



Product Training 5.3

Dynamic Concepts, Inc.

www.dynamic.com

18-B Journey

Aliso Viejo, CA 92656

(949) 215-1200

(800) 822-2742

TRADEMARK NOTICE

Information in this document is subject to change without notice and does not represent a commitment on the part of Dynamic Concepts, Inc. (DCI). Every attempt was made to present this document in a complete and accurate form. DCI shall not be responsible for any damages (including, but not limited to consequential) caused by the use of or reliance upon the product(s) described herein.

The software described in this document is furnished under a license agreement or nondisclosure agreement. The purchaser may use and/or copy the software only in accordance with the terms of the agreement. No part of this manual may be reproduced in any way, shape or form, for any purpose, without the express written consent of DCI.

© Copyright 2004 Dynamic Concepts, Inc. (DCI). All rights reserved.

dL4 is a trademark of Dynamic Concepts, Inc.

UniBasic is a trademark of Dynamic Concepts, Inc.

Dynamic Windows is a trademark of Dynamic Concepts Inc.

BITS is a trademark of Dynamic Concepts, Inc.

IRIS is a trademark of Point 4 Data Corporation.

IMS Basic is a trademark of Information Management Systems, Inc.

c-tree is a trademark of FairCom.

IQ is a trademark of IQ Software Corporation.

UNIX is a registered trademark of UNIX Systems Laboratories.

SYBASE is a registered trademark of Sybase, Inc.

CodeBase is a trademark of Sequiter Software Inc.

Microsoft, MS, MS-DOS, Microsoft Access, and FoxPro are registered trademarks, and ODBC, Windows and Windows NT are trademarks of Microsoft Corporation in the USA and other countries.

MySQL is a trademark of MySQL AB

dL4 Training – Class Outline

1. Introduction to dL4 (Benefits)
2. Overview of Documentation Set
3. Features of Language
 - New Data Types
 - Date
 - Binary (read images, etc.)
 - Structures
 - Arrays of strings and structures
 - Expanded numeric precisions
 - 3A. Structured Programming
 - Blocked Structures
 - New Statements
 - Error Handling
 - 3B. Virtual BASIC Machine
 - 3C. Procedures
 - Callable subprograms
 - Libraries
 - 3D. New & Enhanced Statements
 - OPTION statement
 - CHAIN READ IF
 - 3E. Global Variables
 - 3F. Functions
 - 3G. New Calls
 - 3H. Old Calls
 - 3I. How to
 - Background jobs, phantom ports, etc.
 - SPAWN, PORT, CALL TRXCO
 - Modify Terminal Definition file
4. Debugging
 - Run, Scope & Basic modes
 - Using Debugger
 - Resizing
 - Use of ProgramDump
5. Data file drivers
 - Foxpro Full-ISAM
 - Microsoft SQL
 - MySQL
 - Bridge Driver
6. Other drivers
 - Socket driver
 - Email driver
 - Serial Device driver
7. Windows, GUI & dL4Term
 - GUI Libraries **NEW!**
8. Convert
 - ub2dl4 utility
 - ims2dl4 utility
 - Conversion profile file

OPTION statements
INCLUDE

9. Install & Configure
 - Environment variables
 - dL4Term
 - Terminal Definition Files
 - Printers
 - Pfilter
 - Shared cache
10. Tools Directory
11. New enhancements
12. Protecting your investment with OSN's
13. Development with source control
 - Streamlining development
14. dynamicXport integration to browser
 - File upload and download **NEW!**
15. dynamicPaper **NEW!**
 - Forms overlay
 - Convert to PDF format
 - Archiving

dL4 Product Training

SECTION 1

Benefits

Portable Platform – Unix & Windows
Industry standard structured programming
More modular components & drivers
GUI support
Supports mouse and keyboard input editing
Event driven programming
Database support – Foxpro, Microsoft SQL, MySQL, others
Databases without code changes using bridge drivers
Socket drivers, TCP/IP sockets, send/receive e-mail
Write callable routines in BASIC
Run converted Unibasic or IMS code as is or re-engineer to take advantage of capabilities
Text-based development, edit in vi, notepad, etc.
Unicode character set and other character sets
Includes dL4Term, eliminating need for emulators
Tools to port Unibasic & IMS code to dL4
Integration with HTML, XML, Wireless with DynamicXport
Future Dynamic development focused on dL4

dL4 Product Training

SECTION 2

Documentation

Documentation can be downloaded in PDF format from our web site, www.dl4.com
Below is a list of the currently available documentation.

Language Reference Manual - Statements, Functions, Calls

Command Reference Manual – Compile, Debugger

Files & Devices Manual

GUI Training 5.3 and Windows Class

Installation and Configuration for Windows

Installation and Configuration for Unix

Supporting Documentation

dL4Term Reference Guide

dL4 User CALL & Driver Implementation Guide

dL4 Release notes – latest information & enhancement history

dL4 Training 5.3 – this training document

dL4 Product Training

SECTION 3

Features of Language

Common Syntax Differences:

Listed here are common syntax differences when converting from Unibasic to dL4. These differences are handled by the conversion process. New programs and modifications must use the new syntax.

Variable dimensions & subscripts are performed with brackets [] rather than parenthesis ().

Call statement syntax is CALL NAME(var1, var2) rather than CALL \$NAME,var1,var2
Example CALL TIME(T\$) rather than CALL \$TIME,T\$

Since mnemonics can be more than 2 characters they are space separated.

Example 'CS BU' rather than 'CSBU'

Also, each code may be optionally preceded by a list of one or more numeric constants, separated by commas, to be interpreted as parameters (often a repetition count). The PCHR\$ function provides a means to construct parameters using expressions rather than constants.

Character strings in a DATA statement must be in quotes.

INDEX statements are converted to SEARCH

CREATE statements are converted to BUILD

RESTORE statements are converted to RESTOR (resets DATA statement pointer)

dL4 keyword collisions are corrected by appending an underscore "_"

Functions that return a string value are named with an ending \$ sign. Thus ERM is converted to ERRMSG\$, STR is converted to STR\$, etc. (Note: MSF is converted to MSC\$)

CHN function is converted to identical CHF function

The mod operator % (returns the remainder of a division of the two operands) is converted to the MOD function. For example, A = B % C is converted to A = B MOD C

A DEF FN statement must precede any use of a function. (This is automatically resolved in conversion.)

The Option statement specifies runtime option settings for a program. A sample Option statement would be **Option Arithmetic Iris Decimal**

Option Dialect statements should be used to convert or inherit specific language behaviors.

Differences not automatically resolved by conversion:

Cannot have multiple DEF FN's of the same name in the same program

RND, random number generator creates different results.

TAB on screen displays will properly tab to that position, even if the cursor is currently past that position.

LET TO statement with optional : numeric variable is limited to single character lookup and cannot be combined with LET statement concatenation.

Data Types:

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 six data types are described briefly below.

- ❑ **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.

Table of Numeric Precisions

%	Parameters	Bytes	Decimal Digits	Range of values supported
1	16-bit signed integer	2	4+	± 32768
2	32-bit signed integer	4	9+	$\pm 2,147,483,648$
3	3-word BITS Base 10000 floating ¹	6	9-12 ²	$\pm .9999999999999999 E\pm 63$
4	4-word BITS Base 10000 floating ¹	8	16	$\pm .9999999999999999 E\pm 63$
5	2-word BITS Base 10000 floating ¹	4	6	$\pm .999999 E\pm 63$
6	6-word BITS Base 10000 floating ¹	12	17-20 ²	$\pm .99999999999999999999 E\pm 63$
7	16-bit signed BCD integer	2	4	± 7999
8	2-word IRIS BCD floating	4	6	$\pm .999999 E\pm 63$
9	3-word IRIS BCD floating	6	10	$\pm .9999999999 E\pm 63$
10	4-word IRIS BCD floating	8	14	$\pm .9999999999999999 E\pm 63$
11	5-word IRIS BCD floating	10	18	$\pm .99999999999999999999 E\pm 63$
12	32-bit signed BCD integer	4	8	± 79999999
13	2-word IEEE BCD floating	4	6	$\pm .999999 E\pm 63$
14	3-word IEEE BCD floating	6	10	$\pm .9999999999 E\pm 63$
15	4-word IEEE BCD floating	8	14	$\pm .9999999999999999 E\pm 63$
16	5-word IEEE BCD floating	10	18	$\pm .99999999999999999999 E\pm 63$
17	2-word IEEE floating scaled X 100	4	7 ³	$\approx \pm 99999.99$
18	3-word IEEE floating scaled X 100	6	11 ³	$\approx \pm 999999999.99 E\pm 35$
19	4-word IEEE floating scaled X 100	8	14 ³	$\approx \pm 999999999999.99 E\pm 35$

Programs declare precisions in either the form *%p* or *p%*. 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 *p%* to a real precision is based upon the **Option Arithmetic** declaration within each program.

Training Note Unless specified, the default is **Decimal** (alias **IEEE Decimal**). (Typical for Unibasic conversions.)

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 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.

Precisions of Numeric Variables

The various **Option Arithmetic** statements cause the following mappings to occur:

Option Arithmetic Decimal **Option Arithmetic IEEE Decimal**

Precision n	Translates to	Type description
1%	%7	16-bit signed BCD integer
2%	%13	2-word IEEE BCD floating
3%	%14	3-word IEEE BCD floating
4%	%15	4-word IEEE BCD floating
5%	%16	5-word IEEE BCD floating
6%	N/A	N/A
7%	%12	32-bit signed BCD integer

Option Arithmetic IRIS Decimal

Precision	Translates to	Type description
1%	%7	16-bit signed BCD integer
2%	%8	2-word IRIS BCD floating
3%	%9	3-word IRIS BCD floating
4%	%10	4-word IRIS BCD floating
5%	%11	5-word IRIS BCD floating
6%	N/A	N/A
7%	%12	32-bit signed BCD integer

Option Arithmetic BITS

Precision n	Translates to	Type description
1%	%1	16-bit signed integer
2%	%5	2-word BITS floating
3%	%3	3-word BITS floating
4%	%4	4-word BITS floating
5%	N/A	N/A
6%	%6	6-word BITS floating
7%	%2	32-bit signed integer

Option Arithmetic ICE Binary

Precision n	Translates to	Type description
1%	%1	16-bit signed integer
2%	%17	2-word IEEE floating X 100
3%	%18	3-word IEEE floating X 100
4%	%19	4-word IEEE floating X 100

5%	N/A	N/A
6%	N/A	N/A
7%	%2	32-bit signed integer

Note: During file access, numeric data is converted to/from the type of data contained within the file.

Date Variables

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 *p%* or *%p* specification controlling the currently-selected precision.

Unlike numeric precisions however, there is no mapping from *p%* to *%p* controllable by the Option statement; e.g., 1% always means %1, etc. 1% Precisions default time to Noon GMT.

Training note: Variable names ending with # designate a date variable. There are many intrinsic Functions and CALLs to convert Date variables to viewable formats. Data fields that are not populated will generate an Error 20, Date Overflow, if you attempt to use the field.

Structures

Structures are a pre-defined, fixed grouping of variables defined at compile-time. The items comprising a structure are said to be "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 structname=name { ... }  
Def Struct structname  
    Member name { ... }  
    ...  
End Def
```

structname 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 and the name cannot begin with a digit. **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.

The *name* of a **Member** is any legal variable name, or precision declaration in the form: %p or p%. *name* may be any type of variable, string, numeric, date, binary or another structure. Any given member may 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 may 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 *structname* definition.

The **End Def** statement defines the end of a structure definition.

The following are examples of structure definitions:

```
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

```
DEF STRUCT TEST=Q$ [20] , 1%, R, S
```

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

Dim *variable*. { [expr {, ... }] } **As** *structname*

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 *structname* informs the compiler which compiled structure definition is to be used for *variable*.

Examples of dimensioning structure variables are:

```
DIM A. AS TEST, B.[5] AS TEST      !B is array of 0-5 (6) structures
```

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** *structname2*

name. is the name within *structname* whose members are defined by the structure definition *structname2*. *structname2* 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.

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].Q$ = "quit"
```

"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:

```
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 may 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 and Status variables may be passed within a structure.

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

```
DIM B. [10]
LET B.R = 5
```

! is equivalent to B.[0].R = 5

String Arrays

In a **DIM** statement, if the *var.list* contains a variable in the form *str.var\$[num.expr1,num.expr2]*, it is defining a string array of *num.expr1* elements with a maximum of *num.expr2* characters each. The *num.expr1* within the subscripts is evaluated, truncated to an integer, and used to select the size (number of elements) of the string array. The *num.expr2* within the subscripts is evaluated, truncated to an integer, and used as the maximum size of each string array variable in characters. Any attempt to store data beyond this maximum results in data truncation. String array variables must 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).

Re-dimensioning

Once a variable is allocated, it cannot be changed with one exception: an array variable can be re-dimensioned to a different size, reduced or enlarged, (**number of elements, not length or precision of elements**), 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, null or invalid dates.

Alternatively, a variable can be freed with the **FREE** statement (value is lost), and then re-dimensioned (re-used).

Training Note on Dimensioning Structures :

A structure must be defined in a program using the **DEF STRUCT** statement prior to dimensioning a structure variable. Structure Definitions must appear in the program code before a dimension using the structure variable.

Training Note :

An Option statement, **OPTION STRING REDIM IS LEGAL**, allows strings to be re-dimensioned. This is available for backwards compatibility to other languages and is not a recommended programming method.

Assignment Operator: Colon Equal

The assignment operator, Colon Equal is different from "=" which is compare-for-equality.

Compare-for-equality indicates that dL4 is attempting to determine if the values are equal.

":=" is an assignment operator while "=" is a relational operator except for the initial "=" in a **LET** statement.

Typically useful in **IF** statements, for example :

```
IF (X$ := function())      !X$ receives value from function, then Boolean
true or false if, null is false
  THEN
END IF
```

```
LET A:=B:=C:=1           !Results in A=1 B=1 C=1
```

dL4 Product Training

SECTION 3A

Statements for Structured Programming

A number of new statements afford the ability to structure applications into blocks. Block structuring provides the following benefits over conventional programming:

- ❑ Simplifies readability of application programs
- ❑ Reduces support
- ❑ Speeds development and customization

Throughout this document, the term *expr* and *expression* are used to indicate the evaluation of any legal expression. Expressions are made up of operators, operands and relations, such as >, <, etc.

When evaluating a boolean relation, i.e. $A > B$, within or as an expression, one is returned if the relation is true, and zero if false.

The notion of so-called *structured programming* has become almost synonymous with “**GOTO** elimination”.

Programs can be written in terms of only three control structures, namely the *sequence structure*, the *selection structure* and the *repetition structure*.

The sequence structure is built into dL4 without line numbers. Unless directed otherwise, the computer executes dL4 statements one after the other in the order in which they are written.

dL4 provides **three types of selection structures**.

The IF statement and IF/END IF provides a single-selection structure.

The IF/ELSE IF/END IF provides a double-selection structure.

The SELECT CASE/CASE/END SELECT provides a multiple-selection structure.

dL4 provides **three types of repetition structures**, namely the DO LOOP, WHILE WEND and FOR NEXT structures.

A WHILE is identical in behavior to DO WHILE.

Blocked IF - Else IF Statements

In addition to the familiar blocked IF structure supported in IRIS and UniBasic, an **ELSE IF** clause has been added. The general form of an **IF** block is:

```
If expression
! Perform these statements if the expression is true
Else If expression           !If first expression was false
! Perform these statements if this expression is true
Else If expression           !Second expression was false also
! Perform these statements if this expression is true
Else
! Perform these statements only if all other expressions are false
Endif
```

The first *expression* is evaluated and if true, execution resumes at the next line. No additional statements may follow the *expression* in a blocked **If**. An *expression* is considered true if it evaluates to non-zero.

If the *expression* is false, any next corresponding **Else If** relation is performed. If true, execution resumes at the next line. When corresponding **Else If** blocks are not

included, or should each evaluate false, execution resumes at any corresponding **Else**. If the block does not contain an **Else**, execution resumes following the associated **End If**.

Do, Do While, Do Until & Loop Statements

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 general form of a **Do** loop is as follows:

```
Do { While | Until expression }
    stmts
Loop { While | Until expression }
```

While or **Until** *expression* provides the loop with a specific termination condition. **While** provides for looping as long as the *expression* remains true, whereas **Until** provides for looping as long as the *expression* remains false - that is until it becomes true.

The optional **While** or **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.

stmts are any valid dL4 program statements. 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.

```
Goto Label      !not recommended, but you can branch into a Do Loop
Do Until Value > 100
    Print Value;
    Label: Value = Value + 1
Loop
```

The following example sets up an infinite loop which runs until explicitly exited by an **Exit Do** statement.

```
Do
    statements
    If X Exit Do
Loop
```

Other variations of **Do** loop syntaxes :

```
Do While expr                                ! As long as the expression is true
    stmts! Execute the statements
Loop

Do Until expr                                ! As long as the expression is false
    stmts! Execute the statements
Loop
```

```

Do
    stmts! Execute the statements
Loop While expr                ! As long as the expression is true
Do
    stmts! Execute the statements
Loop Until expr                ! As long as the expression is false

```

While & Wend Statements

Program loops may be established using the **While** and **Wend** statements as a means of blocking a set of repeated statements. These statements provide additional flexibility and looping control beyond the simple **For / Next**. The general form of a **While** loop is as follows:

```

While expr
    stmts
Wend

```

expression is any legal expression which terminates the loop when true .

While provides for looping as long as the *expression* remains true. The *expression* is tested prior to performing each loop. The loop is terminated once the *expression* is false.

Training Note: **While** is identical in behavior to **Do While ... Loop**, but note that there is not an equivalent to the **Exit Do** statement for a **While** loop and you cannot compare on the **Wend** statement.

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

```

Exit Do Statement

The **Exit Do** statement gracefully exits a **Do** loop. It may be used within a with **Do Loop** block . The general form of the **Exit** statement is:

Exit Do

The **Do** loop currently being executed is gracefully terminated. **Exit Do** is the preferable method to terminate a **Do** loop when writing portable code. Branching out of a loop is never recommended.

```

Do
    stmts
If ...
    ...
    If T < 100 Exit Do

```

```
End If
Loop Until expr
```

Exit For Statement

The **Exit For** statement gracefully exits a **For** loop. It may be used within a with **For Next** block . The general form of the **Exit For** statement is:

Exit For

The **For** loop currently being executed is gracefully terminated. **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.

```
For I = 1 to 1000
    stmts
    If ... Exit For
Next I
```

Select Case & End Select Statements

The **Select Case** statement organizes blocks of statements which are dependent upon the value of a single expression. The general forms of **Select Case** are:

Select Case *expression*

```
Case expr | expr TO expr | Is rel-op expr { , ... }
    stmts
```

```
Case Else
    stmts
```

End Select

expression is the primary expression which is evaluated for subsequent selection within the entire block.

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 *expr* which is compared for equality, an inclusive range of values specified in the form *expr* **TO** *expr*, or a value which results in a true relation, such as **Is > 50**. Multiple conditions, separated by comma may be specified and are treated as OR conditions.

stmts are those statements which are to be executed for the selected condition. Once a Case is true and it's block of statements are executed, further Case statements in the Select block are NOT executed.

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

Example 1:

```
Search #Ch,4,1;K$,R,E
Select Case E
    Case 0
        ! Successful key insertion
```

```

Case 1
  If R <> RecNum Error 236          ! Duplicate key in unique index
Case Else
  Stop "Directory 1 key insertion failed"
End Select

```

Example 2:

```

Select Case E$
Case "0" TO "9"          !String to compare to
                          !0 through 9 inclusive
  Print "Numeric"
Case "A" TO "Z"          !A through Z inclusive
  Print "Capital Letter"
End Select

```

Example 3:

```

Select Case E$
Case Is = "0", Is = "1"    !String to compare to
                          !0 or 1   TREATED AS OR
  Print "0 or 1"
Case "A" TO "Z", Is = " "  !A through Z inclusive or is space
  Print "Capital Letter or space"
End Select

```

Try, End Try & Retry Statements

Try provides for the temporarily redirection of error branching within a block. The general form of the statements are:

Try *statement1* **Else** *statement2*

Try

```

    stmnts to be executed
[[ Else If expr
    stmnts
{ ... }

```

Else

stmnts to be executed

End Try

The **Retry** statement can be used within a **Try** block. The syntax for the **Retry** statement is:

Retry

In the first general form, *statement1* is executed. If any error is detected, *statement2* is executed. Otherwise, execution resumes on the next line.

The second general form provides for the attempted execution of multiple statements. If any of the statements cause an error, execution resumes following the associated **Else**. Nested **Try** blocks are permitted, as well as the optional **Else If** block.

Retry may be used within the **Else** block(s) to repeat the last **Try** block.

If any program error branching is in effect, it is temporarily suspended for the duration of the **Try** statement or block. Error branching is restored at the upon the completion of the line or block.

For example:

```

Try
  Open #0,"Filename"          ! Try to open the file
Else if Spc(8) = 42          ! Not found
  Build #0,+"Filename!"

```



```

    Close #0
    Retry                                ! File not found
Else
    Call Error_Function(Spc(8))         ! Give up
End Try

```

Training note: The Try block relinquishes error trapping to outside the block once an error occurs, thus errors within the Else blocks will be trapped by error trapping outside the Try block, ie another Try block or IF ERR.

Training note: If a Try block traps an error, the error will not be seen by a surrounding IF ERR trap, however SPC(8) will still contain the error that occurred.

Error Statement

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. The general form of the statement is:

Error *expr*

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

Should the developer wish to create his own error numbers, use values $\geq 10,000$.

Training note: A developer can use the Error statement to exit a **Try** block of code by generating a 'false' error.

dL4 Product Training

SECTION 3B

VBM Virtual BASIC Machine

VBM VIRTUAL BASIC MACHINE

This discussion helps in understanding:

- ❑ A variable's lifetime, which is defined to be the duration a variable exists during the execution of a program
- ❑ A variable's "visibility" or "scope", which determines which portions of a program can reference a particular variable
- ❑ Prerequisite for understanding dL4 procedures
- ❑ Program libraries

VBM Fundamentals

- ❑ A dL4 BASIC program is made up of one or more program units
- ❑ A program unit is either an external procedure or the main program
- ❑ A program unit consists of one or more program blocks
- ❑ A program block is a procedure, either an internal or an external procedure
- ❑ A program must always contain one main program unit and any number of secondary program units
- ❑ A dL4 BASIC program runs in a Virtual BASIC Machine (VBM)

Program Unit

- ❑ A program unit is either an external procedure or the main program
- ❑ Each running *program unit* is one or more program-blocks and has it's own:
 - Option settings
 - Variables
 - Argument variables
 - TRY stack
 - FOR stack
 - GOSUB stack
 - Current DATA statement position
 - Current numeric precision
 - Current date precision
 - Current LIB setting
 - Current values for:

Last error number: SPC(8)
Line number of last error: SPC(10)
Stmt number on line of last error: ERR(5)
Last END or STOP: SPC(24)
Last determinant: DET(0)
Last input element: MSC(1)
Last input size: SPC(17)
Input pend mode: SYSTEM 26/27

Main Program Unit

- ❑ A main program unit contains statements other than those blocked within named procedure(s)
- ❑ Each running program is one or more program-units and has it's own:
 - Structure definitions
 - Saved disk image (program)
- ❑ The main program unit is considered empty if it does not contain any executable statement
- ❑ The main program unit or secondary program units (External procedures) may contain any number of program blocks (internal procedures).
- ❑ When a program is initiated by Chain or RUN, the main program unit of the selected program is executed.

Program Block:

- ❑ A program block is a procedure, either an internal or an external procedure
- ❑ A program block is one or more lines of BASIC code
- ❑ Each running program block is one or more lines and has it's own:
 - Argument variables
 - Current program position
 - Current TRY stack position
 - Current FOR stack position
 - Current GOSUB stack position
- ❑ Program branching between blocks is not permitted
- ❑ Nesting program blocks is not permitted

Linked Program

- ❑ Each running linked program is one or more programs and has it's own Table of external procedure names

VBM

- ❑ Each running BASIC machine contains one or more linked programs and has it's own:
 - Open channels
 - Standard input channel
 - Standard output channel
 - Trace channel
 - Command line string: INPUT (0,K)
 - Hot-key program
 - Dynamic Window stack
 - Single-step count

Pseudo Program Structure Example

dL4 BASIC program

Program Unit A (ie External Procedure)

Program Block 1 (ie Internal Procedure of Unit A)

Program Block 2 (ie main block of Unit A)

Program Unit B (ie another External Procedure)

Program Block 1 (ie main block of Unit B)

Program Unit C (ie main program unit)

Program Block 1 (ie Internal Procedure of Unit C)

Program Block 2 (ie main block of Unit C)

Typical Program Structure Example

! Comments about program
Option Default statements if needed (Option Default applied to entire program
file

!Define file structures used in program
Def Struct customer
 Member name\$(10) : Item 10
End Def
!etc

!Use a Lib statement to specify directory to find programs if called by filename,
or
!external libraries OR define a LIBSTRING environment variable.
!Is the **directory** definition only!

!Define external library sources if necessary (define filename)
!directories to search for this filename is determined by LIBSTRING or an
overriding
!Lib statement
External Lib "**filename** of library with procedures"

!Declare intrinsic subs & functions used
Declare Intrinsic Sub ProgramDump
Declare Intrinsic Function FindChannel

!Declare external subs & functions used, if resides in external file or is used
before its
!definition
!preferred method is to have them defined before use
!can also be declared at the beginning of the block using it
Declare External Sub *myexternal*

!Declare internal subs & functions used, if used before its definition
!preferred method is to have them defined before use
!can also be declared at the beginning of the block using it
Declare Sub *myinternal*

!first program block
External Sub A(*receiving variables*)
 Option statements that just apply to this block can be stated

 !internal procedure B used in Sub A, like a gosub
 Sub B()
 If ----- Exit Sub
 End Sub !B

 Call B()
End Sub !A recommend putting subroutine name in a comment on End

!second program block

```

External Function C()
    If ----- Exit Function value
End Function value !C

!main program block
Sub D()           !an internal sub within the main block
    Print "Hello"
End Sub !D

Call A()
X = C()
Call D()

End

```

Training Note :

You don't need to declare internal subs & functions or external subs & functions in the same physical file if they appear before they are used. Thus it is common to have procedures appear in the code prior to their use to eliminate the need to Declare them.

dL4
Product Training
Section 3C
Procedures

PROCEDURES

Procedures facilitate the grouping of program statements into organized, re-usable program blocks. The resulting modularity simplifies development and support for the programmer.

Modularity provides the developer with the following benefits:

- Enhances readability of applications software.
- Each program contains only the code for its required functions and does not contain duplicative subroutine code.
- Provides for the organization of procedures into libraries.
- Simplifies customization and support of applications.
- Replaces dependency on **GOTO** and **GOSUB**.

Unlike saved program files under IRIS or BITS which contain only a single program, dL4 *programs* are made up of one or more *program units* each of which contain(s) one or more *program blocks*. A program block is one or more lines of BASIC code, and a program unit is one or more program blocks.

A program is further described herein to contain a *main program unit* when it contains statements other than those blocked within named procedure(s). A program always contains a *main program unit* consisting of all executable statements other than those embedded within procedure(s). If a program contains no such executable statements, the main program unit is considered empty, containing only an implied **End** statement. A main program unit has no actual name.

IRIS programs have one program unit (the main one) consisting of a single program block.

When a program is initiated by **Chain** or **RUN**, the main program unit of the selected program is executed.

Each running *program block* is one or more lines and has its own:

- Argument variables
- Current program position
- Current TRY stack position
- Current FOR stack position
- Current GOSUB stack position

Each running *program unit* is one or more program-blocks and has its own:

- Option settings
- Variables
- Argument variables
- TRY stack
- FOR stack
- GOSUB stack
- Current DATA statement position
- Current numeric precision
- Current date precision

Current LIB setting

Current values for:

Last error number: SPC(8)
Line number of last error: SPC(10)
Stmt number on line of last error: ERR(5)
Last END or STOP: SPC(24)
Last determinant: DET(0)
Last input element: MSC(1)
Last input size: SPC(17)
Input pend mode: SYSTEM 26/27

Each running *program* is one or more program-units and has it's own:

Structure definitions
Saved disk image (program)

Each running *linked program* is one or more programs and has it's own:

Table of external procedure names

Each running *BASIC machine* contains one or more linked programs and has it's own:

Open channels
Standard input channel
Standard output channel
Trace channel
Command line string: INPUT (0,K)
Hot-key program
Dynamic Window stack
Single-step count

Types of Procedures

There are several types of procedures supported by the language. From simple one-line functions, to complex external library routines, procedures simplify programming tasks.

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.

(a)	! Pass 'a' by reference
(a+0)	! Pass an expression equal to 'a'
((a#))	! Force 'a#' into expression with parentheses
(a\$ + "")	! Force a\$ into expression

Single Line Functions

Single line functions are an extension of the familiar **DEF FN(x)** functions and permit the developer to define internal procedures, which return data of any type when passed one or more arguments:

Def function-name (parameter1 {, ...})

A function 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 ?. For example :

```
Dim a$[10],b$[10],c$[10]
b$="123"
c$="456"
Def test$(x$,y$) = x$ + y$
a$=test$(b$,c$)
Print a$                                !results in 123456
```

Structures may be passed and operated upon, but a function cannot return a structure.

Parameter is any numeric, string, date, binary or structure variables (or expressions) to be passed to the function.

As with the familiar **DEF FNA(X) = X * 2**, the caller supplies an expression or variable to the function which itself is referenced by the function for each occurrence of X. Passing Y to the function returns the current value of Y being multiplied by two. Older functions of this type were only permitted to perform calculations on existing variables and the variable passed by reference. No variable assignment was permitted within the function itself.

Unlike the older style single line **DEF FN**, single line **Def** function have the ability to alter existing variables as well as those passed by reference.

Single-line functions are still restricted to expressions (i.e. they cannot have statements), however the new := operator permits the assignment of a value within an expression.

Note: Single line functions are compiled as internal procedures and therefore cannot be redefined at runtime.

