied up to, but not including the **[ENTER]** character. If the input is larger than the specified *str.var*, the extra input characters are discarded. If the input does not fill up the destination *str.var*, a zero-byte terminator is placed after the last character of data. If "KEY" is specified, then the **[ENTER**] character will be returned in *str.var*.

If a *num.var* is specified in the *var.list*, the input characters are converted to numeric and stored into the *num.var*. An error is generated if the input is not numeric or contains characters other than digits **+ - .** or **E** notation. If error branching is in effect, the **MSC(1)** function (Last **INPUT** Element) may be used to determine which input item was in error. For example:

```
10 Errset 40
20 Input A,B,C,D
30 End
40 Print "ERROR IN INPUT VARIABLE";Msc(1)
```

The user would enter the item or items, separating multiple items with a comma "," or **[ENTER]**. If too many items are entered, a non-abortive error is generated and the extra items are ignored.

Numeric values may be entered in scientific notation; however, commas are not allowed within a numeric item; e.g. 1,200 must be entered as 1200. To abort the **INPUT** statement, press **ESC**ape.

Examples

```
Input Tim 10; Len 30; "CUSTOMER NAME >"A$
```

```
Input @10,23;"Press [RETURN]" T$
```

```
Input (-1,K)  "Enter a single character "A$
```

```
Input "4 numbers w/ comma ? "A,B,C,D
```

See also

SYSTEM, READ

INTCLR

Synopsis

Clear interrupt event branching.

Syntax

INTCLR

Parameters

None.

Executable From Keyboard?

No.

Remarks

INTCLR restores normal operation with respect to user interrupts. **[INTERRUPT]**, **SIGNAL 1**, and **SEND** no longer automatically interrupt the program and branch to a specific **INTSET** statement number.

Examples

```
Intclr
```

See also

INTSET, SIGNAL, SEND

INTSET

Synopsis

Enable branch to statement on interrupt events.

Syntax

INTSET *label*: **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

INTSET sets the selected *label:* or *stmt.no* to receive control each time an interrupt character is pressed or a message is waiting to be received. The **[INTERRUPT]** action may be assigned to any character, but it is normally defined as **CTRL-C**. **INTCLR** removes the branching, and further interrupt requests or messages are ignored.

A program branch is defined to transfer execution to a pre-defined statement when either an 'interrupt' character is pressed or a message is transmitted to your port via the **SEND** or **SIGNAL** statements.

The interrupt handling routine can do any processing desired and return to the main program as if the branch never occurred. Secondary interrupts are inhibited until the program clears the initial interrupt. This is done using the **ERR(3)** function, which also yields the original interrupted statement number. Generally, an interrupt handling routine loops until all interrupts or messages are received. The main body of the program is resumed using the statement:

```
stmt.no Jump ERR(3)
```

or

```
stmt.no Jump ERR(3);ERR(7)
```

The latter form is required if multi-statement lines are used within the program.

The interrupt function should not use the **ERR(3)** function other than shown above unless it is re-entrant and stacks multiple return locations.

Examples

```
Intset 1000 ! Branch on Signal, CTRL C
Intset USER ! Branch on Signal, CTRL C
```

See also

INTCLR, SEND, SIGNAL

JUMP

Synopsis

Transfer control immediately to another location.

Syntax

JUMP *stmt.no* {; *sub.stmt*} {, *num.var*}

Parameters

stmt.no is a numeric expression whose integer value is a statement line number.

sub.stmt is a numeric expression that identifies a sub-statement in a statement.

num.var is a variable of numeric type that is set to the statement number following **JUMP**.

Executable From Keyboard?

No.

Remarks

The *stmt.no* is any *num.expr* which, after evaluation is truncated to an integer and used as the statement number to branch to. The optional *sub.stmt* is any *num.expr* which, after evaluation is truncated to an integer and used as the sub-statement on that line. **JUMP** performs an unconditional branch to the selected statement (and sub-statement). On multi-statement lines, sub-statements are numbered starting at 1.

If the optional *num.var* is supplied, it will be set to the statement number of line following **JUMP**. This is similar to the **GOSUB** statement, as a subsequent **JUMP** to this variable will essentially perform a **RETURN**. The *num.var* will is set to zero when the **JUMP** is the last statement of a program.

JUMP statements are in no way affected by the **RENUMB** command. Therefore, they are not an acceptable substitute for **GOTO** or **GOSUB** when a literal *stmt.no* can be used.

JUMP is best used in conjunction with system functions that supply statement numbers, retaining the program's ability to be renumbered.

The **JUMP** statement is illegal in a procedure.

Examples

```
Jump K*10
```

```
Jump Spc(10)
```

```
Jump ERR(1);ERR(4),J
```

See also

ERR, ESCSET, ERRSET, INTSET, GOSUB, GOTO

KILL

Synopsis

Delete file(s).

Syntax

KILL *filenames* {**AS** *driver-class* **|** *driver-name* } {, *filenames* {**AS** *driver-class* **|** *driver-name* } } ...

Parameters

filenames is a string literal or expression containing one or more space separated filenames.

driver-class specifies the driver-class.

driver-name specifies the driver-name.

Executable From Keyboard?

Yes.

Remarks

If an error occurs, the statement is aborted and any remaining filenames within the *str.lit* or *str.expr* are not deleted. Furthermore, other *filenames* are not processed.

The result of deleting a file that is currently in use or open is operating system dependent. On some operating systems, an error will be generated. On other operating systems, the effect is to remove the entry of the *filename* from the system directory preventing it from being opened again. When the last user closes the file, the system releases the disk space. Prior to closing, all types of access, including extending the file, is permitted.

Examples

```
KILL "23/ABC 23/DEF"
```

```
KILL A$,B$,C$
```

See also

LET

Synopsis

Assign values to variables.

Syntax1

{**LET**} *var.name* = *expr* { ; *var.name* = *expr* } ...

Syntax2

{**LET**} *str.var* = *str.expr* **TO** *str.expr* {: *num.var*}

Syntax3

{**LET**} *str.var* = *num.expr* **USING** *str.expr* {,*str.expr* ...}

Parameters

var.name is a variable name.

expr is a series of constants, variables, functions, and operators to define a desired computation.

str.var is a variable of string data type.

str.expr is an expression yielding a string value or a string variable.

num.var is a variable of numeric data type.

Executable From Keyboard?

Yes.

Remarks

The type of *expr* must match that of *var.name* except for the following cases:

if *var.name* is numeric, then *expr* must either be numeric or a string expression that begins with a number in character form.

if *var.name* is a string, then *expr* must either be a string, a number, or a date.

if *var.name* is a date, then *expr* must either be a date or a string expression that begins with a date in character form.

In each of the special cases, *expr* will be converted to the type of *var.name*.

If *var.name* is a structure variable, then *expr* must be a structure variable whose members match the types of the members of *var.name*.

The **LET** verb is optional, and is assumed when not entered. Although entry of the **LET** verb is optional, it is printed whenever the program is listed.

Multiple assignments may appear on a single line separated by semicolons.

```
Z=100;Q=1;N=0;A$="TXXX"
```

Numeric formatting is performed within a **LET** statement with the **USING** operator. This is functionally equivalent to the **EDIT** statement.

```
Let D$=X Using "##,###.##"
Let E$=X Using "##,###.##",Y,Z
```

In the above examples, X is formatted into the **USING** string. This string is then assigned to the *str.var*. If the *str.var* is not **DIM**ed as large as the **USING** string, the **USING** string is truncated. This will result in a loss of the corresponding right most digits of X.

Note that the **USING** operator is not part of the **LET** statement, but is instead a general purpose operator that can be used wherever a string expression is accepted and in any statement.

The **TO** operator allows assignment of string data to terminate upon encountering a given *str.expr*. The *str.expr* may be a single or multiple character string. The optional *num.var* returns the character position at which assignment stopped.

```
Let N$="ABCDEF%GHIJKL"
Let S$=N$ To "%":K
returns:  S$="ABCDEF",K=7
```

If the optional *num.var* is used, only the first character of the second *str.expr* will be used to perform the search. This form of the **TO** operator is recognized only in the **LET** statement.

Examples

```
Let V=1
Let T$=1/3
Let A=42;T=17;R7=91
Let B[7]=(A*T)+(R7/4) Using "#####"
Let A$="1234565";T=A$;B$=A$ To "45":T1
Let D#="January 2, 1996 11:00"
```

See also

DEF STRUCT, COM

LIB

Synopsis

Specify alternate directories to locate program files.

Syntax

LIB *str.expr* | *num.var*

Parameters

str.expr is an expression yielding a string value or a string variable which indicates a space-separated list of relative or absolute directory pathnames.

num.var is a variable of numeric data type which is set to a single directory number.

Executable From Keyboard?

No.

Remarks

A value of -1 may be used to clear a defined library logical unit.

The library unit is the first unit searched by **CALL** for a subprogram file, unless the subprogram filename itself specifies a full pathname.

SPC 23 is used to determine the current library logical unit, however its return value is only valid when the library logical unit is numbered.

Examples

```
Lib -1
```

```
Lib "pgms menus"
```

See also

CHAIN, OPTION CHAIN ALTERNATE DIRECTORIES, SWAP

LINE

Synopsis

Draw a line on a display device.

Syntax

LINE {*chan.no*;} {@*x1,y1*;} **TO** @*x2,y2*; { **TO** @*x2,y2*; } ...

Parameters

chan.no identifies a valid channel number.

x1,y1 are the column, row coordinates of the start of a line.

x2,y2 are the ending column, row coordinates of a line.

Executable From Keyboard?

Yes.

Remarks

Line drawing is a function of the window and printer drivers. If running on a character terminal, your terminal description file must contain a definition for the mnemonic **#,#LINETO**.

If **@***x1,y1* is not specified, the current cursor position is assumed.

TO is a keyword which must be followed by the ending coordinate position of the line segment.

Examples

```
Line @3,3; To @30,3;
Line @3,3; To @3,9; TO @30,9;
Line To @30,1;
```

See Also

BOX

LOOP

Synopsis

End a **DO** loop block.

Syntax

LOOP { **WHILE** *bool.expr* **|** **UNTIL** *bool.expr* }

Parameters

None.

Executable From Keyboard?

No.

Remarks

The **WHILE** or **UNTIL** *bool.expr* provides the loop with a specific termination condition. **WHILE** provides for looping as long as the *bool.expr* remains true, whereas **UNTIL** provides for looping as long as the *bool.expr* remains false - that is until it becomes true.

The optional **WHILE** or **UNTIL** clause may be placed on the line containing the **LOOP** statement to ensure that at least one iteration is performed.

Upon execution of the **LOOP** statement, execution resumes at the statement following the corresponding **DO** if the *bool.expr* is true. If the bool.expr is false, execution resumes at the statement following the **LOOP**.

Each **LOOP** must have exactly one matching **DO** statement. The compiler ensures that all loops are properly matched. Although not recommended, branching from outside to inside a **DO** loop will not cause an error, rather the program will remain in the loop until it terminates. The **DO** statement itself need not be executed to commence looping.

Examples

```
Do
      done = 1
      Print done
      If done Exit Do
Loop
```

See also

DO, DO UNTIL, DO WHILE, EXIT DO

MAP

Synopsis

Assign a logical index or an item number to an index or field name.

Syntax

MAP *chan.expr str.expr*

Parameters

chan.expr is driver-class dependent channel expression.

str.expr is an expression yielding a string value.

Executable From Keyboard?

Yes.

Remarks

Often it is necessary to work with a subset of fields within a database or provide for later changes in the field content or order within the file. The **MAP** statement allows a program to 'marry' a structure definition to the current file's data dictionary.

This kind of dynamic record access not only insulates the application from certain modifications to the file structure, but also could be used by individual programs to limit record accesses to only those fields which are directly used. Depending on the format of the underlying record data (which is subject to the rules of the actual file being driven; FoxPro, etc.), this may circumvent unnecessary data conversion and thereby boost performance.

MAP can also be used to define the logical index or directory number used within the application. This statement allows a program to be written using a hard-coded directory number, which is then logically mapped to the physical directory number within the file.

Examples

```
Map #2, 0, 0, -1; "CustNum"
Map #2, 0, 1, -1; "Name"
Map #2, 0, 2, -1; "YtdSales"
Map #2,1; "ByCustNum"         ! map ByCustNum key to index # 1
```

See also

MAP RECORD

MAP RECORD

Synopsis

Assign an alternate item number mapping.

Syntax

MAP RECORD *chan.no* **AS** *struct.name*

Parameters

chan.no is a valid channel number.

struct.name is a structure tag name which was defined using **DEF STRUCT**.

Executable From Keyboard?

Yes.

Remarks

Often it is necessary to work with a subset of fields within a database or provide for later changes in the field content or order within the file. The **MAP RECORD** statement allows a program to 'marry' a structure definition to the current file's data dictionary.

struct.name is the name of a template **DEF STRUCT** structure definition which is to be aligned with the fieldnames of the database, or named index within the database. *struct.name* members must have **ITEM** fieldname or directory name definitions.

MAP RECORD defines an alternate item number mapping at run-time. This statement allows a custom (sub-) record schema for record access, but does so dynamically by the item's fieldname.

The fieldnames given within the Customer structure are used to align each member to its current item number within the file. For example, if the field "Addr", which is item 1 in the structure, is currently item 4 in the physical record, a **MAP RECORD** would cause the driver to perform the necessary item-number translation so that any further access to item 1 will actually access item 4.

This kind of dynamic record access not only insulates the application from certain modifications to the file structure, but also could be used by individual programs to limit record accesses to only those fields which are directly used. Depending on the format of the underlying record data (which is subject to the rules of the actual file being driven; FoxPro, etc.), this may circumvent unnecessary data conversion and thereby boost performance.

Examples

```
Map Record #2 As CUSTREC
```

See also

MAP

MAT =

Synopsis

Copy an entire matrix.

Syntax

MAT *destination.var.mat* **=** *source.var.mat*

Parameters

destination.var.mat is any destination numeric matrix variable.

source.var.mat is any source numeric matrix variable.

Executable From Keyboard?

Yes.

Remarks

The *destination.var.mat* must be at least as large as the *source.var.mat*. In the following example, matrix A is dimensioned as [5,5] and matrix B as [6,6]:

`Mat B=A` is acceptable.

`Mat A=B` Is illegal since A is not large enough to contain all of the elements in B.

The copy is performed element by element. An error or integer truncation can occur if the precisions are not compatible. Row and column zero are not copied. **MAT =** cannot be used to copy single element arrays.

Examples

```
Mat T=D0
Mat T[4,4] = D9
Mat T[5]=G
```

See also

DIM, FOR, NEXT

MAT +

Synopsis

Add elements from two matrices.

Syntax

MAT *destination.var.mat* **=** *source.var.mat1* **+** *source.var.mat2*

Parameters

destination.var.mat is any destination numeric matrix variable.

source.var.mat1 is any source numeric matrix variable.

source.var.mat2 is any source numeric matrix variable.

Executable From Keyboard?

Yes.

Remarks

The two matrices being added must be exactly the same dimensions (rows and columns). The *destination.var.mat*, if not already defined, is dimensioned at the current default precision for the same number of rows and columns as the *source.var.mat*. An error or integer truncation can occur if the precisions are not compatible. Row and column zero are not added.

The same *matrix* variable may appear on both sides of the equation.

A[X,Y]=A[X,Y]+B[X,Y]

The sum, matrix D, of matrix A and matrix B is:

D[X,Y]=A[X,Y]+B[X,Y]

for each *matrix* element.

Examples

```
Mat T=D0+A9
```

```
Mat D0=D0+J
```

See also

MAT *

Synopsis

Multiply elements of two matrices.

Syntax

MAT *destination.var.mat* **=** [*source.var.mat1* **|** [*num.lit*)]] *source.var.mat2*

Parameters

destination.var.mat. is any destination numeric matrix variable.

source.var.mat1 is any source numeric matrix variable.

num.lit is a numeric literal.

source.var.mat2 is any source numeric matrix variable.

Executable From Keyboard?

Yes.

Remarks

MAT * performs a multiplication, establishing a new matrix equal to the product of two matrices. Scalar multiplication allows each element of a matrix to be multiplied by a constant.

Following the rules of matrix multiplication, if we multiply matrix A dimensioned [X,Y] by matrix B dimensioned [R,S], then the resulting matrix will be dimensioned [X,S]. An error or integer truncation can occur if the two precisions are not compatible. Row and column zero elements are not multiplied.

The same matrix variable may not appear on both sides of the equation.

Scalar multiplication causes each element of the given matrix to be multiplied by the value of the *num.lit*. The *num.lit* must be in parentheses, and immediately follow the equal sign (=).

Examples

```
Mat D=A*B
```

```
Mat Q=X*X
```

```
Mat C=(5)*A
```

See also

MAT CON

Synopsis

Create a constant matrix.

Syntax

MAT *destination.var.mat* **=** **CON** { "["*num.expr1*{, *num.expr2* }"]" }

Parameters

destination.var.mat is any destination numeric matrix variable.

num.expr1 is a numeric expression yielding a dimension.

num.expr2 is a numeric expression yielding a dimension.

Executable From Keyboard?

Yes.

Remarks

Each element of the *destination.var.mat* is set to the constant value one. Row and column zero are not set.

The optional *num.expr1* and *num.expr2* are evaluated, truncated to integer and used to select a new working size. The total number of elements in the new size cannot exceed that of the old. A single element *array* can be converted to a *matrix* or vice versa as long as the total number of elements does not exceed the original **DIM**ensioned size. For example, a [4,4] matrix has 25 actual elements and could be re-declared as **CON[25]**.

A constant other than one can be accomplished using a combination of the **CON** function and Scalar multiplication:

```
Mat A=CON \ Mat B=(5)*A \!Fill B with 5's.
```

Any array created by a **MAT** statement with a single dimensions assumes a second dimension of one. For example, Mat C= ZER[15] and Mat C = ZER[15,1] are equivalent.

Examples

```
Mat A=CON
```

```
Mat D0=CON[7,X/2]
```

See also

MAT IDN

Synopsis

Create an identity matrix.

Syntax

MAT *destination.var.mat* **= IDN** { "["*num.expr1* {, *num.expr2* } "]" }

Parameters

destination.var.mat is any destination numeric matrix variable.

num.expr1 is a numeric expression yielding a dimension.

num.expr2 is a numeric expression yielding a dimension.

Executable From Keyboard?

Yes.

Remarks

The matrix function **`IDN`** establishes an identity matrix of all zeroes with a diagonal of ones.

Any matrix multiplied by an identity matrix of the same size results in the original matrix. For example: If matrix A is dimensioned [3,3] and matrix B is an identity matrix also dimensioned [3,3], the result of: Mat C=A*B produces matrix C equal to A. Row and column zero are not affected by **`IDN`**.

The optional *num.expr1* and *num.expr2* are evaluated, truncated to integer and used to select a new working size for the *array*. The total number of elements in the new size cannot exceed that of the old. A single element *array* can be converted to a *matrix* or vice versa as long as the total number of elements does not exceed the original **DIM**ensioned size. For example, a [4,4] matrix has 24 actual elements and could be re-declared as **`IDN[25]`**. An identity *array* is an array of all zeros.

Any array created by a **MAT** statement with a single dimensions assumes a second dimension of one. For example, Mat C= ZER[15] and Mat C = ZER[15,1] are equivalent.

Examples

```
Mat Q=IDN
```

```
Mat T=IDN[4,4]
```

```
Mat A8=IDN[X,Y]
```

See also

DIM

MAT INPUT

Synopsis

Assign keyboard/file input to a matrix.

Syntax

MAT INPUT {*chan.expr*} *var.list*

Parameters

chan.expr is driver-class dependent channel expression.

var.list is a list of comma separated numeric matrix variables.

Executable From Keyboard?

Yes.

Remarks

MAT INPUT is used to assign values to an entire matrix. The values are accepted from either keyboard (operator) input, or through a channel (file or device).

Execution of a **MAT INPUT** statement pauses the program after output of a **?** to your terminal. The program is then suspended and data input is accepted. The user would enter all matrix items, separating each item with either a comma **,** or **[ENTER]** (return). **MAT INPUT** does not complete until all elements have been accepted.

The *array* elements are assigned by rows, starting with [1,1] thru [1,n], then continuing with [2,1] thru [2,n], etc. Row and column zero are not assigned. For example, a 4 by 4 matrix might be entered as:

```
17,42,87,12 <-
18,14,26,14 <-
15,0,18,29 <-
34,29,86,69 <-
```

Using **MAT INPUT** from a *channel* is similar to terminal **MAT INPUT**, except the data is read from the channel and must <u>include</u> row and column zero elements. The data must be separated by either commas or **[EOL]** (return), and cannot be in the format generated by a **MAT PRINT #**.

Any array created by a **MAT** statement with a single dimensions assumes a second dimension of one. For example, Mat C= ZER[15] and Mat C = ZER[15,1] are equivalent.

Example

```
Mat Input T
Mat Input A,B[4,10],C
Mat Input #3;X
Mat Input #2,R,20;E1,E2
```

See also

INPUT, MAT PRINT

MAT INV

Synopsis

Invert a matrix.

Syntax

MAT *destination.var.mat* = **INV**(*source.var.mat*)

Parameters

destination.var.mat is any destination numeric matrix variable.

source.var.mat is any source numeric matrix variable.

Executable From Keyboard?

Yes.

Remarks

The matrix function **INV** establishes one square matrix as the inverse of another.

Only square matrices (number of rows = number of columns) may be inverted. Both matrices must also be the same precision and dimension. Row and column zero are not affected by **INV**.

The **DET** function supplies the determinant of the last matrix inverted by your program, e.g. if two matrices are inverted before the **DET** function is used, the determinant returned will be from the second inversion.

Examples

```
Mat C=INV(A)
```

```
Mat R7=INV
```

See also

DET, DIM

MAT PRINT

Synopsis

Print contents of matrix(ces).

Syntax

MAT PRINT {*chan.expr* } *var.mat.list* { , **|** ; }

Parameters

chan.expr is a driver-class dependent channel expression.

var.mat is a list of comma or semicolon separated numeric matrix variables.

Executable From Keyboard?

Yes.

Remarks

Each *var.mat* is printed in character form without subscripts. Each variable may be followed by either a comma (,) or a semicolon (;). A comma will cause the matrix variable preceding it to be spaced using comma fields. These are generally 15 characters long. A semicolon will cause minimal spacing between elements. Elements are normally preceded by a space or "-", indicating negative or positive, and will be followed by one space. When all items in a matrix row have been output, two blank lines are output to produce double spacing between rows.

Row and column zero elements are only printed for **MAT PRINT** when the data is directed through a *channel*.

If a *channel* is specified to **MAT PRINT**, output is attempted to that channel. If the selected channel is not open, output is sent to the terminal.

Examples

```
Mat Print A
Mat Print I,J
Mat Print X;Y;Z;
Mat Print #3,T;H1,S1
```

See also

MAT RDLOCK

Synopsis

Read an array, matrix or string with locking.

Syntax

MAT RDLOCK *chan.expr var.list*

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

MAT RDLOCK transfers data into any dL4 data type. The operation is similar to a **READ** statement, except that an entire *array* or *matrix* is transferred; including row and column zero elements. If the specified *var* is a string, its entire specified length is transferred including zero-byte terminators.

If the variable in the list is a simple *num.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a *str.var*, its size may be controlled by subscripts. All characters are transferred including zero-bytes if support by the file type and driver (refer to the dL4 Files and Devices reference manual).

MAT RDLOCK transfers data and unconditionally locks the record.. The data record remains locked until a non-locking operation is performed by that same program to the same channel. While a record is locked, other users will be unable to access the record.

MAT RDLOCK is identical to **MAT READ** omitting the trailing semicolon.

See the **MAT READ** statement for details on the transfer of data to different types of files.

Examples

```
Mat Rdlock #3,R1,100;A
```

```
Mat Rdlock #C,R;A$
```

See also

MAT READ

MAT READ

Synopsis

Read a matrix from DATA or a channel.

Syntax1

MAT READ *chan.expr var.list* { *;* }

Syntax2

MAT READ *var.mat.list*

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

";" unlocks the record after a successful **MAT READ**.

Executable From Keyboard?

Yes.

Remarks

Syntax 1:

MAT READ transfers data into any dL4 data type. The operation is similar to a **READ** statement, except that an entire *array* or *matrix* is transferred; including row and column zero elements. If the specified *var* is a string, its entire specified length is transferred including zero-byte terminators.

If the variable in the list is a simple *num.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a *str.var*, its size may be controlled by subscripts. All characters are transferred including zero-bytes if support by the file type and driver (refer to the dL4 Files and Devices reference manual).

The optional semicolon (;) terminator eliminates the automatic record-lock applied to the supplied *record* in the *chan.expr*. Applications may also utilize **MAT RDLOCK** for operations with locking transfers.

Syntax 2:

MAT READ attempts to transfer data into each dL4 data type listed in the statement. Transfer of each element terminates at a comma (,) or at the end of the **DATA** statement. The format of the data is left to the user. Attempting to read string data into a numeric variable produces the error **DATA** of wrong type (numeric/string).

MAT READ transfers data sequentially from **DATA** statements until the entire matrix has been assigned. Row and column zero are not read.

See the **READ** and **DATA** statements for other rules governing reading from **DATA** statements.

Examples

```
Mat Read #3,R1,100;A,B$,C[12]
Mat Read #C,R;A$
Mat Read A[2,2], B$
Mat Read B$, J
```

See also

READ, DATA, MAT WRITE, READ

MAT TRN

Synopsis

Transpose a matrix.

Syntax

MAT *destination.var.mat* **= TRN**(*source.var.mat*)

Parameters

destination.var.mat is any destination numeric matrix variable.

source.var.mat is any source numeric matrix variable name.

Executable From Keyboard?

Yes.

Remarks

The matrix function **TRN** is used to establish one matrix as the transposition of another.

Transposition causes each element [X,Y] of the original matrix to be moved to element [Y,X] of the transposed matrix. Note that this also causes the dimension of the transposed matrix to be the reverse of the original. For example:

```
Original matrix [3,4]         Transposed matrix [4,3]
1     2     3     4           1     5     9
5     6     7     8           2     6     10
9     10    11    12          3     7     11
                              4     8     12
```

An error or integer truncation can occur if the two matrix precisions are not compatible. Row and column zero are not affected by **TRN**.

Any array created by a **MAT** statement with a single dimensions assumes a second dimension of one. For example, Mat C= ZER[15] and Mat C = ZER[15,1] are equivalent.

Examples

```
Mat C=TRN(A)
Mat R7=TRN
```

See also

DIM

MAT WRITE

Synopsis

Write a variable to a channel.

Syntax

MAT WRITE *chan.expr* *var.list* { *;* }

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

";" unlocks the record after a successful **MAT WRITE**.

Executable From Keyboard?

Yes.

Remarks

MAT WRITE transfers data from any dL4 data type to the file opened on the supplied *chan.expr*. The operation is similar to a **WRITE** statement, except that an entire *array* or *matrix* is transferred; including row and column zero elements. If the specified *var* is a string, its entire specified length is transferred including zero-byte terminators.

If the variable in the list is a simple *num.var*, the transfer size is controlled by the **DIM**ensioned size and precision

If the variable in the list is a *str.var*, its size may be controlled by subscripts. All characters are transferred including zero-bytes if support by the file type and driver (refer to the dL4 Files and Devices reference manual).

The optional semicolon (;) terminator eliminates the automatic record-lock applied to the supplied *record* in the *chan.expr*. Applications may also utilize **MAT WRLOCK** for operations with locking transfers.

Examples

```
Mat Write #3,R1,100;A,B$,C[12]
```

```
Mat Write #C,R;A$
```

See also

MAT READ, WRITE

MAT WRLOCK

Synopsis

Write a variable to a channel with locking.

Syntax

MAT WRLOCK *chan.expr var.list*

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

MAT WRLOCK transfers data from any dL4 data type to the file opened on the supplied *chan.expr*. The operation is similar to a **WRITE** statement, except that an entire *array* or *matrix* is transferred; including row and column zero elements. If the specified *var* is a string, its entire specified length is transferred including zero-byte terminators.

If the variable in the list is a simple *num.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a *str.var*, its size may be controlled by subscripts. All characters are transferred including zero-bytes if support by the file type and driver (refer to the dL4 Files and Devices reference manual).

MAT WRLOCK transfers data and unconditionally locks the record. The data record remains locked until a non-locking operation is performed by that same program to the same channel. While a record is locked, other users will be unable to access the record.

See the **MAT WRITE** statement for details on the transfer of data.

Examples

```
Mat Wrlock #3,R1,100;A
```

```
Mat Wrlock #C,R;A$
```

See also

MAT READ, WRITE

MAT ZER

Synopsis

Zero an entire matrix.

Syntax

MAT *var.mat* **= ZER** { **"["** *num.expr1* {, *num.expr2* } **"]"** }

Parameters

var.mat is any numeric array or matrix variable.

num.expr1 is a numeric expression yielding a dimension.

num.expr2 is a numeric expression yielding a dimension.

Executable From Keyboard?

Yes.

Remarks

The matrix function **ZER** allows each element of a matrix to be set to zero. Row and column zero are not set. To set the elements of row and column zero to a zero use the **CLEAR** statement.

The optional *num.expr1* and *num.expr2* are evaluated, truncated to integer and used to select a new working size for the *array*. The total number of elements in the new size cannot exceed that of the old. A single element *array* can be converted to a *matrix* or vice versa as long as the total number of elements does not exceed the original **DIM**ensioned size. For example, a [4,4] matrix has 25 actual elements and could be re-declared as **ZER[24]**.

Any array created by a **MAT** statement with a single dimensions assumes a second dimension of one. For example, Mat C= ZER[15] and Mat C = ZER[15,1] are equivalent.

Examples

```
Mat C=ZER
```

```
Mat R7=ZER[4,4]
```

See also

CLEAR

MEMBER

Synopsis

Define a member associated with a specific structure.

Syntax1

MEMBER {%*prec* **|** *prec*% ,} *var.list* {, { %*prec* **|** *prec*% ,} *var.list*} ...

Syntax2

MEMBER {%*prec* **|** *prec*% ,} *var.name* [: **ITEM** *id*] { : **DECIMALS** *digits*} { :**RAW** }

Syntax3

MEMBER {%*prec* **|** *prec*% ,} *var.name* [: **KEY** *id option.list*] { : **DECIMALS** *digits*}

Parameters

prec indicates the precision number defined for the variable.

var.list is a list of comma separated variable names of any dL4 data types.

var.name is the name of a variable.

id is a string or a numeric literal identifying a fieldname or an item number.

digits is a numeric literal identifying the number of decimal digits.

option.list is a list of **UPPERCASE**, **ASCENDING**, **DESCENDING**, **DUPLICATES**, **UNIQUE**, **VARLEN**, and/or **PACKED** key options, each preceded by a plus sign ("+").

Executable From Keyboard?

No.

Remarks

MEMBER *var.name* is any legal variable name, or precision declaration in the form: %prec or prec%. *var.name* may be any dL4 data type. The syntax and function of **MEMBER** statements are nearly identical to that of **DIM**.

A structure definition itself may contain one or more structures, arrays, or arrays of structures. To define a structure which includes a structure, a **MEMBER** is expressed as follows:

```
Member var.name. { [expr {, ... }] } As structname
```

*var.name***.** is the name within the structure whose members are defined by the structure definition *structname*. *structname* must be an existing *structname* which has been previously defined.

The names of structure members are distinct from any other names outside the structure; e.g. Data.Q$ is distinct from Q$ which is distinct from Data1.T.Q$.

The members of a structure are physically contiguous in memory, and are ordered in memory as defined by **DEF STRUCT**. Individual structure members cannot be re-dimensioned.

The order in which members of a structure are declared is important because this determines the order in which values are read from a **DATA** statement, or transferred to/from a file, etc.

The **RAW** option enables special file access behavior similar to **OPTION FILE ACCESS RAW** but applied only to the specified structure member when used in an **ADD RECORD**, **READ RECORD**, or **WRITE RECORD** statement.

Examples

```
Def Struct StatMem
      Member CustName$. As FullName
      Member %4, Income
      Member City$[40]
End Def
```

See also

OPTION FILE ACCESS RAW

MODIFY

Synopsis

Change filename or a file's attributes.

Syntax

MODIFY *str.expr* {**AS** *driver-class* **|** *driver-name* }

Parameters

str.expr is a string expression consisting of an original *file.spec.str*, followed by new file attributes or a new filename.

driver-class specifies the driver-class.

driver-name specifies the driver-name.

Executable From Keyboard?

Yes.

Remarks

The original *file.spec.str* specifies the file to be changed. The new filename, if included, selects a new name or location for the original file.

If the file consists of two or more subfiles, each file will be modified. For example, an Index Contiguous file might consist of a data file and an index file. All these files would be copied to the respective destination filename.

If the *source filename* contains a *lu* or *directory* specifier, these must also precede the *destination filename* or the *source filename* is relocated to the current working directory.

Refer to the dL4 Files and Devices reference manual for more information on specific file types.

Examples

```
Modify "2/FILE 23/OLDFILE"! Move the file
Modify "PAYROLL <77>"
A$= "JUNK" \ Modify A$+"<E666>"
```

See also

MOVE

Synopsis

Move the components of a window.

Syntax

MOVE {*chan.expr*} @*x,y*;

Parameters

chan.expr is a driver-class dependent channel expression.

x,y are the destination column, row coordinates for the window components.

Executable From Keyboard?

Yes.

Remarks

The **@***x,y* parameter corresponds to the column, row coordinates of the upper left corner of the window.

Depending on the driver, it is possible to move the window on the screen or control which part of the window is displayed. Refer to the dL4 Files and Devices reference manual for more information about windows.

Examples

```
Move #1;@I,I;
```

See also

NEXT

Synopsis

Iterate a **FOR/NEXT** program loop.

Syntax

NEXT *num.var*

Parameters

num.var is a variable of numeric data type.

Executable From Keyboard?

No.

Remarks

The **NEXT** statement must have been preceded by execution of a **FOR** statement defining the parameters of the loop. Nested **FOR/NEXT** loops are paired based on the *num.var* used as the *index* variable.

Upon execution of the **NEXT**, the loop's *step* value is added to the *index*. If the new *index* exceeds the loop's *final* value, normal program execution resumes at the statement following the **NEXT**; otherwise, the *index* value is updated by the *step* and execution reverts back to the statement following the associated **FOR**. If a *step* was not specified on the associated **FOR** statement, it is assumed to be one.

When a loop terminates, the *index* variable contains the first value not used within the loop.

Examples

```
Next I
```

See also

FOR, WEND, WHILE

ON

Synopsis

Perform conditional branch on value of expression.

Syntax

ON *num.expr* [**GOSUB | GOTO**] *label:* **|** *stmt.no*

Parameters

label: is a user-defined name identifying a statement line.

stmt.no is a unique positive integer that identifies a statement line.

Executable From Keyboard?

No.

Remarks

The *num.expr* is any numeric expression which, after evaluation is truncated to an integer *n*. The program will then branch to the *nth label:* or *stmt.no* in the given list. If no *label:* or *stmt.no* corresponds to *n*, then execution continues with the statement following the **ON**.

GOTO and **GOSUB** work precisely as their singular counterparts. Branching will be to the first sub-statement of the statement number given, and the statement must exist.

Examples

```
On Q Goto 200,300,400,500,600
On Q Goto two, three, four, five
On (Sgn(A)+2) Goto 300,450,1000 ! Neg, Zero, Pos
On (A/100) Gosub 600,750,840,950
```

See also

GOTO, GOSUB

OPEN

Synopsis

Open an existing file.

Syntax1

OPEN *chan.no*, *file.spec.str* {**AS** *driver-class* **|** *driver-name* } {, {*chan.no*,} *file.spec.str* {**AS** *driver-class* **|** *driver-name*}} ...

Syntax2

OPEN *chan.no*, *file.spec.items* **AS** *driver-class* **|** *driver-name* {, {*chan.no*,} *file.spec.items* **AS** *driver-class* **|** *driver-name*} ...

Parameters

chan.no identifies a valid channel number, which the program uses for subsequent references to the file.

file.spec.str, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

driver-class specifies the driver-class, instead of using a default driver-class derived from the file.spec.

driver-name specifies the driver-name, instead of using a default driver-class derived from the file.spec.

file.spec.items, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

Executable from Keyboard?

Yes.

Remarks

The **OPEN** statement links a selected file or device to the specified *channel*. The *file* must already exist on the system or an error is generated.

Multiple *str.expr's* may be specified to open several files on successive channel numbers. Any new *channel* number (*channel*) in the filename list will cause assignment of channels to continue from that number.

In applications, if the specified *channel* is already in use, a **CLOSE** statement must be performed prior to an **OPEN**.

Most files to which a user has access may be opened. The same file may be simultaneously opened by other users, and may be opened on more than one channel. If a file is already opened for exclusive access via **EOPEN** by another process, an error is generated.

OPEN will link the selected file for read/write access and update each file's last access date.

A file may not be **OPEN** if it, or its directory does not have read permission for the user requesting access. If the file is read-only to the user, an implied **ROPEN** is performed and only read operations are allowed.

If a *file.spec.str* begins with a single $ character, the *filename* will be opened as an output pipe and the rest of the *file.spec.str* will be passed to the operating system as parameters to the pipe. If a *file.spec.str* begins with a "$$", the *filename* will be opened as an input pipe and the rest of the *file.spec.str* will be passed to the operating system as parameters to the pipe. Refer to the dL4 Files and Devices reference manual for a description of the pipe driver.

Examples

```
Open #1,"12/DATAFILE","FILE2",#4,"/usr/path/AR.CHECK"
Open #3,"$LPT",L$+A$ !EXPRESSION IS LU+FILENAME
Open #D,""
```

See also

BUILD, CLOSE, EOPEN, ROPEN, WOPEN

OPTION

Synopsis

Specify runtime option(s) for the current program.

Syntax

OPTION { **DEFAULT** } *opt.spec setting* {, *opt.spec setting* } ...

Parameters

opt.spec is a runtime option specifier.

setting is a runtime option parameter.

Executable From Keyboard?

No.

Remarks

The **OPTION** statement is used to specify various runtime options for the current program unit. Each of the options shown below are processed at compile-time and may be set once in each program unit, applying to the whole unit. An **OPTION DEFAULT** statement sets runtime options for all program units within a program file (it does not set options for libraries used by the program).

Unlike global environment variables, **OPTION** settings follow a program from system to system and are preserved in all forms of the program.

Default	Alternatives
OPTION ARITHMETIC DECIMAL	OPTION ARITHMETIC IRIS DECIMAL
	OPTION ARITHMETIC ICE BINARY
	OPTION ARITHMETIC IEEE DECIMAL
	OPTION ARITHMETIC EXTENDED IEEE
	OPTION ARITHMETIC NATIVE
	OPTION ARITHMETIC BITS DECIMAL
OPTION DATE FORMAT STANDARD	OPTION DATE FORMAT NATIVE
OPTION COLLATE STANDARD	OPTION COLLATE NATIVE
OPTION ANGLE RADIANS	OPTION ANGLE DEGREES
OPTION BASE YEAR 1988	OPTION BASE YEAR *numconst*
OPTION FOR NESTING 8	OPTION FOR NESTING *numconst*
OPTION GOSUB NESTING 8	OPTION GOSUB NESTING *numconst*
OPTION TRY NESTING 8	OPTION TRY NESTING *numconst*
OPTION COMMA SPACING 15	OPTION COMMA SPACING *numconst*
OPTION USING DECIMAL IS PERIOD	OPTION USING DECIMAL IS COMMA
OPTION FILE ACCESS STANDARD	OPTION FILE ACCESS RAW
OPTION FILE UNIT IS WORDS	OPTION FILE UNIT IS BYTES
OPTION DISPLAY AUTO LF ON	OPTION DISPLAY AUTO LF OFF
OPTION CHAIN FAILURE IS RETURNED	OPTION CHAIN FAILURE IS ERROR
OPTION CLOSE FAILURE IS ERROR	OPTION CLOSE FAILURE IS IGNORED
OPTION IF BY LINES	OPTION IF BY STATEMENTS
OPTION INPUT TIMEOUT SIGNAL ON	OPTION INPUT TIMEOUT SIGNAL OFF
OPTION ZERO DIVIDED BY ZERO IS ERROR	OPTION ZERO DIVIDED BY ZERO IS LEGAL
OPTION STRINGS STANDARD	OPTION STRINGS RAW
	OPTION STRINGS HAGEN

`OPTION STRING SUBSCRIPTS STANDARD`	`OPTION STRING SUBSCRIPTS IRIS`
`OPTION STRING REDIM IS ERROR`	`OPTION STRING REDIM IS LEGAL`
`OPTION OPEN AUTO CLOSE OFF`	`OPTION OPEN AUTO CLOSE ON`
`OPTION RETURN BY STATEMENTS`	`OPTION RETURN BY LINES`
`OPTION NUMERIC FORMAT STANDARD`	`OPTION NUMERIC FORMAT NATIVE`
`OPTION INPUT BUFFER 256`	`OPTION INPUT BUFFER` *`numconst`*
`OPTION CHAIN ALTERNATE DIRECTORIES ON`	`OPTION CHAIN ALTERNATE DIRECTORIES OFF`
`OPTION ARGUMENT CHECKING STANDARD`	`OPTION ARGUMENT CHECKING IS WEAK`
`OPTION DIALECT STANDARD`	`OPTION DIALECT IRIS`
	`OPTION DIALECT IRIS1`
	`OPTION DIALECT BITS`
	`OPTION DIALECT BITS1`
	`OPTION DIALECT IMS`
`OPTION AUTO DIM ON`	`OPTION AUTO DIM OFF`
`OPTION FLUSH AFTER STATEMENT OFF`	`OPTION FLUSH AFTER STATEMENT ON`
`OPTION RECORD LOCK TIMEOUT –1`	`OPTION RECORD LOCK TIMEOUT` *`numconst`*
`OPTION PROGRAM TAG “”`	`OPTION PROGRAM TAG` *`strconst`*

The **OPTION ARITHMETIC EXTENDED IEEE** is identical to **OPTION ARITHMETIC IEEE DECIMAL** except that it maps 1% variables to 16-bit signed binary integers and 2% variables to 32-bit signed binary integers.

The **OPTION USING DECIMAL** [**IS PERIOD | IS COMMA**] only controls the meaning of period (".") and comma (",") in **USING** mask strings, not which character is output. The output character is controlled by **OPTION NUMERIC FORMAT** [**STANDARD | NATIVE**] and the operating system locale setting.

The **OPTION INPUT BUFFER *numconst*** specifies the size in characters of the input buffer used by the **INPUT** and **MAT INPUT** statements.

The **OPTION ZERO DIVIDED BY ZERO IS** [**ERROR | LEGAL**] controls whether dividing zero by zero is an arithmetic error.

The **OPTION STRING SUBSCRIPTS** [**STANDARD | IRIS**] controls the handling of the subscript if it evaluates to zero. String subscript values of zero are not normalized by default (**STANDARD**). Zero string subscripts are normalized when **OPTION STRING SUBSCRIPTS IRIS** is used, such that a starting subscript of 0 becomes 1, with an ending subscript of 0 being treated as if no ending subscript were given.

The **OPTION STRING REDIM IS** [**ERROR | LEGAL**] controls whether a string variable can be redimensioned without first FREEing the variable. By default, redimensioning a string variable to a different size generates an error.

The **OPTION CHAIN ALTERNATE DIRECTORIES** [**ON | OFF**] controls whether the **CHAIN** and **SPAWN** statements use a search path to locate programs. By default (**ON**) the **Lib** *dirname* of the program unit is searched first. The directory of the calling program is searched next. Finally, the users current working directory is searched. If disabled (**OFF**), no search path is used and the program file is located just as in the **OPEN** statement.

The **OPTION ARGUMENT CHECKING** [**STANDARD | IS WEAK**] controls whether empty brackets (“[]”) are required in order to pass array variables as arguments to subprograms (Call by Filename). Normally, empty brackets are required. This option can only be used in an **OPTION DEFAULT** statement.

The **OPTION DATE FORMAT** [**STANDARD | NATIVE**] controls the date input/output formats. **STANDARD** specifies the USA format of MM/DD/YY and **NATIVE** specifies the format as determined by the system locale setting.

The **OPTION AUTO DIM** [**ON | OFF**] enables or disables auto-dimensioning of variables.

The **OPTION FLUSH AFTER STATEMENT** [**OFF | ON**] enables a flushing of the record buffer at the end of each write statement for those file drivers that support a flush record without unlock operation.

The **OPTION RECORD LOCK TIMEOUT *numconst*** sets the default record lock timeout period in tenth seconds for I/O statements that do not specify a timeout period. This option only effects I/O to disk file and database drivers. The value of ***numconst*** must be between –1 (wait forever) and 36000 inclusive.

The **OPTION PROGRAM TAG *strconst*** places an ASCII string constant in the program file for use by external utilities. Under Unix, this option can be used to place a revision string in the program file for use with standard Unix program utilities.

The **OPTION DIALECT** [**STANDARD | IRIS | IRIS1 | BITS | BITS**1 | **IMS**] sets multiple options. The default option settings should serve best for most IRIS programs. The statement **OPTION DIALECT IRIS** is equivalent to **OPTION STRING SUBSCRIPTS IRIS**. The statement **OPTION DIALECT IRIS1** adds the additional option **OPTION ZERO DIVIDED BY ZERO IS LEGAL** and allows intrinsic **CALL**s to return results into subscripted string arguments.

BITS users should add the following line to each program:

OPTION DIALECT BITS

This is equivalent to adding the lines:

OPTION FILE ACCESS RAW,FILE UNIT IS BYTES,DISPLAY AUTO LF OFF

OPTION CHAIN FAILURE IS ERROR,CLOSE FAILURE IS IGNORED

OPTION IF BY STATEMENTS,INPUT TIMEOUT SIGNAL OFF,STRINGS RAW

OPTION OPEN AUTO CLOSE ON,RETURN BY LINES

For further BITS compatibility, the line

OPTION DIALECT BITS1

is equivalent to **OPTION DIALECT BITS**, but also enables BITS style **FOR/NEXT** behavior, BITS **USING** mask features, returning results to intrinsic **CALL** arguments that are subscripted strings, and an initial precision of 4%.

For IMS compatibility, the line

OPTION DIALECT IMS

is equivalent to **OPTION DIALECT IRIS1** with some minor changes to **USING** mask behavior.

Examples

```
Option Date Format Native
```

See also

FOR, GOSUB, TRY

PAUSE

Synopsis

Suspend program operation.

Syntax

PAUSE *num.expr*

Parameters

num.expr is an expression yielding tenth-seconds pause time.

Executable From Keyboard?

No**.**

Remarks

The *num.expr* is any numeric expression which, after evaluation is truncated to an integer and used to specify a delay in program operation. The delay is limited to an integer between 0 and (2^{32})-1 representing the number of tenth-seconds to delay.

This is the most accurate method of pausing the execution of a program. Other methods, such as finite program loops, will be affected by the current usage of the system and most likely yield varying results.

The program is unconditionally suspended for the number of tenth-seconds specified in *delay*. An **[ESCAPE]** without **ESC**ape branching or **[ABORT]** terminates a pause.

Examples

```
Pause 30
Pause Fna(Q7)
Pause A*10
```

See also

SIGNAL

PORT

Synopsis

Attach and control other ports.

Syntax

PORT *num.expr1*, *num.expr2*, *num.var1* {, *expr*} ...

Parameters

num.expr1 is an expression used to select a target port number.

num.expr2 is an expression used to select the PORT statement mode.

num.var1 is a variable of numeric data type used to received operational status.

expr is an expression or variable.

Executable From Keyboard?

Yes.

Remarks

The **PORT** mode is a *num.expr* which, after evaluation is truncated to an integer and used to select an operation for **PORT**. There are 8 modes as determined by the second parameter:

Mode	Operation Performed.
0	Attach selected port
1	Place an attached port in command mode
2	Transmit a command string to an attached port
3	Return an attached port's operational status
4	Return the name of the current running program of a specified port
5	Return the position of the current running program of a specified port
6	Return record lock status of the program running on a specified port
7	Return user information for a specified port
8	Return information about open channels on a specified port.
9	Determine if a specified file is open on a a specified port.

num.var1 is used to return the exception status of the operation. The meaning of *num.var1* depends upon the mode selected.

The **PORT** statement allows a *port* to be attached to a program. Once attached, commands may be transmitted to the *port* for normal processing, and the current *status* or state of the attached *port* can be controlled and monitored. If the attached *port* has a keyboard, it may be used as any other normal terminal. However, commands transmitted will override any current keyboard operation.

Mode 0—Attach Selected Port

Syntax

PORT *num.expr1*, 0, *num.var1* {, *num.expr2* }

A **PORT** mode 0 statement must be issued once for each port being attached. Once attached, the port remains so until signed-off (sending a **BYE** command or executing **SYSTEM 0** to the port).

num.expr2 is evaluated and truncated to an integer and used to select a different account for the attached port when using *mode* 0. The account should be expressed as G*256+U, where G and U are the desired group and user numbers respectively. The Group and User numbers must be in the range 0 to 255. If not specified, the group and user id of the program executing the attach is set. The meaning of Group and User numbers is operating system dependent. The ability to start a port using group or user ids different from the calling program will require the use of a privileged account on most operating systems.

PORT Mode 0 begins by attempting to attach the *port*. If the *port* is already running under uniBasic, the attach operation is complete and successful.

If the *port* is not currently running dL4, a background process is created as the supplied *port* number. It assumes the callers' environment and current working directory. It then becomes a unique process linked to the supplied *port number*. This port is then available for **CALL $TRXCO** commands, **PORT**, **SEND**, **RECV**, and **SIGNAL** statements from any other dL4 user as well as the program performing the initial **PORT Mode 0**.

When sending commands to a *port* which is connected to a terminal and keyboard, you must ensure that *port* is already running dL4 before sending commands. Otherwise, a *phantom port* is created for the supplied *port number*. If a user later attempts entry into Basic using the same *port number*, entry into Basic will be rejected.

Upon completion, the status variable is set to indicate

0. Successful, port is now attached.
1. The selected port is already logged-on to the system and in-use.
2. All available ports are already in use. In some configurations, the allowed number of concurrent users is set less than the actual number of ports configured. This indicates that either another *port* or *phantom port* must be signed-off, or the number of concurrent users increased on your license.
3. Illegal account number selected. The selected group or user number is out of the range 0-255.

Mode 1—Place an Attached Port in Command Mode

Syntax

PORT *num.expr1*, 1, *num.var1*

PORT Mode 1 sends an ESCape Override Character **[ABORT]** to the selected *port*, terminating any running program and placing the *port* into *command mode*.

Upon completion, the status variable is set to indicate:

0. Successful, the selected port is now in command mode.
1. The select port is not attached.

Mode 2—Transmit Command String to Attached Port

Syntax

PORT *num.expr1*, 2, *num.var1*, *str.expr*

str.expr is used to send the command string to the specified *port*.

PORT Mode 2 requires that a *command string* be supplied following the *status* variable. The string data in *command string* is then transmitted to the selected *port*. This *command string* may contain any legal command input for a terminal. Any command, such as **NEW**, **LIST**, **BYE**, **RUN**, etc., may be transmitted, as well as program statements. If a terminal is connected to the attached *port*, the *command string* is echoed as it is processed on the attached *port*. An attached *port* connected to a terminal may also receive commands from its keyboard.

A *command string* cannot be transmitted unless the attached *port* is in an 'input ready' state. A **PORT Mode 3** status check is suggested prior to sending a command.Upon completion, the status variable is set to indicate:

0. Successful, command transmitted and accepted.
1. The selected port is not attached.

2. The selected port is not in an 'input ready' state.

Mode 3—Return Attached Port's Operational Status

Syntax

PORT *num.expr1*, 3, *num.var1*, *num.var2*

PORT Mode 3 requires that a return value (*num.var2*) be supplied following the status variable(*num.var1*). This variable will receive a value indicating the port's operational status. A **PORT** *mode* 3 should always precede any *mode* 2 command transmission to check for 'input ready'. It may also be used to monitor the current state of the attached port.

0. Successful, operational status returned.
1. The selected port is not attached.

The value returned as the operational status consists of a mode, an 'Input Ready' flag, and an 'Output in Progress' flag.

This value may be divided into its respective parts as follows:

Assume X = value returned by **PORT** mode 3.

```
If X>32767 Then 'Input Ready' on attached port.
```

The 'Input Ready' flag must be removed from the value prior to testing the 'Output in Progress' flag, since both input and output may be in progress.

```
If X>32767 Then X=X-32768 \! Remove flag.
If X>16383 And X<32768 Then 'Output in Progress'
```

The attached port's current mode can be determined by:

```
Let M=X % 16 \! Retrieve mode.
```

Mode	Current State
0	Idle. At command mode or logged-off.
1,2	Command input or execution.
3	Run. Program execution in progress.
4,5	List. Program listing in progress.
6	Statement execution in immediate mode.
7	Get. Program being loaded from text file.
8	Initial Run. Becomes mode 3.
9,10	Enter. Program statement entry using ENTer.

Mode 4—Return Name of Current Program of Specified Port

Syntax

PORT *num.expr1*, 4, *num.var1*, *str.var*

PORT mode 4 returns in *str.var* the name of the current program of a specified port. For example, the statement:

```
Port P,4,S,F$
```

will return in F$ the name of the program running on port P. As with **PORT** mode 3, a status is returned in S indicating success (zero) or failure (one, port not attached). Under some operating systems, only a privileged user (such as the Unix root account) can use **PORT** mode 4 to examine ports that belong to different user ids.

Mode 5—Return Current Program Position of Specified Port

Syntax

PORT *num.expr1*, 5, *num.var1*, *str.var*

PORT mode 5 returns in *str.var* the current execution position of the current program of a specified port. For example, the statement:

```
Port P,5,S,L$
```

will typically return in L$ the line number and library name, if any, of the statement currently being executed by the program running on port P. As with **PORT** mode 3, a status is returned in S indicating success (zero) or failure (one, port not attached). Under some operating systems, only a privileged user (such as the Unix root account) can use **PORT** mode 5 to examine ports that belong to different user ids.

Mode 6—Return Record Lock Status of Specified Port

Syntax

PORT *num.expr1*, 6, *num.var1*, *num.var2, num.var3*

PORT mode 6 returns the record lock status of the specified port in *num.var2* and and the conflicting port number in *num.var3*. For example, the statement:

```
Port P,6,S,B,N
```

will return one in B if port P has been waiting for a record lock for more than 20 seconds and it will return zero in B if the port is not blocked. If the port is blocked, the port number of the program that has locked the record will be returned in N. If the port number is not available, N will be set to -1. As with **PORT** mode 3, a status is returned in S indicating success (zero) or failure (one, port not attached). Under some operating systems, only a privileged user (such as the Unix root account) can use **PORT** mode 6 to examine ports that belong to different user ids.

Mode 7—Return User Information of Specified Port

Syntax

PORT *num.expr1*, 7, *num.var1*, *str.var1, str.var2* {, *var.list* }

PORT mode 7 returns the current user information for the specified port in *str.var1* and *str.var2*. Additional information can be returned in optional string variables in *var.list*. For example, the statement:

```
Port P,7,S,U$,W$
```

will, for port P, return in U$ the user name and in W$ the workstation name. The optional string variables in *var.list*, if specified, receive the group name, current directory, terminal type, account number, and group number. Values not supported by the operating system will be returned as “”. As with **PORT** mode 3, a status is returned in S indicating success (zero) or failure (one, port not attached). Under some operating systems, only a privileged user (such as the Unix root account) can use **PORT** mode 7 to examine ports that belong to different user ids.

Mode 8—Return Open Channel Information for Specified Port

Syntax

PORT *num.expr1*, 8, *num.var1*, *num.expr2, num.expr3, struct.array.var*}

PORT mode 8 returns open channel information for the specified port in *struct.array.var*. A range of channel numbers to examine is specified using *num.expr2* as the starting channel number and *num.expr3* as the ending channel number. The information is returned in the array variable *struct.array.var* which is an array of structures using the following structure definition:

```
Def Struct CHANINFO
    Member 1%,ChanNum
    Member Path$[200]
```

```
        Member 3%,RecordNum
        Member 1%,RecordState
    End Def
```

The member names, dimensioned size of the Path$ member, and the numeric precisions of the other structure members can be varied as desired. The filename returned in Path$ may be truncated if it is longer than Path$ or if it exceeds system limitations. If the number of open channels in the specified range is less than the dimensioned size of 'chaninfo.[]', then the first unused element of the array will have a ChanNum value of -1. If the number of open channels in the specified range is greater than the dimensioned size of 'chaninfo.[]', the extra channels will be ignored. As with **PORT** mode 3, a status is returned in *num.var1* indicating success (zero) or failure (one, port not attached). Under some operating systems, only a privileged user (such as the Unix root account) can use **PORT** mode 8 to examine ports that belong to different user ids.

Mode 9— Determine if a Specified File is Open on a Specified Port

Syntax

PORT *num.expr1*, 9, *num.var1*, *str.expr, num.expr2*, *num.var2*

PORT mode 9 determines which channel, if any, on the port specified by *num.expr1* is open to the file *str.expr* with record *num.expr2* locked. If *num.expr2* is negative, the record lock status will not be checked. If a match is found, the channel number is returned in *num.var2*. If no match is found, *num.var2* is set to -1. As with **PORT** mode 3, a status is returned in *num.var1* indicating success (zero) or failure (one, port not attached).

Examples

```
Port 8,0,S \ If S Stop ! attach & check status
Port P,1,S \ If S Stop ! abort & get ready
Port P*2,2,E,C$[50] \ If E Stop ! send command
Port X,3,Y,Z \ If Y Stop ! get current mode & stat
```

See also

SWAP, SPAWN

PRINT

Synopsis

Format values and output formatted string to a file or a device.

Syntax

PRINT {*chan.expr*} { **USING** *str.expr* ; } *var.list* { , **|** ; }

Parameters

chan.expr is a driver-class dependent channel expression. The standard output channel is used when the *chan.expr* is omitted or the channel number is -4.

str.expr is a string expression used for formatting numeric values.

var.list is a list of comma or semicolon separated variables of any dL4 data types passed to this program.

Executable From Keyboard?

Yes.

Remarks

The *var.list* consists of variables, literals, or expressions; numeric, date, or string. Each item in the *var.list* must be separated by either a comma (,) or a semicolon (;). A comma performs a **TAB** to the next comma field before output of the next item. This is generally 15 characters long, but can be changed with the **OPTION COMMA SPACING** statement. A semicolon prevents additional spacing in the output.

Numerics are output preceded by a '-' or space indicating negative or positive, and followed by one space (The **STR$** function may be used to omit leading and trailing spaces). Strings are output exactly as stored, from the supplied starting position terminating at the first zero-byte terminator. No preceding or trailing spaces are output.

When all items in the *var.list* are output, a new-line is output to advance the terminal to the next line (or mark end of line in a text file). This can be suppressed by using a comma or semicolon as the last character in the **PRINT** statement. In the case of a comma, a **TAB** is still performed.

The **USING** operator formats numeric data for columnar output. It may also be used to imbed commas, asterisk check fill, floating dollar signs and other special output formats. It must be after any *chan.expr* and before the *var.list*, and only one is allowed per statement. For additional information, see the string operator **USING**.

An output column counter (base zero) is maintained for each channel holding the current character position on the output line. This counter is reset anytime a new-line is output (usually a return) or an @0,y cursor positioning operation is performed.

The **TAB** function is used to skip the terminal to a specific column. Its form is:

```
Tab  (num.expr)
```

The *num.expr* must be a positive value. A **TAB** to a position less than the current position or greater than the device width may be ignored depending on the driver.

After all items in the *var.list* are placed into the terminal buffer, it is flushed immediately. No **SIGNAL 3,0** is required to start output, and is ignored if executed.

If a *chan.expr* is specified for **PRINT**, the output is redirected to the selected *channel*. If the *channel* is not open, output is transmitted to the terminal. This allows a program to selectively output to the terminal or a printer by including an **OPEN** of the printer *pipe* on the selected *channel*. A separate output column counter is maintained for each *channel* opened, so that the **TAB** and comma operator will operate on applications doing both screen and file output operations.

PRINT # is generally used to output to a text file, or *pipe* such as a line printer. The most common form used for output to a line printer is:

```
Print #chan.expr; var.list
```

The optional *record*, *byte displacement* and *time-out* specifications of a *chan.expr* are normally unused, as line-oriented data is generally of variable length. Each successive **PRINT #** continues its transfer immediately following the previous, unless a new *record* or *byte displacement* is specified.

Examples

```
Print "AVAILABLE";TAB(40);A*100;"$";Z
```

```
;@0,23;’CL’;"Error in Program";
```

```
Print #K; Using T$;X,Y,Z,Z/10
```

See also

OPTION

RANDOM

Synopsis

Seed random generator for **RND** function.

Syntax

RANDOM *num.expr*

Parameters

num.expr is an expression yielding a numeric random number seed value.

Executable From Keyboard?

Yes.

Remarks

The *num.expr* is evaluated, truncated to a positive integer and used to seed the system's pseudo-random number generator. Seeding implies that a sequence is selected and initiated based on the value supplied. A seed value of zero selects a further random sequence based upon the current system time.

Typically, a non-zero seed value is used during program debugging, causing the **RND** function to yield the same sequence of numbers with each successive run. Once the program is completed, a **RANDOM 0** is issued to produce better random selection.

Examples

```
Radom 5
```

```
Random 0
```

```
Random ((N*100)/E^2)
```

See also

RDLOCK

Synopsis

Read record and keep record locked.

Syntax

RDLOCK *chan.expr var.list*

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

RDLOCK transfers data into user variables.

If the variable in the list is an *array.var*, optional *subscripts* may be specified. If given, these are evaluated, truncated to integer and used to select a single element. If no *subscripts* are supplied, only the first element is transferred. The entire array may be transferred using the notation "[ ]".

If the variable in the list is a simple *num.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a string or binary variable, its size may be controlled by subscripts.

RDLOCK transfers data and unconditionally locks the record. The data record remains locked until a non-locking operation is performed by that same program to the same channel. While a record is locked, other users will be unable to access the record.

RDLOCK is identical to **READ** omitting the trailing semicolon.

Example

```
Rdlock #3,R1,100;A
Rdlock #C,R;A$
```

See also

READ, WRLOCK, OPTION FILE ACCESS

READ

Synopsis

READ variables from **DATA** statements or channel.

Syntax1

READ *var.list*

Syntax2

READ *chan.expr; var.list* { *;* }

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

"*;*" unlocks the record after a successful **READ**.

Executable From Keyboard?

Yes.

Remarks

Syntax1:

An *array.var* or *mat.var* with *subscripts* specifies only that single element. Omission of a *subscript* selects only the first element.

READ begins transferring data sequentially from the lowest numbered **DATA** statement found in the program. Subsequent **READ** statements resume transfer at the next element of the **DATA** statement. After all of the items in a given **DATA** statement have been read, reading continues with the next highest numbered **DATA** statement. When all **DATA** statements have been read, any subsequent will produce the error 'Out of Data'. The **RESTOR** statement can be used at any time to start reading from a specific **DATA** statement.

READ attempts to transfer data into each variable listed in the *var.list*. Transfer of a variable terminates at a comma (,) or at the end of the **DATA** statement. You may not transfer string data into any numeric variable. String items must be enclosed in quotes (" ").

Syntax2:

If the variable in the list is an *array.var* or *mat.var*, only the first element is read. *Subscripts* may be used to select any individual element to be transferred. The entire array may be transferred using the "[ ]" notation. The number of bytes transferred is based upon the variable's dimensioned size. The transfer is performed according the rules for a *num.var*.

If the variable in the list is a simple *num.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a string or binary variable, its size may be controlled by *subscripts*. Refer to the dL4 Files and Devices reference manual for file and driver specific details of data transfer.

The optional semicolon (;) terminator is used by dL4 applications to release the automatic record-lock applied to the supplied *record* in the *chan.expr*.

Examples

```
Read A,B,D[10],A[4,4]
Read A$
Read #3,R1,100;A,B$,C[12];
Read #C,R;A$
```

See also

DATA, INPUT, MAT READ, RDLOCK, READ, SEARCH, WRITE, WRLOCK

READ RECORD

Synopsis

Read an entire record structure.

Syntax

READ RECORD *chan.expr; struct.var*

Parameters

chan.expr is a driver-class dependent channel expression.

struct.var is a variable of structure data type.

Executable From Keyboard?

Yes

Remarks

The **READ RECORD** statement is similar to the normal **READ** of a record except for the requirement that a *struct.var* is supplied and the computation and override of the item number for each member.

Examples

```
Read Record #2,RecAccess;CustRec.
```

See also

WRITE RECORD

RECV

Synopsis

Receive a message.

Syntax

RECV *num.var1*, [*str.var* **|** [*num.var2*, *num.var3*]] {, *num.expr*}

Parameters

num.var1 is a variable of numeric data type to receive the sender's port number.

str.var is a variable of string data type to contain the received message.

num.var2 and *num.var3* are variables of numeric data type to contain the received message.

num.expr is an expression yielding a number specifying a maximum wait period.

Executable From Keyboard?

Yes.

Remarks

num.var1 receives the sender's *port number*, or -1 if no messages are waiting for your *port*.

str.var receives a string message.

num.var2 and *num.var3* receive 2 numeric messages. If the second parameter is a *num.var*, two numeric variables must be specified. Their two values are then received. The two variables need not be the same precision.

The optional *num.expr* is any numeric expression which, after evaluation is truncated to an integer to specify a delay period (in tenth-seconds) during which the program awaits a message. If zero, or not included, no pause is invoked, but any currently waiting message will be received. Any message appearing during a specified delay allows **RECV** to accept the transmitted data and resume program execution immediately. If no message appears during the entire delay, *port* is set to -1.

If the program has an **INTSET** branch enabled, any message sent to your *port* will cause a branch to the selected statement. The interrupt handling routine can then perform a **RECV** to receive the message.

RECV is identical in operation to **SIGNAL 2**.

Examples

```
Recv P,A,B,600 ! Wait 60 seconds
```

```
Recv P,A$
```

See also

SIGNAL, SEND

REM

Synopsis

Insert program comment.

Syntax

REM {*comment* }

Parameters

comment is a sequence of characters.

Executable From Keyboard?

No.

Remarks

The **REM** statement allows the placement of comments within a program. A **REM** statement is ignored during execution, but may be referenced within the program.

When **REM** statements are entered, all characters following the **REM** up to the end of line are considered the comment. This includes leading and trailing spaces and control characters.

A **!** may be used to abbreviate the verb **REM** during entry. During listing, **REM** is listed if it is the first statement of the line, otherwise **!** is displayed. When a **REM** statement is processed during program execution, the statement is ignored. Branching (**GOTO**, **GOSUB**, etc.) to **REM** statements is acceptable with little program overhead.

Note that, since all characters following a **REM** are considered part of the **REM**, the **REM** is always the last statement on it's line.

```
400 Print A  \ Rem OUTPUT TOTAL \ Goto 200
```

Line 400 outputs the value of A and continues with the next program line. The "Goto 200" is considered to be part of the comment.

Examples

```
Rem  Request input of customer name
Gosub 1000 ! Go receive response
```

See also

RESTOR

Synopsis

Reset to first data item in a **DATA** Statement

Syntax

RESTOR *label*: **|** *stmt.no*

Parameters

label is a user-defined name identifying a statement.

stmt.no is any valid dL4 statement.

Executable From Keyboard?

Yes.

Remarks

RESTORE resets the **DATA** statement pointer to the first data item of the first **DATA** statement in the program, just as when the program started.

Including an optional *label:* or *stmt.no* sets the pointer the first data item of the first **DATA** statement encountered at or past that *label:* or *stmt.no*.

If no further **DATA** statements are found, the pointer will be set to return an "Out of DATA" error during the next **READ**.

Examples

```
Restor MIDDLE:
```

```
Restor 2200
```

See also

DATA, READ

RETRY

Synopsis

Re-execute last **TRY** block.

Syntax

RETRY

Parameters

None.

Executable From Keyboard?

No.

Remarks

RETRY may be used within the **ELSE** block(s) to repeat the last **TRY** block.

Examples

```
Try
      Open #2,"cust.master"
      Print "Opened cust.master on channel 2"
Else
      Print "Attempting to open cust.master file again"
      Retry
End Try
```

See also

TRY, END TRY

RETURN

Synopsis

Return from a **GOSUB** subroutine call.

Syntax

RETURN {*num.expr* }

Parameters

num.expr is a numeric value specifying the return point relative to the calling **GOSUB** statement.

Executable From Keyboard?

No.

Remarks

The **RETURN** statement is used with **GOSUB** and indicates the end of a program subroutine.

A normal **RETURN** (or **RETURN +0**) resumes execution at the statement following the matched **GOSUB**. A value of **+1** would branch to the second statement following the **GOSUB** (the first statement past a normal **RETURN**). A value of **-1** would branch to the statement of the **GOSUB** itself.

The **OPTION RETURN BY LINES** statement can be used to enable relative return by lines rather than statements.

Examples

```
Return
```

```
Return +1
```

See also

GOSUB, OPTION

REWIND

Synopsis

Reset a file to the beginning.

Syntax

REWIND *chan.no* {, *chan.no* } ...

Parameters

chan.no is a valid channel number.

Executable From Keyboard?

Yes.

Remarks

The **REWIND** statement resets the selected *channel's* current file position to the beginning of the file. The position is reset to *record* 0, *byte displacement* 0. If the next file transfer does not specify a *record* or *byte displacement*, the transfer will start at the first data byte of the file.

The effect of **REWIND** is to reset the current file position as when the channel was initially opened. **REWIND** is typically used with Text Files accessed sequentially.

A **REWIND** operation is ignored when issued to a *channel* linked to a *pipe*.

REWIND is identical in operation to **SETFP #***channel*, *0*, *0* ;

Examples

```
Rewind #T, #7, #(J*2)
```

See also

SETFP

ROPEN

Synopsis

Open an existing file for Read-Only access.

Syntax1

ROPEN *chan.no*, *file.spec.str* {**AS** *driver-class* **|** *driver-name* } {, {*chan.no*,} *file.spec.str* {**AS** *driver-class* **|** *driver-name*}} ...

Syntax2

ROPEN *chan.no*, *file.spec.items* **AS** *driver-class* **|** *driver-name* {, {*chan.no*,} *file.spec.items* **AS** *driver-class* **|** *driver-name*} ...

Parameters

chan.no identifies a valid channel number, which the program uses for subsequent references to the file.

file.spec.str, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

driver-class specifies the driver-class, instead of using a default driver-class derived from the file.spec.

driver-name specifies the driver-name, instead of using a default driver-class derived from the file.spec.

file.spec.items, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

Executable From Keyboard?