```
! This function is passed two numeric expressions. The first is
! multiplied by the second after the second is incremented by 2.
```

```
Def test(y,z) = y * (z := z + 2)
a = 5;b = 2
c = test(a,b)
Print a;b;c 5 4 20
```

Note: Single line functions are unique program blocks and operate identically with internal procedures, which share everything with their surrounding program unit.

The following example computes the difference between two date variables returning the number of days. This type of function might be used in an aging routine.

```
! Simple function which is passed two dates and returns the
! difference between them, returning the result in days
! Date arithmetic returns seconds; there are 86400 seconds / day

Def days(a#,b#) = (b# - a#) / 86400 !Define the procedure

! To call this procedure, define two date variables:
Dim l%, first#, second#
first# = "Jan 1, 2002"
second# = "Jan 15, 2002"

! Subtracting the dates returns seconds:
Print second# - first#          1209600

! Using our function returns the difference in days:
Print days(first#,second#)      14
```

Training Note: Why is the function name days as opposed to days#?
It is because the result is a number not a date.

The following example computes the difference between two date elements in a structure returning the number of days. When passing a structure to a procedure, the declaration of the procedure must include a definition of the structure being passed.

```
! Simple function which is passed two dates and returns the
! difference between them, returning the result in days
! Date arithmetic returns seconds; there are 86400 seconds / day

Def Struct dates                      ! Assuming we have defined this
structure
  Member l%,first#
  Member l%,second#
End Def

Def days(pass. As dates) = (pass.second# - pass.first#) / 86400

! To call this procedure, setup the dates in a structure:
Dim test. As dates
test.first# = "Jan 1, 2002"
test.second# = "Jan 15, 2002"

! Subtracting the dates returns seconds:
Print test.second# - test.first# 1209600

! Using our function returns the difference in days:
Print days(test.)                14
```

Training Note: DEF statements

Multiple function definitions (DEF statement) with the same function name in the same program are not acceptable in dL4. Unibasic processes DEF statements at execution so the same function name could be reused in a program. dL4 processes DEF statements at

compile time, so it cannot have two functions defined with the same name in a program. Resolution: recode so all functions are uniquely named.

Multi-line Procedures

Multi-line procedures include the familiar **Call "subprogram"** procedures and permit the developer to define procedures, which perform a wider range of operations. There are two types of multi-line procedures available to the developer, SUB and FUNCTION:

- Subprograms that may operate upon and return data through optionally passed arguments.
- Functions that may operate upon and return data through optionally passed arguments and return a value to the caller.

Procedures may be declared **External** making them independent program units within a program and visible to other program units both inside and outside of the program. **External procedures share nothing with the surrounding program, except channels.** Regardless of their physical location, they have their own set of variables, Lib directory, DATA statements, current precision, stacks, OPTIONS, random number seed, etc.

An External procedure is similar to a separate program in dL4 or IRIS. However, an External procedure is a *secondary program unit* that is only executable when called. Attempting to **Chain** or **RUN** a program that contains only External procedure(s) will result in no operation.

Procedures that are **NOT declared External** are internal program blocks within the surrounding program unit and DO share variables,etc. Internal procedures are not visible outside the program unit that contains them. However, they are considered a separate program block and are only executed when called. In addition, they share everything with any surrounding program unit. In some ways, internal procedures are similar to a block-structured GOSUB.

Any given program may include zero or one main program units and any number of secondary program units. Any given program unit may include any number of program blocks, however nesting program blocks is not permitted. Therefore, the main program unit or secondary program units (External procedures) may contain any number of program blocks (internal procedures), however a program block (internal procedure) may not contain another program block (internal procedure) or secondary program unit (external procedure).

Correct nesting of internal and external procedures:

```
External Sub doit()           !This is a secondary program unit
  Sub dosomething()          !This is a block within this unit
    Print "something"
  End Sub
End Sub

Call doit()                  !This is the main prog unit (1 stmt)
```

Incorrect nesting of internal and external procedures:

```
External Sub doit()           !This is a secondary program unit
  Sub dosomething()          !This is a block within this unit
    Print "something"
```

```

    Sub dosomething2()                !Nesting another block is illegal
        Print "something2"
    End Sub
End Sub

Call doit()                        !This is the main prog unit (1 stmt)

```

Call by program name (old method) :

For backward compatibility purposes, the execution of a Unibasic syntax statement **Call "subprogram"** calls the main program unit of the selected subprogram. The **Enter** statement is used to accept all passed arguments.

Note: Any given program may have only one main program unit. A main program unit may be **RUN** or called as a subprogram by filename.

A group of External procedures may be saved in a single program, called a library file. If a program has an empty main program unit, attempting execution via **Chain** or **RUN** results in no operation.

Training Note: It is good practice to name a library file with a .lib extension.

A program that has both an executable main program unit as well as External procedures may also be referenced as a library by other programs. However, it is advisable to segregate shared External procedures into library files that 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 procedure that is called by *filename* (i.e. Call "program") and therefore exists as a main program unit of the library file.

Note: Program branching between blocks is not permitted. Variables are passed to a procedure by reference. A procedure may alter those variables in the callers program as a method of returning data. Procedures that are called by filename are physically external. That is, they are stored in another program as the main program unit of that program. They only share channels and any variables passed by reference with the caller.

Passing variables by reference to a procedure is not the same as passing common variables via **Com** or **Chain Write**. Common variables (**Com**) are passed by name to a program and are literally copied into the variable space of the new program. When that program operates upon the variables, it does so with its own copy. To return or pass those variables back to a caller requires an elaborate use of **Chain** and **Jump**.

When variables are passed by reference to a procedure, that procedure actually points its referenced variables to the caller's supplied variable data space. Any changes to the variable are affected in the caller's program.

For example:

```

! Perform a summation on three numeric arguments.
! Update the first argument to be twice the result.
! If the result is > 1000, set d to 1 otherwise 0
Sub compute(a,b,c,d)
    a = (a + b + c) * 2

```

```

        if a > 1000 let d=1 else let d=0
End Sub

! Call the function with three parameters
Call compute(r,200,g,k)           ! r will be result, k the return flag

! Call the function ignoring the flag
Call compute(g,r*50,b,0)

```

The above example is better written as a **Function** returning the value. In that way, selective ignoring of a return value is readily apparent:

```

! Perform a summation on three numeric arguments.
! Update the first argument to be twice the result.
! If the result is > 1000, return 1 otherwise 0
Function compute(a,b,c)
    a = (a + b + c) * 2
    if a > 1000 Exit Function 1 else Exit Function 0
End Function

```

Multi-line Procedures Stored in Program Files (Unibasic style call by program name)

For compatibility with dL4 and IRIS, subprogram procedures may be saved as separate programs. The main program unit of any program may be called as a procedure.

The following statements are used in conjunction with subprogram procedures stored as programs:

Lib	Specify a directory to search when calling named programs as subprograms.
Call "program",	Invoke a subprogram procedure by program name passing a list of parameters.
Enter	Accept arguments into a subprogram.
End	Terminate a subprogram program and return to the caller.

The **Lib** statement is used to specify an alternate directory to be searched when procedures are called by filename. It is also used by **Chain** and **Swap**. A program unit may contain any number of **Lib** specifications. Only the most recent is in effect.

Lib "dirname"

dirname selects a specific directory which is to be searched when calling a procedure by name. *dirname* is also used when transferring control to another program using **Chain**, **Swap** and **Spawn**. Only one *dirname* may be specified.

Note: The following search mechanism is used to locate programs when called by **Call**, **Chain**, **Swap** and **Spawn**: (If the LUMAP environment variable is set, LUMAP is applied during the initial translation of the program filename and, if mapped, the filename becomes an absolute path which doesn't use any of the search directories.)

1. If the program unit has specified a **Lib** *dirname* , it is searched first. (If the LIBSTRING environment variable is set, LIBSTRING is used unless overridden by a **Lib** statement.)
2. The directory where the calling program resides is searched.
3. The users current working directory is searched.

When calling the main program unit as a multi-line procedure by filename, use the **Lib** statement to specify the directory where the procedure is located. The **Lib** declaration may be changed within a program unit as necessary to call various named subroutines.

When calling a named program as a procedure (**old method**), the calling program includes a statement in the form:

Call "program" {, *parameter1* {, ...}}

program is the name of a saved program. When called, the main program unit of the *program* is executed.

Optional *parameters* may be passed to the procedure. The *parameters* may be any type of data, including a *structure*.

A procedure called by filename looks like any other program with the following exceptions:

1. It usually contains an **Enter** statement to map variables and data passed by reference.
2. It terminates with an **End** statement.

The **Enter** statement maps variables and expressions passed by reference to names used within a procedure. The subprogram accepts and returns the values of each variable. The general form of the **Enter** statement is:

Enter *variable* {, ... }

Normally, all required arguments are specified within a single **Enter**. An error results when a variable's type does not match the type of parameter passed by the caller, or the caller supplied more or less than the specified number of parameters.

Note: The '...' enclosed within braces is used to indicate that an optional list of parameters may be entered. This syntax should not be confused with the '...' parameter itself which permits **Enter** to selectively accept parameters. Refer to Variable Type and Number of Arguments later in this section for more information.

When passing a structure, the procedure must also include its own structure definition of an identical structure and supply the structures designation.

In the following example, the caller passes a structure, date variable, string variable and numeric array using the **Call** statement. The procedure maps the references to its own set of local names using **Enter**.

```
! Within the calling program
Call "testdata", T., v#, s$, X[ ]
! The procedure: Main program unit within "testdata" program
Def Struct Dates
    Member 1%, date1#
    Member 1%, date2#
End def

Enter this. as Dates, date3#, string$, array[ ]
Dim ... !Dimension its own local variables
!T. is passed to this. as a structure
```

In the above example, *this* is a 'Dates' structure and references the callers *T.*, *date3* references *v#*, *string\$* references *s\$* and *array* references *X*. Any changes to any of the referenced variables will affect the caller's program.

A procedure which is called by *filename* causes the main program unit of that *filename* to be executed. The program may also contain other program units, for example:

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

!-----
! Main program
Enter Count
For J = 1 to Count
    If IsPrime(J) Print J;" is a prime number"
Next J
End
```

Multi-line Procedures Stored as Program Units and Blocks

When creating multi-line procedures, the developer decides and declares whether any given procedure is internal or external. **Internal procedures (much like GOSUBs)** are typically used when:

- ❑ The procedure is to operate on any variables passed by reference as well as variables or data

elsewhere within the program unit.

- ❑ The procedure may change or set parameters which are to affect the surrounding program unit,
- ❑ The procedure desires to share a program unit's current state.
- ❑ No other program will be calling the procedure.

Conversely, the developer declares a procedure External whenever:

- ❑ The procedure 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 procedure set its own parameters independent of the caller.
- ❑ The procedure shares nothing with the caller, except channels.
- ❑ Other programs need to call the procedure.

Multi-line procedures make use of the following statements for their declaration, linkage and calling.

Sub <i>name</i> (...) (internal)	Define a named subroutine program block
Function <i>name</i> (...)	Define a named function program block
External Sub <i>name</i> (...)	Define an independent subroutine program unit
External Function <i>name</i> (...)	Define an independent function program unit
Declare {External}. . .	Identify a function/subroutine block or program unit which appears in another program, or later in the current program.
Enter ...	Accept additional arguments into a procedure.
External Lib "file"	Declare named library file(s) in a list
End Sub	End a named subroutine program block.
Exit Sub	Exit a named subroutine program block.
End Function <i>value</i> <i>value</i> .	End a named function program block returning <i>value</i> .
Exit Function <i>value</i> <i>value</i> .	Exit a named function program block returning <i>value</i> .

Sub (Internal implied) declares a subprogram which operates as a separate program block within a program unit. A **Sub** operates upon, and returns values through, supplied parameters passed by reference.

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 procedure *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 procedure. Numeric data has no suffix, strings end with \$, dates with # and

binary variables end with ?. Structures may be passed and operated upon, but a procedure cannot return a structure.

Whenever a procedure 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.

Declare { External | Intrinsic } Sub | Function *name* {, ...}

External identifies the procedure as a separate secondary program unit which shares nothing with its surrounding program and any main program unit.

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. (Typically parameters are not passed to an internal sub since they are already shared.)

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 are opened and the required procedures are dynamically linked into the calling program.

External Lib *filename* {, ...}

filename is the name of a program which is to be opened during the dynamic linking phase when the current program is first executed. **External Lib** declarations may be placed anywhere within a program, and they affect the entire program.

Whenever a program is loaded, via **Chain**, **RUN**, **Call "filename"** or **Swap**, all references to **External** procedures must be resolved prior to execution. 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.

An error is generated if any **External** procedure references are unresolved.

In the following procedure, written as a function returning a value, the caller passes a string parameter and two numeric parameters. The purpose of the function is to prompt the user for input, using the supplied string parameter, and permit the input of a numeric value. The value must be between the two supplied numeric expressions. A numeric value is returned when the input is successful. The function will re-prompt the user if the supplied input is not in range.

The procedure is shown in both internal and External forms.

```
Function InputNum(Prompt$,Min,Max)
  Dim 4%,Z                               ! This may be a problem  !!
  Do
    Print 'CR';Prompt$;
    Input Z
```

```

        If Not (Fra(Z)) And Z >= Min And Z <= Max Exit Do
        Print " VALUE OUT OF RANGE!";
    Loop
End Function Z                                !Return numeric value Z

```

```

External Function InputNum(Prompt$,Min,Max)
    Dim 4%,Z

    Do
        Print 'CR';Prompt$;
        Input Z
        If Not (Fra(Z)) And Z >= Min And Z <= Max Exit Do
        Print " VALUE OUT OF RANGE!";
    Loop
End Function Z

```

This function is designed to be declared External. Omitting this declaration has the following {negative} effects on the caller:

1. DIM 4% specifies a new default precision for the calling program unit each time this function is called.
2. Variable Z is a temporary variable used by this procedure. If the calling program unit has a Z, this procedure will overwrite it's value.

The following example of the BUILDXF program illustrates the use of this procedure and a program with two program units.

```

!  "BUILDXF" == PROGRAM TO CREATE AN INDEXED FILE
!  K = KEY LENGTH
!  L = DATA RECORD LENGTH
!  R = NUMBER OF INDEXED DATA RECORDS
!  X = NUMBER OF DATA RECORDS
!  D = NUMBER OF DIRECTORIES

```

```

External Function InputNum(Prompt$,Min,Max)
    Dim 4%,Z
    Do
        Print 'CR';Prompt$;
        Input Z
        If Not (Fra(Z)) And Z >= Min And Z <= Max Exit Do
        Print " VALUE OUT OF RANGE!";
    Loop
End Function Z

```

```

Dim N$(128)
Dim %1,E,I,D,K(62)
Dim %2,X,L,R,R2

```

```

Print 'CR';"PROGRAM TO CREATE AN INDEXED DATA FILE"
Do
    Input 'CR';"DESIRED FILENAME? ";N$
    If N$ = "" Exit Do
    X = InputNum("NUMBER OF DATA RECORDS",0,2^31)

    If X
        L = InputNum("DATA RECORD LENGTH (#WORDS)",1,2^30)
    Else
        L = 256
    End If

    R = InputNum("NUMBER OF INDEXED RECORDS",0,2^31)
    D = InputNum("NUMBER OF DIRECTORIES",1,62)
    Print 'CR CR';"ENTER KEY LENGTH (#WORDS) FOR EACH DIRECTORY:";

```

```

For I = 1 To D
    K[I] = InputNum("#" + Str$(I),1,61)
Next I

Print 'CR CR';"PLEASE WAIT . . ."

! CREATE FILENAME STRING AND BUILD THE FILE
Build #0,"["+Str$(X)+":"+Str$(L)+"] "+N$

! SET KEY LENGTH FOR EACH DIRECTORY
For I = 1 To D
    Search #0,0,I;N$,K[I],E
    If E Stop 'CR' + "ERROR TYPE " + Str$(E) + " IN DIRECTORY " + Str$(I)
Next I

! STRUCTURE THE DIRECTORIES
Search #0,0,0;N$,R,E
If E Stop 'CR' + "ERROR TYPE" + Str$(E) + "WHILE STRUCTURING DIRECTORIES"

! READ FIRST REAL DATA RECORD NUMBER
Search #0,1,0;N$,R2,E
If E Stop 'CR' + "ERROR TYPE" + Str$(E) + "WHEN READING FIRST REAL RECORD "

Print 'CR';"FILE HAS";X;"DATA RECORDS";'CR'
Print 'CR';"FILE STRUCTURE COMPLETED"
Close #0
Loop
Chain ""

```

This program example places its External function at the start of the program, before the main program unit. In this way, it is defined prior to its first use.

Variable Type & Number of Arguments Passed to Procedures

Procedures 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 procedure to process each of the arguments passed by a caller.

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

Function *name* (...)

Sub *name* (...)

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

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

Function *name* (*parameter1*, *parameter2*\$, ...)

Procedures of this type utilize the **Enter** statement to accept optional parameters.

Subprogram procedures called by *filename* may also accept a variable list of parameters. Unlike functions and subs, the compiler performs no type or parameter checking for procedures called as programs. Checking is only performed during the runtime processing of any **Enter** statement within the called procedure. When calling a procedure of this type, it is the sole responsibility of the procedure to check the passed parameters.

Call "*filename*" {*parameter1* , { ... } }

In any case, a caller's list of arguments is placed into a list to be processed by the actual 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.

The **Enter** statement must end with ... if additional parameters might follow. This preserves any remaining arguments in the list passed by the caller. Do not terminate the **Enter** statement with ... if the procedure is certain that additional parameters are not in the list, or that an error should result if there are additional parameters.

The following example illustrates the use of a variable number of parameters.

```
! Name:
!   VerifyDate() - Verify date inputs
!
! Synopsis:
!   Call VerifyDate(D${ ,R${ ,S}{ })
!
! History:
!   CALL 24 on BITS systems and dL4.
!   CALL 64 on some IRIS systems.

External Sub VerifyDate(D$,...)           !... is part of syntax
  Option Date Format Native
  Dim 2%,D#,N
  Dim %1,NoStatVar

  Try Enter R$,... Else Dim R${6]
  Dim %1
  Try Enter S Else S = 0;NoStatVar = 1
  Try
    Let D# = D$
    N = (Year(D#) Mod 100) * 10000 + Month(D#) * 100 + Monthday(D#)
    R$ = N Using "&&&&&&"
  Else
    S = 1
  End Try
  If S And NoStatVar Error 38
End Sub
```

General Notes on Procedures

Single line procedures, declared by **Def** behave as separate program blocks. For example, the following two program blocks behave identically:

```

Def XYZ(A,B)=Sin(A) + Cos(B)

Function XYZ(A,B)
End Function Sin(A) + Cos(B)

```

When a caller invokes a procedure, other than **Call "name"**, 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 - nothing more. The procedure has no method of identifying whether an expression or a reference to a string variable was passed.

When a procedure is defined including the ... argument option, the procedure itself assumes all responsibility for the type and size of each passed argument.

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. However, if a procedure assumes that it was passed a variable and attempts to modify specific subscripts, an error will be generated to the caller if the argument is not of sufficient size.

The developer can elect to **pass BASIC errors encountered in a procedure back** to the caller (by not explicitly handling them locally), alter the actual error returned (**Error** statement) or change the behavior of the procedure.

```

External Sub Check(S$)
  Dim %2,I

  For I = 1 To Len(S$)
    Select Case S$[I,I]
      Case "0" To "9"
        Rem These are valid characters
      Case Else
        Error 38 ! Force an error to the caller
    End Select
  Next I
End Sub

```

The developer can explicitly handle errors in routines and pass them back to the caller as such :

```

10 Rem a way to trap and report all errors in subs
20 Declare Sub a,b
30 Dim e$(50)
100 Sub a()
110   Try
120     Call b()
130   Else
140     If Spc(8) < 10000 Let e$ = "sub a " + Msc$(2) + " at " +
Str$(Spc(10))
150     Error 10000
160   End Try
170 End Sub
200 Sub b()
210   Try
220     x$ = ""
230   Else
240     If Spc(8) < 10000 Let e$ = "sub b " + Msc$(2) + " at " +
Str$(Spc(10))
250     Error 10000
260   End Try

```

```

270 End Sub
300 Rem main
310 Try
320   Call a()
330 Else
340   If Spc(8) < 10000 Let e$ = "main " + Msc$(2) + " at " +
Str$(Spc(10))
350   Print e$
360 End Try

```

Note: A procedure may deallocate its use of a referenced variable, using the **FREE** statement. However, the procedure cannot deallocate variables in the calling program.

Occasionally, it might be necessary to loop through a variable number of arguments. In this example, the **procedure accepts arguments from the command line input buffer**. This procedure parses a command line, for example:

```

#test 0 123 44 18 20

External Sub GetCommandLineArgs(...)
Dim S$(80)
Dim %1,P

Input (0,K)S$           ! Get the input command string
!special Input mode to input command string
!see INPUT statement in Reference manual
!mode 0=read command line, K=not used in this mode

Do
  ! Skip to next arg in command line string
  ! This skips the first which is the supplied program name
  P = Pos(S$, = " ")
  If P Let S$ = LTrim$(S$[P + 1]) Else S$ = ""
  !Ltrim removes any extra spaces between parameters

  If Not(S$) Exit Sub      ! No more parameters
  ! S$ now has a parameter
  ! Assign to caller's numeric var by referencing as 'N'
  Try Enter N,... Else Exit Do ! No more, or wrong type
  N = S$                   ! Assign to callers var
  Free N                   ! So next time points to next param
  Loop                     ! Loop until no more parameters
End Sub                    ! Done

```

User Calls as Procedures

To facilitate the development of User Calls, dL4 treats procedures written in **C** virtually identical to those written in BASIC. In addition, procedures may be written in **C** as Functions returning a value to the caller.

This streamlined mechanism allows developers the flexibility to create procedures in either C or BASIC while maintaining a uniform calling sequence:

- Procedures written in C are declared Intrinsic
- Procedures written in BASIC are declared internal (implied) or External.

dL4 Product Training

SECTION 3D

New & Enhanced Statements

The following are new statements available in dL4, in addition to the new statements for Structured Programming and new statements for accessing various drivers, which are described in the driver sections. New statements **MOVE** and **SIZE** are described in the GUI section.

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 of the rectangle. If not specified the current cursor position is used.

TO is a keyword which must be followed by the lower right coordinates of the box,

-or-

SIZE is a keyword which must be followed by two integers representing the number of columns and rows.

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. When running on Unix, your terminal description file must contain a definition for these 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;
```

CHAIN READ IF

Synopsis

Like CHAIN READ of specified variables, but without reporting an error if any of the variables weren't passed.

Syntax

CHAIN READ IF *var.list*

Parameters

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

Executable From Keyboard?

No.

Remarks

See **CHAIN READ** for further remarks..

Examples

CHAIN READ IF SubTotal, Filename\$

See also

CHAIN READ,CHAIN WRITE

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#

```
Channel 36, #1; "E" !Channel 36 sets open mode, i.e. "E"  
Exclusive Open
```

Example of use with the Windows driver:

```
Channel 11, #1; !shows window  
Channel 12, #1; !hides window
```

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"
```

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 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").

Examples

Duplicate "PAYROLL PAY1QTRBKUP"

Duplicate "/usr/ub/23/file /u/u1/23/file"

EOPEN

Synopsis

Open an existing file for exclusive access.

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

Similar to the **OPEN** statement except opens a file exclusively. Exclusive open only allows the file to be opened if it is not already open by another user. Once opened exclusively it cannot be opened by other users until closed.