Yes.

Remarks

The **ROPEN** statement opens files for read-only access with record locking disabled. This feature permits an application to read records that are currently locked by other processes. This form of open is supported by the Portable Formatted, Portable Indexed Contiguous, UniBasic Formatted, UniBasic Indexed Contiguous, and FoxPro Full-ISAM drivers. Note: reading records that are currently locked may return partially updated or inconsistent data.

A file may not be **ROPEN**ed if it, or its directory does not have read permission for the user requesting access.

ROPEN is equivalent to an **OPEN** statement which specifies "**<WL>**" as an access option.

Examples

```
Ropen #1,"DATAFILE","FILE2",#4,"AR.CHECK"
Ropen #1,"23/MMFILE" As "Full-ISAM"
```

See also

BUILD, CLOSE, EOPEN, OPEN, WOPEN

SEARCH (String)

Synopsis

Search string for a sub-string.

Syntax

SEARCH *source.str.expr*, *destination.str.expr*, *num.var*

Parameters

source.str.expr is any source string expression.

destination.str.expr is any target string expression.

num.var is a variable of numeric data type which receives the character index of the target within the source, or zero if *destination.str.expr* is not found in *source.str.expr*.

Executable From Keyboard?

Yes.

Remarks

source.str.expr is searched for the first occurrence of *destination.str.expr*. If found, *num.var* is set to the character position of the located substring. If not found, a zero is returned. If the *source* being searched is a single *str.var*, it may include a starting *subscript* if desired, and searching begins at the selected position. Note however that any position returned will be relative to this starting position.

When performing multiple **SEARCH** operations on a single string, it is best to initialize a *num.var* to 1; adjusting for each located identical sub-string.

```
290 Let J=1
300 Search T$[J],"H-",R
310 If R Then Let J=(J+R)-1
```

Here, *destination.str.expr* is adjusted for the offset caused by a starting *subscript*. If the substring is not found, *destination.str.expr* is returned as zero. The adjustment needed for any given starting *subscript* 'A' can be defined as:

actual position in string = starting *subscript* + *location* - 1

Searching terminates when a null character is encountered in the *source.str.expr*. Entry of the verb **SEARCH** followed by a **#** character is interpreted as a file **SEARCH** statement and treated as such.

Example

```
Search P$+A$,".",K
Search A$[J],"TIME",K \ J=J+K-1
```

See also

POS function

SEARCH (Traditional)

Synopsis

Access or create an index in a keyed file.

Syntax

SEARCH *chan.no*, *num.expr1*, *index.no* {, *num.expr2*} ; *str.var*, *num.var1*, *num.var2*

Parameters

chan.no is any valid channel number..

num.expr1 is an expression yielding a number specifying the desired operation.

num.expr2 is an expression yielding a number specifying the timeout value.

index.no is a numeric expression whose integer value identifies an index in the file.

str.var is a variable of string data type which contains the source and destination key.

num.var1 is a variable of numeric data type in which the record number is returned if the operation succeeds.

num.var2 is a variable of numeric data type which contains the return status value.

Executable From Keyboard?

No.

Remarks

In the following tables, mode is the operation as selected by the value of *num.expr1*.

Summary of SEARCH Operations

Mode	OPERATION
0	Define and Create indices within a Contiguous Data File.
1	Return miscellaneous index information.
2	Search for an exact key.
3	Search for the next highest key.
4	Insert a new key into an index.
5	Delete an existing key from an index.
6	Search for the previous key (Search Backward).
7	Unused, included for compatibility.
8	Maintain the B-Tree insertion algorithm for an index.
9	Temporarily same as Mode 6 - Reserved for future use.

Detailed Table of SEARCH Operations

Mode	Index	Status	Operation Performed
0	$1 \le d \le 63$		For a new Indexed File, sets the *key* length of the selected *index* to the number of bytes specified by *num.var1*. Indices must be defined starting at one and proceed sequentially.
0	0		Freeze the file definition and build the ISAM portion of the file. Total number of initial data records is specified by the *num.var1*.

1	>0		Return the *key* length of the specified *index* in bytes.
1	0	=0	Returns the record number of the First Real Data Record.
1	0	=1	Return the number of available records in the file.
1	0	=2	Allocate and return a new record for the application.
1	0	=3	Return a record to the file that is no longer needed. Deleted records will be reused before the file is extended.
1	0	=4	Return in *num.var1* the number of records in the file.
1	0	=5	Return in *num.var1* the number of records in the file.
1	0	=6	Set the First Real Data Record to the value supplied in *num.var1*. This option is only available during file structuring.
1	0	=7	Return the current number of records in use (allocated) in the data portion of the file.
2			Search the specified *index* for the exact match of the supplied *key*. If found, return the full key in the supplied *key* variable, and the associated record number in *num.var1*. *num.va2r* is set to 0 if the *key* was found, and 1 if the *key* was not in the index.
3			Search the specified *index* for the first key whose value logically exceeds the supplied *key*. If found, *num.var2* is set to 0, the full key is returned in *str.var*, and the associated record number is returned in *num.var1*.
4			Insert *key* into the specified *index* using the supplied *num.var1* as the associated pointer. The record should have been previously allocated using *mode* 1, *status* = 2 above. A *status* of 0 indicates a successful operation. If the *key* already exists in the *index*, a 1 is returned as *num.var2*.
5			Delete the supplied *key* from the specified *index*. If successful, *num.var1* is returned as the associated pointer, and the *num.var2* is set to 0. A *num.var2* of 1 indicates an unsuccessful operation; ie, the *key* was not found in the *index*. The *record* should be returned to the file using *mode* 1, *status* = 3 above.
6			Search the specified *index* for the first key whose value is logically less than the supplied *key*. If found, *num.var2* is set to 0, the full key is returned in *str.var*, and the associated record number is returned in *num.var1*.
7			No operation. Reserved for future use.
8			B-Tree algorithm maintenance. If *num.var1* is negative, return in *num.var1* the current B-Tree algorithm for *index*. If *num.var1* is positive, change the insertion algorithm to the value passed in *num.var1*. Set to zero (default) for random insertion, 1 for increasing insertion, 2 for decreasing insertions.
9			Temporarily, the same as Mode 6. Reserved for future use.

Table of SEARCH status return values

Value	Description of *status*
0	No error, the Index operation was successful.
1	Operation was unsuccessful; i.e. *key* not found.
2	End of *index*. Given on *modes* 3, 6 and 9 when the beginning or end of the *index* is reached.
3	End of data; all records are allocated.
4	File has no Indices, cannot perform an Indexed File operation.

5 Indexed file structure error; given when *key* length **DIM** is less than the actual size of the key from an Index on Modes 2, 3, 6 and 9. Indicates a **DIM**ension error or structure problem, possibly a c-tree file structuring error.

6 Index number not in sequence during creation. You must sequentially define all directories.

7 File is not a Contiguous File.

8 File is already Indexed.

9 Value of *record* is negative or too large.

10 Illegal Index Number.

Example

```
Search #5;4,1,K$,R1,E \ If E Call KeyExists
E=3 \ Search #J,1,0,K$,R1,E \ If E Call Process(K$,R1,E)
```

See also

SEARCH (Modern)

SEARCH (Modern)

Synopsis

Locate a key.

Syntax

SEARCH *rel.op*, *chan.no*, *index.no*{,*num.expr*};{ *var.list*}

Parameters

rel.op is a relational operator.

chan.no is any valid channel number.

index.no is a numeric expression whose integer value identifies an index in the file.

num.expr is an expression yielding a number specifying the timeout value.

var.list is a list of comma separated variables of any dL4 data types passed to this program.

Executable From Keyboard?

No.

Remarks

The **SEARCH** statement has been streamlined for use with full ISAM data files.

```
SEARCH relation #c,index; structure
```

Where *relation* is =, >, >=, <, <=, *index* selects the directory for the operation and *structure* is any structure variable which defines the key parts.

```
Search  = #C, I; Key.   !Exact search
Search  > #C, I; Key.   !Search Greater
Search  < #C, I; Key.   !Search Less
Search  >= #C, I; Key.  !Search Greater or Equal
Search  <= #C, I; Key.  !Search Less than or Equal
Search  < #C,1;         !Position to last key of Index 1
Search  > #C,1;         !Position to first key of Index 1
```

The **SEARCH** statement is used with full ISAM data files to specify an index and set a current record position within the file for further **READ** and **WRITE RECORD** statements. It is not necessary to issue repeated **SEARCH** statements unless a random repositioning is required. If the **SEARCH** succeeds, the current record position is set accordingly and the index used becomes the current index. Relative record access forward or backward is then performed using this index.

When used in conjunction with full ISAM files, the application would perform an initial **SEARCH** and read the current record. A loop, such as **WHILE** or **DO** can then used to read next or previous through the file.

When **SEARCH** is used with older-style indexed files, structure variables can still be used by defining a structure containing the traditional parameters supplied to a **SEARCH** statement. Only the modes =, >, < are supported for Indexed files.

Examples

```
! This is an example of the Search statement
Def Struct CUSTREC
      Member CustNum$[6] : Key "CustNum"
      Member Name$[24] : Item "Name"
      Member 3%,YtdSales : Item "YtdSales"
End Def
Dim CustRec. As CUSTREC
Dim %1, RecAccess
Open #2,"cust.masterfi" As "Full-ISAM"
Map Record #2 As CUSTREC
RecAccess = -2                ! read current record
! sequentially read through a Full-ISAM file,
! from beginning to end
Search > #2,1;
Do
      Try Read Record #2,RecAccess;CustRec. Else Exit Do
      Print CustRec.CustNum$, CustRec.Name$, CustRec.YtdSales
      RecAccess = -1          ! read next (ascending) record
Loop
If Spc(8) <> 52 Print "Unexpected Error: "; Spc(8)
! end of sequential search and now about to delete a specific !
! record first delete the record associated with key value
! 011692, and then search for the deleted key to show that the
! key and record were actually deleted
For I = 1 to 2
      Try
            Search = #2,1;"4549DL"
            Read Record #2, -2;CustRec.
            Delete Record #2
            Print "Deleted Customer Number: 4549DL"
      Else
            Print "Key '4549DL' not found" ! look for this key
      End Try
Next I
Close
```

See also

SEARCH (Traditional)

SELECT CASE

Synopsis

Conditionally execute blocks of statements depending upon the value of an expression.

Syntax1

SELECT CASE *expr*

CASE [*num.lit* **|** [*num.lit* **TO** *num.lit*] **|** [**IS** *rel.op num.lit*]] {, [*num.lit* **|** [*num.lit* **TO** *num.lit*] **|** [**IS** *rel.op num.lit*]]} ...
stmts

CASE ELSE

stmts

ENDSELECT

Syntax2

SELECT CASE *expr*

CASE [*str.lit* **|** [*str.lit* **TO** *str.lit*] **|** [**IS** *rel.op str.lit*]] {, [*str.lit* **|** [*str.lit* **TO** *str.lit*] **|** [**IS** *rel.op str.lit*]]} ...
stmts
CASE ELSE
stmts
ENDSELECT

Parameters

expr is an expression which is evaluated for subsequent selection within the entire block.

stmts is any block of dL4 BASIC statements.

num.lit is a numeric literal.

rel.op is a relational operator.

str.lit is a string literal.

Executable From Keyboard?

No.

Remarks

The **SELECT CASE** statement organizes blocks of statements which are dependent upon the value of a single expression.

For each *expr* value which requires further processing by the application, a **CASE** selection is specified. These may be in the form of a single *expression* which is compared for equality, an inclusive range of values specified in the form *expression* **TO** *expression*, or a value which results in a true relation, such as **IS > 50**. Multiple conditions, separated by comma may be specified.

stmnts are those statements which are to be executed for the selected condition.

CASE ELSE is optional and the associated *stmnts* are executed when no other **CASE** *expression* matched the value of the primary *expr*. If present, **CASE ELSE** must be the last **CASE** in the block.

Examples

```
! This is an example of the Select Case statement
Print 'CS'
Choice = 1
Do Until Choice = 6
      Select Case Choice
      Case 1
            Print @15,Choice + 15;"This is case 1"
      Case 2 To 3
            Print @15,Choice + 15;"This is case 2 or 3"
      Case Is > 3
            Print @15,Choice + 15;"This is case greater than 3"
      Case Else
            Print @15,Choice + 15;"This is default case"
      End Select
      Choice = Choice + 1
Loop
```

See also

CASE, ENDSELECT

SEND

Synopsis

Transmit a message to another port.

Syntax

SEND *num.expr1*, [*str.var* **|** [*num.var2*, *num.var3*]]

Parameters

num.expr1 is an expression yielding a number specifying the receiver's port number.

str.var is a variable of string data type containing the message to transmit.

num.var2 and *num.var3* are variables of numeric data type containing messages to transmit.

Executable From Keyboard?

Yes.

Remarks

If the second parameter is numeric, two numeric expressions must be specified. Their two values are then transmitted. The two variables need not be the same precision.

It is up to the program on the receiving port to execute the appropriate **RECV** or **SIGNAL 2** statement to receive the type (string/numeric) of data transmitted. If that program has an **INTSET** branch enabled, **SEND** will cause an interrupt to occur in it.

SEND is identical in operation to **SIGNAL 1**.

Examples

```
Send 12,22,33
```

```
Send P,A$
```

See also

RECV, SIGNAL

SET

Synopsis

Set driver-class dependent information in a channel.

Syntax

SET *chan.expr expr.list*

Parameters

chan.expr is a driver-class dependent channel expression.

expr.list is an arbitrary number of comma separated expressions or variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

Refer to the dL4 Files and Devices reference manual for information on a specific driver.

Examples

```
Set #1,0,0,0;CustRec.Name$, "Name"
Set #1,0,1,0;CustRec.Address1$, "Address1"
Set #1,0,3,0;CustRec.City$, "City"
Set #1,0,4,0;CustRec.State$, "State"
Set #1,0,5,0;CustRec.Zip, "Zip"
```

See also

GET

SETFP

Synopsis

Set file position for next access.

Syntax

SETFP *chan.expr*

Parameters

chan.expr is a driver-class dependent channel expression.

Executable From Keyboard?

Yes.

Remarks

A semicolon must terminate the *chan.expr*.

SETFP specifies a new file position on a *channel* for the next sequential access **READ**, **WRITE**, etc. not specifying a *record* or *byte displacement*. If the next transfer specifies its own *record* and *byte displacement* position, the former position is overridden. The *byte displacement* specification is optional and, if not included, will default to byte zero of the selected record.

SETFP to *record 0*, *byte displacement 0* is identical in operation to a **REWIND**.

Examples

```
Setfp #6,R,I;
```

```
Setfp #5,0,0; ! Same as REWIND #5;
```

See also

REWIND, READ, WRITE

SIGNAL 1 | 2

Synopsis

Transmit/Receive a message.

Syntax1

SIGNAL 1, *num.expr1*, [*str.expr* **|** [*num.expr2*, *num.expr3*]]

Syntax2

SIGNAL 2, *num.var1*, [*str.var* **|** [*num.var2*, *num.var3*]] {, *num.expr4*}

Parameters

num.expr1 is an expression yielding a number specifying the destination port number.

str.expr is an expression yielding a string specifying the destination message.

num.expr2 and *num.expr3* are expressions yielding numbers specifying the destination message.

num.var1 is a variable of numeric data type receiving the sender's port number.

num.var2 and *num.var3* are variables of numeric data types to contain the receive message.

num.expr4 is an expression yielding a number specifying a maximum wait period.

Executable From Keyboard?

No.

Remarks

Syntax1:

The *string* expression or 2 *num.expr values* are placed into the communication buffer for transmission to the selected *port*. Messages may be transmitted to your current *port number*, or any *port number* that is logged on. An error 153 is returned if the destination port is invalid.

Messages are FIFO (First in, First out). Messages include those transmitted using **SEND**, **SIGNAL 1**, and **CALL $TRXCO**.

If numeric data is transmitted, full floating point precision is transmitted. When numeric values are received with **SIGNAL 2**, they are converted to the precision of the supplied *value1* and *value2 num.vars*.

An error is generated if the communication file is full, or an illegal *port number* is specified. Messages transmitted to a *port* not signed into a dL4 process are discarded, and no error is generated.

Messages awaiting a *port* are deleted when that *port* ends its session.

Syntax2:

The optional *delay* for **SIGNAL 2** is any *num.expr* which, after evaluation is truncated to an integer to specify a delay period (in tenth-seconds) during which the program awaits a message. If zero, or not included, no pause is invoked, but any currently waiting message is received. Any message appearing during a specified delay allows **SIGNAL** to accept the transmitted data and resume program execution immediately. If no message appears during the entire delay, *port* is set to -1.

A scan is performed for the oldest **SIGNAL 1** or **SEND** message transmitted to your *port number*. If found, *port* is set to the *port number* of the sender. If no messages are waiting, *port* is set to -1.

The received message is copied into *string* or *value1* and *value2* as specified. It is the programs' responsibility to select the same format (*str.var* or 2 *num.vars*) used by the sender. The sender's *port number* is returned in the supplied *port* variable. Typically, an application designer chooses one format for all message transmission and reception.

If *delay* is specified and no message is waiting, the program is paused for the specified number of tenth-seconds. If any message is transmitted during the *delay*, the pause is terminated allowing immediate reception. A -1 is returned in *port* if no message is received within the *delay* period.

The **[SIGNAL]** input character (usually **CTRL B**) transmits a message of 2 numeric zeros or a null string to your current port which may be retrieved using **SIGNAL 2**.

All messages may be cleared by performing repeated **SIGNAL 2** statements until *port* is returned with -1, or by issuing a **SIGNAL 6**.

If the program has an **INTSET** in effect, transmission of a message by another port or **[SIGNAL]** character performs an interrupt branch.

Messages awaiting a *port number* are deleted when that *port number* ends its session.

Examples

```
Signal 1,P,A,B*100
Signal 2,P,A,B,300 !Wait 30 seconds
```

See also

RECV, SEND

SIGNAL 3

Synopsis

Suspend program operation.

Syntax

SIGNAL 3, *num.expr*

Parameters

num.expr is an expression yielding tenth-seconds pause time.

Executable From Keyboard?

No.

Remarks

The program is unconditionally suspended for the number of tenth-seconds specified in *delay*. An **[ESCAPE]** without **ESC**ape branching or **[ABORT]** terminates a pause. If the application has an **INTSET** defined, the **[INTERRUPT]** or **[SIGNAL]** will terminate the pause and perform the branch.

If *delay* is zero, the statement is ignored and no pause is performed.

Examples

```
Signal 3,30 !Pause 3 seconds
```

See also

PAUSE

SIGNAL 5

Synopsis

Receive system signal.

Syntax

SIGNAL 5, *num.var1*, *num.var2*, *num.var3* {, *num.expr4* }

Parameters

num.var1 is an expression yielding the transmitter's port number.

num.var2 is a variables of numeric data type receiving the type of system message.

num.var3 is a variables of numeric data type receiving specific system message.

num.expr4 is an expression yielding a number specifying a maximum wait period.

Executable From Keyboard?

No.

Remarks

A scan is made for the oldest system message directed to your *port number*. If no system message is waiting, *port* is set to -1.

If a system message is waiting, *port* is set to -2, *value1* is set to the type of system message, and *value2* returns specific information.

The only system message currently implemented is for **INPUT** timed-out. This occurs when an application performs an **INPUT TIM**, and the input times-out without response from the keyboard. *port* is set to -2, *value1* is set to 0, and *value2* is set to the number of characters entered prior to time-out.

Unless **OPTION INPUT TIMEOUT SIGNAL OFF** is used, programs performing an **INPUT TIM** should immediately follow with a **SIGNAL 5** to check the sense of the timed input and prevent overflowing communication resources. If *port* returns -1, a response was entered within the prescribed time limit.

Examples

```
Signal 5,P,A,B,300 !Wait 30 seconds
```

See also

SIGNAL 6

SIGNAL 6

Synopsis

Clear outstanding signals.

Syntax

SIGNAL 6, *num.expr1*, *num.var2*, *num.var2*

Parameters

num.expr1 is an expression yielding a number to specify a signal type.

num.var2 are variables of numeric data type used for syntax only.

Executable From Keyboard?

No.

Remarks

All user messages, system messages or both may be cleared using **SIGNAL 6**. The *type* selects the messages to be cleared from the system:

type	**Function Performed**
-1	Remove all user messages; **SIGNAL 1**, **SEND**.
-2	Remove all system messages.
-3	Remove both user and system messages.

SIGNAL 6 may be used to clear the message queue for this *port number* . Messages are automatically deleted when a *port* ends its session (**BYE**, **SYSTEM 0**, or terminated **SPAWN** commands).

Examples

```
Signal 6,-3,A,A
```

See also

SIGNAL 5

SIZE

Synopsis

Select the size of a window component.

Syntax

SIZE { *chan.expr* } *w,h*

Parameters

chan.expr is a driver-class dependent channel expression.

w,h are the width and height for the window component.

Executable From Keyboard?

Yes.

Remarks

Depending on the driver, it is possible to change the size of the window on the screen or control which part of the window is displayed. Refer to the dL4 Files and Devices reference manual for more information about windows.

Examples

```
! This is an example of the Size statement
Dim S$[1]
Print 'CS'
W = 41 \ H = 12
Open #1,{" Windows ","TITL",W,H} As "Window"
For I=1 TO 5
      Print #1;"1234567890123456789012345678901234567890"
      Size #1; W - (I * 2), H - (I * 2)
      Read #1;S$
      Erase #1
Next I
```

See also

MOVE, WINDOW

SPAWN

Synopsis

Launch a background BASIC program.

Syntax

SPAWN *filename* {, *num.var* }

Parameters

filename is a string literal or expression containing a name which is optionally preceded by a relative or absolute directory pathname.

num.var is a numeric variable in which the program's port number is returned.

Executable From Keyboard?

No.

Remarks

SPAWN creates another process to run the BASIC program. This child process inherits the current environment and current working directory. All channels are closed, and no **COM** or **CHAIN WRITE** variables may be passed.

SPAWN is simpler than the **PORT** or **CALL TRXCO()** functions to launch a *phantom port* into a BASIC program. It is especially suited for launching background reports, spoolers and other programs communicated with using **SEND**, **RECV** or **SIGNAL**.

When the program terminates to *command mode* or BASIC *program mode* from **STOP**, non-trapped error, **END**, **CHAIN ""**, or **SYSTEM 0/1**, the process terminates releasing the *port*.

SPAWN locates an unused *port number* scanning backward from the value of the runtime parameter **MAXPORT**.

The optional *port num.var* is returned with the *port number* assigned to the background program. **SEND** and **SIGNAL**, as well as **CALL TRXCO()** and **PORT** statements may be used to communicate with a *port* initiated by **SPAWN**.

Examples

```
Spawn "1/SPOOLER"
```

```
Spawn A$,K ! Start program, get port number
```

See also

PORT, SIGNAL, SYSTEM

STOP

Synopsis

Abnormally terminate a program.

Syntax

STOP {*str.expr*}

Parameters

str.expr is an expression yielding a string value.

Executable From Keyboard?

No.

Remarks

The **STOP** statement terminates a running program and is functionally identical to the **SUSPEND** statement.

str.expr is an optional string expression to be displayed.

If the program was executed from the SCOPE Interactive Development Environment (IDE) a **STOP** statement causes program execution to cease, and returns the user to *debug mode*.

The **STOP** statement is usually used to indicate an error condition or some other abnormal mode of program termination. A **STOP** statement, non-trapped **[ESCAPE]** or **[ABORT]** causes program execution to cease. The program is left in the partition , channels remain open, and variables retain their values. The user is returned to *debug mode* with the display:

```
--> [0] program:stmt.no;sub-stmt.no
program - Root program
STOP = str.expr

STOP at program:stmt.no statement

Type ? for help
dbg>
```

program is the *filename* of the current BASIC program, *stmt.no* is the statement number containing the **STOP**, *sub-stmt.no* is the statement within the line, and *statement* is the actual BASIC statement.

If the running program was started by **SWAP**, the various levels are displayed:

```
--> [1] program2:80;1
  program2 - SWAPed
    [0] 60;1

STOP = in program2

STOP at program2:80 STOP "in program2"

Type ? for help
dbg>
```

This example indicates that a **STOP** occurred in program2, which was swapped to from a program at line 60;1 in that program.

If the program was executed from another environment, such as the Operating System prompt, via the applicable **RUN** *filename* command, the user is returned to that environment with a display:

```
STOP at program:stmt.no;sub-stmt.no
str.expr
prompt
```

program is the *filename* of the current BASIC program, *stmt.no* is the statement number containing the **STOP**, *sub-stmt.no* is the statement within the line, and *prompt* is the environment prompt.

If the running program was started by **SWAP**, the various levels are displayed:

```
STOP at program2:80;1
SWAP at program1:60;1
in program2
$
```

Other statements may follow a **STOP** in the program.

Examples

```
100 Stop
```

```
220 Stop "Irrecoverable error, contact support"
```

See also

SUSPEND

SUB

Synopsis

Define a subroutine.

Syntax

SUB *proc.name* ({*parm.list* })

Parameters

proc.name is the procedure name.

parm.list is a list of variables associated with parameters passed, optionally followed by three dots ("...").

Executable From Keyboard?

No.

Remarks

SUB declares a subroutine which operates as a separate program block within a program unit. A subroutine operates upon, and return values through, supplied parameters passed by reference.

A *proc.name* may be from one-to-thirty-two characters in length. Structures may be passed and operated upon.

Whenever a subroutine is to be used before its definition within the current program unit or program, or physically resides in another program, a **DECLARE** statement must occur before its first use.

Subroutines may be written to allow the caller to pass other than a fixed list of parameters. Parameter types and number are not checked by the compiler or interpreter. Rather, it is left to the subroutine to process each of the arguments passed by a caller.

To define a subroutine of this type, the following general forms are supported:

```
Sub name (...)
```

The definition of the subroutine itself specifies '**...**' informing the compiler and interpreter to leave the parameter type and number checking to the subroutine.

It is also permitted to define a subroutine which has a known (required) list of parameters, followed by additional optional parameters. Optional parameters must be the last parameters in the function definition. The following example requires a numeric parameter and a string parameter, followed by an optional number of parameters.

```
Sub proc.name (parameter1, parameter2$, ... }
```

Subroutines of this type utilize the **ENTER** statement to accept optional parameters.

Examples

```
Sub VerifyDate(D$, ...)
```

See also

FUNCTION

SUSPEND

Synopsis

Abnormally terminate a program.

Syntax

SUSPEND {*str.expr*}

Parameters

str.expr is an expression yielding a string value.

Executable From Keyboard?

No.

Remarks

The **SUSPEND** statement is functionally identical to the **STOP** statement.

str.expr is an optional string expression to be displayed.

If the program was executed from the SCOPE Interactive Development Environment (IDE) a **SUSPEND** statement causes program execution to cease, and returns the user to *debug mode*.

The **SUSPEND** statement is usually used to indicate an error condition or some other abnormal mode of program termination. A **SUSPEND** statement, non-trapped **[ESCAPE]** or **[ABORT]** causes program execution to cease. The program is left in the partition , channels remain open, and variables retain their values. The user is returned to *debug mode* with the display:

```
--> [0] program:stmt.no;sub-stmt.no
program - Root program
STOP = str.expr

STOP at program:stmt.no statement

      Type ? for help
dbg>
```

program is the *filename* of the current BASIC program, *stmt.no* is the statement number containing the **SUSPEND**, *sub-stmt.no* is the statement within the line, and *statement* is the actual BASIC statement.

If the running program was started by **SWAP**, the various levels are displayed:

```
--> [1] program2:80;1
  program2 - SWAPed
    [0] 80;1
STOP = in program2

STOP at program2:60 SUSPEND "in program2"

Type ? for help
dbg>
```

This example indicates that a **SUSPEND** occurred in program2, which was swapped to from a program at line 60;1 in that program.

If the program was executed from another environment, such as the Operating System prompt, via the applicable **RUN** *filename* command, the user is returned to that environment with a display:

```
STOP at program:stmt.no;sub-stmt.no
str.expr
prompt
```

program is the *filename* of the current BASIC program, *stmt.no* is the statement number containing the **SUSPEND**, *sub-stmt.no* is the statement within the line, and *prompt* is the environment prompt.

If the running program was started by **SWAP**, the various levels are displayed:

```
STOP at program2:80;1
SWAP at program1:60;1
in program2
$
```

Other statements may follow a **SUSPEND** in the program.

Examples

```
100 Suspend
220 Suspend "Irrecoverable error, contact support"
```

See also

STOP

SWAP

Synopsis

Suspend current program and execute another BASIC program.

Syntax

SWAP { *num.expr*,} *filename*

Parameters

num.expr selects whether channels and common variables are to be passed to the **SWAP**ped program.

filename is a string literal or expression containing a dL4 BASIC program filename which is optionally preceded by a relative or absolute directory pathname.

Executable From Keyboard?

No.

Remarks

num.exp is a *mode* which, after evaluation is truncated to an integer to select channel and common variable pass-along into the **SWAP** program. If *mode* is omitted, mode 2 is assumed.

SWAP suspends execution of the current program, saves all open channels and variables, and then executes the child program. This *child* (swapped) program inherits the current environment, variables, open channels, and current working directory from the *parent* (calling program).

The selected *filename.expr* is loaded following the same rules as **CHAIN**. Common variables declared using **COM** or **CHAIN WRITE** statements following the **SWAP** statement, and open channels passed to the child process are processed according to the *mode* as follows:

mode	**Function Performed**
0	Close all open files in the child. Do not pass any common variables, i.e. ignore **COM** and **CHAIN WRITE**.
1	Pass all open channels to the child, and process the common variables according to the rules for **COM** or **CHAIN WRITE**.
2	(default) Close all open files for the child, but process any common variables according to the rules for **COM** or **CHAIN WRITE**.

The *parent* is the initial program that executed the **SWAP** statement.

The *child* is each program executed by the **SWAP** statement . The *parent* is suspended while the *child* runs. When a *child* terminates, the *parent* continues automatically, unaware of the events of the *child*.

A *child* can itself be considered a *parent* if it performs a **SWAP** statement. **SWAP** statements may nest until memory is exhausted. A unique relationship exists between the *parent* and *child* programs. Variables and File Positions all flow forward from *parent* to *child*, however no information is passed back to the *parent* upon termination of a child.

When a *child* inherits open files, the Operating System uses the same entries in the dL4 channel table. A *child* can change its copy of the current pointers as well as add or remove locks on records. These operations may confuse the *parent*.

When the **SWAP** program terminates using **END**, **SYSTEM**, or **CHAIN** "", the calling program resumes execution at the statement immediately following the **SWAP**. To the caller, it appears as if the **SWAP** statement never occurred.

If a non-trapped **[ESCAPE]**, **[ABORT]** or **STOP** statement occurs, the swapped program is terminated to *BASIC debug mode* to allow debugging. Execution of a termination statement while in debug mode (**END**, **SYSTEM**, or **CHAIN** ""), terminates the swap level and resumes execution in the calling program.

Data may be passed from a swapped program back to the calling program using temporary files, or by placing it into the type-ahead buffer using **CALL $INPBUF**. Data may not be transferred to the calling program using common variables.

Important: a child program can communicate with other ports using **CALL 98**, etc., and assumes the same port # as the parent.

Examples

```
Swap "23/PROGRAM3"
```

```
Swap 0,A$
```

See also

CHAIN, SPAWN

SYSTEM

Synopsis

Execute operating system specific commands.

Syntax1

SYSTEM *str.expr* [,*num.var*]

Syntax2

SYSTEM *num.expr* {, *expr*} { ; *num.expr* {, *expr*}} ...

Parameters

str.expr is a command passed to the native operating system.

num.var is a variable of numeric data type to return the status.

num.expr is an expression yielding an operation to be performed.

expr is a numeric or a string expression, or a variable, yielding a parameter.

Executable From Keyboard?

Yes.

Remarks

num.expr may be a *mode* which, after evaluation is truncated to an integer and used to specify the operation to be performed. Some *modes* require a second *parameter* which is any *num.expr* which, after evaluation is truncated to an integer. The *parameters* are separated by the *mode* using a comma.

Multiple **SYSTEM** *modes* may be invoked separating each with a semicolon.

str.expr is passed directly to the Operating System. This command can be used to launch another application, or perform a system command. If an optional *num.var* follows, the status that is returned from the Operating System is stored.

Following execution of the system command by the operating system, the program resumes operation.

If the system command performs any output, your screen will be compromised unless a new Window was opened prior to, and closed after, the **SYSTEM** command.

mode	**Operation Performed**
0	Terminate a session (**BYE** command). You may also terminate other users by including a *port number* as an additional *parameter*. The general form: **SYSTEM 0,N** terminates port **N**.
1	Clear the port's program partition (issue a **NEW** command), and stop the program.
4	Un-assign all non-common variables. This allows re-dimensioning of partition space as long as all variables to be used are re-assigned.
5	Un-assign all variables. Same effect as **SYSTEM 4**, except common variables (**COM** and **CHAIN WRITE**) are also affected.
8	Enable terminal echo. Each character input will be echoed by the system to the terminal.
9	Disable terminal echo. Each character input is received by the system, but not echoed to the terminal. This feature allows for password or other secretive input.
14	Enable Binary Input mode. All characters input are directly accepted as data. This includes **end-of-line**, requiring the use of character limited **INPUT**.
15	Disable Binary Input mode. Normal character processing is resumed.
16	Enable Binary Output mode.
17	Disable Binary Output mode.

20 Enable Trace mode. See Trace Mode.

21 Disable Trace mode.

26 Automatic limited input. Causes character limited input to terminate when the specified number of characters have been entered. Affects **INPUT** statement.

27 Disable Automatic limited input. Causes character limited input to require an **[ENTER]** (usually return) to be entered, even after the specified limit has been reached. Entry of each extra character sounds the terminal bell until **end-of-line** is entered.

28 Get value of Environment Variable. This function requires the special form: **SYSTEM 28, *str.var*** where *str.var* initially contains the name of an environment variable. If found, its value is overwritten in the string, otherwise the *str.var* is set to "". If SYSTEM 29 has been used to set an alternate source and the value is not found in the environment, then the alternate source will be searched.

29 Set alternate sources of Environment Variables. This function requires a special form: **SYSTEM 29, *str.var*** where *str.var* contains an alternate source path for variables that are not defined in the environment. On Windows systems, this path is an application registry key within the user or system software keys. This mode is not supported on Unix systems.

30 Execute the native operating system command specified by the subsequent string parameter and, optionally, return the command status in a numeric parameter. This function requires one of two special forms: **SYSTEM 30,str.expr** or **SYSTEM 30,str.expr,num.var**. The operating system command is not permitted to perform input or output to the user terminal and thus the command execution is invisible to the user.

31 Execute the client operating system command specified by the subsequent string parameter, wait for the command to complete, and, optionally, return the command status in a numeric parameter. This function requires one of two special forms: **SYSTEM 31,str.expr** or **SYSTEM 31,str.expr,num.var**. If the application is running remotely, the command will be executed on the local system. For example, if a user is connecting to the application system via the dL4Term terminal emulator, the command will be executed on the user's Windows system on which dL4Term is running. If the application is running under dL4 for Windows, this mode is identical to '**SYSTEM** "command",status'. This mode can only be used with supported terminal emulators and may require configuration of the client system software to enable local command execution.

32 Get the amount of available space on a file system in units of 512 bytes. This mode requires a special form: **SYSTEM 32, *str.expr,num.var*** where *str.expr* is the path of a directory or file on the file system and *num.var* is a variable that receives the number of available 512 byte blocks.

33 Start the client operating system command specified by the subsequent string parameter and, optionally, return the initialization status in a numeric parameter. Unlike **SYSTEM 31**, the statement does not wait for the completion of the command. This function requires one of two special forms: **SYSTEM 33,str.expr** or **SYSTEM 33,str.expr,num.var**. If the application is running remotely, the command will be executed on the local system. For example, if a user is connecting to the application system via the dL4Term terminal emulator, the command will be executed on the user's Windows system on which dL4Term is running. If the application is running under dL4 for Windows, the command will run on the same system as the application. This mode can only be used with supported terminal emulators and may require configuration of the client system software to enable local command execution.

Each port is returned to its normal operational modes (8, 15, 17, 19, 21, and 26) when a program is completed or aborted.

Examples

```
System 14;16;
```

See also

TRACE

Synopsis

Control non-interactive statement tracing.

Syntax

TRACE [**OFF** | [**ON** { *chan.no* }]]

Parameters

chan.no is a valid channel number.

Executable From Keyboard?

Yes.

Remarks

Trace mode is used when it is desirable to observe the statement number program flow without performing single steps. **SYSTEM 20** or **TRACE ON** enables tracing; **SYSTEM 21** or **TRACE OFF** turns trace off. These statements may be used in *immediate mode*, or imbedded within specific code segments of a program. For each statement executed, the statement number *stmt.no* and sub-statement number *sub-stmt.no* (statements on the same BASIC line) is printed. The current program and procedure names will be printed if the names are available.

The **TRACE ON** statement can be followed by an optional *channel* number for redirecting trace output to a file or driver.

The *channel* number that is given must be opened prior to executing the **TRACE** statement. If the *channel* is subsequently closed, trace output defaults to the terminal. The following information is output during *trace mode*:

```
[statement number; sub-statement number]
```

Tracing is automatically disabled when another program is loaded using **CHAIN**, **SWAP**, or **SPAWN**.

Examples

```
Trace On
```

```
Trace Off
```

```
Trace On #5
```

See also

SYSTEM 20, SYSTEM 21

TRY

Synopsis

Specify a statement/block to execute when an error occurs in a specific statement/block.

Syntax1

TRY *stmt1* **ELSE** *stmt2*

Syntax2

TRY
stmts
ELSE IF *bool.expr*
stmts
ELSE
stmts
END TRY

Parameters

stmt1 and *stmt2* are any valid dL4 BASIC statements.

bool.expr is an expression evaluated to produce a boolean value.

stmts is any block of dL4 BASIC statements.

Executable From Keyboard?

No.

Remarks

TRY provides for the temporarily redirection of error branching within a block. If any program error branching is in effect, it is temporarily suspended for any error other than **ESCAPE** for the duration of the **TRY** statement or block. Error branching is restored upon the completion of the line or block.

Examples

```
Try
      Open #2,"cust.master"
      Print "Opened cust.master on channel 2"
Else
      Print "Unexpected Error: ";Spc(8); " at line ";Spc(10)
End Try
Print "Terminating program"
Close
```

See also

RETRY

UNLOCK

Synopsis

Unlock current locked record.

Syntax

UNLOCK *chan.no*{, *chan.no*} ...

Parameters

chan.no is any valid channel number.

Executable From Keyboard?

Yes.

Remarks

Any record locked by your program on the specified *channel* becomes unlocked. For most drivers, no error is generated if no record has been locked. A record locked by another user cannot be unlocked.

Generally, **UNLOCK** is only used in special circumstances, such as having one file open on two channels. In this case, **UNLOCK** can be used to prevent the program from locking itself out of a record.

The statement **WRITE #** *channel* **;;** is identical to **UNLOCK**.

Examples

```
Unlock #5, #K, #K+1
```

See also

READ, WRITE

WEND

Synopsis

End a **WHILE** block.

Syntax

WEND

Parameters

None.

Executable From Keyboard?

No.

Remarks

Each **WEND** statement must match exactly one previous **WHILE** statement. The compiler ensures that all loops are properly matched.

Examples

```
Print 'CS'
Counter = 5

While Counter
      Print Counter,
      Counter = Counter - 1
Wend
Print
```

See also

DO, ENDIF, LOOP, NEXT

WHILE

Synopsis

Begin a loop to be performed as long as the expression is true.

Syntax

WHILE *bool.expr*

Parameters

bool.expr is an expression evaluated to produce a boolean value.

Executable From Keyboard?

Yes.

Remarks

Program loops may be established using the **WHILE** and **WEND** statements as a means of blocking a set of repeated statements. **WHILE** and **WEND** statements provide additional flexibility and looping control beyond the simple **FOR / NEXT**.

WHILE provides for looping as long as the *bool.expr* remains true. The *bool.expr* is tested prior to performing each loop. The loop is terminated once the *bool.expr* is false.

WHILE is identical in behavior to **DO WHILE ... LOOP.**

Unlike **FOR**, **WHILE** loops may nest indefinitely. In addition, each **WHILE** loop must contain exactly one matching **WEND** statement. The compiler ensures that all loops are properly matched. Although not recommended, branching from outside to inside a **WHILE** loop will not cause an error, rather the program will remain in the loop until it terminates. The **WHILE** statement itself need not be executed to commence looping.

```
Goto Label
While Value > 100
      Print Value;
      Label: Value = Value + 1
Wend
```

Examples

```
Print 'CS'
Counter = 5
While Counter
      Print Counter,
      Counter = Counter - 1
Wend
Print
```

See also

DO, DO LOOP, DO WHILE , FOR, LOOP

WINDOW CLEAR

Synopsis

Clear all Dynamic Windows and screen.

Syntax

WINDOW CLEAR

Parameters

None.

Executable From Keyboard?

Yes.

Remarks

The recommended method for using Windows under dL4 is to open a channel to the Window driver as described in the Window driver section of the dL4 Files and Devices reference manual. The **WINDOW** statements are provided for compatibility and programmer convenience.

WINDOW CLEAR clears all Windows back to Window Zero and clears the screen.

Examples

```
Window Clear
```

See also

WINDOW CLOSE

WINDOW CLOSE

Synopsis

Delete current Dynamic Window and repaint the original underlying data.

Syntax

WINDOW CLOSE

Parameters

None.

Executable From Keyboard?

Yes.

Remarks

The recommended method for using Windows under dL4 is to open a channel to the Window driver as described in the Window driver section of the dL4 Files and Devices reference manual. The **WINDOW** statements are provided for compatibility and programmer convenience.

WINDOW CLOSE deletes the current Window repainting the original underlying data. **MSC(33)** and **MSC(34)** now reflect the size of the previous Window and **MSC(42)** is decremented. A Window must always be deleted at the same *parent* / *child* SWAP level it was created. For example, you perform a **WINDOW OPEN** in program A, then **CHAIN** to program B, which in turn performs a **SWAP** or **[Hot-Key]** swap to program C (a *child* of B). If program C opens any windows, then **WINDOW CLOSE** should be performed before returning control to program B. A **WINDOW CLOSE** will be performed automatically for any windows that program C opened, but did not close.

Examples

Window Close

See also

WINDOW CLEAR

WINDOW MODIFY

Synopsis

Change the size or position of the current Dynamic Window.

Syntax

WINDOW MODIFY *@x1,yl*; [**SIZE** *w,h*; **| TO** *@x2,y2*;] {**USING** *str.expr*}

Parameters

x1,y1 are the column, row coordinates of the upper left corner.

w,h identify the width and height.

x2,y2 are the lower right column, row coordinates.

str.expr is a string expression yielding a window title.

Executable From Keyboard?

Yes.

Remarks

The recommended method for using Windows under dL4 is to open a channel to the Window driver as described in the Window driver section of the dL4 Files and Devices reference manual. The **WINDOW** statements are provided for compatibility and programmer convenience.

WINDOW MODIFY is used to change the size of the current Window based upon the supplied *parameters*. Functions **MSC(33)** and **MSC(34)** are updated to reflect the current size. The size of a Window may be changed as many times as desired but it cannot extend beyond the original *parameters* specified to **WINDOW OPEN**. If the Window must be enlarged, perform a **WINDOW CLOSE**, followed by another **WINDOW OPEN**. **WINDOW MODIFY** may be used to create your own borders, to modify the border created by **WINDOW OPEN**, or implement a series of panes inside a Window that can be accessed randomly.

WINDOW MODIFY merely redefines the writable region inside a window. The window itself is not actually closed and re-opened. No underlying data is revealed or hidden by this statement.

Examples

```
Window Modify @7,7 To @62,18;
```

```
Window Modify @7,7; Size 80,24; Using "Help"
```

See also

WINDOW OPEN

WINDOW OFF

Synopsis

Redirect screen I/O from Dynamic Window to root window.

Syntax

WINDOW OFF

Parameters

None.

Executable From Keyboard?

Yes.

Remarks

The recommended method for using Windows under dL4 is to open a channel to the Window driver as described in the Window driver section of the dL4 Files and Devices reference manual. The **WINDOW** statements are provided for compatibility and programmer convenience.

WINDOW OFF temporarily redirects output to the root window channel. Further screen operations are not output to the current window and access outside the current Window is allowed. If Dynamic Window was previously on and protected fields were used, they won't be protected.

WINDOW OFF and **ON** may also be used when secondary Windows (other than the first full-screen) are opened, and access to the full screen is desired. When Dynamic Windows is turned off, cursor access is to the full screen. When Dynamic Windows is again turned on, the cursor is logically re-positioned to the last tracked position. Turning Dynamic Windows off to modify data outside the screen should be limited to the display of errors or messages in a common area. The Dynamic Window system is unaware of any changes to the screen.

Examples

```
Window Off
```

See also

WINDOW ON

WINDOW ON

Synopsis

Redirect screen I/O to current Dynamic Window.

Syntax

WINDOW ON

Parameters

None.

Executable From Keyboard?

Yes.

Remarks

The recommended method for using Windows under dL4 is to open a channel to the Window driver as described in the Window driver section of the dL4 Files and Devices reference manual. The **WINDOW** statements are provided for compatibility and programmer convenience.

WINDOW ON enables Dynamic Windows and should precede any other **WINDOW** function. The Dynamic Window system is initialized by clearing the screen. Subsequent **WINDOW ON** statements are ignored.

WINDOW OFF and **ON** may also be used when secondary Windows (other than the first full-screen) are opened, and access to the full screen is desired. When Dynamic Windows is turned off, cursor access is to the full screen. When Dynamic Windows is again turned on, the cursor is logically re-positioned to the last tracked position. Turning Dynamic Windows off to modify data outside the screen should be limited to the display of errors or messages in a common area. The Dynamic Window system is unaware of any changes to the screen.

Examples

```
Window On
```

See also

WINDOW OFF

WINDOW OPEN

Synopsis

Create a new Dynamic Window.

Syntax

WINDOW OPEN *@x1,yl*; [**SIZE** *w,h*; **| TO** *@x2,y2*;] {**USING** *str.expr*}

Parameters

x1,y1 are the column, row coordinates of the upper left corner of the Window.

w,h identify the Window width and height.

x2,y2 are the lower right column, row coordinates of the Window.

str.expr is a string expression yielding a Window title.

Executable From Keyboard?

Yes.

Remarks

The recommended method for using Windows under dL4 is to open a channel to the Window driver as described in the Window driver section of the dL4 Files and Devices reference manual. The **WINDOW** statements are provided for compatibility and programmer convenience.

@ specifies a *crt.expr* in the form of a Cursor Address. *x1* is any *num.expr* which, after evaluation is truncated to an integer to select the Upper Left Column for the Window. *y1* is any *num.expr* which, after evaluation is truncated to an integer to select the Upper Left Row. Following the *crt.expr* must be a semicolon.

SIZE selects the size of a Window in columns and rows. **TO** specifies the size using a *crt.expr* in the form of a Cursor Address of the last character position in the Window. Either form may be used. If **SIZE** is used, *w* is any *num.expr* which, after evaluation is truncated to an integer to select the number of columns. *h* is any *num.expr* which, after evaluation is truncated to an integer to select the number of rows. If **TO** is specified, *x2* is any *num.expr* which, after evaluation is truncated to an integer to select the Lower Right Column for the Window. *y2* is any *num.expr* which, after evaluation is truncated to an integer to select the Lower Right Row. Following the *crt.expr* must be a semicolon.

The optional **USING** *str.expr* is any string expression to be centered and printed as the title of a Window. The size must be less than the number of columns in the Window, or it is truncated. The inclusion of **USING** specifies that a graphical border is to be placed around the Window. The *str.expr* may be a null-string for a box without heading. The specification of a graphical border reduces the usable space in the Window by one row, and column on the top, bottom and each side.

Whenever a program terminates, Dynamic Windows is turned off. If a program is terminated by **[ESCAPE]**, **[ABORT]**, **STOP**, or Breakpoint, debugging is permitted and Windows remain open, otherwise all Windows are cleared.

Examples

```
Window Open @5,5; To @60,20; Using "Help"
```

```
Window Open @0,0; Size 80,24;
```

See also

WINDOW MODIFY

WOPEN

Synopsis

Open an existing file for Write-Only access.

Syntax1

WOPEN *chan.no*, *file.spec.str* {**AS** *driver-class* **|** *driver-name* } {, {*chan.no*,} *file.spec.str* {**AS** *driver-class* **|** *driver-name*}} ...

Syntax2

WOPEN *chan.no*, *file.spec.items* **AS** *driver-class* **|** *driver-name* {, {*chan.no*,} *file.spec.items* **AS** *driver-class* **|** *driver-name*} ...

Parameters

chan.no identifies a valid channel number, which the program uses for subsequent references to the file.

file.spec.str, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

driver-class specifies the driver-class, instead of using a default driver-class derived from the file.spec.

driver-name specifies the driver-name, instead of using a default driver-class derived from the file.spec.

file.spec.items, which is described in detail in Chapter 9 of this guide, identifies a valid dL4 file specification used to open a file.

Executable From Keyboard?

Yes.

Remarks

Similar to the **OPEN** statement except access is write-only.

Examples

```
Wopen #2,"cust.masterfi" AS "Full-ISAM"
```

See also

BUILD, CLOSE, EOPEN, OPEN, ROPEN

WRITE

Synopsis

Write variables to a channel.

Syntax

WRITE *chan.expr* *var.list* {;}

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

";" unlocks the record after a successful **WRITE**.

Executable From Keyboard?

No.

Remarks

WRITE transfers data from any *dL4 data type* to the file opened on the selected *chan.expr*.

If the variable in the list is an *array.var* or *mat.var*, only the first element is written. *Subscripts* may be used to select any individual element to be transferred. The number of bytes transferred is based upon the variable **DIM**ensioned size. The transfer is performed according the rules for the array element type.

If the variable in the list is a simple *num.var* or *date.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a *str.var*, its size may be controlled by *subscripts*. Refer to the dL4 Files and Devices reference manual for a description of how each specific file type and driver transfer data.

The optional semicolon (;**)** terminator is used to release the automatic record-lock applied to the supplied *record* in the *chan.expr*.

Examples

```
Write #3,R1,100;A,B$,C[12]
```

```
Write #C,R;A$
```

See also

READ, READ RECORD, MAT WRITE, WRITE RECORD, WRLOCK

WRITE RECORD

Synopsis

Write an entire structure.

Syntax

WRITE RECORD *chan.expr struct.var* {;}

Parameters

chan.expr is a driver-class dependent channel expression.

struct.var is a variable of structure data type.

";" unlocks the record after a successful **WRITE**.

Executable From Keyboard?

Yes.

Remarks

The **WRITE RECORD** statement is similar to normal **WRITE** of a record except that item numbers may be supplied by the **ITEM** option of the **MEMBER** statement.

The example illustrates the use of structures and the new statements on an old-style existing Indexed or Contiguous file.

```
Def Struct DRCR
  Member 3%, Debit  : Item 0 !Note item displacement is
  Member 3%, Credit : Item 6 !relative to where we begin a
                             !transfer
End Def

Def Struct Cust
  Member Number$[8]     : Item 0
  Member Name$[30]      : Item 10
  Member Addr$[30]      : Item 42
  Member Balance. As DRCR     : Item 74
  Member 1%,LastOrderNumb#    : Item 86
End Def

Dim Customer. As Cust

Write Record #c,r,b,t;Customer.
```

is identical to:

```
Write #c,r,b+0,t;Customer.Number$
Write #c,r,b+10,t;Customer.Name$
Write #c,r,b+42,t;Customer.Addr$
Write #c,r,b+74+0,t;Customer.Balance.Debit
Write #c,r,b+74+6,t;Customer.Balance.Credit
```

The starting (or supplied) byte displacement is incremented by any **ITEM** declaration within the structure. Since the structure Customer contains the structure DRCR as Balance beginning at offset 74, the original definition of the structure DRCR has starting offsets of zero. If one were to transfer a DRCR structure separately, a starting offset of 74 would have to be supplied in the transfer statement itself.

Examples

```
Write Record #2, -2;CustRec.
```

See also

READ RECORD

WRLOCK

Synopsis

Write record and keep record locked.

Syntax

WRLOCK *chan.expr var.list*

Parameters

chan.expr is a driver-class dependent channel expression.

var.list is a list of comma separated variables of any dL4 data types.

Executable From Keyboard?

Yes.

Remarks

WRLOCK # transfers data from any dL4 data type into the file opened on *chan.expr*.

If the variable in the list is an *array.var*, optional *subscripts* may be specified. If given, these are evaluated, truncated to integer and used to select a single element. If no *subscripts* are supplied, only the first element is transferred.

If the variable in the list is a simple *num.var* or *date.var*, the transfer size is controlled by the **DIM**ensioned size and precision.

If the variable in the list is a string or binary variable, its size may be controlled by subscripts. All characters are transferred including zero-bytes.

WRLOCK transfers data and unconditionally locks the record. The data record remains locked until a non-locking operation is performed by that same program to the same channel. While a record is locked, other users will be unable to access the record.

WRLOCK is identical to **WRITE** omitting the trailing semicolon.

See the **WRITE** statement for additional details.

Examples

```
Wrlock #3,R1,100;A
```

```
Wrlock #C,R;A$
```

See also

RDLOCK, WRITE

Chapter 8 - Intrinsic CALLs and Functions

Introduction

This chapter presents the standard user defined **CALLs** and functions included with dL4. These procedures and functions must be **DECLARE**d before used in a BASIC program, i.e.:

```
Declare Intrinsic Sub TrxCo, Logic, InpBuf
Declare Intrinsic Function FmtOf
```

This chapter does not describe the **CALL**s, such as **DXOpen** and **DXGET**, that are specific to dynamicXport applications. Please see the dynamicXport manuals for information concerning those **CALL**s.

FUNCTION ADDMD5?

Synopsis

Calculate intermediate MD5 checksum for multiple string or binary values.

Syntax

ADDMD5? (*expr*, {, *bin.expr*})

Parameters

expr is a string or binary expression which specifies the value on which to calculate the MD5 checksum.

bin.expr is an optional expression which is the result of a previous ADDMD5? calculation.

Remarks

ADDMD5? calculates and returns as a 128 byte binary value an intermediate value of the MD5 checksum of *expr*. This intermediate value must be passed to a subsequent call to the MD5? function to generate a final MD5 checksum. The optional binary argument *bin.expr* can be used to pass the intermediate MD5 result value from a previous call to ADDMD5? to calculate a combined checksum of several variables. The checksum is calculated against the dimensioned size of strings so that null characters can be included in the checksum. Subscripts can be used to limit the number of characters included in the checksum. So that string values will produce the same checksum values on all platforms, each UNICODE character of a string is forced into a most-significant-byte-first ordering for checksum calculation. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
Dim CheckSum?[16], Temp?[128]
Temp? = AddMD5?(C$)
CheckSum? = MD5?(X$[1,Len(X$)],Temp?) !Calculate checksum of C$+X$
```

See also

CRC32, **MD5?**

CALL ASC2EBCDIC

Synopsis

Convert string between Unicode and EBCDIC character sets.

Syntax

CALL ASC2EBCDIC (*str.var { ,num.expr}*)

Parameters

str.var is a string variable containing the string to translate to or from EBCDIC.

*num.*expr is an optional expression select the translation mode.

Remarks

The string is translated from EBCDIC to Unicode if *num.expr* is zero or not specified. If *num.expr* is non-zero, then the string is translated from Unicode to EBCDIC. An error 38 is generated if *str.expr* contains any characters that cannot be translated. This procedure is compatible with UniBasic **CALL 53**.

Examples

```
Call Asc2EBCDIC(Rec$)
```

See also

CALL, CALL ATOE, CALL ETOA

CALL ATOE

Synopsis

Convert string from Unicode to the EBCDIC character set.

Syntax

CALL ATOE (*str.var*)

Parameters

str.var is the string to translate.

Remarks

An error 38 is generated if *str.var* contains any characters that cannot be translated. This procedure is compatible with UniBasic **CALL $ATOE**.

Examples

```
Call AtoE(Value$)
```

See also

CALL, CALL ETOA, CALL ASC2EBCDIC

CALL AVAILBLKS

Synopsis

Get amount of available file space

Syntax

CALL AVAILBLKS(*num.expr, num.var*)

Parameters

num.expr is an expression which specifies the logical unit to check.

num.var is a variable that receives the amount of available space, in 512 byte blocks, on the file system that contains the logical unit specifed by *num.expr*.

Remarks

This procedure is compatible with UniBasic **CALL 117**. The **SYSTEM 32** statement provides a more general method of checking for available file space.

Examples

```
Call AvailBlks(LU,NBLKS)
```

See also

CALL, SYSTEM 32

CALL AVPORT

Synopsis

Find available port number.

Syntax

CALL AVPORT (*num.var* {,*num.expr1* {,*num.expr2*}})

Parameters

num.var is a variable which is set to the first available port number in the specified port number range or -1 if no port is available.

*num.expr1*is an optional expression which specifies the beginning of the port number range.

num.expr2 is an optional expression which specifies the end of the port number range.

Remarks

If num.expr2 is not specified, the end of the port number range is assumed to be the maximum port number. If num.expr1 is not specified, the beginning of the port number range is assumed to be zero. If the end of the port number range is less than the beginning, then the port number search will be performed downwards from the end of the range.

Examples

```
Call AvPort(P)
Call AvPort(PortNum, 100)
Call AvPOrt(PortNum, 1000, 900)
```

See also

CALL, PORT, CALL TRXCO

FUNCTION BASE64$

Synopsis

Encode binary value as a printable base 64 value.

Syntax

BASE64$ (*bin.expr*)

Parameters

bin.expr is a binary string expression.

Remarks

BASE64$ encodes the binary string *bin.expr* as a printable base 64 character string. Base 64 is used for some forms of MIME encoding. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
C$ = Base64$(C?)
```

See also

BASE64?

FUNCTION BASE64?

Synopsis

Decode base 64 string into a binary string.

Syntax

BASE64? (*str.expr*)

Parameters

str.expr is a string expression which is a binary string encoded in base 64.

Remarks

BASE64? decodes the base 64 string *str.expr* into a binary string. Base 64 is used for some forms of MIME encoding. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
C? = Base64?(C$)
```

See also

Base64$

CALL BITMANIP

Synopsis

Manipulate Numeric BIT.

Syntax

CALL BITMANIP (*num.expr*, *num.var1*, *num.var2* {, *num.var3*})

Parameters

num.expr is a mode which, after evaluation, is truncated to an integer to specify one of the following operations: Reset, Set, Test, AND, OR, XOR, Complement.

num.var1 is used to select one binary argument to the **CALL**.

num.var2 is used to select a second binary argument to the **CALL**.

The optional *num.var3* is used to return information from the **CALL**.

Remarks

mode is any num.expr which, after evaluation, is truncated to an integer to specify one of the following operations:

mode	**Operation Selected**
0	Reset (zero) bit number *num.var1* in variable *num.var2*. *num.var3* returns bit *num.var1* before reset.
1	Set bit number *num.var1* in variable *num.var2* to one. *num.var3* returns bit *num.var1* before set.
2	Test bit number *num.var1* in variable *num.var2*. *num.var3* returns zero if the bit is zero or $2^{15\text{-}num.var1}$ if the bit is one.
3	AND variable *num.var1* to variable *num.var2* and store result in *num.var2* . A logical AND produces a one in each bit position set in both *num.var1* and *num.var2*.
4	OR variable *num.var1* to variable *num.var2* and store result in *num.var2*. A logical OR produces a one in each bit position set in either *num.var1* or *num.var2* or both.
5	XOR variable *num.var1* to variable *num.var2* and store result in *num.var2*. A logical XOR (exclusive OR) produces a one in each bit position set in either *num.var1* or *num.var2* but not in both.
6	Complement (NOT) variable *num.var1* and store result in variable *num.var2*. Each one bit is set to zero and vice-versa.

CALL BITMANIP provides bit manipulation on integer variables in the range 0 thru 65535 (177777_8). One-word arithmetic and logical operations are also provided.

The following table illustrates the effect of the logical operations:

X	Y	X AND Y	X OR Y	X XOR Y	NOT Y
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Examples

```
Call Bitmanip(M,A,B,F)
```

See also

CALL, CALL LOGIC

CALL BITSNUMSTR

Synopsis

Store/Load BITS representation of a number.

Syntax1

CALL BITSNUMSTR (*num.expr1*, *num.expr2*, *bin.var*)

Syntax2

CALL BITSNUMSTR (*num.expr1*, *num.expr2*, *str.var*)

Syntax3

CALL BITSNUMSTR (*num.expr1*, *bin.expr*, *num.var*)

Syntax4

CALL BITSNUMSTR (*num.expr1*, *str.expr*, *num.var*)

Parameters

num.expr1 is a numeric expression yielding an index into *bin.var* or *bin.expr* at which to copy *num.expr2* or *num.var*.

num.expr2 is a numeric expression yielding a value to copy into *bin.var*.

bin.var is a binary variable into which the value of *num.expr2* is copied.

str.var is a string variable into which the value of *num.expr2* is copied.

bin.expr is a binary expression yielding a binary string from which a value is copied to *num.var*.

str.expr is a string expression yielding a binary string from which a value is copied to *num.var*.

num.var is a numeric variable into which a value is copied from *bin.expr*.

Remarks

CALL BITSNUMSTR may be used to convert between BITS numeric data and binary data.

Syntax1 converts a number to its BITS binary representation and stores it at the index position in the binary string variable.

Syntax2converts a number to its BITS string representation and stores it at the index position in the string variable.

Syntax3 converts a BITS binary representation at the index position to a number and stores it in a variable.

Syntax4 converts a BITS string representation at the index position to a number and stores it in a variable.

The precision of the numeric variable determines the storage requirements.

Examples

```
Declare Intrinsic Sub BitsNumStr
Dim b?[20]
Dim %1,a1
a1 = 3
i = 1
Call bitsnumstr(i,a1,b?)
Print Hex$(b?)
Call bitsnumstr(i,b?,a1)
Print "The magic number was ";a1
End
```

See also

CALL

CALL BYTECOPY

Synopsis

Copy bytes from source to destination up to shorter of the two variables.

Syntax

CALL BYTECOPY (*destination.var.name*, *source.var.name*)

Parameters

destination.var.name is the destination variable name of any dL4 data type.

source.var.name is the source variable name of any dL4 data type.

Remarks

The **BYTECOPY** call may be used for low level manipulations, but should not be used by the BASIC programmer except in special situations, as it will frequently cause a program or its files to become non-portable.

Examples

```
! Demonstration of danger using BYTECOPY
Declare Intrinsic Sub ByteCopy
Dim %1,a1,%2,a2
a2 = 32767
Print 'CS'
For i=1 to 3
      Try
            a1 = a2          ! PRECISION PROBLEM
      Else
            Print a2;" too large for assignment to %1 variable "
            Print " Will use BYTECOPY to force assignment. "
            Call ByteCopy(a1,a2)   ! FORCE THE ASSIGNMENT
      End Try
      Print
      Print " Variable a2 is ";a2;" copied to variable a1 as ";a1
      Print
      a2 = a2 + 1
      If i = 2 then a2 = 50000
Next I
End
```

See also

CALL, Declare Intrinsic Function FmtOf

CALL CALLSTAT

Synopsis

Get **CALL** subprogram level information

Syntax

CALL CALLSTAT (*num.var1*, *str.var*, *num.var2*)

Parameters

num.var1 receives the current **CALL** subprogram level (zero if in the main program).

str.var receives the name of the parent (**CALL**ing) program.

num.var2 receives the line number of the **CALL** statement in the parent program.

Remarks

The arguments are optional and can be placed in various orders with the returned information determined by the variable type and the preceding arguments. An error 38 is generated if the arguments are ilegal.

Examples

```
Call CallStat(level,parentname$,parentline)
```

See also

CALL

FUNCTION CALLSTAT$

Synopsis

Return description of the current program position at a specified level

Syntax

CALLSTAT$ (*num.expr*, *str.var*)

Parameters

num.expr specifies the procedure level to describe.

str.var receives the level type such as “Swap” or “ExtFunc”.

Remarks

The current level is specified as zero, the parent procedure is specified as one, and so on. An error 38 is generated if a non-existent level is specified or it the arguments are ilegal.

Examples

```
Print CallStat$(1, Type$)
```

See also

CALL CALLSTAT

CALL CHECKDIGITS

Synopsis

Validate numeric field.

Syntax

CALL CHECKDIGITS (*str.expr*)

Parameters

str.expr is an expression which specifies the string to validate.

Remarks

An error 38 is generated if *str.expr* contains any non-numeric characters or if the parameter is not a string. A null string (“”) is accepted as valid. This procedure is compatible with UniBasic **CALL 22**.

Examples

```
Call CheckDigits(Cost$)
```

See also

CALL, CALL CHECKNUMBER

CALL CHECKNUMBER

Synopsis

Validate numeric field.

Syntax

CALL CHECKNUMBER (*str.expr*)

Parameters

str.expr is an expression which specifies the string to validate.

Remarks

An error 38 is generated if *str.expr* contains any characters other than digits (0 - 9), plus signs ("+"), minus signs ("-"), or more than one decimal point ("."). This procedure is compatible with UniBasic **CALL 23**.

Examples

```
Call CheckNumber(Cost$)
```

See also

CALL, CALL CHECKDIGIT

CALL CHSTAT

Synopsis

Get SWAP level information

Syntax

CALL CHSTAT (*num.var1*, *str.var*, *num.var2*)

Parameters

num.var1 receives the current **SWAP** subprogram level (zero if in the main program).

str.var receives the name of the parent (**SWAP**ing) program.

num.var2 receives the line number of the **SWAP** statement in the parent program.

Remarks

The arguments are optional and can be placed in various orders with the returned information determined by the variable type and the preceding arguments. An error 38 is generated if the arguments are ilegal.

Examples

```
Call ChStat(level,parentname$,parentline)
```

See also

CALL, SWAP

CALL CKSUM

Synopsis

Calculate file checksum.

Syntax

CALL CKSUM ({*num.expr1*, } *str.expr* , *num.expr2, num.expr3, var, num.var*)

Parameters

num.expr1 is an optional expression selecting the type of checksum.

str.expr is the file path.

num.expr2 is the 16-bit word starting offset of the file area to checksum.

num.expr3 is the 16-bit word ending offset of the file area to checksum. Use -1 to checksum the entire file.

var is a numeric or binary variable that receives the calculated checksum.

num.var is an optional variable that receives the operation status.

Remarks

The checksum algorithm is selected by *num.expr1* as follows:

omitted	UniBasic compatible 16-bit checksum (*var* must be numeric)
0	UniBasic compatible 16-bit checksum (*var* must be numeric)
1	32-bit CRC checksum (*var* must be numeric)
2	16 byte MD5 checksum (*var* must be binary)

If *num.var* is specified, then the following operation status is returned in the variable:

0	Successful
1	*str.expr* is not a string
3	*num.expr1* (start offset) is negative
5	*num.expr2* (end offset) is negative
6	*num.expr1* (start) is greater than *num.expr2* (end)
7	File not found

If *num.var* is not specified and the final status would have been non-zero, an error 38 will occur.

This procedure is compatible with UniBasic **CALL $CKSUM**.

Examples

```
Call Cksum(Filename$,Start,End,Checksum,Status)
```

See also

CALL, CRC32, MD5?

CALL CLEARSTR

Synopsis

Fill string variable with nulls

Syntax

CALL CLEARSTR (*str.var*)

Parameters

Str.var is the string to clear.

Remarks

This procedure is compatible with UniBasic **CALL 57**. String variables can also be initialized to nulls by the **CLEAR** statement.

Examples

```
Call ClearStr(X$)
```

See also

CALL, CLEAR

CALL CLOSEALL

Synopsis

Close all channels

Syntax

CALL CLOSEALL (*expr*)

Parameters

expr is an expression of any type. The expression value is not used by this **CALL**.

Remarks

This procedure is compatible with UniBasic **CALL 116**. All channels can also be closed by the following statement:

CLOSE

Examples

```
Call CloseAll(0)
```

See also

CALL, CLOSE

CALL CLU

Synopsis

Change current logical unit.

Syntax

CALL CLU (*num.expr { , num.var}*)

Parameters

num.expr is an expression which specifies the new logical unit number or -1 to return to the default working directory.

num.var is a numeric variable that receives the operation status. A status of 0 is successful, a status of 1 indicates an invalid logical unit number, and a status of 2 occurs if the logical unit was not found.

Remarks

An error 38 is generated if the type or number of parameters is incorrect This procedure is compatible with UniBasic **CALL $CLU**. The **CHDIR** statement provides a more general method of changing the current directory.

Examples

```
Call CLU(5)
```

See also

CALL, CHDIR

CALL CONVERTCASE

Synopsis

Convert selected characters to upper or lower case.

Syntax

CALL CONVERTCASE (*num.expr1*, *str.expr* {, *num.expr2*})

Parameters

num.expr1 is an expression which selects the function to be performed

str.var is a string variable to be converted.

num.expr2 is an optional expression which specifies the index (origin 0) in *str.var* at which to begin converting.

Remarks

The value of num.expr1 selects one of the following conversion modes:

Mode	Function
1	Convert all letters to upper case.
2	Convert first letter only to upper case.
3	Convert first letter of each word to upper case.
4	Convert all letters to lower case.
5	Convert first letter and any single “I” to upper case.
6	Convert all letters to lower case and any single “I” to upper case.

This procedure is compatible with UniBasic CALL 43.

Examples

```
Call ConvertCase(1,C$)
```

See also

CALL, LCASE$, UCASE$

CALL COPYSTR

Synopsis

Copy string to specified position

Syntax

CALL COPYSTR (*str.var, num.expr, str.expr*)

Parameters

str.var is the destination string.

num.expr is the index value in *str.var* at which the copy is performed. A value of one starts the copy at the first character in *str.var*.

str.expr is the string value to copy.

Remarks

If the source string is longer than the destination area, the copy will be truncated. If *num.expr* is negative or exceeds the size of *str.var*, nothing will be copied, but no error will occur. This procedure is compatible with UniBasic **CALL 30**.

Examples

```
Call CopyStr(Dest$,DestIdx,Src$)
```

See also

CALL

FUNCTION CRC16

Synopsis

Calculate 16 bit cyclic reduncancy code of string or binary value.

Syntax

CRC16 (*num.expr1*, *num.expr2*, *str.expr*, *num.expr3*)

Parameters

num.expr1 is an expression which selects the type of CRC calculation.

num.expr2 is an expression which specifies the CRC polynomial.

Str.expr is a string expression which specifies the value on which to calculate the 16 bit CRC.

num.expr3 is an expression which is the result of a previous CRC calculation.

Remarks

CRC16 calculates and returns as a number the 16-bit CRC checksum of *str.expr* which must be a string value. If *num.expr1* is zero, a simple 8 bit sum is calculated. If *num.expr1* is equal to one, a 16 bit CRC is calculated using *num.expr2* as the CRC polynomial. The numeric argument *num.expr3* can be used to pass the CRC value from a previous call to calculate a combined CRC of several variables. The CRC value is calculated against the **DIM**ed size of strings so that null characters can be included in the CRC value. Subscripts can be used to limit the number of characters included in the CRC. So that string values will produce the same CRC values on all platforms, each UNICODE character of a string is forced into a most-significant-byte-first ordering for CRC calculation. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
CheckSum = CRC16(1,4129,Blk$,0) !Calculate XMODEM CRC of Blk$
```

See also

ADDMD5?, CRC32, MD5?, NCRC32

FUNCTION CRC32

Synopsis

Calculate 32 bit cyclic reduncancy code of string or binary value.