Examples

```
Eopen #2,"cust.masterfi" AS "Full-ISAM"
```

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

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 **reDim** it to the last precision level.

Examples

```
Free N
```

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

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

See also

DIM

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.

The GET statement can be used to retrieve the local or remote IP addresses associated with a socket. Example:

```
Get #c,-1598;RemoteAddress$  
Get #c,-1599;LocalAddress$
```

Get #c,-399;\$ can be used to retrieve the index character set from an indexed-contiguous or Full-ISAM file.

Get #c,-1299;Name\$ can be used to determine the character set used on channel c.

By using the DUPCHANNEL intrinsic CALL, these operations can also be used on the standard input and output channels if they are open to a socket.

Examples

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

See also

SET

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

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 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 <i>numconst</i>
OPTION FOR NESTING 8	OPTION FOR NESTING <i>numconst</i>
OPTION GOSUB NESTING 8	OPTION GOSUB NESTING <i>numconst</i>
OPTION TRY NESTING 8	OPTION TRY NESTING <i>numconst</i>
OPTION COMMA SPACING 15	OPTION COMMA SPACING <i>numconst</i>
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 STRINGS STANDARD	OPTION STRINGS RAW

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 <i>numconst</i>
OPTION CHAIN ALTERNATE DIRECTORIES ON	OPTION CHAIN ALTERNATE DIRECTORIES OFF
OPTION STRING SUBSCRIPTS STANDARD	OPTION STRING SUBSCRIPTS IRIS
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 <i>numconst</i>
OPTION STRING REDIM IS ERROR	OPTION STRING REDIM IS LEGAL
	OPTION STRINGS HAGEN
	OPTION ZERO DIVIDED BY ZERO IS LEGAL
	OPTION DEFAULT ARGUMENT CHECKING IS WEAK
	OPTION PROGRAM TAG "text"

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 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 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 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. When a program that uses **OPTION AUTO DIM OFF** is saved, error messages will be generated for each variable that is not declared in a DIM, COM or CHAIN READ statement, or parameter list.

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 STRING REDIM IS LEGAL** allows re-dimension of simple string variables without FREEing the variable.

The **OPTION STRINGS HAGEN** provides compatibility with Unibasic HAGEN string mode.

The **OPTION ZERO DIVIDED BY ZERO IS LEGAL** allows zero divided by zero to equal zero rather than causing an arithmetic overflow.

The **OPTION DEFAULT ARGUMENT CHECKING IS WEAK** allows passing array variables to subprograms (CALL by filename) without using the empty bracket notation. This improves IMS compatibility. (Only available in OPTION DEFAULT mode.)

The **OPTION PROGRAM TAG “text”** can be used to place user defined text in a compiled program file as ASCII text. This option can be used to add listable revision text strings to be printed by the Unix “what” utility.

The **OPTION DIALECT [STANDARD | IRIS | BITS | IRIS1 | BITS1]** sets multiple options. The default option settings should serve best for all IRIS programs.

The statement **OPTION DIALECT IRIS** is equivalent to **OPTION STRING SUBSCRIPTS IRIS**. **OPTION DIALECT IRIS1** also includes **OPTION ZERO DIVIDED BY ZERO IS LEGAL**.

IMS users should add the following line to each program:

OPTION DIALECT IMS

Which is the equivalent of OPTION DIALECT IRIS1, plus IMS Using “mask” behavior

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

OPTION DIALECT BITS1 is the equivalent of all the above, plus :

OPTION DEFAULT ZERO DIVIDED BY ZERO IS LEGAL

Final value of FOR/NEXT loop is BITs compatible

Using mask is BITs behavior
Initial numeric precision is 4%

Examples

Option Date Format Native

PORT (enhanced)

Synopsis

Attach and control other ports.

Syntax

PORT *num.expr1*, 4, *num.var1*, *str.var*

Parameters

num..expr1 is an expression used to select a target port number.

num.var1 is a variable of numeric data type used to receive operational status.

str.var is a variable of string data type used to receive response.

Executable From Keyboard?

Yes

Remarks

The following modes have been **added** to the PORT statement :

Mode 4-Return name of Current Program of Specified Port

Port mode 4 returns the name of the current program on a specified port. For example, the statement :

```
Port P,4,S,L$
```

will return in L\$ the name of the program running on port P.

Mode 5-Return current line # and library name of Specified Port

Port mode 5 returns the current line number and library name executing on a specified port. For example, the statement :

```
Port P,5,S,L$
```

will return in L\$ the current line number of the program running on port P. If the program is executing a line in a library, then L\$ will have the format "library:line#" where "library" is the name of the library. A status is returned in S indicating success (zero) or failure (one).

Mode 6-Determine if blocked by record lock on Specified Port

Port mode 6 determines whether a port is blocked by a record lock and which other port is holding the needed record lock. The statement:

```
Port portnum,6,status,isblocked,blockingport
```

sets the variable "isblocked" to one if the dL4 program running on port "portnum" has been waiting for over 20 seconds for a record lock and to zero if it is not blocked. If the port is blocked by a record lock, the variable "blockingport" is set to the port number of the program that currently has the record locked or to -1 if the blocking port number

cannot be determined. PORT mode 6 is supported for record locks on formatted, contiguous, and indexed contiguous files.

Mode 7-Return user name and work station name of Specified Port

Port mode 7 returns the user name and work station name of a specified port. For example, the statement :

```
Port portnum,7,status,userid$,station$
```

Will query port "portnum" and returns in "userid\$" the user name, if any, associated with the port and returns in "station\$" the terminal name, if any, used by the port. Mode 7 can optionally return the group name, current directory path, terminal type, numeric account, and numeric group for the selected port. The group name, numeric account, and numeric group values are returned as "" under Windows. Example:

```
Port p,7,status,userid$,station$,group$,dir$,term$,usern$,grpn$
```

Mode 8>Returns the open channels on the Specified Port

Port mode 8 returns the open channels on a specified port. For example, the statement :

```
Port P,8,status,firstchan,lastchan,chaninfo.[ ]
```

will return open channel information into the array 'chaninfo.[]' for channels between 'firstchan' and 'lastchan'. The 'chaninfo.[]' variable must be an array of structures using the following structuredefinition:

```
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.

Mode 9-Determines if a specified file is open on the Specified Port and if a record is locked

Port mode 9 determines if a specified file is open on a specified port and optionally determines if a specified record number is locked in the file. For example, the statement :

```
Port P,9,status,filepath$,recordnum,channum
```

Performs The PORT statement has been extended with a new mode, 9, that determines if a specified file is open on the port and optionally determines if a specified record number is locked in the file. The statement

```
PORT portnum,9,status,filepath$,recordnum,channum
```

performs an inquiry on port 'portnum' to determine if it has 'filepath\$' open on a channel with the record 'recordnum' locked. If 'recordnum' is negative, the search will be performed using only the file path. If a match is found, the channel open to the file will be returned in 'channum'. If a match is not found, then 'channum' will be set to -1.

Examples

```
Port P,4,S,L$ \ If S Stop !get program name
Port P,5,S,L$ \ If S Stop !get current line # and library
```

See also

SWAP, SPAWN

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"
```

Training Note: This is a low level method of defining a file if the program is allowing the user to interactively define the file. It is easier to use defined structures and the **Define Record** command.

See also

GET

SYSTEM (enhanced)

Synopsis

Execute operating system specific commands.

Syntax

SYSTEM *str.expr* [,*num.var*]

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.

Executable From Keyboard?

Yes.

Remarks

The following modes have been **added** to the SYSTEM statement :

SYSTEM 29 sets alternate sources of Environment Variables. This function requires the 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, for example "DynamicConcepts\\dL4\\Environment". Note the extra backslash required to preface a backslash in a string. Also note, the SYSTEM 29 statement must use a string variable, not a string expression. This mode is not supported on Unix systems.

SYSTEM 30 executes system commands with standard input and output assigned to /dev/null. This eliminates most screen output by the system command.

SYSTEM 31 When using dL4Term, it is possible to run programs on the user's PC. The statement **SYSTEM 31, "C:\\Program Files\\Application\\program.exe", S** will try to execute "program.exe" on the user's PC. The variable "S" will be set to zero if the program was successfully started and non-zero if the program couldn't be started. The SYSTEM statement suspends execution of the dL4 program and waits until the program exits. To execute in a non-modal fashion (execution of the dL4 program continues), run the application via the cmd.exe (or preferably use SYSTEM 33), such as

SYSTEM 31, "cmd.exe /c start c:\\Progra~1\\Intern~1\\IEXPLORE.EXE", S

In the default dL4Term configuration, the user will be prompted via a message box to permit or deny running the command. See the dL4Term documentation and readme.txt file for information on changing the message box configuration. The DWORD registry value

HKEY_CURRENT_USER\\Software\\DynamicConcepts\\dL4Term\\AllowSYSTEMCmd

Or

HKEY_LOCAL_MACHINE\\Software\\DynamicConcepts\\dL4Term\\AllowSYSTEMCmd

can be **set to one to automatically accept commands** without displaying the message box (note: if set to zero, an "AllowSYSTEMCmd" value in HKEY_CURRENT_USER will require the message box no matter how the HKEY_LOCAL_MACHINE value is set).

SYSTEM 32 will return the number of available 512 byte disk blocks on a file system. The statement **SYSTEM 32,"path",B** returns in B the number of available blocks on the file system that contains the directory or file "path".

SYSTEM 33,"C:\\Program Files\\Application\\program.exe",S will try to execute "program.exe" on the user's PC. The variable "S" will be set to zero if the program was successfully started and non-zero if the program couldn't be started. Unlike SYSTEM 31, the statement does not wait for the program to finish and exit. This statement is intended for use with dL4Term or dL4 for Windows.

Examples

```
SYSTEM 30,CommandString$
```

See also

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"
```

Changed Statements

BUILD # - Declare Type of File to Create

In addition to the older syntax , BUILD provided application selection of a driver or driver class when creating a new file:

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"
```

CALL – Calls program name or procedure

In addition to calling other programs, a procedure can be called as described in the Procedures section.

CLEAR - Initializing Variables

In addition to clearing channels (CLEAR #1), the CLEAR statement can be used to clear variables, i.e. to initialize them as if they were FREEd and re-DIMensioned.

```
CLEAR X, Y, A$
```

Clearing a variable initializes its value as if the variable had just been **DIMed**. 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.

DATA - Quoted Strings

Character strings in DATA statements must be enclosed in quotes (").

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

OPEN # - Declare Driver to Process a File

In addition to the older syntax , OPEN provides application selection of a driver or driver class when opening a file:

```
OPEN #c; filename AS "FULL-ISAM"
```

The AS clause works as described above for the BUILD statement.

READ # and WRITE # - Binary, Date and Structure Variables

Normal READ and WRITE statements support all of the new data types. Whether a particular type can be transferred to/from a channel is driver/file-specific. When reading or writing a structure variable, each separate element is read or written sequentially.

If CUST. is a structure with members NAME\$, ADDR\$ and CITY\$, then the statement:

```
READ or WRITE #c,r,b; CUST.
```

is identical to:

```
READ or WRITE #c,r,b; CUST.NAME$, CUST.ADDR$, CUST.CITY$
```

The additional functionality of mapping structure elements to fixed or named positions within a file are supplied by the new statements READ RECORD and WRITE RECORD.

SEARCH # - Locate First, Last, >=, <= Keys

For use with full ISAM data files, the SEARCH statement has been streamlined.

```
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.

Removed Statements

EXECUTE	Compile / execute statement immediately from string
INDEX	Use SEARCH (automatically converted by CONVERT command)
RDREL	Use READ binary variable to the 'raw' file driver
WRREL	Use WRITE binary variable to the 'raw' file driver
CREATE	Use BUILD (automatically converted by CONVERT command)

dL4 Product Training

SECTION 3E

Global Variables

Global Variables

Two Intrinsic Call Subroutines are available for the purpose of setting or reading global variables for a user. The variable table will be maintained throughout the dL4 session. Globals can be broken up into named 'sets' or 'groupings'. Up to 1000 different global variables can be stored per user.

You can think of global variables as similar to reading/writing to record 0 of a formatted file, except the field type can change at anytime.

CALL SETGLOBALS

Synopsis

Set the value of a global variable

Syntax

CALL SETGLOBALS (*num.expr1*{*str.expr1* or *num.expr2* {*str.expr2* or *num.expr3* ...} })

CALL SETGLOBALS (*str.expr1*,*num.expr1*,*str.var1* or *num.var1*{*str.var2* or *num.var2* ...})

Parameters

str.expr1 is an optional string variable or expression specifying a named set of global values .

num.expr1 is a numeric variable or expression specifying the global item number to set.

str.expr1 is a string variable or literal specifying the value to be stored and associated with the item number.

num.expr2 is a numeric variable or expression specifying the value to be stored and associated with the item number.

str.expr2 and *num.expr3* are optional subsequent items to be set.

Remarks

Element numbers begin at 0.

Up to 1000 elements can be stored.

If only the item number is specified, the value of that item is cleared.

Otherwise, either a string or numeric expression is provided to set the item.

Additional string or numeric expressions can be listed, comma separated, to set subsequent items.

Examples

```
Call SetGlobals(3) !clears global item 3
```

```
Call SetGlobals(3,"123456")
```

```
Call SetGlobals(fieldnumber,value)
```

```
Call SetGlobals(1,x$,35,y$)
```

```
Call SetGlobals("mylib",0,N$,A) !set values in mylib set
```

CALL GETGLOBALS

Synopsis

Get the value of a global variable

Syntax

CALL GETGLOBALS (*num.expr1*,*str.var1* or *num.var1*{,*str.var2* or *num.var2* ...})

CALL GETGLOBALS (*str.expr1*,*num.expr1*,*str.var1* or *num.var1*{,*str.var2* or *num.var2* ...})

Parameters

str.expr1 is an optional string variable or expression specifying a named set of global values .

num.expr1 is a numeric variable or expression specifying the global item number to get.

str.var1 is a string variable to retrieve the value stored in the global variable.

num.var1 is a numeric variable to retrieve the value stored in the global variable.

str.var2 and *num.var2* are optional subsequent items to be set.

Remarks

Items stored as numerics must be retrieved as numerics.

Items stored as strings must be retrieved as strings.

Additional string or numeric variables can be listed, comma separated, to get subsequent items.

Examples

```
Call GetGlobals(3,customer$)
```

```
Call GetGlobals(fieldnumber,value)
```

```
Call GetGlobals(1,x$,d,y$)
```

```
Call GetGlobals("mylib",0,N$,A) !get values from mylib set
```

dL4 Product Training

SECTION 3F

Functions

Predefined Functions

Name	Parameters of Function
ABS(n)	Absolute value.
ASC(s\$)	Unicode value of first character in string.
ATN(n) ¹	Arctangent.
BSTR\$(n,b)	Returns the 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(1500 + c)	Returns the number of characters read by the last I/O operation.
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. Not supported on operating systems such as Unix that do not provide a file creation time.
CHF#(200 + c)	Driver dependent: typically last access date/time for the file open on channel c. Note: Opening a file will change the last access date/time unless the "Raw" file driver is used.
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 (may be a 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)	Filename, always as the native absolute path of the file open on channel c.
CHR(n)	Returns the decimal characteristic of the argument. This is an integer exponent X such that: $10^{X-1} \leq n < 10^X$

¹ Angles are interpreted as either radians or degrees depending on setting of the **OPTION ANGLE** statement.

CHR\$(n)	Returns the Unicode character whose value is <i>n</i> . Note: when converting BITS programs, CHR() must be manually converted to CHR\$().
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 <i>n</i> .
ERR(n)	Various values pertaining to error, ESCAPE and interrupt branching.
ERR(0)	Number of last error.
ERR(1)	Line number of last error.
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)	Returns -1 (Unibasic compatibility)
EXP(n)	Exponential, the constant <i>e</i> to the power given (<i>e</i> ^{<i>n</i>})
FRA(n)	Fractional portion. For example: FRA(4.5) yields 0.5.
GMT\$(d#) ²	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 <i>s\$</i> , which is assumed to contain a hexadecimal representation of binary data.
HEX\$(b?)	Returns a character string containing the hexadecimal representation of <i>b?</i> .
INT(n)	Returns the greatest integer less than or equal to <i>n</i> . 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 <i>n</i> . For example: IXR(3) returns 1000.
LBOUND(a[],0)	Number of dimensions of array <i>a</i> . Trailing brackets ("[]") must follow array <i>a</i> .
LBOUND(a[],n)	Lower subscript bound of dimension <i>n</i> of array <i>a</i> . Trailing brackets ("[]") must follow array <i>a</i> .
LCASE\$(s\$)	Converts all upper-case letters to lower-case.
LEN(s\$)	Length of string in characters.
LOG(n)	Logarithm base <i>e</i> of <i>n</i> . Logarithm in any base <i>B</i> 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 <i>n</i> in base 10.
MONTH(d#)	Numeric month value from <i>d#</i> ; 1 - 12.
MONTH\$(n) ⁵	Name of month from <i>n</i> , 1 - 12.
MONTHDAY(d#)	Day number of month from <i>d#</i> ; 1 - 31.
MSC	Miscellaneous numeric functions
MSC(0)	Current port number.
MSC(1)	Last logical input element accepted.
MSC(2)	Returns -1 or value of SPC(4) runtime parameter (Unibasic compatibility)
MSC(3)	Line number of last GOSUB executed. Value is returned and removed from the GOSUB stack.
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.

² Exact character representation of date components depends on setting of the **OPTION DATE FORMAT** statement.

MSC(18)	The constant π (3.141592653589793).
MSC(19)	The constant e (2.718281828459045).
MSC(20)	Maximum channels per user; returns 100.
MSC(30)	Current line number.
MSC(31)	Current statement number on line..
MSC(33)	Number of columns on the default I/O channel. (GUI)
MSC(34)	Number of rows on the default I/O channel. (GUI)
MSC(35)	Input buffer size in characters.
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, i.e. current screen row. When MSC(43) is used in a PRINT statement, the initial value of the row counter is returned.
MSC(44)	Returns 1 if Dynamic Windows are on, else will return 0
MSC\$(n)	Miscellaneous string functions.
MSC\$(-1)	Returns -1 or value of SPC(4) runtime parameter formatted as "RLLBBSS" (Unibasic compatibility)
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)	Returns native path separator string. dL4 for Unix returns a forward slash (/)
MSC\$(9)	Returns the native absolute path of the directory containing the current program file
MSC\$(264)	Returns ""
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 < <= > >= = <>. s can be negative to indicate backwards searching from the end of string. Ex. POS($s\%,t\%$) The expression "POS($S\%$, IS $T\%$)" will search for the first character in the string $S\%$ that matches a character in the string $T\%$. The expression "POS($S\%$, EXCEPT $T\%$)" will search for the first character in the string $S\%$ that does not match a character in the string $T\%$.
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)	Returns -1 or value of SPC(4) runtime parameter (Unibasic compatibility)

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(13)	Returns terminal type number from terminal definition file
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 is not representable as a number.
SPC(24)	Line number of last END , STOP or SUSPEND statement.
SPC(264)	Returns -1 or value of SPC(264) runtime parameter (Unibasic compatibility)
SPC(272)	Returns -1 or value of SPC(272) runtime parameter (Unibasic compatibility)
SQR(n)	Square root.
STR\$(n)	Convert the numeric value <i>n</i> 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#(0)	Current real-time.
TIMEZONE(d#)	Local time-zone offset from GMT in seconds in effect as of <i>d#</i> .
TRUNCATE(n,d)	Truncates <i>n</i> to <i>d</i> decimal places.
UBOUND(a[,0])	Number of dimensions of array <i>a</i> . Trailing brackets ("["]) must follow array <i>a</i> .
UBOUND(a[,n])	Upper subscript bound of dimension <i>n</i> of array <i>a</i> . Trailing brackets ("["]) must follow array <i>a</i> .

UCASE\$(s\$)	Converts all lower-case letters to upper-case.
VAL(s\$)	Convert the string value <i>s\$</i> to a number.
WEEKDAY(d#)	Day of week number from <i>d#</i> ; 1 = Sunday, 7 = Saturday.
WEEKDAY\$(n) ⁵	Day of week name for day <i>n</i> ; 1 = Sunday, 7 = Saturday.
YEAR(d#)	Year number from <i>d#</i> .
YEARDAY(d#)	Day of year number from <i>d#</i> ; 1 - 366.

Make note of the following dL4 Functions that were not available in Unibasic :

LCASE\$
UBOUND
UCASE\$
LTRIM\$
RTRIM\$
PCHR\$
POS
REP\$
ROUND
TRUNCATE

Numerous functions related to Dates:

DAT#,MONTH,WEEKDAY,YEAR

Additions to intrinsic Functions (require a DECLARE INTRINSIC statement) :

- o Intrinsic function, FindChannel, returns the first closed channel number in a specified range. If no range is specified, the first channel between 99 and 0 (note descending order) is returned. An error will be generated if an illegal number of parameters, parameter type, or parameter value is used.

BASIC syntax:

```
x = FindChannel()
x = FindChannel(startchan,endchan)
```

- o Intrinsic function, FmtOf, returns a number that is interpreted as the precision (for a numeric variable) or dimension (for a string, binary or array variable) of the *var*.

- o Intrinsic string function, Trim\$(), removes both leading and trailing blanks or other whitespace.

- o Intrinsic string function, ENCFNM\$(), adds quotation marks to a filename if required. The

function can be invoked with either one or two string arguments. With two arguments, the first argument will be treated as a directory prefix to be added to the second filename argument.

- o Two intrinsic functions, MD5? and ADDMD5?, have been added to dL4. MD5?() returns the MD5 checksum of the first argument. The argument must be either a binary or a string value. An optional second binary argument can be used to pass an intermediate value from a previous call to ADDMD5?(). This allows a combined checksum of multiple values to be calculated. Checksums are calculated against the DIMed size of

strings so that zero characters can be included in the checksum. To avoid this, simply pass strings with subscripts. So that string values will produce the same checksums on all platforms, each UNICODE character of a string is forced into a most-significant-byte first ordering for calculation.

BASIC syntax:

```
Dim chksum?[16], intermediate?[128]
```

```
chksum? = MD5?(var)
```

```
chksum? = MD5?(var,intermediate?)
```

```
intermediate? = AddMD5?(var)
```

```
intermediate? = AddMD5?(var,intermediate?)
```

- o Intrinsic function, CRC32() returns a 32-bit CRC checksum code of the first argument which must be either a binary or string variable. An optional second numeric argument can be used to pass the CRC value from a previous call and calculate a combined CRC of several variables. CRCs are calculated against the DIMed size of strings so that zero characters can be included in the CRC. 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.

BASIC syntax:

```
x = CRC32(var)
```

```
x = CRC32(var, oldcrc)
```

- o Intrinsic string function, DATEUSING\$, is available to format date values. The function

syntax is:

```
DateUsing$(DateValue#, FormatString$)
```

where "DateValue#" is a date expression to be formatted and "FormatString\$" is a string expression containing a mask string. The following formatting codes are recognized in the mask string:

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

Formatting codes are replaced by their associated values. Any unrecognized characters will be copied unchanged to the result string.

Example:

```
Print DateUsing$(Tim#(0), "MM/DD/YYYY HH:NN")
```

- o The SPC(n) function has been extended to return the numeric value of the environment variable "SPCn" where "n" is an undefined SPC function number. For example, if the environment variable SPC105 was set to "65", then SPC(105) would return a value of 65. User defined SPC functions should be defined at 100 and above to avoid conflicts with any future standard SPC functions.

- o Intrinsic function, ErrMsg\$, is available to replace the UniBasic ERM() function. The ErrMsg\$() function is used with CALL InitErrMsg() to replace the error message facility provided by UniBasic ERM() function and CALL 40.
- o Intrinsic function, UBMem(), is used to replace the UniBasic MEM() function. Like the UniBasic MEM() function, UBMem() always returns zero.

- o Intrinsic function, CRC16(), calculates 16 bit CRC values. BASIC syntax:
`x = CRC16(mode, polynomial, string, oldcrc)`

where:

- "mode = 0" - calculate 8-bit sum of lower 8 bits of each character
- "mode = 1" - calculate CRC using lower 8 bits of each character

An XMODEM compatible CRC can be calculated using mode one, a polynomial value of 4129 (0x1021), and an initial CRC of zero.

- o Two intrinsic string functions simplify replacing string values within a string. The function

`Replace$(Source$,Old$,New$,Count)`

returns the string 'Source\$' replacing the first 'Count' occurrences of the string 'Old\$' with the string 'New\$'. All occurrences of the string may be replaced by omitting the 'Count' parameter or using 'Len(Source\$)' as the value of 'Count'. The function ReplaceCI\$() is identical to Replace\$() except that it uses a case-insensitive search for 'Old\$'.

- o A new intrinsic CALL, UBSTRING(), and two new intrinsic functions, UBASC() and UBCHR\$(), have been added to provide uniBasic-like character conversion routines. The parameters and return values of the routines are identical to CALL STRING(), ASC(), and CHR\$() except that ASCII characters are mapped to the integer range 129 through 255 and uniBasic compatible mnemonics are mapped to the range 1 through 127. These routines can be used to simplify conversion of uniBasic programs to dL4.

- o A new intrinsic function, CALLSTAT\$(), has been implemented to return a string describing the **program position at a specified level** in the procedure stack. This function would typically be used to generate information for an error log. The procedure stack includes all function, procedure, Call-Subprogram, and SWAP levels. The level type ("Swap", "SubPgm", "ExtSub", "ExtFunc", "IntSub", "IntFunc", or "") is returned in a function argument. The current position is level 0, the caller is level 1, and so on. An error 38 is generated if a non-existent level is specified.

BASIC syntax:

`CallStat$(Level, LevelType$)`

Example:

```
Print "Caller position is ";CallStat$(1,Type$)
Print "Caller type is ";Type$
```

dL4 Product Training

SECTION 3G

New Calls

Listed below are some of the new CALLs implemented in dL4.

See the Debugging section for information on the **PROGRAMDUMP** and **FORCEPORTDUMP** Calls.

CALL CHSTAT

Synopsis

Returns current swap level information

Syntax

```
CALL CHSTAT(SwapLevel, ParentLineNum, ParentName$)  
CALL CHSTAT(SwapLevel, ParentName$, ParentLineNum)  
CALL CHSTAT(SwapLevel, ParentName$)  
CALL CHSTAT(ParentName$, SwapLevel, ParentLineNum)  
CALL CHSTAT(ParentName$, SwapLevel)  
CALL CHSTAT(ParentName$)  
CALL CHSTAT(SwapLevel, ParentLineNum)  
CALL CHSTAT(SwapLevel)
```

Parameters

SwapLevel receives the current SWAP level number (zero if there are no SWAPs in progress)

ParentLineNum receives the line number of the SWAP statement in the parent program (zero if none)

ParentName\$ receives the name of the parent program ("" if none).

Remarks

Implemented for IMS compatibility

Examples

```
Call CHSTAT(C,S$)
```

CALL CALLSTAT

Synopsis

Returns current Call-by-filename level information

Syntax

```
CALL CALLSTAT(CallLevel, ParentLineNum, ParentName$)  
CALL CALLSTAT(CallLevel, ParentName$, ParentLineNum)  
CALL CALLSTAT(CallLevel, ParentName$)  
CALL CALLSTAT(ParentName$, CallLevel, ParentLineNum)  
CALL CALLSTAT(ParentName$, CallLevel)  
CALL CALLSTAT(ParentName$)  
CALL CALLSTAT(CallLevel, ParentLineNum)  
CALL CALLSTAT(CallLevel)
```

Parameters

CallLevel receives the current CALL-by-filename level number (zero if there are no CALLs-by-filename in progress)

ParentLineNum receives the line number of the SWAP statement in the parent program (zero if none)

ParentName\$ receives the name of the parent program ("" if none).

Remarks

Implemented for IMS compatibility

Examples

```
Call CALLSTAT(C,S$)
```

CALL DUPCHANNEL

Synopsis

Duplicate existing open channels onto an available channel number.

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 with a value of an open user channel (0 - 99), i.e. old channel, **standard input channel (-1) or standard output channel (-2)** to

duplicate. (Channel -3 is the current input channel and channel -4 is the current output channel.) 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 the main window.