Syntax

CRC32 (*expr* {, *num.expr*})

Parameters

expr is a string or binary expression which specifies the value on which to calculate the 32 bit CRC

num.expr is an optional expression which is the result of a previous CRC calculation.

Remarks

CRC32 calculates and returns as a number the 32-bit CRC checksum of *expr* which must be either a string or a binary value. The optional numeric argument *num.expr* can be used to pass the CRC value from a previous call to calculate a combined CRC of several variables. The CRC value is calculated against the **DIM**ed size of strings so that null characters can be included in the CRC value. Subscripts can be used to limit the number of characters included in the CRC. So that string values will produce the same CRC values on all platforms, each UNICODE character of a string is forced into a most-significant-byte-first ordering for CRC calculation. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
CheckSum = CRC32(C$) !Calculate CRC of C$ alone
CheckSum = CRC32(X$[1,Len(X$)],CheckSum) !Calculate CRC of C$+X$
```

See also

ADDMD5?, MD5?, NCRC32

CALL CUSTOMCHARACTERSET

Synopsis

Create custom character sets.

Syntax

CALL CUSTOMCHARACTERSET (*num.expr*,*str.expr*{,*num.var*})

Parameters

num.expr is a numeric variable or expression specifying the various call functions.

str.expr is a string variable or expression that contains the path of a dL4 profile text file.

num.var is a numeric variable specifying the status returned by the call.

Remarks

The intrinsic **CALL**, CustomCharacterSet, allows dL4 programs to create their own custom character sets. These characters sets can be used with the **OPEN** and **BUILD** "charset=name" option to read or write data in the custom character set. The character set must support a single byte character set: each character in the character set must consist of a single byte (multibyte codes like UTF-8 can not created).

The call requires a *num.expr* "mode" and a *str.expr* "filename" argument. In addition, the call may receive an optional *num.expr* "status" variable argument.

The *num.expr* "mode" argument represents the various call functions. The available modes or functions are:

Mode	Functions
0	Register or modify a user-defined character set.
1	Register a user-defined character set, but do not modify an existing character set. Return an error if the character set was previously registered.
2	Register a user-defined character set, but do not modify an existing character set. Do not return an error if the character set was previously registered.

Note that a character set can modified, but it can not be deleted. The character set will be available until dL4 is exited.

The call will return an error if it is called with an invalid number of arguments or with an invalid argument type.

The *str.expr* argument contains the path of a dL4 profile text file. This text file must contain three sections: a "CharacterSetName" section, a "ToUnicode" table section, and a "FromUnicode" table section.

```
[CharacterSetName]
Name=
Name=
Name=
  .
  .
  .
[ToUnicode]
  .
  .
  .
[FromUnicode]

  .
  .
  .
```

The "CharacterSetName" section consists of one or more names for the character set. Both the "ToUnicode" and "FromUnicode" sections consist of zero or more lines in the following format:

<StartingUnicodeValue>-<EndingUnicodeValue>=<Custom Character Set Value>

An example of the profile file, using an imaginary character set follows:

```
[Character Set Name]
Name=Imaginary Character Set
Name=Synonym Character Set

[ToUnicode]
0x0020-0x007e=0x20
0x00a0-0x00a0=0xff
0x00a1-0x00a1=0xad
0x00a2-0x00a2=0xbd

[FromUnicode]
0x0020-0x007e=0x20
0x00a0-0x00a0=0xff
0x00a1-0x00a1=0xad
0x00a2-0x00a2=0xbd
```

The optional status variable *num.var* represents the status returned by the call. If the status variable is not used, the call will return a BASIC error if it detects an error. If the status variable is specified, then it will be set to either zero, indicating success, or a positive value indicating a specific error status. The status values are:

Status Value	Meaning
0	No Error
1	Profile file does not exist or cannot be opened
2	Invalid CharacterSetName section

3	Invalid ToUnicode section
4	Invalid FromUnicode section
5	Character set already registered
6	Memory overflow
7	Character set is too complex (this shouldn't occur for any real character set)
8	Unexpected system error (such an I/O error reading the profile file)
9	Unknown error (catchall for any other unexpected error)

Examples

```
Call CustomCharacterSet(0,"chardir/custom")
Call CustomCharacterSet(Mode,CharFn$,Error)
```

See also

CALL

CALL DATE

Synopsis

Verify and reformat a date.

Syntax

CALL DATE (*str.expr, str.var, num.expr, num.var*)

Parameters

str.expr is an expression which specifies the string to validate and reformat

str.var is a string variable that receives the reformatted date.

num.expr is the length of formatted output.

num.var is a numeric variable that receives the operation status.

Remarks

The source date in *str.expr* must have the format MMYY, MMDDYY, or MMDDYYYY. The reformatted date in *str.var* will have the format YYMM, YYMMDD, or YYYYMMDD selected by the length *num.expr*. If **OPTION DATE FORMAT NATIVE** is used, the current locale will be used for date ordering. If the date is valid and reformatted successfully, a zero will be returned in *num.var*, otherwise an error status of one will be returned. This procedure is compatible with UniBasic **CALL $DATE**.

Examples

```
Call Date(srcdate$,destdate$,8,status)
```

See also

CALL, CALL VERIFYDATE

CALL DATETOJULIAN

Synopsis

Convert date string to julian date string.

Syntax

CALL DATETOJULIAN ({*num.expr*,} *str.expr* {,*str.var* {,*num.var*}})

Parameters

num.expr is an optional expression selecting the input and output date formats.

str.expr is an expression which specifies the string to convert.

str.var is an optional variable which receives the converted date string.

num.var is an optional variable that receives the status of the conversion (0 for success, 1 for illegal date).

Remarks

Conversion modes:

num.expr	Input Date	Output Date	Comment
0	yymmdd	yyddd	year and day of year; e.g. 98365
1	yymmdd	ddddd	days since January 1, 1968
2	yymmdd	yyyyddd	4 digit year and day of year; e.g. 1998365
4	yyyymmdd	yyddd	2 digit year and day of year; e.g. 98365
5	yyyymmdd	ddddd	days since January 1, 1968
6	yyyymmdd	yyyyddd	4 digit year and day of year; e.g. 1998365

If *num.expr* is not specified, a conversion mode of 0 is assumed.

If *str.var* is not specified, then *str.expr* must be a string variable into which the converted date is stored.

If *num.var* is not specified, then an illegal date will cause an error 38 to occur.

This procedure is compatible with UniBasic **CALL 25**.

Examples

```
Call DateToJulian(S$)
```

See also

CALL, CALL JULIANTODATE

FUNCTION DATEUSING$

Synopsis

Convert date to string using a mask.

Syntax

DATEUSING$ (*date.expr*, *str.expr*)

Parameters

date.expr is a date expression which specifies the date value to convert to a character string.

str.expr is a string expression that controls the formatting of the date value.

Remarks

The **DATEUSING** function parses the format mask *str.expr* replacing the date codes with the values, derived from *date.expr*, shown in the table below. Any characters in the format mask that are not part of a date code are left unchanged. The final string is returned as the function value.

Code	Replacement value
D	Numeric day of week (0 - 6, 0 is Sunday)
d	Numeric day of week (0 - 6, 0 is Sunday)
DAY	Day name in upper case (SUNDAY, MONDAY, ...)
day	Day name in mixed case (Sunday, Monday, ...)
Day	Day name in mixed case (Sunday, Monday, ...)
DY	Abbreviated day name in upper case (SUN, MON, ...)
dy	Abbreviated day name in mixed case (Sun, Mon, ...)
Dy	Abbreviated day name in mixed case (Sun, Mon, ...)
DD	Numeric day of month zero filled ("01" - "31")
Dd	Numeric day of month space filled (" 1" - "31")
dD	Numeric day of month space filled ("01" - "31")
dd	Numeric day of month ("1" - "31")
DDD	Numeric day of year zero filled ("001" - "366")
Ddd	Numeric day of year space filled (" 1" - "366")
ddd	Numeric day of year ("1" - "366")
HH	Numeric hour of day zero filled ("00" - "23")
Hh	Numeric hour of day space filled (" 0" - "23")
hH	Numeric hour of day space filled (" 0" - "23")
hh	Numeric hour of day ("0" - "23")
MM	Numeric month of year zero filled ("01" - "12")
Mm	Numeric month of year space filled (" 1" - "12")

mm	Numeric month of year ("1" - "12")
MONTH	Month name in upper case (JANUARY, FEBRUARY, ...)
Month	Month name in mixed case (January, February, ...)
month	Month name in mixed case (January, February, ...)
MON	Abbreviated month name in upper case (JAN, FEB, ...)
Mon	Abbreviated day name in mixed case (Jan, Feb, ...)
mon	Abbreviated day name in mixed case (Jan, Feb, ...)
NN	Numeric minute of hour zero filled ("00" - "59")
Nn	Numeric minute of hour space filled (" 0" - "59")
nN	Numeric minute of hour space filled (" 0" - "59")
nn	Numeric minute of hour ("0" - "59")
PM	"AM" for time before noon, "PM" for time afterward
pm	"am" for time before noon, "pm" for time afterward
P	"A" for time before noon, "P" for time afterward
p	"a" for time before noon, "p" for time afterward
Q	Numeric quarter of year ("1" - "4", 1 is Oct - Dec)
q	Numeric quarter of year ("1" - "4", 1 is Oct - Dec)
SS	Numeric second of minute zero filled ("00" - "59")
Ss	Numeric second of minute space filled (" 0" - "59")
sS	Numeric second of minute space filled (" 0" - "59")
ss	Numeric second of minute ("0" - "59")
TH	Ordinal number in upper case ("1ST", "2ND", ...)
th	Ordinal number in lower case ("1st", "2nd", ...)
WW	Numeric week of year zero filled ("01" - "53")
Ww	Numeric week of year space filled (" 1" - "53")
wW	Numeric week of year space filled (" 1" - "53")
ww	Numeric week of year ("1" - "53")
YYYY	Four digit year
YY	Two digit year

Examples

```
Print DateUsing$(Tim#(0),”MM/DD/YY HH:NN:SS”)
```

See also

CALL, CALL DATETOJULIAN

CALL DBASE

Synopsis

Access a dBase file.

Syntax0

CALL DBASE (*num.expr*, *str.expr1*, *num.var*)

Syntax1

CALL DBASE (*num.expr*, *str.expr2*, *str.expr3*, *num.var*)

Syntax2

CALL DBASE (*num.expr*, *str.expr2*, *str.var*, *num.var*)

Syntax3

CALL DBASE (*num.expr*)

Parameters

num.expr is an expression which specifies the mode (0 – 5).

str.expr1 is the path of a dBase file.

str.expr2 is a field name from the dBase file.

str.expr3 is a field value.

str.var is a string variable that receives a field value from the dBase file.

num.var is a numeric variable that receives the status of the operation (0 if successful, 1 if the operation failed).

Remarks

CALL DBASE is provided for compatibility with existing applications. New applications should access dBase files using the **OPEN**, **SEARCH**, **READ**, and **CLOSE** statements.

The modes specified by *num.expr* are as follows:

0	Open a dBase file using syntax 0
1	Search the currently open dBase file using syntax 1 to find a record in which the field specified by *str.expr2* has the value specified by *str.expr3*. The search starts at the beginning of the file.
2	Read a value from the current record using syntax 2. The value of the field specified by *str.expr2* is copied into *str.var*.
3	Close the currently open dBase file using syntax 3.
4	Search the currently open dBase file using syntax 1 to find a record in which the field specified by *str.expr2* has the value specified by *str.expr3*. The search starts at the current record.
5	Reposition the currently open dBase file to the first record using syntax 3.

Examples

```
Call Dbase(0,”test.dbf”,status)
```

See also

CALL, **OPEN**, **READ**, **SEARCH**, **CLOSE**

CALL DECTOOCT

Synopsis

Convert decimal to octal.

Syntax

CALL DECTOOCT (*num.expr*, *var*)

Parameters

num.expr is an expression which specifies the number to convert to octal format.

var is a numeric or string variable that receives the converted octal value.

Remarks

The value of *num.expr* must be between -2^{31} and $2^{31} - 1$ inclusive.

If *var* is a string variable, it should be dimensioned to at least 12 characters. The octal value will be right justified to twelve characters, space filled, and, if negative, prefixed with a minus sign.

If *var* is numeric, each octal digit of *num.expr* will become a decimal digit in *var*. For example, if *num.expr* is 25, then 31 will be stored in *var*.

This procedure is compatible with UniBasic **CALL 126**.

Examples

```
Call DecToOct(value,octalvalue)
Call DecToOct(value,octalstring$)
```

See also

BSTR$, CALL

CALL DEVCLOSE

Synopsis

Close DEVxxxx pseudo-channels.

Syntax

CALL DEVCLOSE ({*num.expr*})

Parameters

num.expr is an optional expression which specifies the pseudo-channel number to close.

Remarks

CALL DEVCLOSE closes the specified pseudo-channel or, if *num.expr* wasn't specified, all pseudo-channels. A pseudo-channel is a hidden channel number opened via **CALL DEVOPEN**. This procedure is compatible with UniBasic **CALL $DEVCLOSE**. New applications should use the **OPEN**, **READ**, **WRITE**, and **CLOSE** statements to access devices.

Examples

```
Call DevClose(5)
```

See also

CALL, CALL DEVOPEN

CALL DEVOPEN

Synopsis

Open a DEVxxxx pseudo-channel.

Syntax

CALL DEVOPEN (*num.expr*, *str.expr* { *expr* … })

Parameters

num.expr is an expression that selects the pseudo-channel number to open.

str.expr is an expression which specifies the device or driver to open.

expr is one of one or more optional driver arguments.

Remarks

This procedure is compatible with UniBasic **CALL $DEVOPEN**. New applications should use the **OPEN**, **READ**, **WRITE**, and **CLOSE** statements to access devices.

Examples

```
Call DevOpen(Cost$)
```

See also

CALL, CALL DEVCLOSE, CALL DEVREAD, CALL DEVWRITE, CALL DEVPRINT

CALL DEVPRINT

Synopsis

Print to a DEVxxxx pseudo-channel.

Syntax

CALL DEVPRINT (*num.expr1*, *num.expr2*, *num.expr3*, *num.expr4* { , *expr* …})

Parameters

num.expr1 is the pseudo-channel number to print to.

num.expr2 is the record number to print to.

num.expr3 is the item number or record offset to print to.

num.expr4 is a timeout value in tenths of a seconds or -1 for no timeout.

expr is one of one or more optional values to print as defined by the driver.

Remarks

This procedure is compatible with UniBasic **CALL $DEVPRINT**. New applications should use the **OPEN**, **READ**, **WRITE**, and **CLOSE** statements to access devices.

Examples

```
Call DevPrint(5, -1, -1, 100, “Hello.”)
```

See also

CALL, CALL DEVOPEN, CALL DEVCLOSE, CALL DEVREAD, CALL DEVWRITE

CALL DEVREAD

Synopsis

Read from a DEVxxxx pseudo-channel.

Syntax

CALL DEVREAD (*num.expr1*, *num.expr2*, *num.expr3*, *num.expr4* { , *var* …})

Parameters

num.expr1 is the pseudo-channel number to read from.

num.expr2 is the record number to read from.

num.expr3 is the item number or record offset to read from.

num.expr4 is a timeout value in tenths of a seconds or -1 for no timeout.

var is one of one or more variables to read into as defined by the driver.

Remarks

This procedure is compatible with UniBasic **CALL $DEVREAD**. New applications should use the **OPEN**, **READ**, **WRITE**, and **CLOSE** statements to access devices.

Examples

```
Call DevRead(7, -1, -1, 100, Rec$)
```

See also

CALL, CALL DEVOPEN, CALL DEVCLOSE, CALL DEVWRITE, CALL DEVPRINT

CALL DEVWRITE

Synopsis

Write to a DEVxxxx pseudo-channel.

Syntax

CALL DEVWRITE (*num.expr1*, *num.expr2*, *num.expr3*, *num.expr4* { , *expr* …})

Parameters

num.expr1 is the pseudo-channel number to write to.

num.expr2 is the record number to write to.

num.expr3 is the item number or record offset to write to.

num.expr4 is a timeout value in tenths of a seconds or -1 for no timeout.

expr is one of one or more optional values to write as defined by the driver.

Remarks

This procedure is compatible with UniBasic **CALL $DEVWRITE**. New applications should use the **OPEN**, **READ**, **WRITE**, and **CLOSE** statements to access devices.

Examples

```
Call DevWrite(5, -1, -1, 100, “Hello.”)
```

See also

CALL, CALL DEVOPEN, CALL DEVCLOSE, CALL DEVREAD, CALL DEVPRINT

CALL DRAWIMAGE

Synopsis

Draw image file on screen or printer.

Syntax0

CALL DRAWIMAGE (*num.expr1*,*str.expr*,*num.expr2*,*num.expr3*,*num.expr4*,*num.expr5*)

Parameters

num.expr1 is an optional numeric variable or expression specifying a user channel (0 - 99) open to a window or printer. An error will be generated if *num.expr1* specifies a channel that is closed.

str.expr1 is a string expression containg the path of a JPEG, BMP, or other image file to be drawn.

num.expr2 and *num.expr3* are numeric variables or expressions that specify the horizontal and vertical coordinates of the upper left corner of a rectangle in which the image will be drawn.

num.expr4 and *num.expr5* are numeric variables or expressions that specify the horizontal and vertical coordinates of the lower right corner of a rectangle in which the image will be drawn.

Remarks

DRAWIMAGE draws image files such as JPEG or BMP files on a window or a printer. The window or printer must support drawing images. Currently, drawing images is supported by dL4 for Windows, the dL4Term terminal emulator, and the dL4/dL4Term Windows Printer driver. The image will be drawn as large as possible within the specified rectangle while preserving the aspect ratio of the image.

Examples

```
Call DrawImage(“pictures/product.jpg”,10,5,20,30)
Call DrawImage(printerchannel,”signature.jpg”,0,55,80,58)
```

See also

CALL

CALL DUPCHANNEL

Synopsis

Duplicate existing open channels onto closed user channel numbers.

Syntax

CALL DUPCHANNEL (*num.expr1*,*num.expr2*)

Parameters

num.expr1 is a numeric variable or expression specifying a closed user channel (0 - 99), i.e. new channel, onto which an open channel will be duplicated. An error will be generated if *num.expr1* specifies a channel that is already open.

num.expr2 is a numeric variable or expression that selects the channel to duplicate. The value must be an open user channel (0 - 99, i.e. old channel), standard input channel (-1), standard output channel (-2), Dynamic Window standard input channel (-3), or Dynamic Window standard output channel (-4). The standard input and output channels are the original base channels and not the window channels used by Dynamic Windows. An error will be generated if *num.expr2* specifies a channel that is not open.

Remarks

Duplicate channels can be used to perform I/O in the same way as the original channels. The primary use of **DUPCHANNEL** is to duplicate the standard input and output channels that are used by **INPUT** and **PRINT** when a channel isn't specified. By duplicating the standard input or output channel onto a user channel number, a program can apply channel oriented statements such as **SET** to a standard channel. Because **DUPCHANNEL** duplicates the base standard input and output channels, it can also be used to avoid window tracking when Dynamic Windows are active. Closing the duplicate or original channel has no effect other than freeing the channel number unless all copies of the original channel are closed.

The following program uses **DUPCHANNEL** to change the title of a window.

```
External Function ChangeWinTitle(oldchannel,NewName$)
    Declare Intrinsic Sub DupChannel
    Call DupChannel(99, oldchannel)
    Set #99,-1073;NewName$
    Clear #99
End Function 0

Open #1,{"--------","TITL",70,23} As "Window"
Input A
B = ChangeWinTitle(1," Test Win Name ")
Input A
Stop
```

Examples

```
Call DupChannel(1,2)
```

```
Call DupChannel(newchannel,oldchannel)
```

See also

CALL

CALL ECHO

Synopsis

Enable, disable, or toggle echo.

Syntax

CALL ECHO (*num.expr*)

Parameters

num.expr specifies how the echo mode is to be changed.

Remarks

Echo mode on the standard input channel is disabled if *num.expr* is zero, enabled if *num.expr* is one, and toggled if *num.expr* is two. This procedure is compatible with UniBasic CALL ECHO.

Examples

```
Call Echo(0)
```

See also

Mnemonics, CALL

CALL EDITFIELD

Synopsis

Verify and format a string according to a format mask.

Syntax

CALL EDITFIELD(*str.expr1*, *str.expr2*, *str.var*)

Parameters

str.expr1 is a string expression which is verified and formatted according to the mask *str.expr2*.

str.expr2 is string expression containing a format mask.

str.var is a string variable that receives the formatted result.

Remarks

The mask *str.expr2* may consist of any combination of the following characters:

A	Fixed length alphabetic (A-Z). The current source character must be alphabetic.
N	Fixed length numeric (0-9). The current source character must be numeric.
X	Variable length alpha-numeric (any character). The current source character may be any character.
V	Variable length alphabetic. The current source character can be alphabetic. If not, comparison continues with the next mask character.
Z	Variable length numeric. The current source character can be numeric. If not, comparison continues with the next mask character.
/	Field separator. The current source character may be any one of "/", ".", or "-".
.	Decimal point. The current source character must be a ".", unless followed by "V" or "Z" in the mask.
-	Minus sign. The current source character must be "-", unless this is the first character of the mask. If so, comparison continues with the next mask character.

Any other character that appears in the mask must appear in the source string in the corresponding position.

CALL EDITFIELD verifies that a given string conforms to the specifications of another string, termed a mask. The edit is performed by comparing the string with the mask , character by character.

The following table illustrates some typical editing examples:

MASK	EFFECT
-ZZZ.ZZ	Allows a number between -999.99 and 999.99 with a maximum of 2 fractional digits.
ANA NAN	This mask is used for the Canadian Postal Code. The source string length must be 7 characters, with a space in the fourth position. Each letter and digit must be in its fixed place.
NZZZ.NZ	Allows a minimum of 1 digit before and after the decimal, and a maximum of 4 before and 2 after. The decimal point must exist. Note that "0.0" is allowed.
VVVNZZ	Source "A45" results in edit of "A045".

In a sequence of fixed and variable length numeric edit characters ("N" and "Z"), the fixed length character must appear before the variable length character. In numeric fields, an edit results in left zero-filling of the field.

An error will occur if:

- o Any parameter is not a string variable.
- o Source does not conform to mask.
- o Destination string dimension is too small.
- o Same string used for source and destination.

This procedure is compatible with UniBasic **CALL 29**.

Examples

```
Call EditField(TelNo$, “(NNN)NNN-NNNN”, Result$)
```

See also

CALL

CALL ENV

Synopsis

Change or retrieve the value of an environment variable.

Syntax

CALL ENV ({*num.expr*,}*str.expr1*,*str.expr2*)

Parameters

num.expr is a numeric expression specifying whether the environment variable should be changed (*num.expr* is two or not specified) or retrieved into *str.expr2* (*num.expr* is one and *str.expr2* is a string variable).

str.expr1 is a string variable or string expression specifying the name of the environment variable to be changed.

str.expr2 is a string variable or string expression specifying the new value to be given to the environment variable named by *str.expr1*.

Remarks

CALL ENV places the definition "*str.expr1* = *str.expr2*" into the environment of your process or returns the value of the environment variable *str.expr1* in the string variable *str.expr2*.

The effect of using **CALL ENV** to change the value of dL4 runtime parameters is undefined for the running process: the change may or may not effect the value used by the running process. Applications must not depend on the current treatment of environment variables by dL4 because that behavior may change in future releases. Applications should only change environment variables defined by the application itself.

When using mode 1 to retrieve environment variable values, the following special environment variable names will be recognized and will return predefined values:

"PID" – Unix or Windows process id

"GID" – Unix group id (Unix only)

"UID" – Unix user id (Unix only)

Examples

```
Call Env("PATH","@")
```

```
Call Env(E$,V$)
```

See also

CALL

FUNCTION ERRMSG$

Synopsis

Return specified message string.

Syntax

ERRMSG$(*num.expr*)

Parameters

num.expr is the message number of the message string to be returned.

Remarks

ERRMSG$ return message number *num.expr* from the message file initialized by **CALL INITERRMSG**. If **CALL INITERRMSG** was not used or if the specified message does not exist, an empty string (“”) will be returned..

Examples

```
Msg$ = ErrMsg$(n)
```

See also

ERM$, CALL INITERRMSG

CALL ETOA

Synopsis

Convert string from EBCDIC to the Unicode character set.

Syntax

CALL ETOA (*str.var*)

Parameters

str.var is the string to translate.

Remarks

An error 38 is generated if *str.var* contains any characters that cannot be translated. This procedure is compatible with UniBasic **CALL $ETOA**.

Examples

```
Call EToA(Value$)
```

See also

CALL, CALL ATOE, CALL ASC2EBCDIC

CALL FILEINFO

Synopsis

Get file information.

Syntax

CALL FILEINFO (*dir.expr, info.var, filename.var { , mode.expr { , index.var }}*)

Parameters

dir.expr is a string expression used when *mode.expr* is zero or omitted.

info.var is a numeric array.

filename.var is a string variable that specifies the file path if *mode.expr* is one and receives the filename and some file attributes in both modes.

mode.expr is an optional numeric expression that specifies the CALL mode.

index.var is a numeric array.

Remarks

If *mode.expr* is omitted or zero, then the string expression *dir.expr* must be at least 14 bytes long and contain a BITS directory.

Most of the file information is returned in *info.var* which is a one dimensional numeric array of at least 25 elements with precision 2% or larger. Information returned is accessed by the elements:

[0] Account group (0-255).

[1] Account user (0-255).

[2] Attribute word as a numeric value Mode 0 only.

[3] File type (0-9), represents “O$BACTSI”.

[4] First disk address.

[5] Record length in bytes. For A[3]=0, returns 512 for text files and 65534 for non-text file.

[6] File size in 512 byte blocks (represents both halves of an indexed file).

[7] Creation date in the form MMDDYY.

[8] Last access date in the form MMDDYY.

[9] Relative sector offset; Mode 0 only.

[10] Size of record map in sectors (INDX files Mode 0 only).

[11] Number of indices (Index files only).

[12] System time at last access in hours.

[13] Secondary attribute word as a numeric value; Mode 0 only.

[14] Logical unit number, as currently installed; Mode 0 only.

[15] DIRECTORY sector number; Mode 0 only.

[16] Word displacement into DIRECTORY sector; Mode 0 only.

[17] Unix style protection bits; Mode 1 only.

[18] Number of items per record; Mode 1 only.

[19] Revision of UniBasic at time file was created; Mode 1 only.

[20] First Real Data Record as built; Mode 1 only.

[21] Byte offset to Record 0; size of header; Mode 1 only.

[22] Returns the files creation time in hours-since-BASEDATE.

Record length in element A[5] is 512 bytes for a text file and 65534 for a non-dL4 file of type A[3]=0. The first block of the file is examined and is only considered text if all bytes are <0x80.

In mode 1, *filename.var* provides the path of the file to examine The variable *filename.var* should be DIMensioned to at least 31 characters. Returned in *filename.var* is a 14-character name, truncated if necessary. Supplemental attributes are returned in characters 15-29; <PRWdsEOxFQUgabKY>. Lower-case letters refer to BITS attributes which are only returned when Mode 0 is used on a BITS directory unpack.

The expression *mode.expr* is truncated to an integer and used to specify the operational mode for the CALL. If omitted or 0, then a BITS DIRECTORY entry in directory is unpacked. Mode 1 is used to locate and return information about the file contained in *filename.var*.

Additional information for Indexed-Contiguous or Formatted files is returned in *index.var*, a numeric array. The array should be DIMensioned as *index.var*[128,1].

If the file is an Indexed-Contiguous file, the following information is returned:

index.var[0,0] Record length in bytes for file.

index.var [0,1] Current actual active record count.

index.var [X,0] Key length for Directory X.

index.var [X,1] Active Keys in Directory X or zero, if not available.

If the file is a Formatted file, the following item information is returned:

index.var [X,0] Item Type

index.var [X,1] Item length in bytes.

This procedure is compatible with UniBasic **CALL 127**. The information returned by mode 1 can also be obtained using the CHF functions, the SEARCH statement, and the GET statement.

Examples

```
Call FileInfo(Dir$,Info[],Path$,1,IdxInfo[])
```

See also

CALL

FUNCTION FINDCHANNEL

Synopsis

Find available (closed) channel number.

Syntax

FINDCHANNEL({*num.expr1*, *num.expr2*})

Parameters

num.expr1 is an optional expression that specifies the beginning of the channel number range.

num.expr2 is an optional expression that specifies the end of the channel number range.

Remarks

FINDCHANNEL returns the channel number of the first closed channel in the specified channel number range. If the start of the range is less than the end of the range, then the channel numbers will be checked in descending order. The default channel number range is 99 to 0 (descending).

Examples

```
Chan = FindChannel()
Chan = FindChannel(80,99)
```

See also

OPEN, BUILD

CALL FINDF

Synopsis

Determine if file exists.

Syntax

CALL FINDF (*str.expr*,*num.var {, str.var}*)

Parameters

str.expr specifies the path of the file to check.

num.var receives the status of the file lookup (0 if the file is not found, 1 if the file is found)

str.var is an optional string variable that receives the absolute path of the file if it is found.

Remarks

This procedure is compatible with UniBasic **CALL FINDF**.

Examples

```
Call FindF(filename$,status)
```

See also

CALL

CALL FLUSHALLCHANNELS

Synopsis

Flush all buffered file data to permanent storage.

Syntax

CALL FLUSHALLCHANNELS ()

Parameters

None.

Remarks

FLUSHALLCHANNELS issues a DCC_SYNC command to each open channel to request the driver to flush all modified data to permanent storage. This **CALL** is operating system dependent and may not do anything on some operating systems.

Examples

```
Call FlushAllChannels()
```

See also

CALL

FUNCTION FMTOF

Synopsis

Return precison or dimension of variable.

Syntax

FMTOF(*var*)

Parameters

var is any non-structure variable.

Remarks

If *var* is a numeric or date variable, **FMTOF** returns the actual precision (“%n”) of the variable. If *var* is a string, binary, or array variable, then **FMTOF** returns the dimensioned size of the variable.

Examples

```
prec = FmtOf(X)
maxsize = FmtOf(T$)
```

See also

UBOUND, DIM

CALL FORCEPORTDUMP

Synopsis

Generate program dump on selected port number.

Syntax

CALL FORCEPORTDUMP (*num.expr1*, *num.expr2*, *num.var)*

Parameters

num.expr1 is the dump mode.

num.expr2 is the port number on which the dump is to be generated.

num.var is the status of the dump request.

Remarks

The **FORCEPORTDUMP** intrinsic **CALL** causes the port number selected by *num.expr2* to produce a dump listing file. The dump format is identical to that of the ProgramDump() intrinsic **CALL** and lists the current execution location of the target program, the **CALL** stack, current variable values, the status of open channels, and various other values. If *num.expr1* is zero, the selected port will exit dL4 after producing the dump file. If *num.expr1* is equal to one, the selected port will resume execution after producing the dump. Because producing the dump interrupts and possibly interferes with program execution, **FORCEPORTDUMP** should only be used for debugging purposes.

FORCEPORTDUMP sets *num.var* to zero if the dump request was successfully sent to the selected port. Sending the request does not guarantee that the dump will actually be produced. If an error occurs while sending the request, *num.var* will be set to one. On some operating systems, such as Unix, the caller of ForcePortDump() must either be the same user as that of the target port or be a privileged user (such as root on Unix)

Because the contents of the program dump could reveal passwords and other restricted data, dump output is controlled by the **DL4PORTDUMP** runtime parameter. If **DL4PORTDUMP** is not defined for the selected port, then ForcePortDump() will not generate a dump. On Unix, **DL4PORTDUMP** is an environment variable that must be set in each users environment (perhaps set by the .profile script). Under Windows, the **DL4PORTDUMP** value can be supplied either as an environment variable or as a string value in the registry:

HKEY_CURRENT_USER\Software\DynamicConcepts\dL4\Environment\dL4PortDump

HKEY_LOCAL_MACHINE\Software\DynamicConcepts\dL4\Environment\dL4PortDump

In any form, DL4PORTDUMP is the filename to which the dump will be written. DL4PORTDUMP must be an absolute path. For example, under Windows, DL4PORTDUMP might be defined as "D:\Dumps\DumpFile.txt". The following macro values can be used in a DL4PORTDUMP path string:

%PORT%	Port number of target port
%DATE%	Current date ("YYMMDD")
%TIME%	Current time ("HHMMSS")
%name%	Value of environment variable "name"

These macro values, if used in the DL4PORTDUMP path, will be replaced by their current values. For example, if DL4PORTDUMP was defined with the value "D:\Dumps\%PORT%.txt" and a dump was triggered on port 15, then the dump would be written to the file "D:\Dumps\15.txt".

Examples

```
Call ForcePortDump(0,PortNum,Status)
```

See also

CALL, PORT, CALL PROGRAMDUMP

CALL FORMATDATE

Synopsis

Format date string.

Syntax

CALL FORMATDATE (*str.expr* {,*str.var* {,*num.var* {,*num.expr*}}})

Parameters

str.expr supplies the input date and, if *str.var* is not specified, receives the formatted date.

str.var is an optional variable that receives the formatted date.

num.var is an optional variable that receives the status of the conversion (0 for success, 1 for illegal date).

num.expr is an expression that selects the input and output date formats.

Remarks

Conversion modes:

num.expr	**Input Date**	**Output Date**
0	yymmdd	mm/dd/yy
1	yyyymmdd	mm/dd/yy
4	yymmdd	mm/dd/yyyy
5	yyyymmdd	mm/dd/yyyy

If *num.expr* is not specified, a conversion mode of 0 is assumed.

If *str.var* is not specified, then *str.expr* must be a string variable into which the converted date is stored.

If *num.var* is not specified, then an illegal date will cause an error 38 to occur.

If **OPTION DATE FORMAT NATIVE** is used, the output date will use day-month-year ordering and the native date separator if specified by the current locale.

This procedure is compatible with UniBasic **CALL 28**.

Examples

```
Call FormatDate(S$)
```

See also

CALL, CALL VERIFYDATE

CALL GATHER

Synopsis

Pack data into a string.

Syntax

CALL GATHER (*str.var, expr …*)

Parameters

str.var is a string variable into which the values from *expr* will be placed.

expr is one of one or more variables or expressions whose values are placed in *str.var*.

Remarks

The values of the *expr* expressions are sequentially copied into *str.var.* The expression *expr* may be of numeric, string, or date type. Numeric values are always stored in BITS formats. This procedure is compatible with UniBasic **CALL 72**.

Examples