```
External Function ChangeWinTitle(NewName$)
  Declare Intrinsic Sub DupChannel
  Call DupChannel(99, -2) !duplicate output channel

  !set mode -1073 changes window title
  Set #99,-1073;NewName$
  Close #99
End Function 0
```

```
Input A
B = ChangeWinTitle(" Test Win Name ")
Input A
Stop
```

Examples

```
Call DupChannel(1,2)
Call DupChannel(newchannel,oldchannel)
```

CALL ENV (enhanced)

Synopsis

Get or change the value of an environment variable

Syntax

The following syntax have been **added** to the Env Call

CALL ENV (*num.expr1*, *str.expr1*\$, *str.expr2*\$)

Parameters

Num.expr1 is a numeric variable or expression indicating the mode. "mode" is 1 to read the value of environment variable "str.expr1\$" into "str.expr2\$" and "mode" is 2 to set a new value for environment variable "str.expr1\$" with the value of "str.expr2\$".

Remarks

Mode 1 is equivalent to a **SYSTEM 28** statement. (read)

Mode 2 is equivalent to original syntax of no mode. (set)

For mode equal to 1, the following special value names will be recognized and will override any environment variables with the same name:

PID Unix process id

GID Unix group id

UID Unix user id

Examples

```
Call ENV(1, "TERM", value$)
```

CALL FLUSHALLCHANNELS

Synopsis

Requests Operating System to write all system buffers to disk for each open channel on a Windows system

Syntax

CALL FLUSHALLCHANNELS ()

Parameters

None

Remarks

Has no effect when called in dL4 for Unix.

The CALL provides the same functionality as a "CHANNEL #c,DCC_SYNC,1;" on each open channel.

It could be used to reduce the risk of losing data if the user were to simply turn their Windows system off.

Examples

```
Call FLUSHALLCHANNELS()
```

CALL NRC32

Synopsis

Compute the NCRC32 checksum of a string

Syntax

CALL NCRC32 (*num.var1*,*str.expr1*{*num.expr1*})

Parameters

num.var1 is a ten digit or larger numeric variable to receive the calculated CRC-32 checksum.

str.expr1 is a string variable or literal to be checksummed.

num.expr1 is an optional numeric value containing a CRC-32 value from a previous calculation.

Remarks

Is available for compatibility with Unibasic 7.

The CALL provides the same functionality as the existing CRC32() intrinsic function.

Examples

```
Call NCRC32(C,S$,0)
```

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:

<u>mode</u>	<u>Operation</u>
-------------	------------------

- | | |
|---|---|
| 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 l%, 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 l%, 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 l%, 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 l%, ErrorCode, ErrorMessage$[200]
Call ProgramCache(3, ErrorCode, ErrorMessage$)
If ErrorMessage$ Print "Cache initialization error:
";ErrorMessage$
```

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.

CALL IMSMEMCOPY

Synopsis

Copies a number of bytes from one variable to another

Syntax

CALL IMSMEMCOPY (*var1*,*var2*,*num.expr1*)

Parameters

var1 is a variable of any type, considered the Destination variable.

var2 is a variable of any type, considered the Source variable.

num.expr1 is a numeric variable or expression indicating the number of bytes to copy.

Remarks

Duplicates the behavior of CALL 90 in IMS.

The CALL copies "ByteCount" bytes from the "Source" variable to the "Destination" Variable.

If both "Destination" and "Source" are string variables or arrays, then "ByteCount" characters are copied.

This CALL is dangerous and will corrupt memory if "ByteCount" exceeds the size of the destination variable. This is allowed so that "Destination" variable can be a subscripted array element identifying the start of a copy range.

Examples

```
Call IMSMemCopy(Destination, Source, ByteCount)
```

CALL RMVSPACES() and RMVSPACESI()

Synopsis

Remove leading and trailing spaces from of a string

Syntax

CALL RMVSPACES (*str.expr1*,*str.var1*{*num.expr1*})

CALL RMVSPACESI (*str.expr1*,*str.var1*{*num.expr1*})

Parameters

str.expr1 is a string variable or literal of the source string.

str.var1 is a string variable if the destination string.

num.expr1 is a numeric variable or expression indicating the MODE.

Remarks

Duplicates the behavior of `CALL $RSPCS()` in UniBasic.

If "MODE" is equal to 1, then source string is copied to destination string with all spaces removed except those within quotes, all characters after and including an unquoted "!" are removed, and a trailing linefeed is appended if the string ends in a "!". This can be helpful in creating text files of code.

Call `RMVSPACESI()` differs in that "MODE" 1 always appends a linefeed and a "MODE" other than 0 or 1 causes an error.

Examples

```
Call RMVSPACES(SOURCE$,DEST$,0)
```

CALL SORTINSTRING

Synopsis

Sort Keys in a String. This CALL has been extended to support sorting arrays of strings or arrays of structures where the first structure member is a string.

Syntax

CALL SORTINSTRING (*num.var*, *num.expr1*, *num.expr2*, *str.var1*, *str.var2*)

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.

str.var1 is a string variable containing the keys to be sorted. It may contain any number of fixed-length 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 **DIMensioned to a minimum of length (*num.expr2*) + 8.**

Remarks

The meaning of the return status value from the sort operation:

<u>status</u>	<u>Description</u>
---------------	--------------------

- | | |
|---|--|
| 0 | Successful sort operation. |
| 1 | Parameter Error. |
| 2 | <i>number</i> or <i>length</i> was passed as zero. |
| 3 | <i>sort</i> string is too small; less than <i>number</i> * <i>length</i> |
| 4 | <i>work</i> string is too small; less than <i>length</i> + 8. |

The resulting sorted string is returned in *str.var1*.

Replaces Unibasic Call 65.

Examples

Call SortInString(E, 100, 10, A\$, W\$)

!where E is status, 100 is number of fields to sort, 10 is length of each field, A\$ is the string containing the 100 ten character fixed-length strings and A\$ will receive the resulting sorted string, W\$ is the work string dimensioned to 18 minimum (10+8).

New formats:

Call SortInString(status,keycnt,keylen,keys\$,work\$)
Call SortInString(status,keycnt,keylen,keys[],work.)

The 'work\$' and 'work.' variables should be identical to individual elements of the 'keys\$[]' and 'keys.[]' arrays. The 'keylen' value specifies the maximum number of significant characters in the sorted values and

can be used to perform a sort on the first 'keylen' characters of each key value.

- o A new intrinsic CALL, WHOLOCK(), is now available to determine what port or process has locked a specific record of a file. The CALL syntax is:

CALL WHOLOCK(Channel, RecordNumber, PortNumber [, ProcessID])

where "Channel" is a channel number open to a file and "RecordNumber" is the record number in that file to be tested. If the record is locked, then "PortNumber" will be set to the port number of the process that has locked the record. If "ProcessID" is specified, it will receive the Unix process id number of the process that locked the record. If the record is not locked, then "PortNumber" and "ProcessID" will be set to -1. "PortNumber" will also be set to -1 if the process that locked the record is not a dL4 process (note that UniBasic processes are not dL4 processes) or if the dL4 process is executing on a remote system. In this release, CALL WHOLOCK() is supported only for formatted, contiguous, and indexed contiguous files.

CALL WHOLOCK() is an OS dependent CALL (**Unix only**) and it should not be used in dL4 for Windows.

- o A new intrinsic CALL, IMSPACK(), has been implemented to provide compatibility with CALL \$PACK in IMS BASIC. The syntax of CALL IMSPACK() is:

Call IMSPack(Mode, Src\$, Dest\$)
Call IMSPack(Mode, Dest\$, Src\$)

where "Mode" equals zero packs characters from "Src\$" into "Dest\$" and a non-zero "Mode" unpacks characters from "Src\$" into "Dest\$". The packing algorithm uses a radix 50 style mechanism.

- o A new intrinsic CALL, TRANSLATE(), has been added to translate strings to or from binary strings according to a specified character set. The syntax and arguments of CALL TRANSLATE() are:

Call Translate(DestCnt, Dest\$, SrcCnt, Src?, CharSet\$)
Call Translate(DestCnt, Dest?, SrcCnt, Src\$, CharSet\$)

DestCnt - receives the number of characters translated into the destination.

Dest\$ - destination string that receives the characters translated from Src?.

Dest? - destination binary string that receives the characters translated from Src\$.

SrcCnt - receives the number of characters translated from the source.

Src\$ - source string of characters to be translated. String terminator characters are copied as data and the source size is controlled by the total size of the variable and any double subscripting.

Src? - source binary string of bytes to be translated. The source size is controlled by the total size of the variable and any double subscripting.

CharSet\$ - character set name such as "ASCII", "ANSI", or "UTF-8".

If a character cannot be translated, translation will stop. Translation errors can be detected by comparing the returned source translation count to the source size.

dL4

Product Training

SECTION 3H

Ported Old Calls

Almost all documented and undocumented Unibasic CALLs are now available in dL4, including CALL 97 (read file header), CALL 127, and the legacy BASIC programs which use them, such as LIBR, QUERY, etc.

The most notable newly ported CALL is the implementation of CALL WHOLOCK.

The following are Unibasic Calls that are implemented as **intrinsic** Calls :

```
Call AvPort(PORTNUM {,MINPORT {,MAXPORT}})
Call CheckDigits(STRING$)
Call CheckNumber(STRING$)
Call ConvertCase(MODE, STRING$ {,START})
Call DateToJulian({MODE,} DATE$ {,CONVERTED_DATE$ {,STATUS}})
Call DecToOct(NUMBER, OUTPUT_NUMBER_OR_STRING_VARIABLE)
Call Echo(MODE)
Call FindF(PATH$, STATUS)
Call FormatDate(DATE$ {,CONVERTED_DATE$ {,STATUS {,MODE}}})
Call JulianToDate({MODE,} JULIAN$ {,CONVERTED_DATE$ {,STATUS}})
Call MiscStr({M,}S${,...})
Call ReadRef(CHAN, MODE)
Call Rename(LU, OLDNAME$, NEWNAME$, CHANNEL, STATUS)
Call String(MODE,...)
Call StringSearch({F,}A${,S},T$,P{,N{,S1{,T1}}})
Call Time(T$)
Call VerifyDate(DATE$ {,CONVERTED_DATE$ {,STATUS {,MODE}}})
```

The following are additional Unibasic Calls that are implemented as intrinsic Calls, as of vers 5.1 :

UniBasic CALL	New equivalent dL4 Intrinsic
CALL \$ATOE	Call AToE()
CALL \$CKSUM	Call Cksum()
CALL \$CLU	Call CLU()
CALL \$DATE	Call Date()
CALL \$DEVOPEN	Call DevOpen()
CALL \$DEVCLOSE	Call DevClose()
CALL \$DEVREAD	Call DevRead()
CALL \$DEVWRITE	Call DevWrite()
CALL \$DEVPRINT	Call DevPrint()
CALL \$ETOA	Call EToA()
CALL \$LOCK	Call Lock()
CALL \$MEMCMP	Call MemCmp()
CALL \$RDFHD (97)	Call Rdfhd()
CALL \$VOLLINK (91)	Call VolLink()
CALL 1	Call StrSrch1()
CALL 2	Call BitsNumStr()
CALL 5	Call MemCopy()
CALL 7	Call SetEcho()
CALL 15	Call PkUnPkDec()
CALL 18	Call PkRdx5018()
CALL 19	Call UnPkRdx5019()
CALL 20	Call PkDec20()
CALL 21	Call UnPkDec21()
CALL 29	Call EditField()

CALL 30	Call CopyStr()
CALL 40	Call InitErrMsg()
CALL 44	Call StrSrch44()
CALL 45	Call PkDec45()
CALL 46	Call UnPkDec46()
CALL 47	Call Misc47()
CALL 48	Call PkRdx5048()
CALL 49	Call UnPkRdx5049()
CALL 53	Call ASC2EBCDIC()
CALL 57	Call ClearStr()
CALL 72	Call Gather()
CALL 73	Call Scatter()
CALL 81	Call StrSrch81()
CALL 95	Call IRISOS95()
CALL 116	Call CloseAll()
CALL 117	Call AvailBlks()
CALL 118	Call NextAvPort()
CALL 127	Call FileInfo()

Unibasic CALL \$MONITOR has been implemented as a dL4 subroutine in the tools/oldcalls.lib. It is not an intrinsic Call. library. This is a partial implementation that does not support returning the full list of open channels.

dL4 Product Training

SECTION 3I

How To's

Background Jobs

There are several methods to launch and control programs on another port.

The easiest method to launch a program is by using the **SPAWN** statement.

Messages can be sent to and received by the launched program using the **SEND** and **RECV** statements (identical to the SIGNAL 1 and 2 statements).

CALL TRXCO() and **PORT** statements can also be used to communicate to a program initiated by **SPAWN**.

CALL TRXCO() and **PORT** statements can also be used to launch programs on another port.

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**.

For UNIX users, in order to use the **SPAWN** statement, the executable file “run” must be within one of the directories in your **PATH** environment variable. Otherwise, the environment variable **RUN** must be set to the path of the “run” executable, e.g.:

```
RUN=/usr/bin/run
export RUN
```

The **SPAWN** statement uses the **LIBSTRING** environment variable to find **BASIC** program files unless an **OPTION CHAIN ALTERNATE DIRECTORIES OFF** statement is used. An error 206 (“subprogram file not found”) will be returned if the specified program cannot be located.

Examples

```
Spawn "1/SPOOLER"
```

```
Spawn A$,K ! Start program, get port number
```

- o The **SPAWN** statement uses **LIBSTRING** to find **BASIC** program files unless an **OPTION** statement with "CHAIN ALTERNATE DIRECTORIES OFF" is used. An error 206 ("subprogram file not found") will be returned if the specified program cannot be located.
- o 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
```

Modify Terminal Definition File

How would I edit the default terminal definition file to activate the Esc key as it is typically used in Unibasic?

The primary change would be in the "[InputActions]" section where ESCAPE would be associated with the ESCAPE action. For example, in the vt100 definition file, the line

```
'EOT'=Escape
```

would be changed to:

```
'ESC'=Escape
```

There can be multiple characters defined as the ESCAPE action, but only one will interrupt running programs (the others will be recognized only if an INPUT is in progress). The vt100 is an important case, since the function keys definitions (that use an ESCAPE character as a leadin) must be removed. So, in the "[FunctionKeys]" section, the lines

```
'MU'=\E[A  
'MD'=\E[B  
'MR'=\E[C  
'ML'=\E[D  
'MH'=\E[H  
'F1'=\EOP  
'F2'=\EOQ  
'F3'=\EOR  
'F4'=\EOS
```

would have to be removed (otherwise ESCAPE will only be recognized in INPUT statements). This is not a problem on Wyse terminals where function keys do not use the ESCAPE character.

Refer to an ASCII chart to determine mnemonics to keyboard conversion, For example,

```
'EOT' = Control D  
'ESC' = Escape  
'FS' = Control \
```

To change Abort from Control \ to Control D change 'FS' = Abort to 'EOT' = Abort

Sample program to display what a key is outputting :

```
Print "IOBI" !binary mode  
Input Len(1);""A$ !press key  
Print "IOEI" !end binary input mode  
Print ASC(A$) !print ascii value of A$
```

dL4 Training Class

SECTION 4

Debugging

Commands

New or changed commands available in SCOPE:

DRIVERS is available at SCOPE to display the dL4 drivers available. The command can be followed by a specific search string, ie DRIVERS ISAM.

EXEC allows you to execute the contents of a text file.

LEVEL is available at SCOPE to display the dL4 revision level and the license number.

! (exclamation point), followed by an external command, executes the command. For example,
!ls -l on Unix executes the ls command or !notepad on Windows executes notepad.

There is a history of commands that is maintained within scope that can be accessed with the up and down arrows.

New or changed commands available in BASIC mode:

BREAK, XBREAK and NOBREAK to set break points

CHECK verifies code is okay without saving. By default it **will verify that all string variables have been dimensioned**. If a "-u" option is used undeclared numeric variables will also be listed.

EDIT is not vi oriented, use right arrow key instead of spaces to move right, use insert & delete keys, carriage return when done. You can also use the up and down arrow keys to select additional lines for editing once in EDIT mode.

LABEL converts line # references to labels. You can then DUMP to text to eliminate line numbers.

LOAD command has been extended to load programs from saved program files as well as source text files.

SHOW is similar to FIND to show where a variable name is used, ie C\$, ABC\$ or X.

EXAMINE allows you to select the library file name or main program name to view. Similar to debugger LEVEL command which allows you to move up and down the CALL stack to view various program files.

Basic Debugger Commands

A Debugger session is started whenever any of the following events occur:

- Step (".") Or "..") command line count reached
- Non-trapped BASIC error or forced termination (**ESCAPE** or **CTRL D**)

- Breakpoint
- **STOP** or **SUSPEND** statement
- Abort
- Untrapped **ESCAPE** event

To resume execution, type **GO**. To exit debugger, type **END**. To exit debugger and BASIC type **EXIT**.

To display a list of commands, type "?".

The Debugger is available only through the SCOPE Command Line IDE. Those programs that are run from outside the Command Line IDE do not have access to the debugger.

Abbreviations of commands can be used by using enough characters to make the command unique.

For example, command STATUS can be abbreviated to ST.

Refer to the Command Reference Manual for Debugger command details. The table below lists and briefly describes the commands.

Command	Description
?	Display a list of commands or a description of <command>
;	Display values of specified variables of the current running program
!	Execute external operating system command
.	Execute the next n program lines
..	Execute next program line and step through function
BREAK	Create a breakpoint at specified position or positions or BREAK
XBREAK	IF
	ERR
CONTINUE	Same as BREAK but applies to current program and programs entered via CHAIN, CALL subprogram, or SWAP.
DISPLAY	Resume execution of stopped program
	Display values of specified variables of the current running program
DUMP	List a program to a text data file or pipe driver (printer).
	-u option to include line #'s even if line #'s not used
END	Exit from Debugger to BASIC
EXAMINE	Examine and select which is the current program mode
EXIT	Abort program and exit debugger and BASIC to SCOPE
FILE	Display current program and opened all files
FIND	Search and list selected program statements (case insensitive)
GO	Resume execution of stopped program. <line#> GO to continue execution at a specific line.
HELP	Print text description of an error

LET	Change variable value
LEVEL	Moves current Debugger view between levels in CALL/SWAP stack
LIST	Decode/display dL4 statements. -c option disables paging
NOBREAK	Delete a breakpoint at specified position or positions
OEM	If no line number, all breaks deleted
PDUMP	Lists the currently authorized OSNs (OEM Security Number)
RETURN	Print variable values & other status information to a file
SHOW	Continue execution until current procedure exits
SIZE	Display where a variable name is used
	Display memory usage for current program/data
STATUS	(-l option displays sizes and names of all linked libraries)
	Print the name of the current program file and execution status
TRACE	(ST B shows breakpoints set)
WB	Enable statement trace debugging
WF	Move the debug window to the bottom of the screen
WH	Resize the debug window to full screen
WINDOW	Resize the debug window to half screen
WS	Move,resize, or change treatment of the debug window
WT	Resize the debug window to quarter screen
	Move the debug window to the top of the screen

Use Of Call Program Dump

- ❑ ProgramDump is an Intrinsic procedure
- ❑ Replaces UniBasic SAVE with variables
- ❑ Used to print stack, variables, open channel, and other miscellaneous information of a running program to a text file
- ❑ May aid in debugging a program
- ❑ ProgramDump itself is not a debugger
- ❑ The BASIC interpreter does not automatically call ProgramDump
- ❑ ProgramDump must be explicitly called from within a program

DESCRIPTION

The intrinsic procedure ProgramDump() prints program context such as current location, open channels, and variable values to the file filename\$. The dump output consists of the following sections:

Title	The date and time at which the dump was generated and the name of the program.
Last Error The	The location and text of the last statement on which an error occurred.
Call Stack	values of each variable used in the statement are also listed. If there is no error line (SPC(10) equals 0), this section is not produced. Beginning with the current procedure or program, the locations of all active function calls, subroutine (SUB) calls, subprogram calls, and program swaps are listed. For each level, the GOSUB stack, if non-empty, is also displayed.
Miscellaneous Status	The values for all SPC(), MSC(), ERR(), and MSC\$() functions.
Open Channels	This section begins with a table of all open channels, the driver used by the channel, and the filename, if any, open on the channel. Each open channel is then examined in detail by listing all CHF() and CHF\$() function values.
Variables call	The values of all local variables are displayed for each call level from the stack. The variable listing is ordered according to type and name.

CALL PROGRAMDUMP

Synopsis

Print stack, variables, open channels and other miscellaneous information.

Syntax

CALL PROGRAMDUMP ({*str.expr1*}{*str.expr2*})

Parameters

str.expr1 is the name of the text file in which to write the dump information.

str.expr2 is an optional string value. If set to "append" will cause PROGRAMDUMP() to append the new output to the output file rather than replacing the output file.

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. 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.

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.

Restriction: If the last error is to be displayed, ProgramDump() should be called within the same program unit as the error. If it is called indirectly via a subprogram CALL or a call to an external procedure, the last error will be that of the subprogram or the external procedure.

The command "**PDUMP** filename" while in the **BASIC** mode of **SCOPE** outputs a **PROGRAMDUMP()** style listing of the current program status to "filename".

The intrinsic CALL PROGRAMDUMP, the PDUMP command, and the force port dump feature support options to control line width and to enable printing string values that include nulls. The options are:

```

COLUMNS=<width>
NULLS=<boolean-value>

```

Options are specified in the CALL PROGRAMDUMP options parameter or in an options ("options. . .") field in the filename. The default line width is now 78 columns and long lines will be broken into multiple lines to improve readability. A width of zero will print lines of any width in a single line. The NULLS options may be used with or without a boolean value (the option "NULLS" is equivalent to "NULLS=TRUE"). If enabled, null characters in a string are printed as "\0". Examples:

```

Declare Intrinsic Sub ProgramDump
Dim a$(100)
a$ = "first"
a$(20) = "second"
Call ProgramDump("dump.txt!","columns=50,nulls")

pdump (nulls,columns=60)t.txt

```

The options can be used in parentheses in the DL4PROGRAMDUMP variable: DL4PORTDUMP="(nulls,columns=40)/tmp/dump.txt!"

Examples