```
Call Gather(E$,A,B,C$,D)
```

See also

CALL, CALL SCATTER

CALL GETGLOBALS

Synopsis

Retrieve session global values.

Syntax

CALL GETGLOBALS({*str.expr,*}*num.expr* {,*var.list*})

Parameters

str.expr supplies the name of the global set. If *str.expr* is not specified, the default set (named “”) is used.

num.expr specifies the starting global item number.

var.list is a list of one or more variables of any type except for array or structure. The type of each variable in the list must match that of the global item copied into to the variable.

Remarks

GETGLOBALS copies global values from the selected global set starting with global item *num.expr* and continuing sequentially through the list of global values. An error 38 will occur if one or more of the values do not exist or do not match the variable type.

Examples

```
Call GetGlobals(3,S$,X,User$)
```

See also

CALL, CALL SETGLOBALS

CALL GETREGISTRY

Synopsis

Retrieve Windows registry values.

Syntax

CALL GETREGISTRY(*str.expr*, *var)*)

Parameters

str.expr is the name of the registry key and value to retrieve.

var is a numeric, string, or binary variable.

Remarks

GETREGISTRY copies a Windows registry value from the registry key and value name specified in *str.expr*. An error 38 will occur if the value does not exist or if it does not match the variable type. This **CALL** always returns an error 38 if used on a Unix system. The value of *str.expr* must begin with one of the following root key names:

HKEY_CLASSES_ROOT\ (or HKCR\)

HKEY_CURRENT_CONFIG\ (or HKCC\)

HKEY_CURRENT_USER\ (or HKCU\)

HKEY_LOCAL_MACHINE\ (or HKLM\)

HKEY_USERS\ (or HKUS\)

HKEY_PERFORMANCE_DATA\ (or HKPD\)

HKEY_DYN_DATA\ (or HKDD\)

Examples

```
Call GetRegistry("HKEY_CURRENT_USER\\Software\\MyCompany\\Value",S$)
```

See also

CALL, CALL SETREGISTRY

CALL IMSMEMCOPY

Synopsis

Copy bytes from source to destination variable.

Syntax

CALL IMSMEMCOPY (*destination.var*, *source.var, num.expr*)

Parameters

destination.var is the destination variable of any dL4 data type.

source.var is the source variable of any dL4 data type.

num.expr is the number of bytes to copy.

Remarks

The **IMSMEMCOPY CALL**can be used to copy data between any two variables, but it is best used to quickly copy portions of one array to another array. If used to copy data between arrays, the arrays must be identical in layout, data types, and data precisions. When copying between two string variables, *num.expr* will be treated as the number of Unicode characters to copy rather than the number of bytes. This **CALL** may overwrite memory if *num.expr* is incorrect.

Examples

```
Call IMSMemCopy(D$,S$,20)
```

See also

CALL

CALL IMSPACK

Synopsis

Pack or unpack radix 50 data.

Syntax0

CALL IMSPACK(0, *str.expr, str.var*)

Syntax1

CALL IMSPACK(1, *str.var, str.expr*)

Parameters

str.expr is the source string expression.

str.var is the destination string variable.

Remarks

The **IMSPACK CALL** packs character data from *str.expr* into *str.var* (syntax 0) or unpacks data from *str.expr* to *str.var*(syntax 1). The packed data is in a radix 50 format. The IMSPACK CALL is compatible with CALL $PACK in IMS BASIC.

Examples

```
Call IMSPack(0, S$, D$)
```

See also

CALL PKRDX5018, CALL PKRDX5048

CALL INITERRMSG

Synopsis

Initialize the error message file for **ERRMSG$**.

Syntax

CALL INITERRMSG (*num.expr*, *str.expr*)

Parameters

num.expr must be a numeric expression, but is otherwise ignored.

str.expr is an expression which specifies the path of the error message text file.

Remarks

The error message file must be a text file in which each line begins with an message number, followed by a colon, and ending with the message text. This procedure is compatible with UniBasic **CALL 40**.

Examples

```
Call InitErrMsg(0, Filename$)
```

See also

CALL, ERRMSG$

CALL INPBUF

Synopsis

Place data into type-ahead buffer.

Syntax

CALL INPBUF (*str.expr)*

Parameters

str.expr is copied (appended) to the contents of the current type-ahead buffer.

Remarks

INPBUF may be used to pass data from a child process back to the parent when using **SWAP** statements or **[Hot-Key]** swapping.

If the window driver receives a '**Begin**' mnemonic character, the cursor will be moved to the first character of the current input line ("**Home**" action) and then a special input mode will be entered for the next input character. If the next input character is an edit action (such as "**Forward**"), the user is allowed to edit the current input line. If the next character is a data character, the current input line is deleted and the data character becomes the first input character. If the next character is an "enter" action, the current input line is returned to the program. A dL4 program uses the "Begin" action by calling the **INPBUF** procedure with a string consisting of a default input value followed by the 'Begin' mnemonic character. The next input by the program will then treat the default input as described above.

Examples

```
Call Inpbuf(A$) !Copy data to type-ahead
```

```
Call Inpbuf(A$ + "\215\")
```

See also

CALL, WINDOW, SWAP

CALL

CALL IRISOS95

Synopsis

Satisfy references to IRIS CALL 95.

Syntax

CALL IRISOS95 (*expr …*)

Parameters

expr is one of zero or more expressions of any type.

Remarks

This procedure is compatible with UniBasic **CALL 95**. As in UniBasic, this procedure has no actual function and is provided simply to satisfy any references to **CALL 95**.

Examples

```
Call IRISOS95()
```

See also

CALL

FUNCTION ISSQLNULL

Synopsis

Determine if a value is an SQL driver NULL value.

Syntax

ISSQLNULL (*expr*)

Parameters

expr is an expression of any type.

Remarks

ISSQLNULL returns 1 if *expr* is an SQL driver NULL value and 0 if it is not a NULL value. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
If IsSQLNull(Rec.Value) Print “Value is NULL”
```

See also

SQLNULL, SQLNULL$, SQLNULL#

CALL JULIANTODATE

Synopsis

Convert julian date string to formatted date.

Syntax

CALL JULIANTODATE ({*num.expr*,} *str.expr* {,*str.var* {,*num.var*}})

Parameters

num.expr is an optional expression selecting the input and output date formats.

str.expr is an expression which specifies the string to convert.

str.var is an optional variable which receives the converted date string.

num.var is a optional variable that receives the status of the conversion (0 for success, 1 for illegal date).

Remarks

Conversion modes:

num.expr	**Input Date**	**Output Date**	**Comment**
0	yyddd	mm/dd/yy	year and day of year; e.g. 98365
1	ddddd	mm/dd/yy	days since January 1, 1968
2	yyyyddd	mm/dd/yy	4 digit year and day of year; e.g. 1998365
4	yyddd	mm/dd/yyyy	2 digit year and day of year; e.g. 98365
5	ddddd	mm/dd/yyyy	days since January 1, 1968
6	yyyyddd	mm/dd/yyyy	4 digit year and day of year; e.g. 1998365

If *num.expr* is not specified, a conversion mode of 0 is assumed.

If *str.var* is not specified, then *str.expr* must be a string variable into which the converted date is stored.

If **OPTION DATE FORMAT NATIVE** is used, the output date will use day-month-year ordering and the native date separator if specified by the current locale.

If *num.var* is not specified, then an illegal date will cause an error 38 to occur.

This procedure is compatible with UniBasic **CALL 27**.

Examples

```
Call JulianToDate(S$)
```

See also

CALL, CALL DATETOJULIAN

CALL LOCK

Synopsis

Change exclusive/shared open mode on an open file.

Syntax

CALL LOCK (*num.expr1, num.expr2, num.var*)

Parameters

num.expr1 is an expression which specifies the channel number of an open file.

num.expr2 is an expression which selects the new open mode: 0 for shared open, non-zero for exclusive open.

num.var is a variable which receives the operation status.

Remarks

The status value returned in *num.var* is defined as follows:

0	Operation successful
1	Illegal Channel Number
2	Channel not open
6	File is already Locked
7	File is not locked

This procedure is compatible with UniBasic **CALL $LOCK**.

Examples

```
Call Lock(5, 1, status)
```

See also

CALL, EOPEN

CALL LOGIC

Synopsis

Perform logical operations.

Syntax

CALL LOGIC (*num.expr*, *var1*, var2, *var3*)

Parameters

num.expr is any operator which, after evaluation, is truncated to an integer and used to specify the operation for **LOGIC**: 1 = AND; 2 = OR; 3 = XOR; 4 = NOT.

var1 and *var2* select two identical types of variables (numeric, string, or binary) to perform an operation upon.

var3, the result, must be the same type as the supplied *var1* and *var2*, and will hold the resulting data from the operation.

Remarks

If the supplied variables are numeric, they are truncated to unsigned integers (shorts) to perform the operation. String and binary variables are processed a byte at a time until the **DIM**ensioned length of the shortest argument passed is reached.

An AND operation results in a 1 bit when the corresponding bit of both variables is 1.

An OR operation results in a 1 bit when either of the corresponding bits is 1, or when both are 1.

An XOR (exclusive OR) results in a 1 bit when only one of the corresponding bits of both variables is 1.

A NOT operation only requires *variable1*. *variable2* must be specified for syntactical reasons (use the same variable), but is not used. NOT results in a 1 bit if the bit of *variable1* is zero, and results in 0 if the bit is 1.

Entire strings (including zero bytes) can be operated upon using **LOGIC**. To copy a string in its entirety, AND the string to itself. To fully zero fill (zero byte) a string, XOR it with itself.

X	Y	X AND Y	X OR Y	X XOR Y	NOT Y
0	0	0	0	0	1
0	1	0	1	1	0
1	0	0	1	1	
1	1	1	1	0	

Examples

```
Call Logic(1,A$,B$,C$)      ! AND 2 strings
Call Logic(1,A[0],32768,J) ! Is value negative
Call Logic(1,A$,A$,B$)      ! Copy string A$ to B$
```

See also

FUNCTION MD5?

Synopsis

Calculate MD5 checksum of string or binary value.

Syntax

MD5? (*expr*, {, *bin.expr*})

Parameters

expr is a string or binary expression which specifies the value on which to calculate the MD5 checksum.

bin.expr is an optional expression which is the result of a previous ADDMD5? calculation.

Remarks

MD5? calculates and returns as a 16 byte binary value the MD5 checksum of *expr* which must be either a string or a binary value. The optional binary argument *bin.expr* can be used to pass the intermediate MD5 result value from a call to ADDMD5? to calculate a combined checksum of several variables. The checksum is calculated against the dimensioned size of strings so that null characters can be included in the checksum. Subscripts can be used to limit the number of characters included in the checksum. So that string values will produce the same checksum values on all platforms, each UNICODE character of a string is forced into a most-significant-byte-first ordering for checksum calculation. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
Dim CheckSum?[16], Temp?[128]
CheckSum? = MD5?(C$) !Calculate checksum of C$ alone
Temp? = AddMD5?(C$)
CheckSum? = MD5?(X$[1,Len(X$)],Temp?) !Calculate checksum of C$+X$
```

See also

ADDMD5?, **CRC32**

CALL MEMCMP

Synopsis

Compare strings.

Syntax

CALL MEMCMP (*str.expr1, str.expr2, num.var*)

Parameters

str.expr1 is an expression which specifies a string to compare.

str.expr2 is an expression which specifies a string to compare.

num.var is a variable that receives the result of the string comparison.

Remarks

CALL MEMCMP performs a character by character comparison of *str.expr1* and *str.expr2* including all characters in the DIMensioned length of the strings. The result is returned in *num.var* as follows:

Relation	Result
str.expr1 < *str.expr2*	-1
str.expr1 = *str.expr2*	0
str.expr1 > *str.expr2*	1

This procedure is compatible with UniBasic **CALL $MEMCMP**.

Examples

```
Call MemCmp(A$,B$,Result)
```

See also

CALL

CALL MEMCOPY

Synopsis

Copy 16 bit words between variables

Syntax

CALL MEMCOPY (*expr, var, num.expr*)

Parameters

expr is an expression of any type.

var is a variable of any type.

num.expr is a numeric expression specifying the number of 16 bit words to copy.

Remarks

CALL MEMCOPY moves *num.expr* 16 bit (2 byte) words from the value *expr* to the variable *var*. Because the original IRIS **CALL** used 8-bit strings in which ASCII characters had their most significant bit inverted and dL4 uses 16-bit Unicode characters, the rules for compatible copying are complex. If both *expr* and *var* are not strings, the copy is performed as a memory image without any conversion. If *expr* is a string and *va*r is not a string, then only the lower 8 bits of each Unicode character from *expr* are copied, the most significant bit of each 8 bit value is inverted, and two Unicode characters from *expr* are processed for each 16 bit word. If *expr* is not a string and *var* is a string, then an 8 bit byte from *expr* is expanded to a Unicode character in *var*, the most significant bit in each byte is inverted, and two Unicode characters in *var* are modified for each 16 bit word copied. If both *expr* and *var* are strings, then *num.expr* times two characters are copied from *expr* to *var* without any conversion. This procedure is compatible with UniBasic **CALL 5**.

Examples

```
Call MemCopy(Cost$)
```

See also

CALL

CALL MISC47

Synopsis

Perform miscellaneous operations.

Syntax

CALL MISC47 (*num.expr, num.var*)

Parameters

num.expr is an expression which specifies the operation to perform.

*num.*var is a variable that receives the operation result, if any.

Remarks

CALL MISC47 performs the following operations as specified by *num.expr*:

Num.expr	Operation
0	Pop top of GOSUB stack and return the line number in *num.var*
3	Return current terminal type (SPC(13)) in *num.var*
4	Disable terminal echo
5	Enable terminal echo

An error 38 will occur if *num.expr* is an unsupported operation number. This procedure is compatible with UniBasic **CALL 47**.

Examples

```
Call Misc47(4,Status) ! disable echo
```

See also

CALL

CALL MISCSTR

Synopsis

Miscellaneous string functions.

Syntax

CALL MISCSTR({*num.expr*,} *str.var* {,{*num.expr*},*str.var* }...)

Parameters

num.expr is an optional expression selecting the function to be performed.

str.var is a string on which to perform the current function.

Remarks

Conversion modes:

num.expr	**Function**
0	Convert the string to lower case.
1	Replace all characters with nulls.
2	Zero bit 7 of each character (AND each character with $0FF7F_{16}$)
3	Toggle bit 7 of each non-null character (XOR each character with 00080_{16})

If *num.expr* is not specified, a conversion mode of 0 is assumed. If *num.expr* is specified, it sets the function to be performed on all following strings until the next *num.expr*.

This procedure is compatible with UniBasic **CALL 60**.

Examples

```
Call MiscStr(S$)
Call MiscStr(1,S$)
Call MiscStr(S$,3,D$)
```

See also

LCASE$, CALL, CALL LOGIC, CALL BITMANIP

CALL NCRC32

Synopsis

Calculate 32 bit cyclic redundancy code of a string or binary value.

Syntax

CALL NCRC32 (*num.var*, *expr*, {, *num.expr*})

Parameters

*Num.*var is a numeric variable that receives the calculate CRC value.

expr is a string or binary expression which specifies the value on which to calculate the 32 bit CRC

num.expr is an optional expression which is the result of a previous CRC calculation.

Remarks

NCRC32 calculates and returns as a number the 32-bit CRC checksum of *expr* which must be either a string or a binary value. The optional numeric argument *num.expr* can be used to pass the CRC value from a previous call to calculate a combined CRC of several variables. The CRC value is calculated against the **DIM**ed size of strings so that null characters can be included in the CRC value. Subscripts can be used to limit the number of characters included in the CRC. So that string values will produce the same CRC values on all platforms, each UNICODE character of a string is forced into a most-significant-byte-first ordering for CRC calculation. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
Call CRC32(CheckSum,C$) !Calculate CRC of C$ alone
Call CRC32(CheckSum,X$[1,Len(X$)],CheckSum) !Calculate CRC of C$+X$
```

See also

ADDMD5?, CRC32, MD5?

CALL NEXTAVPORT

Synopsis

Find available port number.

Syntax

CALL NEXTAVPORT (*num.var*)

Parameters

Num.var is a numeric variable that receives the lowest available port number.

Remarks

An error 38 will occur if there are no available port numbers. This procedure is compatible with UniBasic **CALL 118**.

Examples

```
Call NextAvPort(PortNum)
```

See also

CALL, CALL AVPORT

CALL PKDEC20

Synopsis

Pack numeric data.

Syntax

CALL PKDEC20 (*str.expr, str.var*)

Parameters

str.expr is an expression which specifies the string to pack.

str.var is a string variable that receives the packed data.

Remarks

CALL PKDEC20 packs each pair of characters in *str.expr*, which is a string of decimal digits, into a character in *str.var*. Each digit is stored as a 4 bit nibble with the value of the digit plus one (thus 0 is stored as the nibble 1). If the length of *str.expr* is odd, a zero nibble will fill the final character. An error 38 is generated if *str.expr* contains any characters other than digits (0 – 9). This procedure is compatible with UniBasic **CALL 20**.

Examples

```
Call PkDec20(Number$,PackedNumber$)
```

See also

CALL, CALL UNPKDEC21, CALL PKDEC45

CALL PKDEC45

Synopsis

Pack or unpack numeric data.

Syntax

CALL PKDEC45({*num.expr*, } *str.expr, str.var {, num.var}*)

Parameters

num.expr is an optional expression that specifies whether to pack (0 or omitted) or unpack (non-zero).

str.expr is the source expression string.

str.var is the destination string variable.

num.var is an optional numeric variable that receives the operation status (0 if successful, 1 if failed).

Remarks

If *num.expr* is omitted or zero**, CALL PKDEC45** sequentially packs each pair of characters from *str.expr* into a character in *str.var*. Each character is stored as a 4 bit nibble with the character translated as shown in the table below. If the length of *str.expr* is odd, a zero nibble will fill the final character. If *str.expr* contains an unsupported character, then an error status will be report in *num.var* or, if *num.var* was omitted, an error 38 will occur.

If *num.expr* is non-zero, **CALL PKDEC45** sequentially unpacks each character from *str.expr* into two characters in *str.var*. Each character in *str.expr* is treated as a pair of nibbles which are translated into characters as shown in the table below.

This procedure is compatible with UniBasic **CALL 45**.

Character	Nibble	Character	Nibble
Space	0001	3	1001
,	0010	4	1010
-	0011	5	1011
.	0100	6	1100
/	0101	7	1101
0	0110	8	1110
1	0111	9	1111
2	1000		

Examples

```
Call PkDec45(data$, packeddata$)
```

See also

CALL, CALL UNPKDEC46, CALL PKDEC20

CALL PKRDX5018

Synopsis

Pack characters into radix 50 .

Syntax

CALL PKRDX5018 (*str.expr, str.var*)

Parameters

str.expr is an expression which specifies the string to pack.

str.var is a string variable that receives the packed string.

Remarks

CALL PKRDX5018 packs character triplets from *str.expr* into radix 50 character pairs in *str.var*. Each character from *str.expr* is translated to radix 50 values as shown in the table below and then a character triplet value is calculated as (Char1 * 40 + Char2) * 40 + Char3. The upper 8 bits of this triplet value is then stored as a character in *str.var* followed by a character containing the lower 8 bits. The resulting string is approximately one third smaller than the original string. An error 38 is generated if *str.expr* contains untranslatable characters. This procedure is compatible with UniBasic **CALL 18**.

Character	Radix 50	Character	Radix 50	Character	Radix 50	Character	Radix 50
0	01	A	11	K	21	U	31
1	02	B	12	L	22	V	32
2	03	C	13	M	23	W	33
3	04	D	14	N	24	X	34
4	05	E	15	O	25	Y	35
5	06	F	16	P	26	Z	36
6	07	G	17	Q	27	,	37
7	08	H	18	R	28	-	38
8	09	I	19	S	29	.	39
9	10	J	20	T	30	Space	00

Examples

```
Call PkRdx5018(src$,packed$)
```

See also

CALL, CALL UNPKRDX5019, CALL PKRDX5048

CALL PKRDX5048

Synopsis

Pack characters into radix 50 .

Syntax

CALL PKRDX5048 (*str.expr, str.var*)

Parameters

str.expr is an expression which specifies the string to pack.

str.var is a string variable that receives the packed string.

Remarks

CALL PKRDX5048 packs character triplets from *str.expr* into radix 50 character pairs in *str.var*. Each character from *str.expr* is translated to radix 50 values as shown in the table below and then a character triplet value is calculated as (Char1 * 40 + Char2) * 40 + Char3. The upper 8 bits of this triplet value is then stored as a character in *str.var* followed by a character containing the lower 8 bits. The resulting string is approximately one third smaller than the original string. An error 38 is generated if *str.expr* contains untranslatable characters. This procedure is compatible with UniBasic **CALL 48**.

Character	Radix 50	Character	Radix 50	Character	Radix 50	Character	Radix 50
,	01	7	11	H	21	R	31
-	02	8	12	I	22	S	32
.	03	9	13	J	23	T	33
0	04	A	14	K	24	U	34
1	05	B	15	L	25	V	35
2	06	C	16	M	26	W	36
3	07	D	17	N	27	X	37
4	08	E	18	O	28	Y	38
5	09	F	19	P	29	Z	39
6	10	G	20	Q	30	Space	00

Examples

```
Call PkRdx5048(src$,packed$)
```

See also

CALL, CALL UNPKRDX5049, CALL PKRDX5018

CALL PKUNPKDEC

Synopsis

Pack or unpack numeric data.

Syntax

CALL PKUNPKDEC(*src.str, dest.str*)

Parameters

src.str is the source expression string.

dest.str is the destination string variable.

Remarks

If *src.str* is dimensioned larger than *dest.str***, CALL PKUNPKDEC** sequentially packs each pair of characters from *src.str* into a character in *dest.str*. Each character is stored as a 4 bit nibble with the character translated as shown in the table below. If the length of *src.str* is odd, a zero nibble will fill the final character. If *src.str* contains an unsupported character, then an error 38 will occur.

If *src.str* is dimensioned smaller than or equal to *dest.str*, **CALL PKUNPKDEC** sequentially unpacks each character from *src.str* into two characters in *dest.str*. Each character in *src.str* is treated as a pair of nibbles which are translated into characters as shown in the table below.

This procedure is compatible with UniBasic **CALL 15**.

Character	Nibble	Character	Nibble
+	0001	3	1001
,	0010	4	1010
-	0011	5	1011
.	0100	6	1100
Space	0101	7	1101
0	0110	8	1110
1	0111	9	1111
2	1000		

Examples

```
Call PkUnPkDec(data$, packeddata$)
Call PkUnPkDec(packeddata$, data$)
```

See also

CALL, CALL PKDEC20, CALL PKDEC45

CALL PROGRAMCACHE

Synopsis

Manipulate and/or read status of the current shared program cache.

Syntax0

CALL PROGRAMCACHE (**0**, *num.var1*, *num.var2*, *str.var1*, *num.var3*)

Syntax1

CALL PROGRAMCACHE (**1**, *num.var1*, *str.expr*)

Syntax2

CALL PROGRAMCACHE (**2**, *num.var1*)

Syntax3

CALL PROGRAMCACHE (3, *num.var1, str.var2*)

Parameters

num.var1 is a numeric variable to contain the return code.

num.var2 is a numeric variable that determines which cache entry (starting at 0) is read.

str.var1 is a string variable that will receive a program file path.

str.expr is a string expression that will supply a program file path.

num.var3 is a numeric variable set to the number of users of the program.

str.var2 is a string variable that will receive the cache error message.

Remarks

The intrinsic procedure **ProgramCache**() is used to read the current shared program cache status and to manipulate the cache. An error will be generated if improper arguments or argument values are passed to **ProgramCache**(). Any error that occurs while processing the operation will be reported by setting the error code argument to a non-zero dL4 error code.

The first parameter to the ProgramCache function specifies the mode of operation as:

mode	Operation
0	Read next entry in cache.
1	Load program into cache as a permanent entry.
2	Delete cache when the current process exits.
3	Get cache error status message, if any

The return code in *num.var1* will be set to 0 if the operation is successful or to a standard dL4 error code if not. For example, if the cache is not available, the statement Call ProgramCache(0,e,p,f$,c) will set the variable "e" to 42 (file not found).

num.var2 should be set to zero to read the first entry. Each mode 0 call will update the value of *num.var2* so that the next call will read the next cache entry. The precision of *num.var2* must be such that it can contain any value between 0 and 2^{32-1} without any loss of precision (a 3% variable is adequate). The caller should only pass *num.var2* values of zero or those returned by the previous mode 0 call to **ProgramCache**().

num.var3 is a usage count and if set to -1 indicates that the program has been added to the cache as a permanent entry.

Examples

Example 1: Adding a program to the cache as a permanent entry

```
Declare Intrinsic Sub ProgramCache
Dim 1%, ErrorCode
Call ProgramCache(1, ErrorCode, "MenuLibrary.lib")
```

Users in static cache mode can only use cached programs and libraries that have been added as permanent entries. These permanent entries must be created by a user in dynamic cache mode using mode 1 of **ProgramCache**(). Once made, permanent entries cannot be individually deleted because there is no way to determine whether or not a static mode user is currently executing the program or library. See the program cache description in the dL4 Installation and Configuration Guide for more information on dynamic and static cache modes.

Example 2: List entries in cache

```
Declare Intrinsic Sub ProgramCache
Dim 1%, ErrorCode, 3%, CachePos, File$[200], Usage
CachePos = 0
Do
      Call ProgramCache(0, ErrorCode, CachePos, File$, Usage)
      If ErrorCode Exit Do
      If Usage < 0
            Print "Permanent  ";
      Else
            Print Using "#########  ";Usage;
      End If
      Print File$
Loop
If ErrorCode = 73 Print "The program cache is not enabled"
```

Example 3: Deleting the program cache

```
Declare Intrinsic Sub ProgramCache
Dim 1%, ErrorCode
Call ProgramCache(2, ErrorCode)
```

This example will delete the program cache when the current user exits dL4. The program cache should be deleted if it is desired to increase the size of the cache or if the cache has become corrupted. The cache can be deleted only by the owner of the cache or by the root user. Since the cache cannot be deleted until the user exits, no error is returned if the caller lacks delete permission. All other users should exit dL4 before the cache is deleted.

Example 4: Printing the cache error message

```
Declare Intrinsic Sub ProgramCache
Dim 1%, ErrorCode, ErrorMsg$[200]
Call ProgramCache(3, ErrorCode, ErrorMsg$)
If ErrorMsg$ Print “Cache initialization error: “;ErrorMsg$
```

Configuration errors can prevent the program cache from being successfully initialized. If this happens, dL4 will run, but with reduced performance. This example determines whether such an error has occurred and prints a message describing the error.

See also

CALL

CALL PROGRAMDUMP

Synopsis

Print stack, variables, open channels and other miscellaneous information.

Syntax

CALL PROGRAMDUMP ({*str.expr1* {,*str.expr2*}})

Parameters

*str.expr1*is the path of the text file in which to write the dump information.

str.expr2 is a string containing dump options.

Remarks

The intrinsic procedure **PROGRAMDUMP** is called by an application to dump the current program status, variable values, and channel information to a text file. If *str.expr1* is specified, then it is used as the filename of text file and the optional *str.expr2* is treated as an option list. If *str.expr2* contains the option "append", the dump will be appended to the end of the dump text file. If *str.expr1* is not specified, the current value of the DL4PORTDUMP runtime parameter determines the filename (see **CALL FORCEPORTDUMP** for a description of the DL4PORTDUMP parameter). In the example below, any unexpected error will cause **PROGRAMDUMP** to be called and the dump information written to the text file "dumpfile" in the directory "dumpdir":

```
Declare Intrinsic Sub ProgramDump
If Err 0 Goto UnexpectedError
Dim InFile$[40], 3%, X
InFile$ = "TestFile"
Build #1,+InFile$+"!"
X = 17
X = 4 / 0 ! Divide-by-zero error which will trigger a dump
Close #1
Chain ""
UnexpectedError: Call ProgramDump("dumpdir/dumpfile!")
Print "Unexpected error";Spc(8);"at line";Spc(10)
Chain ""
```

Note that, in this example, the directory "dumpdir" must exist in the current working directory or the call to **PROGRAMDUMP** will fail.

Formatting options can be specified in either *str.expr2* or in the options ("(xxx)") portion of the filename. The "COLUMNS=n" option specifies the output width (default 78 columns). The "NULLS=TRUE" option is used to enable printing null characters in strings as "\0\". The "BYNAME=TRUE" option sorts variables only by name instead of by type and name.

The **PROGRAMDUMP** intrinsic **CALL** will print repeated array values on a single line using an array slice notation. For example, if the array V had 10 elements and all of the elements were zero except for V[4]=7 and V[8]=9, then **PROGRAMDUMP** would produce the following output:

* V[0;3],%13 = 0

V[4],%13 = 7

* V[5;7],%13 = 0

V[8],%13 = 9

V[9],%13 = 0

Note that all lines with repeated data are prefixed with an asterisk.

Examples

```
Call ProgramDump(d$)
```

```
Call ProgramDump("dumpdir/dumpfile")
Call ProgramDump("dumpdir/dumpfile","append")
```

See also

CALL, CALL FORCEPORTDUMP

CALL RDFHD

Synopsis

Read file directory.

Syntax

CALL RDFHD(*dir.expr, fileno.var, name.var, acnt.var, type.var, size.var, stat.var, cost.var, income.var, create.var, lastaccess.var, fileid.var*)

Parameters

dir.expr is a string or numeric expression which specifies the directory path or the logical unit number.

fileno.var is a numeric variable which selects which file entry to examine in the directory.

name.var is a string variable which receives the file name.

acnt.var is a numeric variable that receives the file owner user id (-1 if there is no numeric id).

type.var is a numeric variable that receives the file type code (see below).

size.var is a numeric variable that receives the file size in 512 byte blocks.

stat.var is a numeric variable that receives the file status code (see below).

cost.var is a numeric variable that receives the file access cost (always zero).

income.var is a numeric variable that receives the file income (always zero).

create.var is a numeric variable that receives the file creation date in hours since the SPC(20) base year.

lastaccess.var is a numeric variable that receives the file last access date in hours since the SPC(20) base year.

fileid.var is a numeric variable that receives an operating system dependent file identification number

Remarks

CALL RDFHD is used to read file directories and returns information about a selected file in the directory specified by *dir.expr*. The file is selected by *fileno.var* which is the entry number in the file directory. Each call to **RDFHD** increments *fileno.var* to the next entry or to -1 if there are no more entries. The value of *fileno.var* should be initialized to zero before the first call to **RDFHD** for a given directory. This procedure is compatible with UniBasic **CALL 97** and **CALL $RDFHD**.

Type.var	Meaning
0	Not a unrecognized file type
2	dL4 program file
24	Text file
25	Formatted file
26	Indexed-Contiguous file

Stat.var	Meaning
0	Other
2	Indexed-Contiguous file
4096	Formatted file

Examples

```
Call RdFhd(dir$,fileno,f$,acnt,type,fsz,stat,c,i,create,access,fid)
```

See also

CALL, CALL FILEINFO

CALL READREF

Synopsis

Change channel access mode.

Syntax

CALL READREF(*num.expr1*, *num.expr2*)

Parameters

num.expr1 selects the new access mode

num.expr2 is the number of the channel to modify.

Remarks

If *num.expr1* is zero, the channel access mode is changed to read/write with record locking enabled. If *num.expr1* is non-zero, the access mode is changed to read-only with record locking disabled.

If a channel was originally opened for read-only access, it may not be possible to change the access mode to read/write.

The effect of **READREF** on record locking is driver and operating system dependent. New programs should use the **ROPEN** statement and avoid dependence on disabling record locking.

This procedure is compatible with UniBasic CALL $READREF.

Examples

```
Call ReadRef(1,10)
```

See also

CALL, OPEN, ROPEN

CALL RMVSPACES

Synopsis

Copy string and remove spaces.

Syntax

CALL RMVSPACES(*str.expr*, *str.var*, *num.expr*)

Parameters

str.expr is the source string.

str.var is the destination string.

num.expr is the copy mode (0 or 1)

Remarks

If *num.expr* is not equal to one, *str.expr1* is copied to *str.var* with all leading and trailing spaces removed.

If *num.expr* is equal to one, then *str.expr* is copied to *str.var* with all spaces removed except those in quoted strings. If an exclamation mark (“!”) appears outside of a quoted string, then the exclamation mark and all characters after it will be removed and a linefeed character will be appended.

This procedure is compatible with UniBasic CALL $RSPCS.

Examples

```
Call RmvSpaces(A$,B$,0)
```

See also

CALL, **CALL RMVSPACESI**, **LTRIM$**, **RTRIM$**, **TRIM$**

CALL RMVSPACESI

Synopsis

Copy string and remove spaces.

Syntax

CALL RMVSPACESI(*str.expr*, *str.var*, *num.expr*)

Parameters

*str.expr*is the source string.

str.var is the destination string.

num.expr is the copy mode (0 or 1)

Remarks

If *num.expr* is zero, *str.expr1* is copied to *str.var* with all leading and trailing spaces removed.

If *num.expr* is equal to one, then *str.expr* is copied to *str.var* with all spaces removed except those in quoted strings. If an exclamation mark ("!") appears outside of a quoted string, then the exclamation mark and all characters after it will be removed. A linefeed character will be appended to the end of *str.var*.

If *num.expr* is not equal to zero or one, then an error 38 will occur.

Examples

```
Call RmvSpacesI(A$,B$,0)
```

See also

CALL, **CALL RMVSPACES**, **LTRIM$**, **RTRIM$**, **TRIM$**

CALL RENAME

Synopsis

Rename a file.

Syntax

CALL RENAME(*num.expr1*, *str.expr1*, *str.expr2*, *num.expr2*, *num.var*)

Parameters

num.expr1 specifies the logical unit number to prefix the old and new filenames.

str.expr1 is the old filename.

str.expr2 is the new filename.

num.expr2 is a channel number (ignored).

num.var is a variable which will be set to 0 if operation succeeds or to 1 if it fails.

Remarks

If *num.expr1* is negative, it is ignored.

This procedure is compatible with UniBasic CALL $RENAME.

Examples

```
Call Rename(1,"A","B",0,S) ! Rename 1/A to 1/B
```

See also

CALL, MODIFY

FUNCTION REPLACE

Synopsis

Change occurences of a target string to a replacement string.

Syntax

REPLACE$ (*str.expr1*, *str.expr2*, *str.exp3* {, *num.expr*})

Parameters

str.expr1 is the original string value to be modified.

str.expr2 is the string value to find and replace in *str.expr1*.

str.expr3 is the replacement string value.

num.expr is an optional number of occurences of *str.expr2* to be replaced.

Remarks

The **REPLACE$** function returns the modified value of *str.expr1* without changing the value in *str.expr1*. If *num.expr* is not specified, then all occurences of *str.expr2* in *str.expr1* will be replaced by *str.expr3*. If *num.expr* is zero, then *str.expr1* will be returned without any modifications.

Examples

```
A$ = Replace$(C$, “old”, “new”) ! replace all “old“ with “new”
```

See also

REPLACECI$, **POS**

FUNCTION REPLACECI

Synopsis

Change occurences of a target string to a replacement string ignoring case.

Syntax

REPLACECI$ (*str.expr1*, *str.expr2*, *str.exp3* {, *num.expr*})

Parameters

str.expr1 is the original string value to be modified.

str.expr2 is the string value to find and replace in *str.expr1*.

str.expr3 is the replacement string value.

num.expr is an optional number of occurences of *str.expr2* to be replaced.