```
Call ProgramDump(d$)
```

```
Call ProgramDump("dumpdir/dumpfile")
```

See also

CALL, CALL FORCEPORTDUMP

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 user's 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 **to create a variable filename**:

%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

dL4 Product Training

SECTION 5

Data File Drivers

dL4 Training Class

Channel Expression & File Spec

Channel

- ❑ Channels allow unification of dL4 I/O statements
- ❑ The dL4 BASIC interpreter remains purposely oblivious of the actual effect or behavior of most channel I/O statements allowing flexibility to add new drivers
- ❑ dL4 BASIC interpreter simply collects I/O arguments and passes them to the controlling driver
- ❑ A channel expression combines a channel number followed by three optional numeric parameters. It begins with a # and ends in a semicolon
- ❑ The interpretation of optional parameters in a channel expression are driver dependent, except for the third READ/WRITE parameter which represents a timeout value in tenth of a second

File Spec

- ❑ 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
- ❑ The standard list of items can be specified either as a single string expression or as a list of items
- ❑ A list of items must be used when opening a driver that requires a non-standard list of items
- ❑ The *file.spec* may be a single string expression, referred to here as *file.spec.str*, if it is a standard list of items, or it may be a list of comma separated items, referred to here as *file.spec.items*. *file.spec.items* can be used for either a standard or non-standard list of items.
- ❑ **Examples:**

BUILD #9, "(charset=ebcdic) <62> \$99.99 [100:10] myfile!" !as single string

BUILD #9,{"myfile!", "charset=ebcdic", "62", 99.99, 100,10} As "Contiguous"

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. See the Files and Devices manual for a list of standard character sets. The character set is typically set when building the file. The default is IRIS high bit set. Except for text files, which have no header, the system will know by the file header which character set to use and do the translation automatically without specifying the character set in the OPEN statement.

The "U" option letter ('BUILD #1,"<U>[10:100]file") can be used to select the "Universal" format when creating a Portable Formatted, Contiguous, or Indexed file.

The "H" option letter ('BUILD #1,"<H>[10:100]file") can be used to select the "Huge" format when creating a Portable Formatted, Contiguous, or Indexed file. This option allows a file to grow to more than 2 gigabytes in size on operating systems that support user files of that size.

- ❑ 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.

Training Note : Use character set UTF-8 to store full set of dL4 mnemonics in a 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. Various character sets available are "ASCII", "ANSI", "IRIS", "uniBasic", "Windows", "EBCDIC", and "UTF-8".

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.

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:

40	Owner can read the file.
0	
20	Owner can write to the file.
0	
10	Owner can execute the file.
0	
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

Training NOTE :File protections on Unix after a BUILD statement. BUILD applies umask to the requested permissions. So building a file with IRIS standard <00> or Unix standard <666>, the end result is a file protection of <644>.
A MODIFY statement can be added after the BUILD statement (and after the indexes are created) to change a file's protection. Example, MODIFY "filename <666>"

Protection on Windows NT

This section is for those users using NTFS file system under Windows NT. File permissions other than read-only will be ignored unless the file is part of an NTFS (Windows NT) file system.

File permissions can be specified when building or modifying a file. The file permissions can be specified using Unix octal notation, IRIS octal notation, BITS permissions letters, or the new Win32 access control list format. The example below builds a file which can be read by any user, but can only be modified by the users "Fred" and "Alex":

Build #1,"<+Fred(rw)+Alex(rwpo)+everyone(r)>testfile"

An access control list consists of one or more entries. Each entry begins with a plus "+" sign which is followed by the user or group name and then one or more permissions letters surrounded by parentheses. The permissions letters used by the access control list format are:

"r" read access allowed
"w" write access allowed
"x" execute access allowed
"p" change permissions access allowed
"d" delete allowed
"o" change ownership access allowed

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.

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}
- ❑ 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
- ❑ 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
- ❑ 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 an 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.

dL4

Training Class

Data Files

CONVERTING EXISTING DATAFILES TO UNIVERSAL FORMAT

- ❑ Universal formatted files are platform independent (files can be copied to any supported Unix and Windows platform)
- ❑ Universal formatted files are read/write accessible by Unibasic and dL4. (The character set used is IRIS ASCII/IEEE BCD, allowing ASCII and basic Unibasic mnemonics, no European special characters.)
- ❑ To convert non-BCD (BITS) files to Universal format, a utility called ctool is used. Record definitions must be provided to use this utility.
- ❑ To convert IRIS BCD (Unibasic) files to Universal format, a utility called ubconvert is used. (If on query <Q> bit is set.
- ❑ Portable format was available prior to Universal format, which was platform independent, but based on the more limited ANSI character set (no Unibasic mnemonics).
- ❑ The most extensive character set available for dL4 would be the "UTF-8" Unicode character set.

SUPPORT FOR HUGE FILES

The maximum size of the index portion of Portable and Universal Indexed-Contiguous files has been increased to 256 gigabytes on Unix systems that support files larger than 2 gigabytes. The size of the data portion is limited only by the amount of storage space available. To use this feature a file must be created as a "Portable Huge Indexed-Contiguous" or "Universal Huge Indexed-Contiguous" file. Example:

Build #1,"[1:40]File" As "Portable Huge Indexed-Contiguous"

Currently, huge files are supported on UnixWare 7, AIX 4.3 and RedHat 7. They are not supported on SCO OpenServer because OpenServer does not allow user files to be larger than 2 gigabytes. In order to build huge files on UnixWare 7 or AIX, the file system must be configured to support files larger than 2 gigabytes. Please see the UnixWare or AIX OS documentation for instructions on how to configure file systems. To share huge files across a network, both the file server and the client systems must support sharing files larger than 2 gigabytes in size.

RECORD LOCKING

- ❑ Record locking works across most commonly used network software
- ❑ A record should be read with lock and should remain locked while its various fields are being updated
- ❑ Except for Text and Rawfile drivers, any read or write operations will obtain a record lock
- ❑ If the read or write operation end in a semicolon (;), then the record lock is released after the read or write is performed
- ❑ If a timeout value is not specified, the program execution is suspended until the record becomes available or until the program execution is interrupted
- ❑ If a timeout value is specified, the program execution is suspended until either the record becomes available or the program times out
- ❑ An error is generated if a timeout value is specified and the record is not available within the timeout period
- ❑ At most one record can be locked on any given channel
- ❑ Attempting to lock a second record on a channel unlocks any previously locked record i.e. any read or write will release any previously existing lock
- ❑ A different record can be locked on each channel when the same file is opened on multiple channels
- ❑ Attempting to lock the same record on different channels when the same file is opened on multiple channels is O/S dependent. This is strongly discouraged
- ❑ the BASIC OPEN statement has options to prohibit record locking
- ❑ SWAP and SWAPF statements no longer fork and thus it is the same process. Therefore, all the rules above apply to record locking in a SWAP or SWAPF program. The usage of the fork system call was eliminated since the fork system call is not available on Windows systems.

AUTOSELECT DRIVER

- ❑ dL4 uses the autoselect driver to open a file when the “As” specifier is not used
- ❑ The driver may first open a file and read the file header to determine the file type
- ❑ After determining the file type, it closes the file and relinquishes itself to the proper driver
- ❑ A file’s access time is updated since the driver accesses the file’s header

DIRECTORY DRIVER

- ❑ Used to open or create a directory
- ❑ Allows reading from the directory
- ❑ Does not allow writing to the directory
- ❑ Only sequential access
- ❑ Random access not available
- ❑ Can rewind to the beginning by reading record number zero
- ❑ Returns only filenames
- ❑ Open the file and use CHF or other functions to get specific file information

Examples:

```
open #0,"c:\\dl4"  
read #0;a$  
read #0,0;a$                                ! rewind to beginning  
build #1,"c:\\dl4training" as "directory"    ! create dl4training directory
```

- ❑ Driver, "Sorted Directory", is available to provide a sorted directory listing. The driver is otherwise identical to the normal directory driver. The driver is used by opening a directory with an AS clause.

Example:

```
Rem List the current directory contents in sorted order  
Dim F$(255)  
Open #1, "." As "Sorted Directory"  
Do  
    Read #1;F$  
    If F$ = "" Exit Do  
    Print F$;  
Loop  
Close#1
```

RAWFILE DRIVER

- ❑ May be used to open a device
- ❑ Allows functionality for UniBasic RDREL and WRREL statements
- ❑ Does not provide record locking on a READ or a WRITE statement
- ❑ Records can be locked using dL4 Channel statements
- ❑ Raw Regular File driver is used to open a disk file
- ❑ Raw File driver can open both a device and a disk file
- ❑ Does not provide character set support
- ❑ Strings are converted as 8-bit binary data in the least significant bits (lsb)
- ❑ Most significant bits (msb) are set to all zeros
- ❑ Must specify record length on an OPEN statement

Example:

```
!open a file called rawfile using the Rawfile driver. Record  
length=5 words
```

```
!Note that the record count is ignored by the Rawfile driver
```

```
open #0, "[100:5] rawfile" as "raw"
```

TEXT DRIVER

- ❑ No record locking is available
- ❑ Can specify a character set on an OPEN statement
- ❑ Cannot specify a character set on a BUILD statement
- ❑ Default character set is UniBasic ASCII
- ❑ Can create a file with a different character set by first creating the file, then closing and opening the file with character set option
- ❑ Can read a UNIX, DOS or a Macintosh format transparently
- ❑ A line of text is read from the file and it is appended with a carriage return character
- ❑ Builds a file for native O/S format, e.g. DOS, UNIX, etc.
- ❑ Can build a DOS format file by specifying AS "DOS Text"
- ❑ Can build a UNIX format file by specifying AS "UNIX Text"
- ❑ Can build a Macintosh format file by specifying AS "Macintosh Text"
- ❑ Can position within a text file using the SETFP #c,p1p2; command, where p1 is record and p2 is byte-displacement, assuming 512-byte records. Thus SETFP positioning is interpreted as $p1*512+p2$.
- ❑ The record number -4 has been defined in the text file driver to position to the end of the file. A program can append to a text file with a statement such as :

 PRINT #1,-4;"message"
- ❑ The driver, "ANSI Text", will create and access text files that uses the ANSI character set. Using the driver is equivalent to opening a text file with the "charset=ansi" option. This can be set as the default using the DL4DRIVERS environment variable.

PIPE DRIVER

- ❑ This is not specifically a printer driver, but is typically used to access printers
- ❑ Output pipe is opened with a single leading \$
- ❑ Input pipe is opened with double leading \$\$
- ❑ Bi-directional pipe is opened with no leading \$ or \$\$\$. A good example of using the bi-directional pipe is to perform ftp within a program.
- ❑ Finds the command using PATH if a relative filename is specified
- ❑ An error is generated if the command does not exist
- ❑ Character set and lock options may be specified as the first line in a script or a batch file
- ❑ Character set option can also be specified on an OPEN statement
- ❑ The script or batch file takes precedence if a character set exists in the file and if the character set option is also used in the OPEN statement
- ❑ Default character set is UniBasic ASCII character set
- ❑ The lock option in the script or the batch file prevents concurrent open
- ❑ The lock option specifies the absolute filename of a lock file to be built by the driver on a UNIX system
- ❑ UNIX users may have to manually remove the lock file in the event of a system crash
- ❑ The lock option specifies record locking in the form of TRUE or FALSE on dL4 for Windows
- ❑ The default lock option is set to none on both UNIX and Windows, meaning allow concurrency
- ❑ If an OPEN statement uses a "\$" (output pipe) or "\$\$" (input pipe) filename to open a script then the name of the current program will be passed to the script in the environment variable "DL4PROGRAM". The script can then use the program name to control printer spooling or special handling.
- ❑ The CLOSEWAIT option sets the time in seconds that the driver will wait for the child process to exit before the channel is closed. The option "CLOSEWAIT=0" will not wait at all. The pipe driver does NOT terminate the child process when the CLOSEWAIT period expires; the driver simply closes the channel and leaves the child process running. If not set the channel will remain open indefinitely or until closed.
Example: Open #1,"(closewait=120)\$\$script.sh"
- ❑ The option, "binary=true" option, will cause all data written to or read from the pipe will be passed as 8-bit binary characters without any formatting or end-of-line processing. Example: Open #1,"(binary=true)\$program"

- ❑ The pipe driver ('OPEN #1,"\$xxxx"') accepts multiple "# dL4opts=" lines at the beginning of a shell script. This makes it easier to specify multiple pipe driver options. Each option line can contain multiple options, but an individual option cannot span lines.

Driver examples (see Installation Guide, Printer Configuration for more details) :

```
0 dL4 for UNIX:
  # dL4opts=charset=utf-8,lock=/tmp/lpt1.lk

0 dL4 for Windows:
  rem dL4opts=charset=utf-8,lock=true
```

Bi-directional Pipe Driver example program :

```
Open #1,"/bin/sh" As "Bidirectional pipe command"
Print #1;"date"
Read #1;L$
Print L$
```

PROFILE DRIVER

Overview

A driver within the *Profile* class provides facilities to process textual data stored within specially organized text files. Such special text files are called *profile files*.

Some examples of *profile files* include Posix tty terminal description files and Full-ISAM Bridge Profiles. *Profile files* are well suited for the organization and retrieval of configuration information and are easily maintained by a text-editor or word-processor. *Profile files* are a portable feature of dL4.

A *profile file* contains one or more lines of text in the following general forms:

- blank lines
- ; Comments
- [Section Name]
- left=right

where *blank line* is any empty line.

; specifies a comment line.

Blank lines and comment lines are ignored during normal processing of the file.

Section Name is any set of valid characters stored within []. All lines following a *Section*, up to either the end-of-file or the start of [*Another Section*] are considered part of the named *section Name*.

left is any label, terminating with an '=' whose leading and trailing spaces are ignored.

right is any text, including spaces, to be returned as the value of the label specified by *left*.

The *Profile Driver* supports the following operations: **OPEN**, **ROPEN**, **READ**, and **SEARCH**.

Accessing a profile file

In order to access a text *profile file*, it must first be opened for read-access.

Synopsis: OPEN #chan, "profilefile" AS "Profile"

Where *chan* is any valid channel number, *profilefile* is any valid *Profile file*. The "AS Profile" clause is required to prevent the dL4 autoselection mechanism from opening the file as a standard text file.

Profile files are designed to be read-only. The current implementation does not support writing to the file. In addition, the driver does not support any locking mechanism, due to the read-only implementation.

Following a successful **OPEN**, an initial **SEARCH** must be performed prior to using **READ** to access data.

```
Open #0, "/usr/lib/dl4/term/wyse50" As "Profile"
Dim a$(100), b$(100), c$(100)
Search #0; a$           !Position to the first section
Print a$               !Display the name of the section
Read #0, -2; a$, b$     !Read the first record
Print a$; b$           !Display the items
```

Reading profile data

Synopsis: `READ #chan, record, item; svar {, svar}`

Where *record* (-2) specifies the current record, and (-1) selects the next record. Following a **SEARCH** operation, an initial *record*=-2 read is required to load the current record. Subsequent records are read by specifying record -1.

item selects the item to read, 0 or 1. 0 selects the *left* label, with leading and trailing spaces removed, and 1 selects the *right* label, with all data to the right of the = .

svar is the name of any *string* variable into which to read data.

Only the Current (-2), or Next (-1) record may be accessed. A Record Not Written error occurs at the end of a *Section*, or physical end-of-file.

Searching profile data

Synopsis: `SEARCH #chan; svar`

Synopsis: `SEARCH #chan; svar1, svar2, svar3`

The first form is used to perform a search for a specific *section name*. *svar* contains the named *Section Name* to locate. The search operation is case insensitive. If *svar* is null, a search is performed forward to the next *section*. *svar* is returned with the name of the located *section*, if any. An error is returned if the *section* is not found, or the end-of-file was reached with a null string search.

The second search form is used to locate a specific *left* label within a *section*. *svar1* contains the named *section*, *svar2* the *left* label, and *svar3* is returned with the *right* value.

If the operation is successful, the file is positioned to the first element of the named *section*. An error occurs, if the named *section* is not within the file. The

search operation is circular, that is, the file is searched from the current position forward. If an end-of-file is reached, the search continues from the beginning of the file up to the current position. A search for the next *section* is not circular in nature, resulting in an end-of-file error.

FULL-ISAM DATABASE FILES

Full-ISAM database files are designed to offer the developer a powerful alternative to Indexed files. Some of the immediate benefits include:

- Providing a structured approach to data storage and retrieval.
- Access to a file is field-oriented using named fields.
- Indices are maintained automatically and may be added and deleted as required.
- Fields may be expanded, added or deleted with little or no programming.
- Directly accessible by industry-standard third-party applications and programming languages.
- Capable of supporting a number of underlying data-base engines without reprogramming.

Full-ISAM database files represent a new class of object with which applications may interact. An extensive set of language components, interface and statements are included for applications {and drivers} supporting Full-ISAM files.

- Record access to Full-ISAM files is field-oriented and operates on similar principles as do formatted files. Each field has an associated type and an error results should an application attempt to read or write the wrong type of data. Fields are numbered, starting at zero.
- Include a data dictionary which defines field names, types and sizes. Access to a given field is performed by specifying it's item number, or alternately a structure variable which may be mapped by field name to the dictionary definition.

When operating on Full-ISAM files, the application is responsible for adding, deleting, reading and writing records. Record allocation/deallocation and key maintenance is performed by the file structure. For the designer, it is no longer necessary to modify applications when adding a new index to the file, or when changing the size of a data field.

Data fields may be added to or deleted from a file with little or no rewriting of application code.

Full-ISAM database files rely extensively on the use of *structure* variables. They are the preferred method of communication with the file structure.

```
! Define a structure Customer with 8 members, two of which are
! themselves structures.
Def Struct ExtraInfo=1%,X,Y,Z
Def Struct Test1 = Q$(20),%1,R,S
Def Struct Customer                                ! Define the customer structure
  Member Name$(25)                                ! ie larger structures.
  Member Address$(25)
  Member City$(25),State$(2),Zip$(10)
  Member 3%,Balance
  Member Xtra. As ExtraInfo                        ! Member is another structure
  Member T. As Test1                              ! Member is another structure
End Def
! Dimension B. as an array of 11 Customer structures (0-10)
Dim B.[10] AS Customer
```

The names of structure members are distinct from any other names outside the structure; e.g. A.Q\$ is distinct from Q\$ which is distinct from B.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. For example:

```
Def Struct Test = Q$(20),%1,R,S
Dim A. As Test
Write #1;A.                ! This WRITE executes exactly
Write #1;A.Q$,A.R,A.S      ! like this one
```

Indeed, many older-style statements which operate upon a fixed number of parameters may 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 and Status variables may be passed within a structure.

```
Def Struct SearchVar
  Member Key$(100)
  Member %3, RecordNumber
  Member %1, StatVar
End Def

Dim Key. As SearchVar
Search = #channel, index; Key.
Select Case Key.StatVar
Case 0:
  ! Success
Case Else
  ! Failure
End Select
```

Using 'item' Designations in Structure Variables

Structure elements may also declare an 'item number' or 'key option' assignments, for use by file drivers, associated with each member. While most statements ignore the item designation, file drivers utilize such information for record definitions and positioning.

```
Def Struct Customer                ! Define structure for Indexed file
  Member Name$(25)      : ITEM 0    ! supplying byte displacements.
  Member Address$(25)   : ITEM 25
  Member City$(25)      : ITEM 50
  Member State$(2)      : ITEM 75
  Member Zip$(10)       : ITEM 77
  Member %3,Balance     : ITEM 100
End Def
```

The above example might be used to access older style contiguous or indexed data files. To access database files, the same structure definition may define items using 'fieldnames', such as:

```
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
```

Directories may also be defined and managed using structure definitions. By defining the named key CustKey as a unique, packed directory, one can define a structure as follows:

```
Def Struct CustKey      : KEY "NameCtyBal" + Unique + Packed
  Member Name$(25)      : KEY "Name" + Ascending + Uppercase
  Member City$(25)      : KEY "City" + Ascending
  Member 3%,Balance     : KEY "CurrBal" + Descending
End Def
```

Training Note: A colon (:) is used to separate ITEM, KEY and DECIMALS specifiers. Multiple ITEM or KEY specifications are concatenated with a +. Key member fieldnames MUST be defined and MUST match the name of the fields that make the key in the database. In the example above NameCtyBal is the name of the key itself in database.

The **VARLEN** option can be used with Full-ISAM files to create variable length or memo fields instead of fixed length character fields. An example would be :

```
Member Comment$(100) : ITEM "Comment" : Varlen
```

Building a Full-ISAM Database File

Creating a Full-ISAM file is performed by first building the file, followed by the definition of the record layout and indices. The **Build** statement is used to create a Full-ISAM database file. The General form of the **Build** statement for this class of object is:

Build #*channel*, *filename* **As** "Full-ISAM"

Build #*channel*, *filename* **As** "FoxPro Full-ISAM"

channel is any numeric expression which, after evaluation is truncated to an integer specifying an unopened channel on which to build a new Full-ISAM database file.

filename is any *filename* expression including the name of the file.

The string given in an **As** clause is interpreted either as a driver-class name or a specific driver-description, whichever is found first in the main driver table. When a specific driver is desired, it should be specified. Otherwise, specification of the class only results in the selection of the default driver assigned to the class.

If no error occurs, the file is created. The following parameters outline the capabilities of the FoxPro compatible Full-ISAM database driver supplied with dL4.

```
!   MAXIMUM LENGTH OF FIELD NAME = 10 CHARACTERS (10 is correct for FoxPro)
!   MAXIMUM NUMBER OF FIELDS PER RECORD = 255
!   MAXIMUM LENGTH OF A CHARACTER FIELD = 32767 CHARACTERS
!   MAXIMUM NUMBER OF DIRECTORIES = 47
!   NUMBER OF DECIMAL PLACES IN NUMERIC FIELDS IS REQUIRED
!   RECORD NAME PARAMETER IS IGNORED
!   BINARY FIELDS NOT DEFINABLE IN BASIC
!   KEY PART OPTIONS ALLOWED:  UPPERCASE (FOR STRING FIELDS)
!                               NUMBER OF DECIMALS (FOR NUMERIC FIELDS)
!   DIRECTORY OPTIONS ALLOWED:  DESCENDING SEQUENCE
!                               DUPLICATES ALLOWED
```

Defining a Full-ISAM Record Definition

The **Define Record** statement is used to establish the record definition and data dictionary of a newly built Full-ISAM database file. The general form is:

Define Record # *channel* ; *structvar*

channel is any numeric expression which, after evaluation is truncated to an integer specifying an opened channel with a newly built Full-ISAM data file.

structvar is the name of a structure variable including **Item** "Fieldname" specifications for each member of the structure template.

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"
End Def

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

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

Adding an Index to a Full-ISAM File

Indices may be added and deleted to a Full-ISAM file at any time. The process of defining an index requires defining a structure which identifies the various parts of the key. The general form is:

Add Index # *channel*, *index*; *structvar*

channel is any numeric expression which, after evaluation is truncated to an integer specifying an opened channel with a newly built Full-ISAM data file.

index is any numeric expression which, after evaluation is truncated to an integer and used to select an unused index (directory) number within the opened Full-ISAM database file.

structvar is the name of a structure variable including **Key** "Definition" specifications for each member of the structure template.

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

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


```

Def Struct CustKey1      : KEY "NameCtyBal" + Duplicates + Packed
  Member Name$[25]       : KEY "Name" + Ascending + Uppercase
  Member City$[25]       : KEY "City" + Ascending + Uppercase
  Member 3%,Balance      : KEY "CurrBal" + Descending
End Def
Dim Key1. As CustKey1
Add Index #5,1;Key1.

```

In this example, the structure CustKey1 is named "NameCtyBal" and represents an index of possibly duplicate keys packed to save space within the file.

The member Name\$ is a 25-character string from the data field with the same name. It is to be uppercased and stored in ascending order. The field City\$ is a 25-character string from the data field with the same name. It is also to be uppercased and stored in ascending order. The last part of this key, Balance, is a 3% numeric field from the field named "CurrBal" which is to be collated in descending order.

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

Deleting an Index from a Full-ISAM File

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

Delete Index # *channel*, *index*;

channel is any numeric expression which, after evaluation is truncated to an integer specifying the channel of an opened Full-ISAM data file.

index is any numeric expression which, after evaluation is truncated to an integer and used to select an existing index (directory) number within the opened Full-ISAM database file which is to be deleted.

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

Aligning a Structure to a Full-ISAM File

Often it is necessary to work with a subset of fields within a database or provide for later changes in the field order within the file. The **Map Record** statement allows a program to 'marry' a structure definition to the current file's data dictionary. The general form is:

Map Record #*channel* **As** *struct*

channel is any numeric expression which, after evaluation is truncated to an integer specifying the channel of an opened Full-ISAM data file.

struct is the name of a template **Def Struct** structure definition which is to be aligned with the fieldnames of the database. *struct* members must have **Item** fieldname 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.

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, e.g., FoxPro, etc.), this may circumvent unnecessary data conversion and thereby boost performance.

An example of Map Record is presented at the end of this section.

You can also map a key name using the syntax `Map #channel,keyno;"keyname"`,
For example, `Map #1,1;"CUSTID"`
Which will map the key named "CUSTID" to key 1, even if it is not truly key 1 in the file.

Adding a new Record to a Full-ISAM File

A new record is added to a Full-ISAM file using the **Add Record** statement. The general form is:

Add Record *#channel ; structvar*

channel is any numeric expression which, after evaluation is truncated to an integer specifying an opened channel with a newly built Full-ISAM data file.

structvar is the name of a structure variable containing the new record.

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.

Deleting a Record within a Full-ISAM File

A record may be deleted from a Full-ISAM file using the **Delete Record** statement. The general form is:

Delete Record *#channel ;*

channel is any numeric expression which, after evaluation is truncated to an integer specifying an opened channel with a newly built Full-ISAM data file.

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.

Locating Records within a Full-ISAM File

To access Full-ISAM files, the **Search** statement is used 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.

When performing a search operation on a Full-ISAM file, the arguments to the **Search** statement represent the parts of the selected key, rather than the familiar "<key\$>,<record>,<status>". A structure, such as the one used to actually create the index, can also be used; supplying a structure is equivalent to explicitly supplying each of its members.

```
Def Struct CustKey1      : KEY "NameCtyBal" + Duplicates + Packed
  Member Name$(25)       : KEY "Name" + Ascending + Uppercase
  Member City$(25)       : KEY "City$" + Ascending + Uppercase
  Member 3%,Balance      : KEY "CurrBal" + Descending
End Def
Dim Key. as CustKey1
Key.Name$ = "Acme" ; Key.City$ = "Toledo" ; Key.Balance = 0
I = 1
Search = #C, I; Key.      !Exact search (NO PARTIAL MATCH!)
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
```

Note: You must MAP the key # to the actual key name, see previous page.

KEY definitions in a key structure are only needed if using the structure to build the file.

You do not have to use a structure variable to do the search, the Search statement can be a variable list of the key parts.

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 be 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.

```
Def Struct Key           ! Old-style Key structure
  Member Key$(20)        ! Contains string for Key
  Member 3%,V1           ! V1 for record number
  Member 1%,V2           ! V2 for returned status
End Def
Dim K. AS Key
SEARCH = #1,1;K. \ IF K.V2 ... ! etc.
```

Managing Records within a Full-ISAM File

The management (changing) of data records within a Full-ISAM database file is accomplished by simply reading and writing a record. The indices are updated automatically.

The general forms are:

Read Record # *channel* , *record* {, *item* {, *timeout* } } ; *structvar*

Write Record # *channel* , *record* {, *item* {, *timeout* } } ; *structvar*

channel is any numeric expression which, after evaluation is truncated to an integer specifying the channel of an opened Full-ISAM data file.

record is any numeric expression which, after evaluation is truncated to an integer specifying a numbered record or record selection choice. *record* may select an actual record number for dL4 files or may be specified as -1 or -2. Full-ISAM files may only select one of the following:

- 1 Read next (ascending) record
(relative to the index ordering of index last searched)
- 2 Read current record
- 3 Read previous (descending) record.

item is any numeric expression which, after evaluation is truncated to an integer specifying the item number or *byte displacement* within the record to begin the transfer.

timeout is any numeric expression which, after evaluation is truncated to an integer specifying the number of tenth-seconds to wait for a record which is locked.

structvar is the name of a structure variable the contents of which is to be read or written.

The **Read** and **Write Record** statements are similar to normal **Read** and **Write** of a record except for the requirement that a *structvar* is supplied and the computation and override of the item number for each member.

The first 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 relative
  Member 3%, Credit : ITEM 6 !to where we begin a transfer
End Def
Def Struct Cust
  Member Number$(8) : ITEM 0
  Member Name$(30) : ITEM 10
  Member Addr$(30) : ITEM 42
  Member Balance. As DRCR : ITEM 74
  Member 1%,LastOrderNumb# : ITEM 86
End Def
Dim Customer. As Cust
Write Record #c,r,b,t;CUSTOMER. ! identical to:
Write #c,r,b+0,t;Customer.Number$
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.

The following example defines the same sample structure using item 'fieldnames' instead of numbers. In this case, or whenever ITEM clauses are not supplied in a structure definition, the item designation is equivalent to the member number, i.e. sequential from zero. Full-ISAM file access is provided by supplying an item number, therefore a structure to be used for accessing such files must define the items in the order that they exist in the file.

To provide an even greater degree of database-style flexibility, the **Map Record #** statement can be used to align a defined structure with an open file. Most applications would be wise to use this statement upon opening all Full-ISAM files to "marry" the current file definition to the expected structure. In this way, changes to the order of fields, or the addition of new fields, will have minimal impact on existing application code.