Remarks

The **REPLACECI$** function returns the modified value of *str.expr1* without changing the value in *str.expr1*. When searching *str.expr1*, the case of characters in *str.expr1* and *str.expr2* is ignored. If *num.expr* is not specified, then all occurences of *str.expr2* in *str.expr1* will be replaced by *str.expr3*. If *num.expr* is zero, then *str.expr1* will be returned without any modifications.

Examples

```
! Change all occurences of “No”, “no”, “NO”, or “nO” with “yes”
A$ = ReplaceCI$(C$, “No”, “yes”)
```

See also

REPLACE$, **POS**

CALL SCATTER

Synopsis

Unpack data from a string.

Syntax

CALL SCATTER (*str.expr, var …*)

Parameters

str.expr is a string expression containing values from a previous **CALL GATHER**.

var is one of one or more variables that will receive values from *str.expr*.

Remarks

The value of *str.expr* must be the result of a previous **CALL GATHER** or in a compatible format. The packed values from *str.expr* are sequentially unpacked and copied to the variables *var*. The variables *var* must be of the numeric, string, or date type and match the data type packed in *str.expr*. Numeric values are always stored in BITS formats. This procedure is compatible with UniBasic **CALL 73**.

Examples

```
Call SCATTER(E$,A,B,C$,D)
```

See also

CALL, CALL GATHER

CALL SETECHO

Synopsis

Enable or disable terminal echo.

Syntax

CALL SETECHO (*expr*)

Parameters

expr is a string or numeric expression.

Remarks

CALL SETECHO disables echo if *expr* is a string and enables echo if *expr* is numeric. This procedure is compatible with UniBasic **CALL 7**.

Examples

```
Call SetEcho(C$)
```

See also

CALL, CALL ECHO

CALL SETGLOBALS

Synopsis

Set session global values.

Syntax0

CALL SETGLOBALS({*str.expr*,}*num.expr*,*var.list*)

Syntax1

CALL SETGLOBALS({*str.expr*,}*num.expr*)

Syntax2

CALL SETGLOBALS(*str.expr*)

Parameters

str.expr supplies the name of the global set. If *str.expr* is not specified, the default set (named “”) is used.

num.expr specifies the starting global item number.

var.list is a list of one or more variables of any type except for array or structure.

Remarks

When using syntax 0**, SETGLOBALS** copies values to session global variables in the selected global set starting with global item *num.expr* and continuing sequentially through the list of values. The values can be retrieved by using **CALL GETGLOBALS**. Unless they are explicitly deleted (see below), the values persist throughout a dL4 session until dL4 exits. The values types do not need to match any existing type for the specified global item number. Global item numbers do not need to be sequential; setting item *num.expr* does not require setting values for item *num.expr* – 1 or for item *num.expr* + 1. Global item numbers must be in the range 0 through 999. Global set names cannot be longer than 32 characters. An error 38 will occur if there is insufficient memory available to store the value.

To delete a value, use syntax 1. To delete an entire global set, use syntax 2. Deleting a non-existent value or global set is not an error.

Examples

```
Call SetGlobals(3,S$,X)
```

See also

CALL, CALL GETGLOBALS

CALL SETREGISTRY

Synopsis

Set Windows registry values.

Syntax

CALL SETREGISTRY(*str.expr*, *expr)*)

Parameters

str.expr is the name of the registry key and value to set.

expr is a numeric, string, or binary expression.

Remarks

SETREGISTRY set the Windows registry value selected by the registry key and value name specified in *str.expr*. If the registry value already exists, an error 38 will occur if the value does not match the variable type. This **CALL** always returns an error 38 if used on a Unix system. The value of *str.expr* must begin with one of the following root key names:

HKEY_CLASSES_ROOT\ (or HKCR\)

HKEY_CURRENT_CONFIG\ (or HKCC\)

HKEY_CURRENT_USER\ (or HKCU\)

HKEY_LOCAL_MACHINE\ (or HKLM\)

HKEY_USERS\ (or HKUS\)

HKEY_PERFORMANCE_DATA\ (or HKPD\)

HKEY_DYN_DATA\ (or HKDD\)

Examples

```
Call SetRegistry("HKEY_CURRENT_USER\\Software\\MyCompany\\Value",S$)
```

See also

CALL, CALL GETREGISTRY

CALL SORTINSTRING

Synopsis

Sort Keys in a String or elements in an array.

Syntax0

CALL SORTINSTRING (*num.var*, *num.expr1*, *num.expr2*, *str.var1*, *str.var2*)

Syntax1

CALL SORTINSTRING (*num.var*, *num.expr1*, *num.expr2*, *str.array.var*, *str.var2*)

Syntax2

CALL SORTINSTRING (*num.var*, *num.expr1*, *num.expr2*, *struct.array.var*, *struct.var*)

Parameters

num.var is a numeric variable to receive a return status from the sort operation.

num.expr1 is a numeric variable or expression which, after evaluation , is truncated to an integer to specify the number of strings to be sorted.

num.expr2 is a numeric variable or expression which, after evaluation , is truncated to an integer to specify the length of each string. For string or structure arrays, this is the number of significant characters in each string array element or the first structure member.

str.var1 is a string variable containing the keys to be sorted. It may contain any number of fixed-length binary fields to be sorted. Sorting is based upon the supplied length (*num.expr2*) of each item, up to number (*num.expr1*) of items.

str.var2 is any temporary work string **DIM**ensioned to a minimum of length +8.

str.array.var is a string array variable containing the keys to be sorted. If *num.expr2* is less than the dimensioned size of the array elements, then only the first *num.expr2* characters will be significant when sorting.

struct.array.var is an array of structures. The first member of the structure must be a string and sorting will be performed using the first *num.expr2* characters of that structure member.

struct.var is a structure variable identical to the members of *struct.array.var*.

Remarks

The meaning of the return status value from the sort operation:

status	Description
0	Successful sort operation.
1	Parameter Error.
2	*number* or *length* was passed as zero.
3	*sort* string is too small; less than *number* * *length*
4	*work* string is too small; less than *length + 8*.

The resulting sorted string is returned in str.var1.

Examples

```
Call SortInString(E, 100, 10, A$, W$)
```

See also

CALL

FUNCTION SQLNULL

Synopsis

Return numeric SQL NULL value for SQL driver I/O.

Syntax

SQLNULL ()

Parameters

None.

Remarks

SQLNULL returns a numeric value that is recognized by SQL drivers as an SQL NULL. The function currently returns the value –1E62, but, for future compatibility, this function should always be used instead of the literal value. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
Rec.Value = SQLNull()
```

See also

ISSQLNULL, SQLNULL#, SQLNULL$

FUNCTION SQLNULL#

Synopsis

Return date SQL NULL value for SQL driver I/O.

Syntax

SQLNULL# ()

Parameters

None.

Remarks

SQLNULL# returns a date value that is recognized by SQL drivers as an SQL NULL. The function currently returns the value “January 1, 0001”, but, for future compatibility, this function should always be used instead of the literal value. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
Rec.Value# = SQLNull#()
```

See also

ISSQLNULL, SQLNULL, SQLNULL$

FUNCTION SQLNULL$

Synopsis

Return string SQL NULL value for SQL driver I/O.

Syntax

SQLNULL$ ()

Parameters

None.

Remarks

SQLNULL returns a string value that is recognized by SQL drivers as an SQL NULL. The function currently returns the value "\xffff\", but, for future compatibility, this function should always be used instead of the literal value. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

Examples

```
Rec.Name$ = SQLNull$()
```

See also

ISSQLNULL, SQLNULL, SQLNULL#

CALL STRING

Synopsis

Perform miscellaneous string functions.

Syntax0

CALL STRING (*num.expr1*, *str.var*)

Syntax1

CALL STRING(*num.expr1*, *str.expr*, *num.var*)

Syntax2

CALL STRING(*num.expr1*, *num.expr2, str.var*)

Parameters

num.expr1 specifies the function to be performed.

str.var is a variable on which to perform a function or into which to return the result.

num.expr2 is a value to be converted into characters.

num.var is a variable into which a converted character value is stored.

Remarks

num.expr1	Syntax	Function
1	0	Convert characters in *str.var* to upper-case.
2	0	Convert all characters in *str.var* to lower-case.
3	1	Store value of the first character of *str.expr* into *num.var*.
4	2	Store value of *num.expr2* as a character into the first character of *str.var*.
5	0	Copy the command line into *str.var*.
6	1	Store value of the first two characters of *str.expr* into *num.var*. The value is formed by multiplying the value of the first character by 256 and adding the value of the second character.
7	2	Store value of *num.expr2* divided by 256 into the first character of *str.var* and store the value of *num.var* modulo 256 into the second character of *str.var*.

This procedure is compatible with UniBasic **CALL $STRING**.

Examples

```
Call String(1,A$)
```

See also

ASC, **INT**, **LCASE$**, **UCASE$**, **CALL**, **CALL UBSTRING**, **CONV**

CALL STRINGSEARCH

Synopsis

Perform string search.

Syntax

CALL STRING ({*num.expr1*,} *str.expr1* {,*num.expr2*} ,*str.expr2, num.var* {,*num.expr3* {,*num.expr4* {,*num.expr5*}})

Parameters

num.expr1 controls whether the search end at the first null in *str.expr1*. If *num.expr*, truncated to an integer is non-zero, then the search is performed on all characters in the dimensioned length of *str.expr1*. Default value 0..

str.expr1 is the string which is searched for *str.expr2*.

num.expr2 is a starting index in *str.expr1* at which to begin the search. If *num.expr2* is negative, the search is performed backwards from the end of *str.expr1*. Default value: 1.

str.expr2 is the string to search for in *str.expr1*.

num.var is a variable into which the relative index of the matching substring is stored. *num.var* is set to -1 if no match is found.

num.expr3 is the number of the match to search for. If *num.expr3* is positive, *str.expr1* is searched for the Nth occurrence of *str.expr2*. If *num.expr3* is negative, *str.expr1* is searched for the Nth non-occurrence of *str.expr2*. Default value: 1.

num.expr4 is source step value. If specified, *str.expr1* is tested only at positions that are multiples of *num.expr4*.

num.expr5 is the target step value. If specified, *str.expr2* is treated as multiple strings of *num.expr5* characters each and each step in *str.expr1* is searched for each substring.

Remarks

This procedure is compatible with UniBasic **CALL 56**.

Examples

```
Call StringSearch(S$,”dog”,P)
```

See also

POS, **CALL**

CALL STRSRCH1

Synopsis

Search string.

Syntax

CALL STRSRCH1 (*num.expr*, *str.expr1*, *str.expr2*, *num.var*)

Parameters

num.expr is a numeric expression controlling the search mode. Only mode 2 is supported.

str.expr1 is the string to search for.

str.expr2 is the string to search.

num.var is a numeric variable which contains the start position for the search and receives the matching position.

Remarks

If a substring that matches *str.expr1* is found in *str.expr2*, then *num.var* is set to the starting index of that substring. If a match is not found, *num.var* is set to zero. The search starts at index *num.var* minus one in *str.expr2* (zero based indexing) This procedure is compatible with UniBasic **CALL 1**.

Examples

```
Call StrSrch1(2,T$,S$,P)
```

See also

CALL, POS

CALL STRSRCH44

Synopsis

Search string.

Syntax

CALL STRSRCH44(*num.expr1 {, str.expr1, str.expr2, num.var {, num.expr2}}*)

Parameters

num.expr1 is the **CALL** mode (see below).

str.expr1 is the optional string to search for or to swap.

str.expr2 is the optional string to search.

num.var is an optional numeric variable that supplies the search start position and receives the result.

num.expr2 is an optional expression that controls the search step value.

Remarks

Num.expr1	Operation Performed
0	Compare *str.expr1* to *str.expr2*.
1	Search *str.expr2* for the first occurrence of *str.expr1*.
2	Search *str.expr2* for the first non-occurrence of *str.expr1*.
3	Swap target. Reverses position of all characters in *str.expr1*.
4	Disable terminal echo.
5	Enable terminal echo.

If *num.expr1* is zero, the comparison status is returned in *num.var* as follows:

-2 = *str.expr2* is logically less than *str.expr1*

-1 = *str.expr2* is shorter than *str.expr1*

0 = *str.expr1* and *str.expr2* are exactly equal

1 = *str.expr1* is shorter than *str.expr2*

2 = *str.expr1* is logically less than *str.expr2*

If *num.expr1* is 1 or 2, then *num.var* supplies the starting position for the search and receives the matching position. If there is no matching position, then *num.var* is set to zero. If *num.expr2* is supplied, it is used as a step value in *str.expr2* between each search.

If *num.expr1* is 3, then *str.expr1* must be a string variable.

This procedure is compatible with UniBasic **CALL 44**.

Examples

```
Call StrSrch44(4) ! Disable echo
Call StrSrch44(1,T$,S$,P)
```

See also

CALL, POS

CALL STRSRCH81

Synopsis

String Search.

Syntax

CALL STRSRCH81(*num.expr, str.expr1, str.expr2, num.var*)

Parameters

num.expr is an expression that controls the search type.

str.expr1 is the string to search for..

str.expr2 is the string to be searched.

num.var supplies the starting search position and receives the search result.

Remarks

If *num.expr* is zero, a search is performed to match the first character of *str.expr1*. If *num.expr* is one, a search is performed to match the entire *str.expr1* string. The start position for a search if supplied by *num.var* using zero based indexing. If a match is found, the match position is returned in *num.var*. If a match is not found, *num.var* is set to zero. This procedure is compatible with UniBasic **CALL 81**.

Examples

```
Call StrSrch81(1,T$,S$,P)
```

See also

CALL, POS

CALL SWAPF

Synopsis

Control hot-key swapping.

Syntax

CALL SWAPF (*num.expr* {, *str.expr*})

Parameters

num.expr is the mode, which selects the function performed whenever the **[Hot-Key]** is pressed during **INPUT**.

The optional *str.expr* is the program file path defining a program to **SWAP** to whenever the **[Hot-Key]** is pressed, and the mode is non-zero. This can be any BASIC program pathname.

Remarks

num.expr is any mode which, after evaluation, is truncated to an integer to select the function performed whenever the **[Hot-Key]** is pressed during **INPUT**. Depending on the operating system, pressing a **[Hot-Key]** may have no effect until an **INPUT** statement is reached.

mode	Description
0	Disable the **[Hot-Key]** operation.
1	**SWAP** on **[Hot-Key]** with channels **OPEN** with normal common variables as contained in **COM** statements.
2	**SWAP** on **[Hot-Key]** with normal common variables as contained in **COM** statements.
3	**SWAP** on **[Hot-Key]** with channels **OPEN** and no common variables.

An error is generated if a **[Hot-Key]** is pressed and the specified *program name* does not exist.

Examples

```
Call Swapf(0)            ! Disable Hot-key for this program
Call Swapf(2,"AR.CUST") ! To Cust maint, no files
```

See also

CALL, **WINDOW**

CALL SYSRC

Synopsis

Return status of the last **SYSTEM** statement command.

Syntax

CALL SYSRC (*num.var*)

Parameters

num.var is a variable that receives the operating system dependent status of the last SYSTEM statement command.

Remarks

The command status value can also be obtained directly in the **SYSTEM** statement by using the optional status variable (‘SYSTEM “command”,status’).

Examples

```
Call SysRC(status)
```

See also

CALL, **SYSTEM**

CALL TIME

Synopsis

Get date and time.

Syntax

CALL TIME(str.*var*)

Parameters

str.var is a variable into which the current date and time is returned.

Remarks

An error will occur if *str.var* is dimensioned to less than 22 characters.

The format of the returned string is “Mon dd, year HH:MM:SS”.

This procedure is compatible with UniBasic **CALL $TIME** and **CALL 99**.

Examples

```
Call Time(T$)
```

See also

CALL, **TIM#**

CALL TRANSLATE

Synopsis

Translate characters to or from a byte string

Syntax0

CALL TRANSLATE(*num.var1*, *str.var, num.var2*, *bin.expr*, *str.expr1*)

Syntax1

CALL TRANSLATE(*num.var1*, *bin.var, num.var2*, *str.expr2*, *str.expr1*)

Parameters

num.var1 is a variable which receives the number of characters or bytes stored.

str.var is a variable that receives characters translated from *bin.expr*.

num.var2 is a variable which receives the number of source bytes or characters translated.

bin.expr is a binary expression that supplies bytes to be translated.

str.expr1 is a string expression that specifies the character set name (such as EBCDIC or UTF-8).

bin.var is a variable that receives bytes translated from *str.expr2*.

str.expr2 is a string expression that supplies characters to be translated.

Remarks

CALL TRANSLATE is used to convert between a string of bytes and a string of Unicode characters. The number of bytes or characters to be translated is controlled by the size or double subscripting of the source expression (*bin.expr* or *str.expr2*). Null characters in the source expression will be translated as data. Translation will stop at the end of the source expression or at the first byte or character that cannot be translated.

Examples

```
Call Translate(DestCnt,Dest$,NumXltd,Src?[1,40],”EBCDIC”)
```

See also

CALL

FUNCTION TRIM$

Synopsis

Delete leading and trailing spaces from a string value.

Syntax

TRIM$(*str.expr*)

Parameters

str.expr is the string expression to be trimmed.

Remarks

TRIM$ returns *str.expr* with all leading and trailing spaces removed.

Examples

```
X$ = Trim$(X$)
```

See also

CALL, **LTRIM$, RTRIM$**

CALL TRXCO

Synopsis

Control phantom port.

Syntax

CALL TRXCO (*num.expr*, *str.expr*, {, *num.lit* {, *num.expr*}})

Parameters

num.expr is the port, which is used to select the port number for this operation.

str.expr is the command, which selects a command to be sent to the specified port. The supplied command is copied into the specified ports' type-ahead buffer to be processed the next time port is awaiting input. The command may be any system command or prompt response for a running program. Multiple commands, separated by \15\ may be included in the command string.

The optional *num.lit* is the status, an exception value returned to the caller providing completion status of the desired operation.

The optional *num.expr* is the port execution priority, which, after evaluation is truncated to an integer. The valid range is from a low of 1 to a high of 7. The exact effect, if any, of port priority is operating system dependent.

Remarks

For UNIX users, in order to use **CALL TRXCO** or the **PORT** statement, the executable file "scope" must be within one of the directories in your **PATH**. Otherwise, the environment variable **SCOPE** must be set to the path of the "scope" executable, e.g.:

```
SCOPE=/usr/bin/scope
export SCOPE
```

The *status* returned to the caller providing completion status of the desired operation:

Status	Description
0	Successful operation; command transmitted.
1	*port* is not a numeric expression.
2	Specified *port* is out of range 0 to 1023.
3	Specified *port* is not running Basic.
4	Specified *port* is the user's own port.
5	*command* is not a valid *str.expr*.
6	unix fork() operation failed, or *port* is not ready for input.
7	Specified *port* has input already in progress.

TRXCO begins by attempting to attach the *port*. If the *port* is already running Basic, the command is copied into the *port*s' type-ahead buffer. A carriage return is appended to the *string* supplied.

If the *port* is not currently running a Basic process, a background process is created as the supplied *port* number. It assumes the callers identity, environment and current working directory. It then becomes a unique process linked to the supplied *port* number. This *port* is then available for **CALL TRXCO** commands, **PORT**, **SEND**, **RECV**, and **SIGNAL** statements from any other Basic user as well as the program performing the initial **CALL TRXCO**.

When sending commands to a *port* which is connected to a terminal and keyboard, you must ensure that *port* is within Basic before sending commands. Otherwise, a *phantom port* is created for the supplied *port* number. If a user later attempts entry into Basic on a terminal designated as the same *port*, entry will be rejected.

Always pause at least 2 seconds between subsequent **TRXCO** calls to the same or different ports. This permits the receiving *port* time to respond.

Examples

```
A$="Run hello"                  ! dL4 saved program hello
Call Trxco(10,A$,E,2)           ! Low priority
If E Stop                       ! Error trying to start
```

See also

CALL, **PORT**

FUNCTION UBASC

Synopsis

Emulate the UniBasic **ASC** function.

Syntax

UBASC(*strexpr*)

Parameters

str.expr is an expression that specifies a single character to be converted to its UniBasic integer value.

Remarks

This procedure is compatible with the UniBasic **ASC(**n**)** function and always returns values between 0 and 255. ASCII characters are converted to integers between 128 and 255. UniBasic compatible mnemonics are converted to integers between 0 and 127. All other character values are truncated to 8-bits before conversion.

Examples

```
X = UBASC(S$)
```

See also

ASC, CALL, **DECLARE**

FUNCTION UBCHR$

Synopsis

Emulate the UniBasic **CHR** function.

Syntax

UBCHR$(*num.expr*)

Parameters

num.expr is an expression that specifies the character value.

Remarks

This procedure is compatible with the UniBasic **CHR(**n**)** function. Values between 128 and 255 are converted to ASCII values. Values between 0 and 127 are converted to UniBasic compatible mnemonics. All other values are converted to "\177777\".

Examples

```
X$ = UBChr$(193)
```

See also

CHR$, CALL, **DECLARE**

FUNCTION UBMEM

Synopsis

Emulate the UniBasic **MEM** function.

Syntax

UBMEM(*num.expr*)

Parameters

num.expr is an expression that specifies the memory location.

Remarks

This procedure is compatible with the UniBasic **MEM(**n**)** function and always returns zero.

Examples

```
X = UBMem(6)
```

See also

CALL, **DECLARE**

CALL UBSTRING

Synopsis

Perform miscellaneous string functions.

Syntax0

CALL UBSTRING (*num.expr1*, *str.var*)

Syntax1

CALL UBSTRING(*num.expr1*, *str.expr*, *num.var*)

Syntax2

CALL UBSTRING(*num.expr1*, *num.expr2, str.var*)

Parameters

num.expr1 specifies the function to be performed.

str.var is a variable on which to perform a function or into which to return the result.

num.expr2 is a value to be converted into characters.

num.var is a variable into which a converted character value is stored.

Remarks

For modes 3, 4, 6, and 7, ASCII characters are treated as having integer values between 128 and 255. UniBasic compatible mnemonic characters are treated as having integer values between 1 and 127. For modes 3 and 6, Unicode characters outside of the ASCII or UniBasic mnemonic subsets will be truncated to 8-bit values. For modes 4 and 7, integer values outside of the ASCII and UniBasic mnemonic subsets will be translated to “\177777\”.

num.expr1	**Syntax**	**Function**
1	0	Convert characters in *str.var* to upper-case.
2	0	Convert all characters in *str.var* to lower-case.
3	1	Store value of the first character of *str.expr* into *num.var*.
4	2	Store value of *num.expr2* as a character into the first character of *str.var*.
5	0	Copy the command line into *str.var*.
6	1	Store value of the first two characters of *str.expr* into *num.var*. The value is formed by multiplying the value of the first character by 256 and adding the value of the second character.
7	2	Store value of *num.expr2* divided by 256 into the first character of *str.var* and store the value of *num.var* modulo 256 into the second character of *str.var*.

This procedure is compatible with UniBasic **CALL $STRING**.

Examples

```
Call UBString(1,A$)
```

See also

ASC, **INT**, **LCASE$**, **UCASE$**, **CALL**, **CALL STRING**, **CONV**

CALL UNPKDEC21

Synopsis

Unpack numeric data.

Syntax

CALL UNPKDEC21 (*str.expr, str.var*)

Parameters

str.expr is an expression which specifies the string to unpack.

str.var is a string variable that receives the unpacked data.

Remarks

CALL UNPKDEC21 unpacks each character in *str.expr* as a pair of 4 bit nibbles into two characters in *str.var*. Each 4 bit nibble is translated to the equivalent Unicode digit minus one(thus the nibble 0001 is stored as the Unicode character “0”). This procedure is compatible with UniBasic **CALL 21**.

Examples

```
Call PkDec21(PackedNumber$,Number$)
```

See also

CALL, CALL PKDEC20, CALL UNPKDEC46

CALL UNPKDEC46

Synopsis

Unpack numeric data.

Syntax

CALL UNPKDEC46 (*str.expr, str.var*)

Parameters

str.expr is the source expression string.

str.var is the destination string variable.

Remarks

CALL UNPKDEC45 sequentially unpacks each character from *str.expr* into two characters in *str.var*. Each character in *str.expr* is treated as a pair of nibbles which are translated into characters as shown in the table below.

This procedure is compatible with UniBasic **CALL 46**.

Character	Nibble	Character	Nibble
Space	0001	3	1001
,	0010	4	1010
-	0011	5	1011
.	0100	6	1100
/	0101	7	1101
0	0110	8	1110
1	0111	9	1111
2	1000		

Examples

```
Call PkDec46(packeddata$, data$)
```

See also

CALL, CALL PKDEC45, CALL UNPKDEC21

CALL UNPKRDX5019

Synopsis

Unpack characters from radix 50 .

Syntax

CALL UNPKRDX5019 (*str.expr, str.var {,num.expr}*)

Parameters

str.expr is an expression which specifies the string to unpack.

str.var is a string variable that receives the unpacked string.

num.expr is an expression that controls space filling of the *str.var*.

Remarks

CALL PKRDX5019 unpacks character triplets from *str.expr* into *str.var*. Each character pair from *str.expr* forms a 16 bit value by taking the upper 8 bits from the first character and the lower 8 bits from the second character. The 16 bit value contains three radix 50 characters as the sum (Char1 * 40 + Char2) * 40 + Char3. The values of CharN are translated to Unicode as shown in the table below. If *num.expr* is zero or omitted, *str.var* will be space filled. If *num.expr* is one, trailing spaces will be removed. This procedure is compatible with UniBasic **CALL 19**.

Character	Radix 50	Character	Radix 50	Character	Radix 50	Character	Radix 50
0	01	A	11	K	21	U	31
1	02	B	12	L	22	V	32
2	03	C	13	M	23	W	33
3	04	D	14	N	24	X	34
4	05	E	15	O	25	Y	35
5	06	F	16	P	26	Z	36
6	07	G	17	Q	27	,	37
7	08	H	18	R	28	-	38
8	09	I	19	S	29	.	39
9	10	J	20	T	30	Space	00

Examples

```
Call PkRdx5019(packed$,unpacked$)
```

See also

CALL, CALL PKRDX5018, CALL UNPKRDX5049

CALL UNPKRDX5049

Synopsis

Unpack characters from radix 50 .

Syntax

CALL UNPKRDX5049 (*str.expr, str.var*)

Parameters

str.expr is an expression which specifies the string to unpack.

str.var is a string variable that receives the unpacked string.

Remarks

CALL PKRDX5049 unpacks character triplets from *str.expr* into *str.var*. Each character pair from *str.expr* forms a 16 bit value by taking the upper 8 bits from the first character and the lower 8 bits from the second character. The 16 bit value contains three radix 50 characters as the sum (Char1 * 40 + Char2) * 40 + Char3. The values of CharN are translated to Unicode as shown in the table below. This procedure is compatible with UniBasic **CALL 49**.

Character	Radix 50	Character	Radix 50	Character	Radix 50	Character	Radix 50
,	01	7	11	H	21	R	31
-	02	8	12	I	22	S	32
.	03	9	13	J	23	T	33
0	04	A	14	K	24	U	34
1	05	B	15	L	25	V	35
2	06	C	16	M	26	W	36
3	07	D	17	N	27	X	37
4	08	E	18	O	28	Y	38
5	09	F	19	P	29	Z	39
6	10	G	20	Q	30	Space	00

Examples

```
Call PkRdx5049(packed$,unpacked$)
```

See also

CALL, CALL PKRDX5048, CALL UNPKRDX5019

CALL VERIFYDATE

Synopsis

Verify date and convert to standard format.

Syntax

CALL VERIFYDATE(*str.expr* {,*str.var* {,*num.var* {,*num.expr*}}})

Parameters

str.expr is an expression which specifies the string to verify and convert.

str.var is an optional variable which receives the converted date string.

num.var is an optional variable that receives the status of the conversion (0 for success, 1 for illegal date).

num.expr is an optional expression that specifies the output format.

Remarks

The input formatof *str.expr* must be one of the following where MONTH is a month name or three letter abbreviation:

MONTH DD, YYYY

DD MONTH YYYY

MM/DD/YY

MM/DD/YYYY

If *num.expr* is not specified or, when truncated to an integer, is zero, then the output format is "YYMMDD". If the value is non-zero, then the format is "YYYYMMDD".

If *str.var* is not specified, then *str.expr* must be a string variable into which the converted date is stored.

If *num.var* is not specified, then an illegal date will cause an error 38 to occur.

Any non-numeric character will be accepted as the date separator ("/").

If **OPTION DATE FORMAT NATIVE** is used, the input date will use day-month-year ordering if specified by the current locale.

This procedure is compatible with UniBasic **CALL 24**.

Examples

```
Call VerifyDate(D$)
```

See also

CALL, **CALL DATETOJULIAN**

CALL VOLLINK

Synopsis

Create polyfile.

Syntax

CALL VOLLINK (*num.expr1, num.expr2, num.expr3, num.var, array.var*)

Parameters

num.expr1 specifies the channel number open to the file or polyfile.

num.expr2 is ignored.

num.expr3 is the polyfile volume number.

num.var receives the operation status.

array.var is a numeric array which receives volume information (see below).

Remarks

If the volume number *num.expr3* is zero and the channel number *num.expr1* is non-negative, then the indexed-contiguous file open on the channel will be marked as a polyfile. If the channel *num.expr1* is less than zero, the first element of the array *array.var* will be zeroed. If *num.expr1* is not an open channel number, the status *num.var* will be set to 1. If the volume number *num.expr3* is not zero when marking a polyfile, the status *num.var* will be set to 16. This procedure is compatible with UniBasic **CALL 91** and **CALL $VOLLINK**.

Examples

```
Call VOLLINK(5,0,0,S,P[])
```

See also

CALL

CALL WHOLOCK

Synopsis

Determine which port or process has locked a record.

Syntax

CALL WHOLOCK(*num.expr1, num.expr2, num.var1 {,num.var2*})

Parameters

numr.expr1 is a numeric expression which specifies a channel open to a file.

num.expr2 is a numeric expression which specifies a record number in the file open on channel *num.expr1*.

*num.var1*is a variable that receives the port number that currently has the specified record locked or -1 if the record is not locked by a dL4 process.

num.var2 is an optional variable that receives the operating system defined process id number of the process that has the specified record locked or -1 if the record is not locked by another process.

Remarks

CALL WHOLOCK is supported only for Formatted, Contiguous, and Indexed-Contiguous files.

CALL WHOLOCK is not supported on Windows due to operating system limitations and will always return -1 as in *num.var1* and *num.var2*.

This procedure is compatible with UniBasic **CALL $WHOLOCK**.

Examples

```
Call WhoLock(ChanNo,RecNbr,PortNo)
```

See also

CALL, **PORT**

Chapter 9 - File Specification

file.spec Definition

A *file.spec* is an expression used in a dL4 BASIC program to either open or build a file. The expression consists of a list of items. The standard list of items consists of a Filename Item, an Option Item, a Protection Item, a Number of Records Item, and finally a Record Length Item. These items can be specified either as a single string expression or as a list of items. The single string expression and the list of items are referred to as a *file.spec.str* and a *file.spec.items*, respectively in this manual.

The *file.spec.str* is internally parsed into the standard list of items. Thus, a non-standard list of items cannot be specified in a *file.spec.str*. Unlike a *file.spec.str*, a *file.spec.items* can use both the standard and a non-standard list of items. Thus, a *file.spec.items* must be used when opening a driver that requires a non-standard list of items.

This chapter includes a detailed discussion with examples for both a *file.spec.str* and a *file.spec.items*. In addition, it provides a detailed description of each individual items and concludes with a small running program.

file.spec.str

A *file.spec* expressed as a single string expression is referred to as a *file.spec.str*. A generic and a specific example of a *file.spec.str* respectively would be:

"(option item) <protection item> $cost item [number of records item : record length item] filename item!"

"(charset=ebcdic) <62> $99.99 [100:10] myfile!"

The following rules apply to a *file.spec.str:*

- Except for the filename item which is required and must be the last item, the remaining individual items are discretionary and can be expressed in any order, but they must be grouped together as a single string expression.
- The exclamation point (!) in the filename item is used only with the **BUILD** statement to replace an existing file.
- The option item, the protection item, and the cost item must be surrounded by parentheses (()), angle brackets (<>), and must begin with a leading dollar sign ($), respectively.
- The dollar sign ($) is the only allowable currency designator in the cost item.
- The number of records and the length of each record are specified as a single item, enclosed by square brackets ([]), and are separated by a colon (":").

An example of a *file.spec.str* using the **BUILD** statement is as follows:

```
BUILD #9, "(charset=ebcdic) <62> $99.99 [100:10] myfile!"
```

The **BUILD** statement above builds a new Contiguous file, called myfile, by replacing myfile if it already exists. An explanation of each individual item in this example follows:

- Option Item - selects an EBCDIC character set instead of the default character set.
- Protection Item - set to 62, prohibiting reading and writing by other groups, and prohibiting writing by the same group.
- Cost Item - 99.99 is selected.
- Number of Records Item - create 100 initial records.
- Record Length Item - create a file with a record length of 10 words each.
- Filename Item - the name of the file is myfile, which is created in the user's current directory. The exclamation point replaces myfile if it already exists.

file.spec.items

A *file.spec*, which begins in a "{" and ends in a "} and is expressed as a list of items, is referred to as a *file.spec.items.* A generic and a specific example of a *file.spec.items* respectively would be:

{"filename item!", "option item", "protection item", cost item, number of records item, record length item}

{"myfile!", "charset=ebcdic", "62", 99.99, 100,10}

Although the typical usage is *file.spec.str*, the actual interpretation of each item in the list of items is driver-class dependent. A *file.spec.items* must be used if the driver-class interprets the list of items differently.

Unlike a *file.spec.str*, each individual item in a *file.spec.items* must be defined separately. Each item has a data type associated with it, and the appropriate data type must be used for each particular item. In addition, the As "driver-class" must be used with the **BUILD** statement. The data types of each individual items in a *file.spec.items* are as follows:

ITEM	DATA TYPE	COMMENTS
Filename Item	String	A required item with an optional exclamation point (!) to replace and build an existing file. "" is allowed, but will generate an error since "" is not a valid filename.
Option Item	String	"" is allowed, meaning no option specified. Surrounding parentheses () are not allowed.
Protection Item	String	"" is allowed, meaning no protection specified. Surrounding angle brackets (<>) are not allowed.
Cost Item	Numeric	Must specify a legal value. A zero is allowed.
Number of Records Item	Numeric	Specified as a single numeric item.
Record Size Item	Numeric	Specified as a single numeric item.

The following rules apply to a *file.spec.items*:

- A standard list of items must be in the following order: Filename Item, Option Item, Protection Item, Cost Item, Number of Records Item, Record Length Item.
- Surrounding parentheses (()) are not allowed in a Option Item.
- Surrounding angle brackets (<>) are not allowed in a Protection Item.
- The interpretation of each item is driver-class-specific. Therefore, the way each item is interpreted depends upon which specific driver-class is in use.
- The list of items must always appear in order.
- Any discretionary item after the last specified item may be omitted while attempting to open a file. Thus, a file may be opened without write access as follows:

```
OPEN #9,{""myfile"", "", "w"}
```

- The driver-class/name must be specified with an AS clause if the list is used in a **BUILD** statement.