```
Def Struct DRCR
  Member 3%, Debit   : ITEM "Debit" ! By Field Name
  Member 3%, Credit : ITEM "Credit"
End Def
Def Struct Cust
  Member Number$[8]           : Item "Customer Record"
  Member Name$[30]            : ITEM "Number"
  Member Addr$[30]            : ITEM "Name"
  Member Balance. As DRCR     : ITEM "Addr"
  Member 1%,LastOrderNum#     : ITEM "Balance"
End Def
Dim Customer. As Cust
Open #5, "Customers" As "Full-ISAM"
Map Record #5 As Cust
Search > #5,1;
Read Record #5,-2; Customer.      !Read entire structure
Write Record #5,-2; Customer.
```

Training Note: In the example above, Item names Debit and Credit would be **appended** to Item name Balance and then in the case of FoxPro truncated to 10 characters. In FoxPro, the fieldnames would be BalanceDeb and BalanceCre.

To determine FoxPro file field names you can use the query utility in the tools directory, for example:

```
#/usr/lib/dl4/tools/query customers.dbf
```

When used in conjunction with Full-ISAM and **Search**, the application performs an initial **Search** and reads the current record. Specific sets of records can then be processed by reading/writing the next or previous record. A loop, such as **While** or **Do** could be used to traverse the file.

The use of **Read Record** and **Write Record** on older-style indexed files will still rely on the paired operations of **Search** and **Read Record** or **Write Record** . A file is traversed using **Search** mode 3 or 6 or > or < (next or previous) followed by a **Read Record** with a returned record number.

Simple Example

How to do a sequential search

Old Unibasic method :

```
20 SEARCH #1,3,1;V$,V1,V2
```

```

30 IF V2 GOTO 60
40 READ #1,V1; fields
50 GOTO 20
60 REM etc

```

New way with DO/LOOP is to position to a location in the file and then do sequential reads :

```

Try
  Search > #1,1;
  X = -2
Do
  Try Read Record #1,X;record. Else Exit Do
  Etc
  X=-1
Loop
Else
End Try

```

Space filled fields

Full-ISAM files typically will automatically right space fill defined fields.

An open option, "RTRIM=<boolean>", is available for Full-ISAM drivers. If the option "RTRIM=TRUE" is used, all trailing spaces are removed when reading fields. The option is case insensitive and an argument of "T" is identical to "TRUE".

Example:

```
Open #1,"(rtrim=t)test.dbf"
```

dL4 BRIDGE DRIVER

- ❑ A new driver in the Indexed file class
- ❑ Designed to provide developers gradual incorporation of Full-ISAM
- ❑ Transparent to applications
- ❑ Gives restricted access to a Full-ISAM file as if that file were Indexed
- ❑ Simplify the transition into non-proprietary database systems
- ❑ **No immediate reprogramming – use MS-SQL files with existing code!**
- ❑ **Adapt on a file by file basis rather than program by program**
- ❑ Develop new Full-ISAM applications over time
- ❑ Interface immediately with industry-standard tools
- ❑ **Ability to map to any underlying Full-ISAM database**

REQUIREMENTS WHEN USING THE BRIDGE DRIVER

- ❑ Emulated Indexed File must have a single, fixed record layout
- ❑ All indices must be balanced (there is a variation to this)
- ❑ **Each directory must have one and only one key per record (there is a variation to this)**
- ❑ Data fields must be numeric or character
- ❑ Other types (packed) not supported
- ❑ Can not link or chain records by record number (there is a variation to this)
- ❑ Character fields cannot have significant data past first null

ACCESSING AN EMULATED INDEXED-CONTIGUOUS

- ❑ Cannot BUILD a Full-ISAM file using the Bridge driver; A normal Indexed-contiguous file is always BUILD by default
- ❑ File operations on a given channel must be grouped into consecutive operations on a single record - a Transaction
- ❑ Transactions begin with a SEARCH which returns a record number
- ❑ Transactions end with either:
 - © All indices balanced and consistent values present between key and record fields, where applicable.
 - © All keys removed and the associated record deleted.
- ❑ All indices balanced and consistent values present between key and record fields, where applicable
- ❑ All keys removed and the associated record deleted
- ❑ Record numbers are not physical within the Full-ISAM file, thus the driver generates a 'pseudo' record number which may vary for the same record
- ❑ Record numbers are valid within a single transaction
- ❑ Record numbers may vary the next time the same record is accessed

BRIDGE PROFILE - "DATA DICTIONARY"

- ❑ Maps certain Full-ISAM fields to selected byte displacements of an emulated contiguous file
- ❑ Maps certain Full-ISAM fields to selected parts of emulated Index keys
- ❑ Dictionary stored in a text "profile" file, termed a Bridge Profile
- ❑ Profile is typically stored under the same name as the emulated Indexed File so that Open statements need not be modified. However the profile file can have any filename. The Open statement syntax is simply, OPEN #1,"*profile filename*"
- ❑ Application opens a profile believing it to be an Indexed file (open statement is simply Open #1,"profile filename")
- ❑ Profile contains the filename, and optional driver, of underlying Full-ISAM database file

ISAM Bridge Profile

```
[FullISAMBridge]

; Presence of [FullISAMBridge] shall be used to trigger driver
; auto-selection
File=sample.dbf

; format for MS-SQL file would be
; File= (user=xxx, pswd=xxx) servername:databasename.tablename
; OpenAs=... may not be necessary, depending on the database in question
OpenAs=FoxPro Full-ISAM
; or OpenAs=Microsoft SQL Server Full-ISAM


; Sample bridge profile for Full-ISAM Bridge driver
; Entries for fields look like:

;   Field=<fldnam>,<pos>,<fmt>,<align>,<fill>,<opts>
;   <fldnam>  Name of field in Full-ISAM file.
;   <pos>     Byte displacement in emulated idx-ctg record or key.
;   <fmt>     Character length for strings, precision number for numerics.
;   <align>   "L" to left-align string data (default).
;   "R" to right-align string data.
;   <fill>     fill character to use if string field sizes
;             mismatch,
;   default is " ".
;   <opts>    various option keywords separated by "+".  Current options
;             are:
;             "strip" Strip trailing spaces from string fields after
;                   reading them
;                   from the Full-ISAM file.

[Record]

Field=NAME,0,25

Field=ADDRESS1,25,25

Field=ADDRESS2,50,25
```

```

Field=CITY,75,15
Field=STATE,90,2
Field=ZIPCODE,92,5
Field=BALANCE,98,4%
Field=CREDTLIMIT,106,4%
Field=ORDER,114,6
Field=INVOICE,120,6

```

; Entries for keyparts look like:

```

;      KeyPart=<fldnam>,<pos>,<fmt>,<prefix>,<charset>,<opts>
;      <fldnam>      Name of field in Full-ISAM file.
;      <pos>         Byte displacement in emulated idx-ctg record or key.
;      <fmt>         Character length for strings, precision number for
;                   numerics.
;      <prefix>      prefix character strings to use to generate random
;                   unique key fields when a new record is allocated.
;      <charset>     character set to use to generate random unique key
;                   fields when a new record is allocated.
;                   e.g "0123456789" will generate numeric values.
;      <opts>        Various option keywords separated by "+".  Current
;                   options are:"strip" - Strip trailing spaces from
;                   string fields after reading them from the file.

```

[Index1]

Name=BYORDER

KeyPart=ORDER,0,6,"zz","abcdefghij"

[Index2]

Name=BYSTATE

KeyPart=STATE,0,2,"","0123456789"

KeyPart=ORDER,2,6,"zz","abcdefghij"

[Index3]

Name=BYINVOICE

KeyPart=INVOICE,0,6,"zz","abcdefghij"

Training note :

Explanation of <prefix> and <charset>. Sometimes the bridge driver needs to create a temporary key in the FULL-ISAM file until all the fields are written to the file. These parameters are used to help create a temporary key that will be unique (not duplicating a possible existing key). In example above, Index 1 will create a unique temporary key beginning with 'zz' and the remaining 4 characters will be generated with the letters a through j.

Sample Record Layout

File: Cust.Master

Type: Indexed, Record Length: 140 bytes

Directory 1: Key Length 6-characters "Customer Number"

Fieldname	Name\$	Address1	Address2	City	State	Zip	Phone	Date	Ytd Sales	Last Yrsales	Custnum
Offset	0	25	50	61	86	89	94	109	116	120	128
Variable Type	T\$ [24] String	T1\$ [24] String	T2\$ [10] String	T3\$ [24] String	T4\$ [2] String	T5 2%	T6\$ [14] String	T7\$ [6] String	T8 2%	T9 4%	T10\$ [6] String

Sample Bridge Profile

[FullISAMBridge]

File=custmasterfi.dbf

OpenAs=FoxPro Full-ISAM

[Record]

Field=NAME,0,24

Field=ADDRESS1,25,24

Field=ADDRESS2,50,10

Field=CITY,61,24

Field=STATE,86,2

```
Field=ZIP,89,2%
Field=PHONE,94,14
Field=DATE,109,6
Field=YTDSALES,116,2%
Field=LASTYRSALE, 120, 4%
Field=CUSTNUM, 128, 6
```

```
[Index1]
```

```
Name=BYCUSTNUM
```

```
KeyPart=CUSTNUM,0,6,"","0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

o A new option has been added to the Full-ISAM Bridge driver profile to support unbalanced indexes by allowing such indexes to be placed in separate Indexed-Contiguous or Full-ISAM files outside the main Full-ISAM file. For example, if index 3 of an Indexed-Contiguous file is used as a scratch index with entries for only some of the file records (and is thus unbalanced), that index could not be emulated by the Bridge driver within the main Full-ISAM file because Full-ISAM files do not support unbalanced indexes. Using the new option, the index can be emulated by declaring an external index in the bridge profile. For example:

```
[Index3]
File=filename
Index=1
KeyPart=name,0,10
```

This example directs the bridge driver to perform all SEARCH operations on index 3 by applying the SEARCH operations to index 1 of the Indexed Contiguous file "filename". The bridge profile would also have to use the "RealRecordNumbers" option described below so that the record numbers in the keys of index 3 could be used to reference records in the main Full-ISAM file. External indexes can have multiple keys for the same record or use multiple key formats. The data in the key is completely controlled by the application and does not need to be present in any of the fields of the main Full-ISAM file. An external index definition can only have one "KeyPart" entry and, if the external index file is indexed contiguous, the field name is ignored (but it must be specified).

External index files can either be Indexed Contiguous files or Full-ISAM files. If an Indexed-Contiguous file is used, the bridge driver will only access the index portion of the file. To use a Full-ISAM file as an external index file, the following format must be used in the bridge profile:

```
[Index3]
File=filename
Name=indexname
```

```
KeyPart=keyfieldname,0,10,,,Strip
RecPart=recnbrfieldname
```

where "filename" is the name of the external Full-ISAM file, "indexname" is the name of the index within the Full-ISAM file, "keyfieldname" is the name of the Full-ISAM field used for the key (this must be a character field), and "recnbrfieldname" is the name of the Full-ISAM field used for the record number (this must be a numeric field). External index definitions can include "Filename", "Protection", "Options", "OpenAs", and "OpenInProfileDirectory" entries using the same format and function as such entries in the main bridge profile section. If the file protection option is not specified, the protection options used to open the main Full-ISAM file will be applied when opening the external index file.

- o A new option has been added to the Full-ISAM Bridge driver profile to support programs reading records by record number. When set to TRUE, the option "RealRecordNumbers" causes the driver to return the actual Full-ISAM file record number when performing SEARCH statements that return a record number. The record numbers returned by SEARCH can then be used to perform random reads from the file. The option can only be used with Full-ISAM drivers and files that support searching index 0 for equal record numbers. Example:

```
[FullISAMBridge]
File=filename
OpenAs=FoxPro Full-ISAM
RealRecordNumbers=True
```

- o The Full-ISAM Bridge driver has been enhanced to support keys that contain the current record number when the Bridge driver is used with the MySQL Full-ISAM driver or the Microsoft Full-ISAM driver. In the Bridge profile, a record number segment of a key must reference the IDENTITY column of the Full-ISAM table and use the NTOC(), STR(), NTVNTOC(), or NTVSTR() functions to convert the IDENTITY column value to a character format. To use this feature, the option "RealRecordNumbers" must be enabled.

- o The Full-ISAM Bridge driver has been enhanced with two new conversion functions: STR() and NTVSTR(). These functions are similar to NTOC() and NTVNTOC(), but they use a simpler mask consisting only of spaces and a single "#" character. The "#" character is replaced with a default conversion of the number and any spaces are copied. The purpose of the STR() and NTVSTR() functions is to duplicate fields that are created by simple assignment or concatenation of numeric values:

```
TheKey$ = "Name", RecNo
```

Bridge profile example:

```
KeyPart=IDCOL,5,7,"","0123456789",STR("#")
```

- o A new option has been added to the Full-ISAM Bridge driver profile to support programs that accidentally span a record boundary when reading string values. When set to TRUE, the option "TruncateSpanning" disables the normal "Illegal item number" (error 53) error that is reported when a program reads a character variable whose length extends beyond the end of the record. The driver will treat such reads as if the read ended exactly at the end of the record. Example:

```
[FullISAMBridge]
File=filename
OpenAs=FoxPro Full-ISAM
TruncateSpanning=True
```

o The Full-ISAM Bridge driver has been extended to support Full-ISAM date fields and translation of field types. A date field is used by specifying a translation function in the field definition that defines how to translate a date value to or from a character or numeric field. Translation functions also support converting Full-ISAM numeric fields to Indexed-Contiguous character key fields. Additional functions support subscripted character fields for key fields or case-insensitive key fields. The translation functions are:

DTOC(mask) Convert the Full-ISAM date field to a character string in the record image using "mask". All dates use local date/time. When the DTOC() function is used in an index definition, the "mask" must define a sortable date. For example, "YYMMDD" is a legal index mask, but "MMDDYY" is not because it would not sort correctly. "mask" is a quoted string in which the following substrings have special meaning:

YYYY	Four digit year
YY	Two digit year with the century set so it is within 50 years of the current date.
AA	Two digit year in which years after 1999 are specified as "A0" through "E9".
MM	Zero filled month, 1 - 12
DD	Zero filled day of month, 1 - 31
DDD	Zero filled day of year, 1 - 366
DDDDD	Zero filled day relative to base year 1968. January 1, 1968 is "00001". Six or more "D"s can be also be used.
HH	Zero filled hour of day
MM	Zero filled minute of hour
SS	Zero filled second of minute

DTON(mask) Convert Full-ISAM date field to a decimal numbe in the record image using "mask". All dates use local date/time. "mask" is a quoted string defining the decimal digits of the number. The following substrings in the mask have special meanings:

YYYY	Four digit year
YY	Two digit year with the century set so it is within 50 years of the current date.
MM	Month, 1 - 12
DD	Day of month, 1 - 31
DDD	Day of year, 1 - 366 or 0 - 365
DDDDD	Day relative to base year 1968. January 1, 1968 is 1. Six or more "D"s can be also be used.
HH	Hour of day
MM	Minute of hour
SS	Second of minute

NTOC(mask) Convert Full-ISAM numeric field to a character string according to the USING mask "mask". The mask must use a period (".") for any decimal point and comma as any grouping separator.

NTVNTOC(mask) Convert Full-ISAM numeric field to a character string according to the USING mask "mask" and using locale information to determine the decimal point character. The mask must use a period (".") for any decimal point and comma as any grouping separator.

LEFT(len) Use the first "len" characters of the field. This function

can only be used in index definitions and the field must also be used in the "[Record]" section.

UCASE(len) Use the first "len" characters of the field converted to uppercase. This function can only be used in index definitions and the field must also be used in the "[Record]" section.

LCASE(len) Use the first "len" characters of the field converted to lowercase. This function can only be used in index definitions and the field must also be used in the "[Record]" section.

Examples:

```
Field=LASTPAYMNT,114,2%,,DTON("YYDDD")
KeyPart=DATE1,0,10,"","0123456789",DTC("YYYYMMDD")
KeyPart=NUM2,5,7,"","0123456789",NTOC("####.##")
```

o The Full-ISAM Bridge driver profile has been extended to support two conversion functions, IFNULL and IFERR, that convert SQL NULL or invalid values to and from the values needed in the emulated indexed contiguous file record. The IFNULL function takes a single string argument which defines a string or number that is stored into the converted field whenever a NULL is read. When a record is written, the same value will be converted to a NULL. Examples:

```
Field=COUNT,28,4%,,IFNULL("-1")
Field=ACCTID,16,12,,NTOC("-----#.##"),IFNULL("N/A"),Strip
```

In order to read or write NULLs, the SQL driver options in the bridge profile must be set to enable reading and writing nulls.

The IFERR function is similar to the IFNULL function, but it is applied to invalid numeric or date values. In this release of dL4, the only possible invalid value is the special MySQL date value of "0000-00-00". If no IFERR function is specified, an invalid value results in an error. In the following example, MySQL "0000-00-00" dates are converted to -1 while actual dates are converted to decimal numbers in the form "YYYYMMDD":

```
Field=DATE1N,60,3%,,DTON("YYYYMMDD"),IFERR("-1")
```

In this example, "0000-00-00" dates in a key part are converted to the string "00000000" while actual dates are converted to strings in the format "YYYYMMDD":

```
KeyPart=DATE1,0,8,"","0123",DTC("YYYYMMDD"),IFERR("00000000")
```

Both IFNULL and IFERR functions can be used in the same field or key part definition.

o The Full-ISAM Bridge driver profile has been extended to support filler fields. A filler field is any record field with a field name beginning with an asterisk. Filler fields are not read from the Full-ISAM file, but they are initialized to the fill character or zeroed if the fill character is not defined. The filler name is treated as a

comment.

Example:

```
Field=*fillerwithnulls*,54,5
Field=*fillerwithblanks*,59,5,, " "
```

o The numeric precision syntax in Full-ISAM Bridge profile files has been extended to specify the number of decimal places needed. The new, optional format is "p.d%" where "p" is the dL4 numeric precision (1-4) and "d" is the number of decimal places. Thus "3.2%" would specify a 10 digit floating point format with two decimal places. The decimal place information is used by the tools/ic2fi utility to create Full-ISAM files when the Full-ISAM driver supports only fixed point, rather than floating point, numbers. Example:

```
Field=COST,80,3.2%
```

o A new option has been added to the Full-ISAM Bridge driver profile to disable use of temporary record values when inserting or modifying records. The temporary values are normally used to reserve key values after SEARCH mode 4 insert statements and thus prevent other programs from inserting the same key. If the new "ProtectKeys=False" option is specified in the initial section of a bridge profile, SEARCH mode 4 statements will check for the current existence of a key value, but the Full-ISAM file will not be modified until all keys have been inserted for the record. This option improves Bridge driver performance and allows use of foreign key constraints in SQL tables. If the option is used, duplicate key errors may occur after a successful SEARCH mode 4 insertion of the key. Bridge profile example:

```
[FullISAMBridge]
File=filename
OpenAs=FoxPro Full-ISAM
ProtectKeys=False
```

Microsoft SQL Server Full-ISAM

- ❑ Available only on dL4 for Windows product
- ❑ A Full-ISAM interface to SQL Server Tables
- ❑ A SQL Server Table appears as a Full-ISAM file to a Basic program
- ❑ Can use the Bridge Driver to access the server because it is a Full-ISAM interface
- ❑ Can add, change and delete records in a SQL table
- ❑ **Do not use identity fields in tables. dL4 cannot write to identity fields**
- ❑ A table must have at least one index with unique key
- ❑ **The table to be opened must allow NULL values in all fields that are not used in keys**

- ❑ Cannot change database or table definitions. Must be done within SQL administration
- ❑ Cannot open database views
- ❑ Cannot issue SQL statements
- ❑ Cannot use auto selection
- ❑ Does not support record number, record size and file size channel functions (CHF)

Example

Here is an example program which reads a text file, creates a dL4 Foxpro file and then compares the Foxpro file to a MS-SQL file and updates the SQL file with any changes.

```
!      "DEMOSQL" === Program to create dl4 file of open orders and update SQL table
!
! *Declare dL4 Intrinsic Subs & Functions
Declare Intrinsic Sub ProgramDump
Declare Intrinsic Function FindChannel
!
! *Define file structures Def Structs
!
! dl4 file to be built from existing Unibasic files
! foxpro fieldnames limited to 10 characters!
Def Struct dl4orders      :Item "Orders"
    Member Orderstatus$[16]      :Item "OrderID"
    Member PMOrder$[10]          :Item "PMOrder"
    Member PMOrderline$[10]      :Item "PMLine"
    Member PMAccount$[10]        :Item "PMAcct"
    Member %3,Orderdate#         :Item "OrderDate"
    Member Productcode$[10]      :Item "Product"
    Member CustomerPO$[15]       :Item "CustPO"
    Member QtyOrdered$[10]       :Item "QtyOrdered"
    Member QtyShipped$[10]       :Item "QtyShipped"

End Def
!
Def Struct dl4ordersK1      :KEY "OrderID"
    Member Orderstatus$[16]    :KEY "OrderID"
End Def
!
! SQL file to update from dl4 file
```

```

Def Struct sqlorders
    Member Orderstatus$[16]
    Member PMOrder$[10]
    Member PMOrderline$[10]
    Member PMAccount$[10]
    Member %3,Orderdate#
    Member Productcode$[10]
    Member CustomerPO$[15]
    Member QtyOrdered$[10]
    Member QtyShipped$[10]

    :Item "Orders"
    :Item "Order Status ID"
    :Item "PM Order Number"
    :Item "PM Order Line Number"
    :Item "PM Account Number"
    :Item "Order Date"
    :Item "Product Code Number"
    :Item "Customer PO Number"
    :Item "Quantity Ordered"
    :Item "Quantity Shipped"

End Def
!
Def Struct sqlordersK1
    Member Orderstatus$[16]
End Def
!
! *Define Subs & Functions
!
!
!
! Name:
!     BuildFile() - Build dl4 file
!
! Synopsis:
!     BuildFile(dl4File$)
!     dl4File$ = filename to build
!     Builds dl4 file for current order status
!
! Returns:
!     >0: Channel #
!     -1: Error
!
External Function BuildFile(dl4File$)

    Dim %1,C0,S,%3
    Dim dl4orders. As dl4orders
    Dim dl4ordersK1. As dl4ordersK1

    !dl4 orders file
    !dl4 KEY 1

    C0=FindChannel()
    Build #C0,dl4File$ As "Full-ISAM"
    Define Record #C0;dl4orders.
    Add Index #C0,1;dl4ordersK1.
    Close #C0
    Open #C0,dl4File$ As "Full-ISAM"
End Function C0 !BuildFile
!
!
! Name:
!     ReadUnibasic() - Read Unibasic text file and write dl4 file
!
! Synopsis:
!     ReadUnibasic(C0)
!     C0=dl4 file channel #
!

```

```

!      Returns:
!      T=# of records created
!
External Sub ReadUnibasic(C0,T)

Dim %1,C1,C2,C3,S,%3
Dim Stri$[300],Tmp$[300]
Dim dl4orders. As dl4orders           !dl4 orders file
Dim dl4ordersK1. As dl4ordersK1       !dl4 KEY 1

C1=FindChannel()

ROpen #C1,"ordertextfile"

Print "Creating new open orders file"
Do
    Read #C1;Stri$
    If Stri$="" Then Exit Do
    Let tmp$=Stri$ To "|" : Spos
    Let dl4orders.PMOrder$=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.PMOrderline$=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.PMAccount$=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.OrderDate#=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.Productcode$=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.CustomerPO$=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.QtyOrdered$=tmp$
    Let tmp$=Stri$[Spos+1] To "|" : Spos
    Let dl4orders.QtyShipped$=tmp$
    Let dl4orders.orderstatus$=dl4orders.PMOrder$,dl4orders.PMOrderline$

    Add Record #C0;dl4orders.
    T=T+1           !count # of records to processed
Loop

End Sub !ReadUnibasic
!
!
!
!      Name:
!      DeleteSQL() - Read sql records and delete if no matching dl4 record
!
!      Synopsis:
!      DeleteSQL(C0,C9)
!      C0=dl4 file channel #
!      C9=sql file channel #
!
!      Returns:
!      D=# of records deleted

```

```

!
External Sub DeleteSQL(C0,C9,D)
!
Dim dl4orders. As dl4orders           !dl4 orders file
Dim dl4ordersK1. As dl4ordersK1       !dl4 KEY 1
Dim sqlorders. As sqlorders           !sql orders file
Dim %1,c9seq,%3
!
Print "Deleting closed orders in SQL"
c9seq=-2                               !read first rcd
Try
    Search > #C9,1;                    !position to first key of index 1 in sqlorders
file

    Do
        Try Read Record #C9,c9seq;sqlorders. Else Exit Do !loop thru sql file
        dl4ordersK1.Orderstatus$ = sqlorders.Orderstatus$
        Try
            Search = #C0,1;dl4ordersK1.           !look for match in dl4
        Else
            Delete Record #C9                     !if no match delete from Sql
            D=D+1                                  !delete old order
            !count of deleted
        End Try

        c9seq=-1                                  !read next rcd
    Loop

Else
End Try

End Sub !DeleteSQL
!
!
!    Name:
!        UpdateSQL() - Read dl4 records and change/add sql records
!
!    Synopsis:
!        UpdateSQL(C0,C9)
!        C0=dl4 file channel #
!        C9=sql file channel #
!
!    Returns:
!        A=# of records added, C=# of records changed
!
External Sub UpdateSQL(C0,C9,A,C)
!
Dim dl4orders. As dl4orders           !dl4 orders file
Dim dl4ordersK1. As dl4ordersK1       !dl4 KEY 1
Dim sqlorders. As sqlorders           !sql orders file
Dim sqlordersK1. As sqlordersK1       !sql orders key 1
Dim %1,c0seq,%3
Dim tmp$[10]
!
Print "Adding/Changing Orders in SQL"
c0seq=-2                               !read first rcd

```

```

Try
    Search > #C0,1;                                !position to first key of index 1 in dl4orders

Do
    Try Read Record #C0,c0seq;dl4orders. Else Exit Do !loop thru dl4 file
    sqlordersK1.Orderstatus$ = dl4orders.Orderstatus$

    Try
        Search = #C9,1;sqlordersK1.                !look for match in Sql
        Read Record #C9,-2;sqlorders.

        ! Check for any changes
        If hex$(sqlorders.) <> hex$(dl4orders.)
            sqlorders. = dl4orders.

            Write Record #C9,-2;sqlorders.
            C=C+1                                    !count of rcds changed
        End If
    Else
        sqlorders.=dl4orders.                        !if no match add to Sql
                                                !set rcd

        Add Record #C9;sqlorders.                    !add new order
        A=A+1                                          !count of rcds added
    End Try
    c0seq=-1                                          !read next rcd
Loop

Else
End Try

End Sub !UpdateSQL

!
! **Main Program
!
Dim SQLFile$[50],dl4File$[50],auditfile$[50]
!
!
! Build audit file
C8=FindChannel()
auditfile$="audit",TIM(8) USING "&&",TIM(9) USING "&&",TIM(10) USING "&&",TIM(11)
USING "&&",TIM(12) USING "&&",".txt!"
BUILD #C8,+auditfile$
CLOSE #C8
OPEN #C8,auditfile$
PRINT auditfile$;" audit file built"
PRINT #C8;USING "&&","Process started ";TIM(9);"/";TIM(10);"/";TIM(8);"
";TIM(11);":";TIM(12)
PRINT USING "&&","Process started ";TIM(9);"/";TIM(10);"/";TIM(8);"
";TIM(11);":";TIM(12)

! Build new dl4 file
dl4File$="dl4orders!"
C0=BuildFile(dl4File$)                                !Build new dl4 file
! Open SQL file

```

```

C9=FindChannel()
SQLFile$="(user=sa, pswd=) "server1:Finance.Order"
Open #C9,SQLFile$ As "Microsoft SQL Server Full-ISAM"
!
! Read Unibasic files and build new dl4 file
Call ReadUnibasic(C0,T)
Print #C8;"Total # of orders      ";T USING "####,###"
Print "Total # of orders      ";T USING "####,###"

! Search thru sql file, if not in dl4 file delete sql record
Call DeleteSQL(C0,C9,D)
Print #C8;"# of orders deleted    ";D USING "####,###"
Print "# of orders deleted    ";D USING "####,###"

! Search thru dl4 file, match to sql file, if not found add, if different change
Call UpdateSQL(C0,C9,A,C)
Print #C8;"# of orders added      ";A USING "####,###"
Print #C8;"# of orders changed    ";C USING "####,###"
Print "# of orders added      ";A USING "####,###"
Print "# of orders changed    ";C USING "####,###"
!
Print #C8;USING "&&";"Finished ";TIM(9);"/";TIM(10);"/";TIM(8);" ";TIM(11);":";TIM(12)
Print USING "&&";"Finished ";TIM(9);"/";TIM(10);"/";TIM(8);" ";TIM(11);":";TIM(12)
End

```

MySQL Server Full-ISAM

- ❑ Available on dL4 for Unix or Windows product
- ❑ A Full-ISAM interface to MySQL Server Tables
- ❑ Creates, reads, writes to **InnoDB type** MySQL Tables
- ❑ A table must have at least one index with unique key
- ❑ A SQL Server Table appears as a Full-ISAM file to a Basic program
- ❑ Can use the Bridge Driver to access the server because it is a Full-ISAM interface
- ❑ Can add, change and delete records in a SQL table
- ❑ Can create a new MySQL table within dL4
- ❑ Cannot change database or table definitions. Must be done within MySQL administration
- ❑ Can issue SQL statements with separate SQL driver
- ❑ Does not support record number, record size and file size channel functions (CHF)

MySQL licensing

Refer to www.mysql.com for licensing information. Requirements indicate a commercial license is required if the MySQL **drivers** are used in an application, meaning Dynamic must license any for-sale dL4/MySQL distributions.