An example of a *file.spec.items* using the BUILD statement is as follows:

BUILD #9,{"myfile!", "charset=ebcdic", "62", 99.99, 100,10} As "Contiguous"

In addition to grouping the list of items within braces, "{}", the list of items can also be specified in a structure variable. Thus, the previous example can also be written as:

```
BUILD #0, struct.var  As "Contiguous"
```

The **BUILD** statements above build a Contiguous file, called myfile, and replace myfile, if it already exists. An explanation of each individual item for the above example follows:

- Filename Item - the name of the file is myfile, which is created in the user's current directory. The exclamation point (!) replaces the file that may already exist.
- Option Item - selects an EBCDIC character set instead of the default character set.
- Protection Item - set to 62, prohibiting reading and writing by other groups, and prohibiting writing by the same group.
- Cost Item - 99.99 is selected.
- Number of Records Item - create 100 initial records.
- Record Length Item - create a file with a record length of 10 words each.
- Each item in the list of items must be specified, even if it is not used, while building a file.

The Standard List of Items

The standard list of items in a file specification, or *file.spec*, is described in the following paragraphs.

Filename Item

A filename is a string literal or expression containing a filename which is optionally preceded by a relative or absolute directory pathname. A filename must always be specified in a *file.spec*. A filename that contains embedded spaces must be enclosed in quotation marks.

The final optional exclamation point (!) allows creation of a new file, even if a file already exists. This creation is performed by first deleting the old file, if it already exists, then creating the new file. The exclamation point is used only with the **BUILD** statement.

If the final optional exclamation point (!) is omitted, an error will occur while attempting to build an existing file.

Option Item

An Option Item changes driver-class dependent behavior of the driver-class. The general syntax for an Option Item is:

```
option-name=value {, option-name=value}...
```

For example, to create a file with the EBCDIC character set, the option item in the **BUILD** statement is set to charset=ebcdic. In the absence of the Option Item, the driver-class would have built the file with its own default character set.

The syntax optionally allows for additional comma separated options.

Protection Item

A Protection Item allows for the manipulation of file permissions. It can be specified to change the default read and write protection during the building or opening of a file. The methods for specifying protection during **BUILD** and **OPEN** are described in the following paragraphs.

File protection is ultimately Operating System dependent, therefore the Protection Item specified is translated to be compatible with the Operating System format.

Specifying Protection During BUILD

There are three (3) methods to specify a protection string while building a file. These methods are described in the following paragraphs.

Protection by Attribute Letters

The first method is to specify attribute letters. The meaning of each letter is listed below:

A	Allow reading by any member of the group.
B	Allow writing by any member of the group.
D	Prohibit deletion of the file. (operating system specific.)
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.

The attributes are created by combining the above letters, where each letter is used only once. In other words, "RR" is an illegal protection value.

For example, "AW" allows reading by any member of a group, and prohibits writing by anyone except the file owner.

Protection by Two-Digit Number

The second method to specify protection is to use a two-digit number. The meaning of each digit is described below:

40	Prohibit reading by other groups.
20	Prohibit writing by other groups.
10	Prohibit copying by other groups. (operating system specific.)
04	Prohibit reading by the same group.
02	Prohibit writing by the same group.
01	Prohibit copying by the same group. (operating system specific.)

The two-digit attributes are calculated by summing the desired digits, where each digit is added only once in a valid operation. In other words, 48 (40 + 4 + 4) is an illegal protection value, because 4 is added twice. Thus, 77 is the highest available legal value.

For example, if the desired attributes are "Prohibit reading by other groups" and "Prohibit writing by the same group", then these attributes can be summed as 40 plus 02 to equal a sum of 42.

Protection by Three-Digit Number

The third method to specify protection is to use a three-digit number. The meaning of each digit is described below:

400	Owner can read the file.
200	Owner can write to the file.
100	Owner can execute the file.
40	Group can read the file.
20	Group can write to the file.
10	Group can execute the file.
04	Others can read the file.
02	Others can write to the file.
01	Others can execute the file.

The meaning of the execute permission is operating system specific.

The three-digit attributes are calculated by summing the desired digits, where each digit is added only once in a valid operation. In other words, 448 (400 + 40 + 4 + 4) is an illegal protection value, because 4 is added twice. Thus, 777 is the highest available legal value.

Examples are shown below:

PROTECTION	MEANING
777	owner, group, and public can read, write, and execute file
744	owner can read, write, and execute; group and public can read file
644	owner can read and write; group and public can read file
711	owner can read, write, and execute; group and public can execute file

Specifying Protection During OPEN

When a file is opened, protection is specified by selecting a combination of the letters listed below.

R	Open a file without read permission
W	Open a file without write permission
E	Open a file in exclusive mode (driver-class dependent)
L	Open a file and disable record locking (driver-class dependent)

Up to four unique letters can be selected.

For example, "RW" protection value prohibits reading from and writing to the file. A "RWW" protection value is an illegal combination, because the letter W is selected twice.

Cost Item

A Cost Item is a floating point monetary unit whose meaning is driver-class dependent.

Number of Records Item

A Number of Records Item provides a method to specify the number of records.

Record Length Item

A Record Length Item provides a method to specify the record size.

Example of *file.spec*

The program below demonstrates the use of a *file.spec.str* to build and open a Contiguous file.

```
10 DIM S$[20], B?[20]
20 BUILD #9, "(charset=ebcdic) <62> $99.99 [100:10] myfile!"
30 WRITE #9,0; "My File"
40 CLOSE #9
50 OPEN #9, "<W> myfile"   \ REM Open without Write permission
60 READ #9, 0; S$
70 READ #9,0;B?
80 PRINT S$, HEX$ (B?)  \ REM Verify that data was written/read
correctly
90 CLOSE
```

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 - dL4 Reserved Words

The following list shows dL4 reserved words, also called keywords. You cannot use any of these words as a variable, label, or procedure name. Each of the reserved words has a fixed function and cannot be used for any other purpose.

ABS
ACCESS
ADD
ALL
ALTERNATE
AND
ANGLE
ARITHMETIC
AS
ASC
ASCENDING
ATN
AUTO
AUTO
BSTR$
BVAL
BASE
BINARY
BOX
BUFFER
BUILD
BY
BYTES
CALL
CASE
CHDIR
CHAIN
CHANNEL
CHF
CHF$
CHR
CHR$
CLEAR
CLOSE
COLLATE
COM
COMMA
CON
CONV
COS
DAT#
DATA
DATE
DECIMAL
DECIMALS
DECLARE
DEF
DEFINE
DEGREES
DELETE
DESCENDING
DET
DIM
DIRECTORIES
DISPLAY
DO
DUPLICATE
DUPLICATES
EDIT
ELSE
END
ENTER
EOFCLR
EOFSET
EOPEN
ERASE
ERM$
ERR
ERRCLR
ERROR
ERRSET
ERRSTM
ESCCLR
ESCDIS
ESCSET
ESCSTM
EXCEPT
EXIT
EXP
EXTERNAL
FAILURE
FILE
FOR
FORMAT
FRA
FREE
FUNCTION
GET
GMT#
GMT$
GOSUB
GOSUB
GOTO
HEX$
HEX?
IDN

IF	NATIVE	SETFP
IGNORED	NESTING	SGN
INDEX	NEXT	SIGNAL
INPUT	NOT	SIN
INT	NUMERIC	SIZE
INTCLR	OFF	SPACING
INTRINSIC	ON	SPAWN
INTSET	OPEN	SPC
INV	OPTION	SQR
IS	OR	STANDARD
ITEM	PCHR$	STATEMENTS
IXR	PAUSE	STEP
JUMP	PERIOD	STOP
KEY	PORT	STR$
KILL	POS	STRING
LBOUND	PRINT	STRINGS
LCASE$	RTRIM$	STRUCT
LTRIM$	RADIANS	SUB
LEN	RANDOM	SUBSCRIPTS
LET	RAW	SUSPEND
LIB	RDLOCK	SWAP
LIKE	READ	SYSTEM
LINE	RECORD	TAB
LINES	RECV	TAN
LOG	REM	THEN
LOOP	REP$	TIM
MAN	RESTOR	TIM#
MAP	RETRY	TIMEOUT
MAT	RETURN	TIMEZONE
MEMBER	RETURNED	TO
MOD	REWIND	TRACE
MODIFY	RND	TRN
MONTH	ROPEN	TRUNCATE
MONTH$	ROUND	TRY
MONTHDAY	SEARCH	UBOUND
MOVE	SELECT	UCASE$
MSC	SEND	UNIQUE
MSC$	SET	UNIT

UNLOCK

UNTIL

UPPERCASE

USING

VAL

WEEKDAY

WEEKDAY$

WEND

WHILE

WINDOW

WOPEN

WORDS

WRITE

WRLOCK

YEAR

YEARDAY

ZER

Appendix C - BASIC Error Codes

The BASIC error messages, preceded by their numbers, are listed below. All errors have in common the fact that they are recognized from a statement.

0 - No such error.

1 - Syntax error.

2 - Illegal string operation.

3 - Storage overflow.

4 - Format error.

5 - Character is illegal or not supported by driver.

6 - No such line.

7 - Line too long.

8 - Too many variable names.

9 - Unrecognizable word.

10 - GO is illegal before an initial run.

11 - Incorrect parentheses closure.

12 - Program is list/copy protected.

13 - Number out of range.

14 - Out of data.

15 - Arithmetic or date overflow.

16 - GOSUBS nested too deep.

17 - RETURN without GOSUB.

18 - FOR-NEXT loops nested too deep.

19 - FOR without matching NEXT.

20 - NEXT without matching FOR.

21 - Expression too complex.

22 - Illegal numeric or date precision.

23 - No such error.

24 - Too many dimensions.

25 - Variable not dimensioned.

26 - Directory not found.

27 - Too many procedure parameters.

28 - Parameter out of range.

29 - Illegal function usage.

30 - Procedure not declared or defined.

31 - Procedures nested too deep.

32 - Matrices have different dimensions.

33 - Argument is not a matrix.

34 - Dimensions are not compatible.

35 - Matrix is not 'square'.

36 - Intrinsic procedure not found.

37 - No such error.

38 - Error detected by CALLed subroutine.

39 - Formatted output exceeded buffer size.

40 - Channel in use.

41 - Illegal filename.

42 - File not found.

43 - Syntax error in file specification.

44 - Incompatible file type (can't open or replace).

45 - File is read-protected.

46 - File is write-protected.

47 - Disk or directory is full.

48 - Accounts disk block allotment is insufficient

49 - Channel not open.

50 - File is copy-protected.

51 - Illegal record number.

52 - Record not written.

53 - Illegal item number.

54 - Item types don't match.

55 - Statement is illegal from keyboard.

56 - No current program.

57 - Variable already dimensioned.

58 - Error in format string.

59 - Variable is in-use.

60 - Too many numbers entered for INPUT.

61 - Illegal data type.

62 - Signal buffer is full or no such port.

63 - Illegal number/types of args for specified dri....

64 - Illegal line number.

65 - Filename in use for different type file.

66 - Filename in use, being built or replaced.

67 - Filename in use and no exclamation point ('!').

68 - Filename in use by a different account.

69 - File is a processor or driver.

70 - Data read error.

71 - No such driver.

72 - Device not accessible.

73 - Device not on line.

74 - Device requires manual intervention.

75 - Line exceeds buffer size.

76 - File or device is open elsewhere.

77 - Directory access denied.

78 - File is being built, replaced, or deleted.

79 - Illegal driver operation.

80 - Disk does not have enough contiguous blocks.

81 - Device profile not set up properly.

82 - Too many channels in use.

83 - Component file deleted or inaccessible.

84 - Internal error in driver.

85 - Array dimension(s) too large.

86 - Illegal subscript value.

87 - Illegal subroutine name (length or illegal characters).

88 - Illegal usage of multi-statement line.

89 - Program not authorized to use privileged function.

90 - Driver resource exhausted.

91 - Variable in CHAIN READ not passed by CHAIN WRITE.

92 - Variable from CHAIN WRITE not in this program.

93 - Variable in CHAIN READ already contains data.

94 - Variable in CHAIN WRITE contains no data.

95 - Input timed out.

96 - Aborted by ALTESCAPE or MESSAGE event.

97 - Unexpected error status returned by system call.

98 - Illegal value entered for input.

99 - ESCAPE trapped by error branch.

100 - Operation interrupted by abortive channel event.

101 - No such error.

102 - No such error.

103 - No such error.

104 - No such error.

105 - No such error.

106 - No such error.

107 - No such error.

108 - No such error.

109 - No such error.

110 - No such error.

111 - No such error.

112 - No such error.

113 - No such error.

114 - No such error.

115 - No such error.

116 - No such error.

117 - No such error.

118 - No such error.

119 - No such error.

120 - No such error.

121 - No such error.

122 - No such error.

123 - Record is locked.

124 - Record is not locked.

125 - No such error.

126 - No such error.

127 - No such error.

128 - No such error.

129 - No such error.

130 - No such error.

131 - No such error.

132 - No such error.

133 - No Dynamic Window open.

134 - Dynamic Windows not enabled.

135 - Variable is not a structure.

136 - Structure definition not found.

137 - Structure variable has no declared type.

138 - Structure variable already declared.

139 - No such structure member.

140 - Procedure not found.

141 - Procedure is not a function.

142 - Procedure is not a subprogram.

143 - Procedure parameter multiply declared.

144 - Statement is illegal in a procedure.

145 - Illegal procedure nesting.

146 - Inconsistent procedure declaration or definitio....

147 - Illegal variable name declared as procedure.

148 - Illegal procedure name declared as variable.

149 - Type of return value does not match function ty....

150 - Procedure calls are illegal from keyboard.

151 - Message too large.

152 - Port is already in-use.

153 - Illegal port number.

154 - No ports available.

155 - No messages waiting.

156 - Port is not in-use.

157 - Duplicate line label.

158 - Duplicate line number.

159 - Illegal line reference.

160 - Not an indexed file.

161 - Invalid or non-existent index specified.

162 - Key size larger than destination string.

163 - BASIC program has not been successfully compiled.

164 - Unable to load program - invalid file version.

165 - Unable to load program - file can be corrupted.

166 - COM statement out of order.

167 - COM or CHAIN READ variable type mismatch.

168 - TRY blocks nested too deep.

169 - TRY without ELSE.

170 - TRY without END TRY.

171 - RETRY without TRY.

172 - END TRY without TRY.

173 - Statement is illegal in TRY.

174 - DEF STRUCT without END DEF.

175 - MEMBER without DEF STRUCT.

176 - Statement is illegal in DEF STRUCT.

177 - Duplicate member definition.

178 - No members defined.

179 - END DEF without DEF STRUCT.

180 - DO without LOOP.

181 - UNTIL/WHILE at both ends of DO/LOOP.

182 - EXIT DO without DO.

183 - LOOP without DO.

184 - Duplicate OPTION setting.

185 - Illegal OPTION setting.

186 - SELECT CASE without END SELECT.

187 - CASE without SELECT CASE.

188 - Lines between SELECT CASE and first CASE.

189 - Missing CASE.

190 - END SELECT without SELECT CASE.

191 - SUB without END SUB.

192 - EXIT SUB not inside a subprogram.

193 - END SUB without SUB.

194 - FUNCTION without END FUNCTION.

195 - EXIT FUNCTION not inside a function.

196 - END FUNCTION without FUNCTION.

197 - WHILE without WEND.

198 - WEND without WHILE.

199 - Statement is illegal in IF.

200 - No such error.

201 - IFs without END IF.

202 - ELSE without IF or TRY.

203 - END IF without IF.

204 - Can't insert line; program must be renumbered.

205 - Line numbers are illegal or overlap lines.

206 - Subprogram file not found.

207 - No such error.

208 - Number/types of arguments do not match param list.

209 - ENTER is illegal if not in a subprogram.

210 - No such error.

211 - Program filename must be specified (no current ....

212 - Subprogram file is read protected.

213 - Subprogram file is not a BASIC program.

214 - No such error.

215 - No such error.

216 - Param variable in ENTER statement has already b....

217 - The ENTER statement can only be executed once i....

218 - Cannot execute command, all channels are in use.

219 - Program was not interrupted by a SUSPEND statem....

220 - Program change would invalidate running program.

221 - Statement, function, or feature not implemented.

222 - No such character set.

223 - Duplicate character set name.

224 - Directory not empty.

225 - Directory has too many links.

226 - Error executing device macro.

227 - Illegal or missing field name.

228 - Illegal DECIMALS setting.

229 - DECIMALS option must be specified for this file....

230 - No field of that name exists.

231 - Duplicate of existing field name.

232 - Field already mapped.

233 - Field is too long for this file type.

234 - Duplicate of existing index name.

235 - Key option not supported by this file type.

236 - Duplicate key in unique index.

237 - File must be empty to define record or index.

238 - Error in source file.

239 - Error in source line.

240 - Unable to link program.

241 - Duplicate procedure name.

242 - Unsatisfied reference to procedure.

243 - Error in link file.

244 - Intrinsic procedure not declared as intrinsic.

245 - Duplicate of intrinsic procedure name.

246 - Intrinsic procedure table contains duplicate symbols.

247 - Long CHAIN attempted.

248 - Procedure not active.

249 - No such variable.

250 - Resource in use.

251 - Program in use.

252 - Breakpoint not in current program.

253 - No such breakpoint.

254 - Open mode not supported by this driver.

255 - Licensing failure.

256 - File position limit exceeded.

257 - System file position limit exceeded.

258 - Illegal record length.

259 - Illegal sequence of operations.

260 - Error in index.

261 - Error on channel.

262 - Invalid access name or password.

263 - Unexpected value returned by system call.

264 - Record data is out of date (modified by other user).

265 - Not licensed to load or create this program.

266 - Procedure declared as both intrinsic and non-intrinsic.

267 - Operation would corrupt file

268 - Default option changed after options used

269 - Duplicate structure definition

270 - Include file not found

271 - Include files nested too deep

272 - Procedure not defined in conversion profile

273 - Not licensed to use this feature

274 - SQL syntax error

275 - Additional system error information

276 - Field definition overlaps an existing field

277 - Index field definition does not match record field definition

278 - Index definition does not match actual index

279 - SQL implementation or configuration limit exceeded

280 - SQL procedure error

281 - SQL constraint not satisfied

32768 - Impossible state detected, interpreter abort.

Appendix D - dL4 Statements (Quick Reference)

ADD	Define structure of file, or expand file.
ADD INDEX	Add an index to a file.
ADD RECORD	Add new record to file.
BOX	Draw rectangular figure on display device.
BUILD	Create and open a file.
CALL BASIC Program	Call a BASIC program.
CALL Procedure	Call a procedure.
CASE	Control complex conditional and branching operations.
CHAIN	Transfer control to another program.
CHAIN READ	Read variables from a previous program.
CHAIN READ IF	Read variables from a previous program.
CHAIN WRITE	Write variables to the next program.
CHANNEL	Perform a driver-specific command.
CHDIR	Change default directory to the path specification.
CLEAR	Clear an open channel or initialize variables.
CLOSE	Close {all} open channel{s}.
COM	Specify common variables.
CONV	Convert binary data to decimal, or convert decimal data to binary.
DATA	Define internal program data.
DECLARE	Declare a procedure which precedes the actual definition.
DEF FN	Define user function.
DEFINE RECORD	Define a record in a file.
DEF STRUCT	Define a structure.
DELETE INDEX	Delete an index in a file.
DELETE RECORD	Delete current record from a file.
DIM	Allocate space for variables.
DO	Establish program loops.
DO UNTIL	Perform a loop as long as the expression is false.
DO WHILE	Perform a loop as long as the expression is true.
DUPLICATE	Duplicate a file.
EDIT	Format numeric and string expressions.
ELSE	Control conditional branching.
END	Terminate a running program.
END DEF	Define the end of a structure definition.
END FUNCTION	End a FUNCTION definition.
END IF	End conditional branch.
END SELECT	End complex conditional branch.
END SUB	End a procedure or function.
END TRY	End redirection of error branching.
ENTER	Accept arguments into a procedure.
EOFCLR	Clear end-of-file branching.
EOFSET	Enable end-of-file error setting.
EOPEN	Exclusively OPEN a data file.
ERASE	Perform driver-class dependent function(s).
ERRCLR	Clear error branching.
ERROR	Create a dL4 BASIC error to the current running program.
ERRSET	Enable branch to statement on error.
ERRSTM	Specify statements to execute on an error.
ESCCLR	Clear any ESCape branching in effect.
ESCSET	Enable branch to statement on ESCape.
ESCDIS	Disable Escape key.

ESCSTM	Specify statements to execute on Escape.
EXIT DO	Exit a DO loop.
EXIT FOR	Exit a FOR/NEXT loop.
EXIT FUNCTION	Exit a named function.
EXIT SUB	Exit a named subroutine.
EXTERNAL FUNCTION	Define an independent function.
EXTERNAL LIB	Declare named library file.
EXTERNAL SUB	Define an independent subroutine.
FOR	Repeat a group of statements.
FREE	Deallocate (undimension) variables.
FUNCTION	Define a multi-procedure which returns a value.
GET	Obtain class-driver dependent parameters from a channel opened to a file.
GOSUB	Unconditional branch to internal group of statements, saving return point.
GOTO	Unconditional branch to statement.
IF	Control conditional branching.
IR ERR 0	Specify a line of statements to execute on the occurrence of an error
IF ERR 1	Specify an error branch.
INPUT	Retrieve keyboard or channel input.
INTCLR	Clear program interrupt branch.
INTSET	Define a branch for program interrupts.
JUMP	Transfer control immediately to another location.
KILL	Delete a data or program file.
LET	Assign values to variables.
LIB	Specify a directory name for callable subprograms.
LINE	(A function of drivers)
LOOP	Mark the end of a group of statements enclosed in a DO loop.
MAP	Define the logical index or directory number used within the application.
MAP RECORD	Define an alternate item number mapping at run-time.
MAT=	Copy an entire matrix.
MAT+	Add elements from two matrices.
MAT*	Multiply elements of two matrices.
MAT CON	Establish a constant matrix.
MAT IDN	Establish an identity matrix.
MAT INPUT	Assign keyboard/file input to a matrix.
MAT INV	Invert a matrix.
MAT PRINT	Print contents of an array or matrix.
MAT RDLOCK	Read an array, matrix, or string with locking.
MAT READ	Read an array, matrix, or string from DATA or a channel.
MAT TRN	Transpose a matrix.
MAT WRITE	Write array, matrix, or string to a channel.
MAT WRLOCK	Write an array, matrix, or string with locking.
MAT ZER	Zero an entire matrix.
MEMBER	Define a member associated with a specific structure.
MODIFY	Change filename or attributes/permissions.
MOVE	Move a window.
NEXT	Continuation of FOR loop statement.
ON	Conditional branch on value of expression.
OPEN	Open {a file for Read and Write access}{a Driver ...}
OPTION	Specify a runtime option for the current program unit.
OPTION DEFAULT	Specify a runtime option for all program units in the current program.
PAUSE	Suspend program execution.
PORT	Attach and control other ports.
PRINT	Output ASCII to screen, file, or device.
RANDOM	Seed random generator for RND function.
RDLOCK	Read and unconditionally lock a record.
READ	Read variables from DATA structures.
READ RECORD	Read an entire structure and update indexes.
RECV	Receive communication message.
REM	Make a non-executed program comment.

RESTOR	Reset DATA pointer for READ statement.
RETRY	Repeat last TRY statement.
RETURN	Return from previous GOSUB subroutine call.
REWIND	Reset a file to the first data byte.
ROPEN	Open a file for Read-only access.
SEARCH (String)	Search string for sub-string.
SEARCH (Traditional)	Maintain index of an Indexed file.
SEARCH (Modern)	Locate a key.
SELECT	Select the size of a window, in columns and rows.
SELECT CASE	Organize blocks of statements.
SEND	Transmit a message to another port.
SET	Read and write class-driver dependent parameters on a channel.
SETFP	Set file position for sequential access.
SIGNAL	Transmit/receive ported messages and pause.
SIZE	Select the size of a window in columns and rows.
SPAWN	Launch a background BASIC program.
STOP	Abnormally terminate a program.
SUB	Define subroutine procedure.
SUSPEND	Abnormally terminate a program.
SWAP	Pause and execute another BASIC program.
SYSTEM	Execute system functions and commands.
TRACE	Enable statement trace debugging.
TRY	Perform single-line or blocked, nested error handling.
UNLOCK	Unlock any records on a channel.
WEND	With WHILE, block a set of repeated statements.
WHILE	With WEND, block a set of repeated statements.
WINDOW	Maintain Dynamic Windows.
WOPEN	Open a file/device for Write-only.
WRITE	Write array, matrix, or string from a channel.
WRITE RECORD	Write entire structure and update indexes.
WRLOCK	Write and unconditionally lock a record.

Appendix E - dL4 Statement Groups

Introduction

This appendix describes the dL4 statement set by dividing the statements into groups. Each of these groups, such as File and Device Handling or Windows, should be familiar to you from your previous programming experience.

Groups

The dL4 statements have been divided into meaningful groups according to function. A subset of all the statements listed below includes statements that communicate with a channel; these statements are **boldfaced**.

GROUP NAME	dL4 STATEMENTS IN GROUP
1. File and Device Handling	ADD, **ADD INDEX** , **ADD RECORD**, **BUILD**, CHANNEL, CHDIR, **CLEAR** , **CLOSE**, **DEFINE RECORD**, **DELETE INDEX**, DELETE RECORD, DUPLICATE, EOPEN, GET, INPUT, KILL, MAP, **MAP RECORD**, MODIFY, **OPEN**, **RDLOCK** , **READ**, **READ RECORD**, **ROPEN**, **REWIND**, **SEARCH**, SET, **SETFP**, **UNLOCK**, WOPEN, **WRITE** , **WRITE RECORD** , **WRLOCK**
2. User Subroutines and Functions	DECLARE, DEF, END FUNCTION, END SUB, ENTER, EXIT FUNCTION, EXIT SUB, EXTERNAL FUNCTION, EXTERNAL LIB, EXTERNAL SUB, FUNCTION, GOSUB, INTRINSIC FUNCTION, INTRINSIC SUB, LIB, SUB
3. Error and Interrupt Handling	END TRY, EOFCLR, EOFSET, ERRCLR, ERROR, ERRSET, ERRSTM, ESCCLR, ESCDIS, ESCSET, ESCSTM, IF ERR 0, IF ERR 1, INTCLR, INTSET, TRY, RETRY
4. Arrays and Matrices	MAT, MAT INPUT, MAT PRINT, **MAT RDLOCK** , **MAT READ, MAT WRITE**, **MAT WRLOCK**
5. Data Structures	COM, DEF STRUCT, DIM, END DEF, **ERASE**, MEMBER, FREE, LET
6. Program Flow	CALL, CHAIN, CHAIN READ, CHAIN READ IF, CHAIN WRITE, END, GOTO, JUMP, PAUSE, RETURN, SPAWN, STOP, SUSPEND, SWAP
7. Blocks and Loops	CASE, CASE ELSE, DO, DO UNTIL, DO WHILE, ELSE, END IF, END SELECT, EXIT DO, EXIT FOR, FOR, IF, LOOP, LOOP UNTIL, LOOP WHILE, NEXT, ON, THEN, SELECT CASE, WEND, WHILE
8. Communications	PORT, RECV, SEND, SIGNAL
9. Windows	MOVE, SIZE, WINDOW CLEAR, WINDOW CLOSE, WINDOW MODIFY, WINDOW OFF, WINDOW ON, WINDOW OPEN
10. Formatting Output	PRINT, EDIT
11. Miscellaneous Statements	BOX, CONV, DATA, LINE, OPTION, RANDOM, REM, RESTORE, SYSTEM, TRACE

In grouping these statements by function, no presumption of evenness is implied, as each group contains both statements with broad and also others with very specific functionality. No presumption is made about importance, either, because the relative importance or influence of a statement is dependent on the individual developer's perception. The statements are grouped only according to the kinds of effects they have on development.

File and Device Handling

ADD	Define structure of file, or expand file.
ADD INDEX	Add an index to a file.
ADD RECORD	Add new record to file.
BUILD	Create and open a file.
CHANNEL	Perform a driver-specific command.
CHDIR	Change default directory to the path specification.
CLEAR	Clear an open channel or initialize variables.
CLOSE	Close {all} open channel{s}.
DEFINE RECORD	Define a record in a file.
DELETE INDEX	Delete an index in a file.
DELETE RECORD	Delete current record from a file.
DUPLICATE	Duplicate a file.
EOPEN	Exclusively OPEN a data file.
GET	Obtain class-driver dependent parameters from a channel opened to a file.
INPUT	Retrieve keyboard or channel input.
KILL	Delete a data or program file.
MAP	Define the logical index or directory number used within the application
MAP RECORD	Define an alternate item number mapping at run-time.
MODIFY	Change a filename or attributes/permission.
OPEN	Open {a file for Read and Write access}{a Driver...}
RDLOCK	Read and unconditionally lock a record.
READ	Read variables from DATA structures.
READ RECORD	Read entire structure and update indexes.
ROPEN	Open a file for Read-only access.
REWIND	Reset a file to the first data byte.
SEARCH (String)	Search string for sub-string.
SEARCH (Trad.)	Maintain index of Indexed file.
SEARCH (Mod.)	Locate a key.
SET	Read/write class-driver dependent parameters on channel.
SETFP	Set file position for sequential access.
UNLOCK	Unlock any records on a channel.
WOPEN	Open a file/device for Write-only.
WRITE	Write array, matrix, or string from a channel.
WRITE RECORD	Write entire structure and update indexes.
WRLOCK	Write and unconditionally lock a record.

User Subroutines and Functions

DECLARE	Declare a procedure which precedes the actual definition.
DEF FN	Define user function.
END FUNCTION	End a FUNCTION definition.
END SUB	End a procedure or function.
ENTER	Accept arguments into a procedure.
EXIT FUNCTION	Exit a named function.
EXIT SUB	Exit a named subroutine.
EXTERNAL FUNCTION	Define an independent function.
EXTERNAL LIB	Declare named library file.
EXTERNAL SUB	Define an independent subroutine.
FUNCTION	Define a multi-procedure which returns a value.
GOSUB	Unconditional branch to internal group of statements, saving return point.
LIB	Specify a directory name for callable subprograms.
SUB	Define subroutine procedure.

Error and Interrupt Handling

END TRY	End redirection of error branching.
EOFCLR	Clear end-of-file branching.
EOFSET	Enable end-of-file error setting.
ERRCLR	Clear error branching.
ERROR	Create a dL4 BASIC error to the current running program.
ERRSET	Enable branch to statement on error.
ERRSTM	Specify statements to execute on an error.
ESCCLR	Clear any Escape branching in effect.
ESCDIS	Disable Escape key.
ESCSET	Enable branch to statement on Escape.
ESCSTM	Specify statements to execute on Escape.
IF ERR 0	Specify a line of statements to execute on the occurrence of an error.
IF ERR 1	Specify an error branch.
INTCLR	Clear program interrupt branch.
INTSET	Define a branch for program interrupts.
TRY	Perform single-line or blocked, nested error handling.
RETRY	Repeat last TRY statement.

Arrays and Matrices

MAT=	Copy an entire matrix.
MAT+	Add elements from two matrices.
MAT*	Multiply elements of two matrices.
MAT CON	Establish a constant matrix.
MAT IDN	Establish an identity matrix.
MAT INPUT	Assign keyboard/file input to a matrix.
MAT INV	Invert a matrix.
MAT PRINT	Print contents of an array or matrix.
MAT RDLOCK	Read an array, matrix, or string with locking.
MAT READ	Read an array, matrix, or string from DATA or a channel
MAT TRN	Transpose a matrix.
MAT WRITE	Write array, matrix, or string to a channel.
MAT WRLOCK	Write an array, matrix, or string with locking.

Data Structures

COM	Specify common variables.
DEF STRUCT	Define a structure.
DIM	Allocate space for variables.
END DEF	Define the end of a structure definition.
ERASE	Perform driver-class dependent function(s).
MEMBER	Define a member associated with a specific structure.
FREE	Deallocate (undimension) variables.
LET	Assign values to variables.

Program Flow Statements

CALL BASIC Pgm	Call a BASIC program.
CALL Procedure	Call a procedure.
CHAIN	Transfer control to another program.
CHAIN READ	Read variables from a previous program.
CHAIN READ IF	Read variables from a previous program.
CHAIN WRITE	Write variables to the next program.
END	Terminate a running program.
GOTO	Unconditional branch to statement.
JUMP	Transfer control immediately to another location.
PAUSE	Suspend program execution.
RETURN	Return from previous GOSUB subroutine call.
SPAWN	Launch a background BASIC program.
STOP	Abnormally terminate a program.
SUSPEND	Abnormally terminate a program.
SWAP	Pause and execute another BASIC program.

Blocks and Loops

CASE	Control complex conditional and branching operations.
DO	Establish program loops.
DO UNTIL	Perform a loop as long as the expression is false.

DO WHILE	Perform a loop as long as the expression is true.
ELSE	Control conditional branching.
END IF	End conditional branch. .
END SELECT	End complex conditional branch.
EXIT DO	Exit a DO loop.
EXIT FOR	Exit a FOR/NEXT loop.
FOR	Repeat a group of statements.
IF	Control conditional branching.
LOOP	Mark the end of a group of statements enclosed in a DO loop.
NEXT	Continuation of FOR loop statement.
ON	Conditional branch on value of expression.
WEND	With WHILE, block a set of repeated statements.
WHILE	With WEND, block a set of repeated statements.

Communications

PORT	Attach and control other ports.
RECV	Receive communication message.
SEND	Transmit a message to another port.
SIGNAL	Transmit/receive ported messages and pause.

Windows

MOVE	Move a window.
SIZE	Select the size of a window in columns and rows.
WINDOW CLEAR	Maintain Dynamic Windows.
WINDOW CLOSE	
WINDOW MODIFY	
WINDOW OFF	
WINDOW ON	
WINDOW OPEN	

Formatting Output

EDIT	Format numeric and string expressions.
PRINT	Output ASCII to screen, file, or device.

Miscellaneous Statements

BOX	Draw rectangular figure on display device.
CONV	Convert binary data to decimal, or convert decimal to binary
DATA	Define internal program data.
LINE	(A function of drivers.)
OPTION	Specify a runtime option for current program unit.
OPTION DEFAULT	Specify a runtime option for all program units in the current program.
RANDOM	Seed random generator for RND function.
REM	Make a non-executed program comment.
RESTOR	Reset DATA pointer for READ statement.
SYSTEM	Execute system functions and commands.
TRACE	Enable statement trace debugging.

Appendix F - Unicode Character Set

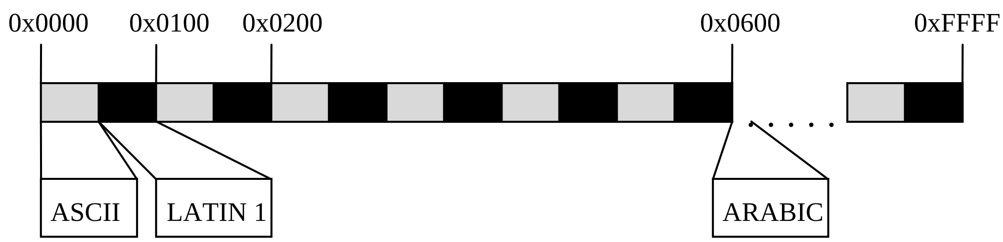
Introduction

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. Unicode values 0x2500 - 0x257F and 0x2580 - 0x27BF, represent forms and charts, and special graphics characters, respectively.

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