MySQL comments

Why MySQL?

It's fast, easy to use, widely used, and inexpensive.

To install MySQL go to www.mysql.com/downloads It will be a tarball file for Unix or a .zip file for Windows.

FYI, with Microsoft Access there is an option to use something other than it's Jet engine. You can configure a Windows machine so that MySQL serves and stores the information that is viewable through Access.

MySQL basic commands

Before you do anything with MySQL, you need to start the server. Run the following from the /bin directory:

```
./mysqld_safe &
```

To interact with the server you can use the MySQL command-line client in the /bin or /usr/bin directory.

Type *./mysql -u root* to start. The *-u root* flag identifies you as the root user of MySQL.

MySQL stores each separate database as a separate directory, for most in the /usr/local/mysql/var directory.

All information for tables within a database is stored in files in the database directory.

.frm files contain file descriptions, .MYI files contain table indexes, and .MYD files contain table data. InnoDB table types create different types of files with different extensions.

To create a database use the command :
mysql> *CREATE DATABASE dbname;*

To use a database use the command :
Mysql> *USE dbname;*

To create a table you can use BUILDFI utility in dL4, use the BUILD statement in dL4, use the ic2fi utility with a bridge profile or use the CREATE TABLE command in MySQL.

You will have the most control on field types by using the MySQL interface. Always specify TYPE=InnoDB

Sampling of column types (this is not a complete list) :

Text column types

char(length)	fixed-length column type, maximum of 255 characters.
text	maximum length of 65,535 characters. (varlen option would be used in a structure definition)

Numeric column types

int(display size) [unsigned] [zerofill]
decimal(M[,D]) [zerofill]

Date column types

date

Viewing Commands :

```
mysql> SHOW DATABASES;
mysql> USE dbname;
mysql> SHOW TABLES;
mysql> SHOW COLUMNS FROM tablename;
mysql> SHOW INDEX FROM tablename;
```

Altering Commands :

Changing a table name:

```
mysql> ALTER TABLE tablename RENAME newtablename;
```

Adding columns:

```
mysql> ALTER TABLE tablename ADD COLUMN columnname column attributes
```

Dropping columns:

```
mysql> ALTER TABLE tablename DROP COLUMN columnname
```

Adding indexes:

```
mysql> ALTER TABLE ADD INDEX definition
```

Dropping indexes:

```
mysql> ALTER TABLE tablename DROP INDEX indexname
```

Changing column definitions (to leave column name the same use the same name for oldcolumn and newcolumn):

```
mysql> ALTER TABLE tablename CHANGE oldcolumnname newcolumnname  
newattributes;
```

MySQL GUI Interfaces

mysqlfront tool is a Windows based MySQL GUI interface available at

<http://www.tucows.com/business/preview/223002.html>

Many others are available, links can be found at mysql.com

DL4MYSQLISAM

New environment variable **DL4MYSQLISAM** can be used to set the default server, database, user and password instead of specifying within the OPEN statement.

Example :

```
DL4MYSQLISAM="server=servername,database=dbname,user=username,pswd=password"
```

```
export DL4MYSQLISAM
```

```
OPEN #1,"tablename" As "mysql full-isam"
```

MySQL Auto Increment

- o The MySQL Full-ISAM driver has been extended to support a SEARCH-equal operation on index 0 using a record number as the key value. Such searches can be performed only on tables that contain an AUTO_INCREMENT column and a unique index based only on that column. The value of the AUTO_INCREMENT column is treated as the record number of the row.

Example:

```
SEARCH = #5,0;R
```

- o The MySQL Full-ISAM driver has been enhanced to support tables whose only unique index contains an AUTO_INCREMENT column.
- o The MySQL Full-ISAM driver has been enhanced to support the creation of tables with numeric IDENTITY (AUTO_INCREMENT) columns. A table can contain only one IDENTITY column which must be an integer type (such as 1%, 7%, %1, or %2). Creating a table with an IDENTITY column will automatically create a unique index based on that column (the primary key). Example:

```
Def Struct REC  
  Member $$ : Item "LABEL"  
  Member %2,Id : Item "IDCOL" : Identity
```

```

End Def
Dim Rec. As REC
Build #2,"test.table" As "MySQL Full-ISAM"
Define Record #2;Rec.
Close #2

```

MySQL NULL & Date fields

- o The new MySQL Full-ISAM driver has an open option, "nulls=true", to support read and writing NULL values to table columns. When a table is opened with the "nulls=true" option, NULL values in numeric, date, or character columns are converted to special values when they are read. When adding new records or modifying existing records, NULL values can be written by writing the same special values. Currently, the special values are -1E62 for numeric values, "January 1, 0001" for date values, and "\xffff\" for strings. Programs should not test for or set these values directly. Instead, new intrinsic functions have been provided to test for NULL values and to set NULL values. The intrinsic function IsSQLNull() returns 1 when its argument is a NULL value and 0 for all other values. The intrinsic functions SQLNull(), SQLNull#(), and SQLNull\$() return special NULL values for numbers, dates, and strings. NULL values cannot be read into or written from integer numeric variables or 1% date variables. NULL values are not supported for binary variables ("B?"). NULL values can be used in keys and index columns, but it is not recommended.

Example:

```

Declare Intrinsic Function IsSQLNull,SQLNull,SQLNull#,SQLNull$
Open #1,"(nulls=true)server:database.table" As "MySQL Full-ISAM"
! Display table with possible NULL values
Do
  Try Read Record #1;R. Else Exit Do
  Print "Name = ";R.CustomerName$
  Print "Appointment = ";
  If IsSQLNull(R.AppointmentDate#)
    Print "none"
  Else
    Print R.AppointmentDate#
  End If
Loop
! Add new record with NULL value
R.CustomerName$ = "John Quinn"
R.AppointmentDate# = SQLNull#()
Add Record #1;R.

```

- o The MySQL Full-ISAM driver supports the special MySQL date value of 0000-00-00. Such date values can now be read into date variables and will set the date variable to be "Not-A-Date". The special value of 0000-00-00 can be written by writing a date value of "Not-A-Date". A date variable can be set to "Not-A-Date" by the CLEAR statement ("CLEAR D#"). An error 15 will occur if a "Not-A-Date" value is used in a date function or date expression.

MySQL SQL Driver

- o A new driver, "MySQL SQL", has been implemented so that applications can use SQL statements to issue commands and queries to a MySQL server. The driver allows an application to access the full capabilities of MySQL including both standard SQL syntax and MySQL specific features. Due to MySQL licensing requirements, the driver cannot be used without a special SSN product option. Please contact the Dynamic Concepts Sales department for information on obtaining the required SSN.

The OPEN statement uses a special filename syntax with two formats: "server:database" and "database". Rather than opening a specific table, the OPEN statement creates a connection to a MySQL server and sets the default database to be used by SQL statements. If the server name is not specified, the system on which the program is running will be used as the server (unless a default server is specified in the DL4MYSQL runtime parameter, see below). The OPEN statement also supports four comma separated options: "user=name", "password=string", "pswd=string", and "rtrim=boolean". These options supply server login identification and, in the case of "rtrim", control whether character fields are returned space filled (default) or with trailing spaces removed ("rtrim=true").

Examples:

```
OPEN #1,"mysystem:accounting" AS "MySQL SQL"
```

```
OPEN #5,"(user=bill,pswd=secret)testdb" AS "MySQL SQL"
```

The server name and login information can be specified in the environment variable "DL4MYSQL" which supports the comma separated options "server=name", "user=name", "password=name", and "pswd=name". Example for a Unix command line shell:

```
$DL4MYSQL="server=myserver,user=anonymous"  
$export DL4MYSQL  
$scope
```

SQL statements are executed by using SEARCH statements. Each SEARCH statement specifies a channel open to a MySQL server and an SQL statement as a character string. Examples:

```
SEARCH #1;"select * from testtable"
```

```
SEARCH #5;"update acctgtbl set balance=123.45 where account=19765"
```

```
SEARCH #5;"drop table testtable"
```

If the statement fails, an error will occur. Syntax errors in SQL statements are reported as error 274, "SQL syntax error".

After an SQL SELECT statement is successfully executed, the number of rows in the result set can be determined by using the CHF(channel) function. The result set itself is read by using normal READ and READ RECORD statements. An error 52, "record not found", will be reported by any statement attempting to read beyond the end of the

result set. Example:

```
Search #7;"select account, balance from acctgtbl"
Print Chf(7);"rows returned by query"
Do
  Try Read #7;Account,Balance Else Exit Do
  Print "Account =";Account;" ";Balance =";Balance
Loop
```

Note: the result set of the current SQL SELECT statement is copied into memory by the dL4 SEARCH statement. SELECT statements should be written so as to limit the size of the result set to a reasonable value. An SQL LIMIT clause can be used in the SQL SELECT statement to restrict the maximum size of the set.

The MAP RECORD statement can be used to map structure variable members according to their item names to the columns returned by a query. In the following example, the SQL select statement returns a two column result "account, balance" which is mapped into a structure variable that uses the opposite ordering:

```
Def Struct RSET
  Member 3%,Balance : Item "balance"
  Member 3%,Acct : Item "account"
End Def
Dim R. As RSET
Search #7;"Select account, balance from acctgtbl"
Print Chf(7);"rows returned by query"
Map Record #7 As RSET
Do
  Try Read Record #7;R. Else Exit Do
  Print "Account =";R.Acct;" ";Balance =";R.Balance
Loop
```

NULL values in numeric, date, or character columns are converted to special values when they are read. When adding new rows or modifying existing rows, NULL values can be written by using the same special values. In this version of dL4, the special values are -1E62 for numeric values, "January 1, 0001" for date values, and "\xffff\" for strings. Programs should not test for or set these values directly. Instead, new intrinsic functions have been provided to test for NULL values and to set NULL values. The intrinsic function IsSQLNull() returns 1 when its argument is a NULL value and 0 for all other values.

The intrinsic functions SQLNull(), SQLNull#(), and SQLNull\$() return

the special NULL values for numbers, dates, and strings. NULL values cannot be read into or written from integer numeric variables or 1% date variables. NULL values are not supported for binary variables ("B?").

Three new intrinsic functions, SQLV\$(), SQLN\$(), and SQLNV\$() are

provided to make it easier to construct SQL statements. The SQLV\$() function takes one or more arguments of any non-array type and returns a string containing the argument values encoded for use by an SQL driver. The SQL driver detects such encoded values in the SEARCH statement string and formats the values as required by the SQL server. This formatting guarantees proper quoting of character string values and places commas between each value. If the argument is a structure variable, each member of structure is encoded. The SQLN\$() function takes a single structure variable argument and returns a string containing the member item names encoded for the SQL driver with commas separating each name. The SQLNV\$() function takes a single structure variable argument and returns a string containing the member names and values encoded for the SQL driver with equals signs ("=") and commas.

Examples:

```
Search #1;"Insert test (count,label) Values (" +SQLV$(C,L$)+")"
```

```
Search #1;"Insert test (" +SQLN$(R.)+" ) Values (" +SQLV$(R.)+" )"
```

```
Search #1;"Update test Set " +SQLNV$(R.)+" where count=19"
```

MySQL SQL Driver Example

```
!samples of sql commands
!
Def Struct orders
  Member Order$[10]:Item "Order"
  Member Account$[10]:Item "Account"
End Def
!
Dim orders. as orders
!
open #1,"cochrane:test" as "MySQL SQL"
!
Sub showorders()
  Search #1;"select * from orders"
  Print Chf(1);" rows returned"
Do
  Try Read #1;orders. else exit do
  Print orders.order$,orders.account$
Loop
End Sub !showorders()
!
Call showorders()
!
Search #1;"update orders set `Order`=55555 where Account="DCI""
!
Call showorders()
```

ic2fi Utility

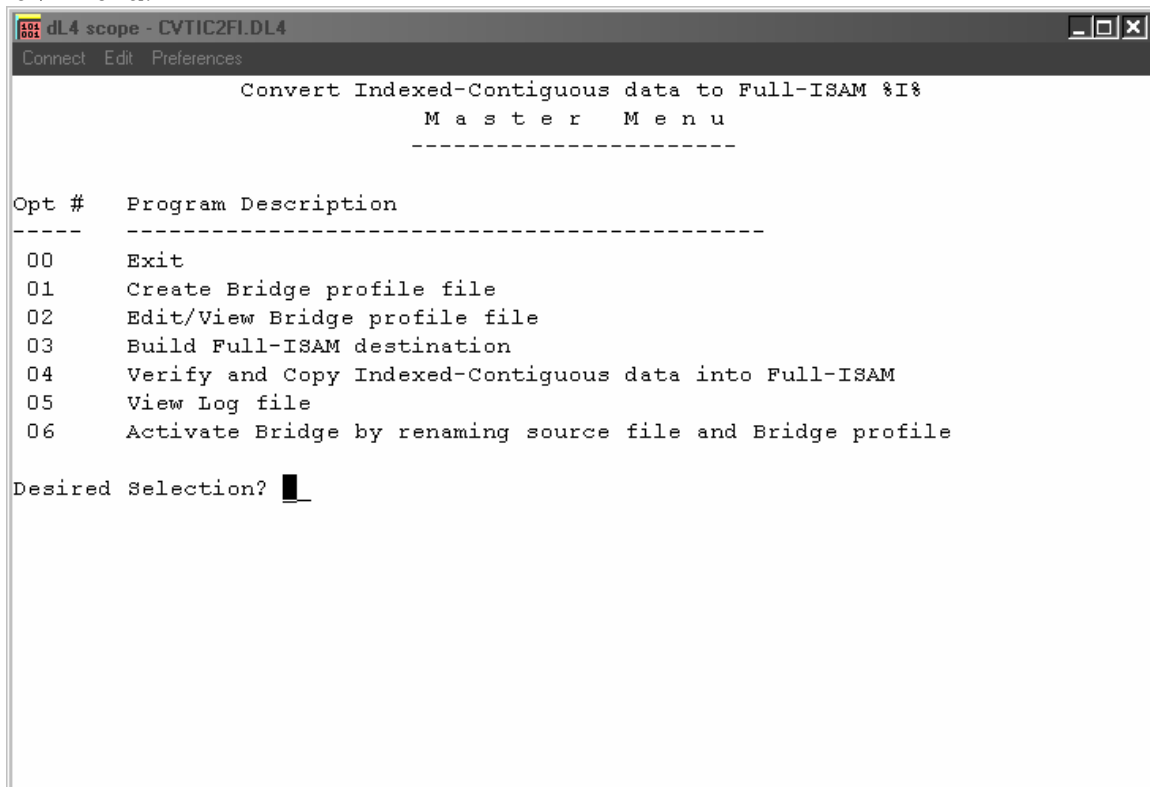
The ic2fi utility is available to easily convert Indexed-Contiguous files to Full-ISAM files.

Is included with dL4 in the tools directory with a name of ic2fi.dl4

Replaces the previous release of ictofi.

Is based on a bridge profile file definition, which can then be used to access the new file with your existing application.

ic2fi menu:



The options should be run sequentially.

Option 01 Allows you to create a new bridge profile file. Basically it copies the ic2fi.prf template file and then allows you to begin editing. The profile is identical to the bridge profile definition with an optional section to specify the default source filename. Post conversion the profile can be used as the bridge file without modification.

Items must be listed in byte displacement sequence.

A [conversion] section can be specified in the profile file, with a default sourcefilename specified.

Option 02 Allows you to view and edit an existing bridge profile.

Option 03 Allows you to build a new destination Full-ISAM file. It will not overwrite an existing file. This mechanism can be used to simply build new Full-ISAM files instead of using the buildfi utility.

Option 04 The guts of the utility.

This option will copy to an existing destination, but will warn you that the file exists. If the file exists, the utility will only add records to the destination file if the record can be added. It will not modify or delete records in the destination file, thus it is not a replication utility.

This option will ask if you want to run in interactive or silent mode. Silent mode will not require any user intervention during a potentially lengthy verification/copy process and will assume defaults on prompts.

This option will first do a quick verification of the profile definition and it will warn you if there are any gaps found between fields. This is a warning only, you can continue the process if you are confident your profile is as you desire.

The option will then go through several quick verification routines to confirm that your profile definition coincides with the actual data in the file.

It will warn you of any field size discrepancies and it will also compare the first 3 and last 3 indexes to confirm they match the profile definition of the indexes. These are warnings only, you can continue the process if you are confident your profile is as you desire.

You will then have an option to have the utility search through entire selected indexes to verify the integrity of the indexes. It will compare the index values to the associated records and confirm balanced indexes. You have the option to skip this process as a time-saving measure. In silent mode this option is skipped.

You will be prompted to enter a log filename which will record any errors encountered during index verification or copying.

Finally, you will be prompted to select a primary index number. This will be the source file index to search when copying the records.

When verification is complete, the utility will proceed to copy the data.

Option 05 Allows you to view the log file for warnings or errors.

Option 06 This option renames the existing Indexed Contiguous files to have an ic. prefix and then moves or renames the bridge profile file to the source filename. Your applications would then function without code modification, now using the bridge file to communicate to the Full-ISAM file.

ic2fi Call program interface

Options of the ic2fi can be access from a command line interface to automate the process of building Full-ISAM or copying Indexed-Contiguous to Full-ISAM files.

You must pass the option number, bridge profile filename and the mode (silent or interactive) as parameters.

Optionally you can provide a status field (0=OK, 1=warnings, 2=errors) and a log filename to overwrite the default.

dL4 Product Training

SECTION 6

Other Drivers

TCP/IP Socket Driver

- o A TCP/IP socket driver is available to communicate to other systems through a port using the TCP/IP protocol. A connection can be created using an OPEN statement similar to:

```
OPEN #c,"system:port" As "Socket"
```

where "system" is a network system name or IP address (xx.xx.xx.xx) and "port" is a service name or port number. Character and binary data can be read or written. The following example program reads the current date and time from the host system:

```
10 Dim L$(100)
20 Open #1,"localhost:daytime" As "Socket"
30 Read #1;L$
40 Close #1
50 Print L$
```

- o The socket driver supports a "partial=<boolean>" option to control whether a read terminates as soon as at least one character has been read (default behavior) or waits until the destination variable has been filled ("partial=true" behavior). This option is specified in the option field when opening a socket. Example:

```
Open #1,"(partial=true)server:servername" As "Socket"
```

EXAMPLE OF USING SOCKET DRIVER FOR DATA MINING

Take Advantage of the web

Enhance your application for e-business with Dynamic Internet Technologies and Professional Services.

Integrate information from the web directly into your application using the dL4 socket driver.

Bring the power of the web into your application and bring a world of information to your clients!

Add Data Mining capabilities to your application to automatically pull valuable information from the web directly to your Unibasic or dL4 application. Utilizing the dL4 TCP/IP Socket Driver included in dL4, your application can connect to internet sites across the globe to query and import information into your application's database. The database can be Microsoft SQL, Foxpro Full-ISAM or dL4/Unibasic Universal files.

You can query UPS, FedEx and U.S.P.S. sites to obtain shipping costs and package tracking information.

You can query your supplier's sites for updated pricing or stock availability. Give your client a competitive edge. Think of the possibilities! Quickly upgrade your application today and bring the power of the web into your world.

As XML proliferates the internet as the standard mechanism for B:B (business to business) communication, you'll be ready to meet your client's needs with the TCP/IP Socket Driver.

Note to Unibasic users:

Data Mining requires a dL4 license to utilize the TCP/IP Socket Driver. However, you do not have to migrate your entire application to dL4 immediately to take advantage of the technology today. A dL4 license can reside on the same machine as an existing Unibasic licensed application. dL4 programs can run as background CRON jobs or Unibasic programs can Call dL4 programs to run to update the Unibasic data files.

Open Socket Syntax :

Open #1,"(partial=true) server:servicename" As "Socket"

By default read statements on the socket channel terminate as soon as at least one character has been read.

The (partial=<boolean>) is optional to control whether a read terminates as soon as one character is read or waits until the destination variable has been filled (partial=true behavior).

Check out the example below.

Yahoo Stock Quote demo - Socket access using dL4 socket driver :

```
If Err 0 GOSUB L8990
! yahoo.bas - example program of using Socket Driver to receive information from
Yahoo Stock Quotes Web Site
! Rev 1.0 6/14/00
!
! All rights reserved. (C) Copyright 2000 by:
! Dynamic Concepts Inc. Aliso Viejo, California USA
!
! This file should be loaded and saved as "yahoo.dl4"
!
!
! As always Dimension the variables used in the program
Dim i$(2000),L1$(2000),l2$(6000),l3$(30),z$(20)
Dim i$(20)
Dim ticker$(10),name$(20),time$(10),3%,price,volume,tmp$(256)
!
! Open the Web site to attach to as a socket using the TCP/IP socket driver
! Open statement format is OPEN #c,"system:port" As "Socket"
! (finance.yahoo.com is the Yahoo Stock Quotes site)
! (The standard port # for HTTP requests is 80)
Print 'CS';"Real-time Stock Quote"
Print
Print "This is a demo of a dL4 Application accessing Yahoo's Stock Prices through an"
Print "HTML Socket using dL4's TCP/IP Socket Driver."
Print
Print "Enter a Stock Symbol to display it's current price."
another: Print
Input "Enter a Stock Symbol (i.e. ARBA) (enter 0 to exit) : "i$
If i$ = "0" GOTO L9000
i$ = UCASE$(i$)
Print
Print "Please wait. . .Retrieving price"
! Prepare to request information by defining the string to send through the socket
! GET is HTTP protocol to request /d/quotes.csv is the file to request at the site
Open #1,"finance.yahoo.com:80" As "Socket"
l$ = "GET /d/quotes.csv"
!
! Characters following ? are CGI parameters the Site Page expects
l$ = l$,"?s=",i$,"+~~~&f=snlv&e=.csv"
!
! The following ends the HTTP request
l$ = l$," HTTP/1.0\12\12\"
! Send the request
Print #1;l$;
!
! Wait for response, use DO LOOP to receive back full string
Clear l2$
Do
  Try !Dl4 Try statement
    Read #1;l1$ !receive some characters
  Else
    Pause 30 !if nothing to read,pause & try again
  Retry
End Try
```

```

12$ = 12$ + L1$ !concatenate to make one string
13$ = "~~~" !this string indicates end of message
x = Pos(12$, = 13$) !new D14 function POS looks in string L2$ for match to L3$
and returns string location if found
If x Exit Do !if end of message, quit reading
Loop
!
!Now parse the stock price out
13$ = """,i$,""" !this string indicates where Stock Info is returned, ticker symbol
x = Pos(12$, = 13$)
Let ticker$ = 12$[x] To "," : Spos
Let name$ = 12$[Spos + 1] To "," : Spos
name$ = LTrim$(RTrim$(name$[2,Len(name$) - 1]))
Let tmp$ = 12$[Spos + 1] To "," : Spos
ypos = Pos(tmp$, = "N/A")
xpos = Pos(tmp$, = "<b>")
If ypos Print \ Print "Invalid Ticker Symbol entered!" \ Print \ Goto done
price = Val(tmp$[xpos + 3])
time$ = LTrim$(RTrim$(tmp$[2] To "-"))
volume = Val(12$[Spos + 1])
Print
Print i$," - ";name$," price is ";price Using "$$$,$$$$.##";" at ";time$;
Print " volume of ";volume Using "###,###,###"
Print
done: Close
Goto another
L8990: If Spc(8) = 99 Goto L9000
Stop
L9000: Rem END
Close
End

```

EXAMPLE OF USING SOCKET DRIVER FOR RECEIVING EMAIL

Receive text email demo - Socket access using dL4 socket driver :

```
! dL4 / Sample program to receive POP3 mail
! "rcvemail.bas" 1.1 11/27/01 09:35:18
!
! All rights reserved. (C) Copyright 2001 by:
! Dynamic Concepts Inc. Aliso Viejo, California USA
!
! This sample program demonstrates the usage of the dL4 socket driver to
! receive POP3 mail. It implements a simple, interactive e-mail reader.
!
! POP3 is discussed in RFC 1939 which may be downloaded from
! http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1939.html
External Function GetLine$(ChannelNo,Buf$)
!
  Dim LineBuf$(65535)
  Dim %2,StreamSize,P
  !
  If (P := Pos(Buf$, = 'CR LF')) = 0
  Then
    StreamSize = 0
    LineBuf$ = Buf$
  Do
    Read #ChannelNo,-1,-1,1800;Buf$[StreamSize + 1]
    !reads in 1 character at a time and adds to Buf$ until <cr><lf> reached
    StreamSize = Len(Buf$)
    P = Pos(Buf$, = 'CR LF')
  Loop Until P > 0
  End If
  LineBuf$[Len(LineBuf$) + 1] = Buf$[1,P + 1]
  Buf$[1,P + 1] = ""
End Function LineBuf$

External Function GetMultiLine$(ChannelNo, Buf$)
!
  Dim MultiLine$(65535)
  Dim %2,P
  !
  Do
    P = Len(MultiLine$) + 1
    MultiLine$[P] = GetLine$(ChannelNo,Buf$)
    !reads in 1 line at a time and adds to MultiLine$ until .<cr><lf> reached
  Loop Until MultiLine$[P] = "." + 'CR LF'
End Function MultiLine$

External Function GetHeader$(Buf$,Tag$)
!
  Dim Header$(1000)
  Dim %1,P
  !
  If P := Pos(Buf$, = Tag$)
```

```

Then
    Header$ = Buf$[P]
    If P := Pos(Header$, = 'CR LF')
    Then
        Header$ = Header$[1,P - 1]
    End If
End If
End Function Header$

External Sub GetPOP3Mail(ChannelNo, User$, PassWord$)
!
Dim Buf$[65535],Response$[65535]
Dim %1,NumberOfMessages,MessageNumber,P
!
Try
    !
    ! Should receive:
    ! +OK SCO POP3 server (version 2.1.4-R3) at POP3_server_name starting.
    !
    Response$ = GetLine$(ChannelNo,Buf$)
    If Not(Pos(Response$, = "+OK"))
    Then
        Print "Sorry, no POP3 server not responding"
        Exit Sub
    End If
    Print #ChannelNo;"USER " + User$ + 'CR LF'
    !
    ! Should receive:
    ! +OK Password required for user_name.
    !
    Response$ = GetLine$(ChannelNo,Buf$)
    If Not(Pos(Response$, = "+OK"))
    Then
        Print "Sorry, Incorrect Account"
        Exit Sub
    End If
    Print #ChannelNo;"PASS " + PassWord$ + 'CR LF'
    !
    ! Should receive something like this:
    ! +OK user_name has 20 message(s) (20790 octets).
    !
    Response$ = GetLine$(ChannelNo,Buf$)
    If Not(Pos(Response$, = "+OK"))
    Then
        Print "Sorry, Incorrect Password"
        Exit Sub
    End If
    Print #ChannelNo;"STAT " + 'CR LF'
    !
    ! +OK number_of_messages size_of_maildrop
    !
    Response$ = GetLine$(ChannelNo,Buf$)
    If Not(Pos(Response$, = "+OK"))
    Then
        Print "Sorry, STAT failed"
    End If
End Try
End Sub

```



```

Exit Sub
End If
Response$ = Response$[5]           ! skip over +OK
NumberOfMessages = Val(Response$[1,Pos(Response$, " ") - 1])
For MessageNumber = 1 To NumberOfMessages
    ! Get 10 lines for this particular messagenumber
    Print #ChannelNo;"TOP " + Str$(MessageNumber) + " 1" + 'CR LF'
    Response$ = GetLine$(ChannelNo,Buf$)
    If Pos(Response$, "= +OK")
    Then
        Response$ = GetMultiLine$(ChannelNo,Buf$)
        ! Extract mail header information
        Print MessageNumber;GetHeader$(Response$,"From"),
        Print GetHeader$(Response$,"Date"),
        Print GetHeader$(Response$,"Subject")
    End If
Next MessageNumber
If NumberOfMessages > 0
Then
    ! Retrieve message number 1
    Print #ChannelNo;"RETR 1" + 'CR LF'
    Response$ = GetLine$(ChannelNo,Buf$)
    If Pos(Response$, "= +OK")
    Then
        Response$ = GetMultiLine$(ChannelNo, Buf$)
        If P := Pos(Response$, "." + 'CR LF', -1)
        Then
            Print Response$[1,P - 1];
        Else
            Print Response$;
        End If
    End If
    ! Delete message number 1
    Input "Delete Message Number 1? [N/Y] "; Response$
    If Response$ = "Y"
    Then
        Print #ChannelNo;"DELE 1" + 'CR LF' !Deletes email
        Response$ = GetLine$(ChannelNo, Buf$)
    End If
Else
    Print "No mail"
End If
Else
    Print Msc$(2) + " at " + Str$(Spc(10))
End Try
Print #ChannelNo;"QUIT" + 'CR LF'      ! sign off
End Sub

```

```

! Main program
Dim POP3Server$[100], PortNum$[3]
Dim User$[100], PassWord$[20]
Dim %1,ChannelNo
!
Declare Intrinsic Function FindChannel

```

```

!
Input "Please Enter POP3 server name: "POP3Server$
Print
Input "Please Enter POP3 server's port number [110] "PortNum$
If Not(PortNum$)
Then
    PortNum$ = "110"
End If
While Not(User$)
    Print
    Input "Please Enter User's POP3 account: "User$
Wend
Print 'IOEE'
Input "User's password: ";PassWord$
Print 'IOBE'
Try
    ChannelNo = FindChannel()      ! get unused channel number
    ! Make a socket connection to the POP3 server
    Open #ChannelNo, "(partial=true, opentime=1800)" + POP3Server$ + ":" +
PortNum$ As "Socket"
    Call GetPOP3Mail(ChannelNo, User$, PassWord$)
Else
    Print Msc$(2) + " at " + Str$(Spc(8))
End Try
Try Close #ChannelNo Else Rem

```

Listening Socket

- o A driver, "TCP Listen Socket" is available. The "Listen" driver allows a dL4 program to be a server for socket requests. For example, dL4 program S running on ServerA might use the "Listen" driver to open a listening socket on port 9631 of ServerA. Programs on other systems could then open sockets to port 9631 of ServerA to exchange data with program S.

The "Listen" driver is used by opening a port number using the "Listen" driver:

```
Open #1,":9631" As "TCP Listen Socket"
```

A "REUSE" option can be used to open a socket with a port number that is already in use. This option is needed to support certain network protocols and to avoid error 76 ("File or device is open elsewhere") when re-opening a previously used TCP port number. Example:

```
Open #1, "(reuse):9631" As "TCP Listen Socket"
```

The open will return immediately. To accept the next queued connection to that port, the program opens a socket using channel 1 as the parent socket:

```
Open #2,{1} As "Socket"
```

This open to channel 2 will not return until another program on the local or a remote system opens a socket to port 9631 of the local system. Once the open returns, channel 2 can be used to perform normal socket read and write operations to transfer data from or to the client program. When the transaction is finished, the server program closes channel 2 and performs another open against parent channel 1 to accept the next queued client request.

The example program below provides a date and time service on port 9631. While the program is running, any telnet utility can be used to connect to port 9631 and receive the current date and time. The current date and time will be printed to the client once every ten seconds until the client closes the connection.

```
Dim I$[100],3%,I
Open #1,":9631" As "TCP Listen Socket"
I = 0
Do
  Open #2,{1} As "Socket"
  I = I + 1
  Do
    Try Print #2;"Session";I;Tim#(0);"\15\12\"; Else Exit Do
    Try Read #2,-1,-1,100;I$ Else Rem
```

```

        If Spc(8) = 123 Exit Do
    Loop
    Close #2
Loop

```

If multiple clients attempt to open port 9631 at the same time, only one request will be accepted at a time. The other clients will be queued by the operating system until the current connection on channel 2 is closed. The client program will receive an error if the total number of queued requests exceeds an operating system defined number (usually a small number). Client programs should be prepared to retry opens if the server is busy.

Normally, a null server name (":9631") should be used to open a listening socket on local system. The server name can be specified to open a listening socket on a specific network interface.

The "opentime" option can be used to specify a maximum number of seconds to wait when opening a new queued connection:

```
Open #2,{1,"opentime=10"} As "Socket"
```

A record locked error (error 123) will be generated if the open times out.

Lists of pre-defined TCP port numbers can be found at many web sites include www.iana.org. When writing a "listening" socket program, the port number should not conflict with a port number used by a needed or common pre-defined service. In general, the port number should be greater than 1023.

Inetd alternative

Note: on most Unix systems, a "listening" server can be implemented more robustly by having the system **inetd** process listen for requests and start dL4 processes as required. For example, adding the line below to /etc/inetd.conf will cause inetd to listen on the port "test" and start the dL4 program "pgm.dl4" whenever anyone attempts to open a socket on the port "test".

```
test stream tcp nowait user /usr/bin/run run -t "" /home/user/pgm.dl4
```

The dL4 process will run as the user "user". **The socket will be open on the standard input and output channels and accessible through normal INPUT and PRINT statements. The port number "test" must be defined in the file /etc/services.** The '-t ""' option to run tells run to use the default terminal definition. After modifying the file /etc/inetd.conf, the system must be rebooted or a SIGHUP signal must be sent to the inetd process.

New GET statements

Two new GET statement operations have been defined for use with TCP socket drivers. The following statement will retrieve as a character string (usually "nnn.nnn.nnn.nnn") the remote IP address connected to the socket open on channel C:

Get #C,-1598;S\$

This function would typically be used with the "TCP Listen Socket" driver to determine the IP address of a client system. The statement below will retrieve the local IP address connected to the socket open on channel C:

Get #C,-1599;S\$

Email Driver

- o A driver class, "Email", is available to send email. The driver requires sending email through an SMTP server such as provided by Unix servers or most ISPs. Non-SMTP mechanisms may be added in the future and should not require any application changes unless SMTP specific options (such as "SERVER=") are used (use the DL4EMAILSERVER environment variable instead). After opening the driver, a program simply prints text to be emailed to the channel. Files can be attached and sent as part of the email by using 'ADD #c;"Filename"' statements. If a file consists of multiple files, such as the data and index portions of an Indexed Contiguous file, each subfile must be sent with a separate ADD statement.

The path argument to the driver is an email address list. An email address list consists of one or more space separated email addresses. Email address lists are also used in the options such as "TO=" described below. The driver options parameter ("xxx") can be used to pass the following options:

"TO=addresses" one or more destination email addresses. May be used in addition to placing addresses in the path.
Addresses should be space separated.

"BCC=addresses" one or more "BCC" email addresses. These are similar to "CC" addresses, but they are not included in the email header.

"CC=addresses" one or more "CC" email addresses.

"FROM=addresses" one or more sender addresses.

"REPLYTO=addresses" one or more reply addresses. This option would only be used if the reply address was different from the sender ("FROM=") address.

"SUBJECT=text" email title or subject. The text may be placed in quotation marks if necessary.

"CONTENT=name" content type. "HTML" allows HTML formatting commands. "TEXT" selects non-HTML text format (the default).

"SERVER=name" SMTP server name. **Defaults** to the value defined by the DL4EMAILSERVER environment variable or the host system.

"PORT=n" SMTP port number. **Defaults** to the standard SMTP port (25).

"PROTOCOL=name" Email protocol name. Only "smtp", the default protocol, is supported by this release.

"ATTACHAS=name" File attachment encoding type. This option must be specified if file attachments will be used. The value of "name" must be either "mime" or "default" (which is "mime" in this release). (all attachments are sent as binary)

"TIMEOUT=n" timeout period in tenth-seconds for communication with the SMTP server. This option is used to change the default 5 minute timeout period and should not be needed.

A program using the email driver must specify at least one destination email address in the path or a "TO=" option. At least one "FROM=" email address must be provided. The SMTP server name is optional and can be defined via the "SERVER=" option or in the DL4EMAILSERVER environment variable. If the server name is undefined, the host Unix system will be used.

After all email text has been output and all attachments added, the email channel should be CLOSEd to actually send the email. If the channel is CLEARed, the driver will attempt to cancel the email.

Example:

```
Open #1,"(From=name@domain,AttachAs=Mime) nobody@dynamic.com" As
"Email"
Print #1;"Test the email driver"
! Append the file "Filename" as an attachment
Add #1;"Filename"
Close #1
```

Please change the "From=" and destination email addresses to your own email address before using this example.

Another example :

```
! dL4 / Sample program to send e-mail
! "sendemail.bas" 1.1 11/27/01 09:35:02
!
! All rights reserved. (C) Copyright 2001 by:
! Dynamic Concepts Inc. Aliso Viejo, California USA
!
! This sample program demonstrates the usage of the dL4 email driver. It
! uses SMTP protocol to send e-mail to a user supplied recipient's address
! from a user supplied sender's address. It sends a test message and an
! attachment file, sendmail.txt, if it exists. This program assumes that
! the SMTP server is on the current system. If the SMTP server is not on
! the current system, then use the DL4EMAILSERVER environment variable or
! the "server=" open option.
```

```
External Sub SendMail(Sender$,Recipient$)
!
```

```

Dim AttachmentFile$[100]
  Dim %1,ChannelNo,Attachment
  !
Declare Intrinsic Function FindChannel
  Declare Intrinsic Sub FindF
  !
  Try
    AttachmentFile$ = "sendmail.txt" ! name of attachment file
    ! Check for attachment file
    Call FindF(AttachmentFile$,Attachment)
    ChannelNo = FindChannel()          ! get unused channel number
    !
    Open #ChannelNo,"(From=" + Sender$ + ",Subject=dL4
sendmail,AttachAs=Mime)
+ Recipient$ As "Email"
    Print #ChannelNo;"Thank you for participating in our test program."
    Print #ChannelNo;"This e-mail was sent to you using a dL4 program."
    Print #ChannelNo;"Please reply with an acknowledgement to the sender."
    If Attachment                      ! attachment file found?
    Then
      Add #ChannelNo;AttachmentFile$ ! yes, send attachment
    End If
  Else
    Print Msc$(2) + " at " + Str$(Spc(8))
  End Try
  Try Close #ChannelNo Else Rem
End Sub

! Main program
Dim Sender$[100],Recipient$[100]
!
Input "Please Enter sender's e-mail address: ";Sender$
Print
Input "Please Enter recipient's e-mail address: ";Recipient$
Call SendMail(Sender$,Recipient$) !call subroutine above

```

Serial Device Driver

- o A driver, "Serial Terminal", allows programs to open serial communication devices to a Window class driver. Using this driver, input and output to a serial device will follow the same rules as screen and keyboard I/O. By default, end of line characters will terminate input, input edit characters such as backspace will be processed, and data characters will be echoed. The standard 'IOxx' mnemonics can be used to control input characteristics. Similarly, cursor positioning can be used on output if mnemonics are defined in a terminal definition file.

The "Serial Terminal" driver accepts the options listed below when opened:

Option	Argument	Use
TERM	Filename or path	Specify terminal definition file to be used with device
SPEED	Numeric ("9600")	Set device dependent line speed
DATA	String ("8n1", "7e1")	Set device dependent data format
XONFLOW	Boolean ("T" or "F")	Enable XOFF/XON output flow control

If not specified, all options except TERM use the current system default value of the device. If the TERM option is not specified, the driver uses a simple default terminal definition in which carriage return is recognized as an input terminator.

Example:

```
F$ = "(speed=38400,data=8n1,term=/usr/lib/dl4/term/vt100) /dev/tty1a"  
Open #1,F$ As "Serial Terminal"
```

!/dev/tty1a in statement above is the device name to open
!in Windows it may be COM

dL4 Product Training GUI

(Refer to dL4 Version 4.3 GUI Training document)

dL4

Product Training

Conversion

Ub2dl4 Conversion tool

The ub2dl4 conversion tool is available for converting Unibasic programs to a dL4 environment on a Unix platform. It also attempts to convert terminal definition files, printer scripts and UNIX profiles.

The tool is intended for mass conversion of programs and ease of conversion.

The tool is a separate download from [ftp.dynamic.com](ftp://dynamic.com) and can be found in the /dist/pub/FF/ub2dl4 directory.

After downloading, first uncompress the product using uncompress and then extract the product by running "cpio -imcdv <product" (or on Linux cpio -imdv <product)

1. Set the dL4 environment variable TERMDIR to point to the directory (usually /usr/lib/dl4/term) of the dL4 terminal definition files.
2. Change (cd) to the ub2dl4 directory.
3. The conversion package must be run from a UniBasic account so that all UniBasic environment variables, particularly LUST, are set.
4. The main menu screen is displayed by typing the following from a UNIX shell:

run ub2dl4.dl4

5. Follow the menu options sequentially, meaning run menu option 1, then menu option 2, then menu option 3, and so on.

If you are remaining on the same platform, there is no need to convert data files.

If you will be moving data files to a different platform, ie Windows, indexed contiguous, contiguous and formatted files must be in Universal Data file format prior to moving the files.

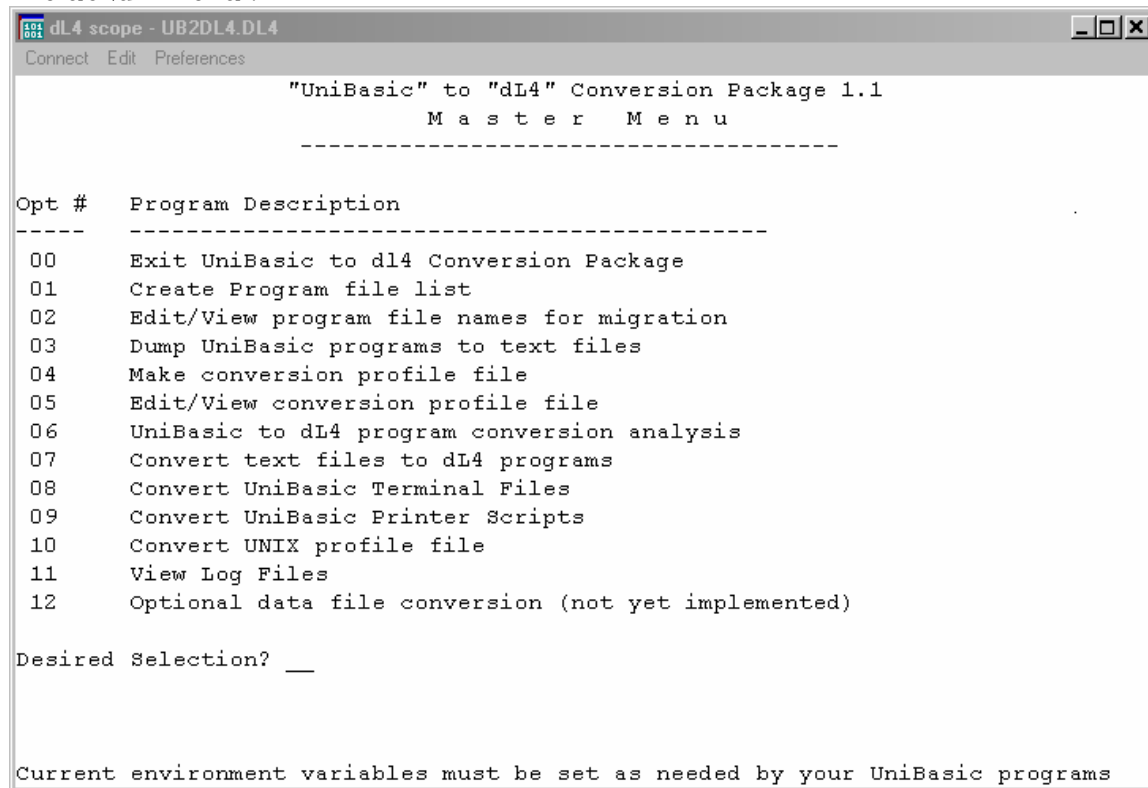
If the files are in IRIS style BCD Data file format and the Key file is NOT IRIS style keys, the **ubconvert** program can be used to convert to Universal Data file format. **Ubconvert** is a user friendly interface to the **ubconvertfiles** command line utility. Both are part of Unibasic version 6 or greater.

To determine if a file is IRIS style BCD and non-IRIS style keys, QUERY the file to verify the attribute <Q> has been set and the attribute <K> is not set.

If the files are in BITS style format (the <Q> bit is not set), the **ctool** utility can be used to convert to Universal Data file format. **Ctool** is downloadable from the ftp site.

Training Note The best way to configure the proper environment is to make a copy of a .profile that sets up a current Unibasic runtime user. Then modify that .profile to set TERMDIR and start dL4 scope. Then at the system prompt type ub2dl4.dl4

The ub2dl4 menu :



```
dL4 scope - UB2DL4.DL4
Connect Edit Preferences

"UniBasic" to "dL4" Conversion Package 1.1
M a s t e r   M e n u
-----

Opt #   Program Description
-----
00      Exit UniBasic to dL4 Conversion Package
01      Create Program file list
02      Edit/View program file names for migration
03      Dump UniBasic programs to text files
04      Make conversion profile file
05      Edit/View conversion profile file
06      UniBasic to dL4 program conversion analysis
07      Convert text files to dL4 programs
08      Convert UniBasic Terminal Files
09      Convert UniBasic Printer Scripts
10      Convert UNIX profile file
11      View Log Files
12      Optional data file conversion (not yet implemented)

Desired Selection? __

Current environment variables must be set as needed by your UniBasic programs
```

The options should be run sequentially.

Option 01 Allows you to create a Program file list that is to be converted. The directory containing the Unibasic programs to be converted is entered (programs in subdirectories will also be converted). Then you can narrow the list by specifying a filename with wildcards. A text file list of files to be converted will be created.

Option 02 Allows you to view the list that was created by Option 01. You can edit this list before dumping to text and converting. The list will contain the program type (BITS or IRIS) and the filename.

Option 03 Will dump all the files in the convert list to text files. The dL4 converter converts from a text file of a Unibasic program. You will be prompted to select a directory to dump the text files to. A subdirectory called dumpdir will be created under your specified directory to dump the program files to.

Option 04 Will create the conversion profile file specific to your Unibasic environment.
The conversion profile file is basically a text file that contains dL4 conversion rules.

Option 05 Allows you to view and/or edit the conversion profile file.

Option 06 Provides a pre-conversion analysis of the files dumped in Option 03. A summary or detail report can be displayed or printed to a printer or text file. The analysis provides details of any conversion issues that will occur so that the profile or source files can be modified prior to actual conversion.

Sample Option 06 display:

```

dL4 scope - UB2DL4.DL4
Connect Edit Preferences

Date: OCT 7, 2002 12:37:57.524
dL4 Analysis Summary Report
loadsave 5.1.1

Total # of program files processed:      336
Total # of program files without errors:  328      97.6%
Total # of program files with errors:    8        2.4%

Total # of program files with warnings:  0        0.0%

Approximate disk space for programs:      3.183 MB
Approximate disk space for text files:    2.068 MB

Call Errors      Occurrences  Programs  Program Name

Errors

Syntax error                        41        8
ar.chargepost
1280 LET A1$=A$ TO "\211\";X \! CUSTOMER #

Enter printer filename, text filename or [RETURN] to continue

```

Option 07 Will do the actual conversion of the Unibasic text files to dL4 and save as a compiled dL4 program.

Option 08 (optional) This option will look at your Unibasic terminal definition files and create dL4 Terminal files to match your environment. If dL4 Terminal files already exist in the destination directory they will be renamed with a .dl4 extension.

Option 09 (optional) This option will look at your Unibasic printer scripts and create dL4 printer files to match your environment. Any text files with execute permissions are considered to be printer scripts.

Option 10 (optional) This option will look for .profile and profile files in the directory specified and then modify and save them with a .ub2dl4 extension to be used in the dL4 environment. It adds lines to set environment variables like TERMDIR and DL4LUST. It also remarks lines to launch Unibasic and replaces them to launch run or scope appropriately.

Option 11 (optional) Allows you to view the analysis log or any error logs that have been generated.

You may need to edit the conversion profile file to perform a more successful conversion.

Many issues are resolved by having the converter add OPTION statements to each program.

The following are possible issues post-conversion that are not resolved by the conversion tool when converting from Unibasic:

The LET TO statement with the optional : numeric variable is limited in dL4 to a single character lookup and cannot be combined with LET statement concatenation.

For example:

```
LET X$="TEST:", Y$ TO "x" : X
```

Must be changed to :

```
LET TMP$=Y$ TO "x" : X  
LET X$="TEST" : " , TMP$
```

You cannot have multiple DEF FN's of the same name in the same program, thus functions cannot be dynamically defined.

Unibasic only allowed letters, digits, dash and period as valid filename characters. Unibasic would stop reading the filename when it reached an invalid character, thus Open #1,"customers@01" would open file named "customers".

By default, dL4 will try to open file named "customer@01" and fail.

To correct, statements can be changed, files can be renamed or an optional parameter, unibasic can be specified in parentheses at the beginning of the LUMAP or DL4LUST parameter string to treat filenames in Unibasic mode. Also the pfchar=x optional parameter may be used.

Terminal ESCape sequences will work with dL4 but will not be understood by dL4Term.

Unibasic supports User-defined Graphical Mnemonics defined as G12 through G28 in the terminal definition file. The octal codes associated with these will not function in dL4 unless they are first defined in the dL4 terminal definition files. The mnemonics associated with these definitions are not supported in dL4 as mnemonic 'BG' and 'EG' switches are ignored. In Unibasic, these mnemonics, when appearing between 'BG' and 'EG' where interpreted differently than their default meaning.

ims2dl4 Conversion tool

The ims2dl4 conversion tool is available for converting IMS Basic programs to a dL4 environment on a Unix platform.

The tool also converts Formatted, Contiguous and Indexed files to Universal Data files.

The tool is intended for mass conversion of programs and data files.

Included with the tool is **imscalls.lib** which is a library containing Calls and Functions specific to IMS. **imscalls.bas** is the editable source code for imscalls.lib.

The tool is a separate download from ftp.dynamic.com and can be found in the /dist/pub/FF/ims2dl4 directory.

After downloading, first uncompress the product using uncompress and then extract the product by running "cpio -imcdv <product" (or on Linux cpio -imdv <product)

The tool is self-documented with a /H or ? option.

1. Set the dL4 environment variable TERMDIR to point to the directory (usually /usr/lib/dl4/term) of the dL4 terminal definition files.
2. Change (cd) to the ub2dl4 directory.
3. The main menu screen is displayed by typing the following from a UNIX shell:

run ims

The following are some common edits that may be needed to the IMS conversion profile file:

Under [Header] section add :

Line=OPTION STRING REDIM IS LEGAL

Line=OPTION DEFAULT ARGUMENT CHECKING IS WEAK

The following are possible issues post-conversion that are not resolved by the conversion tool when converting from IMS:

dL4 **always** creates and opens relative path files with lowercase filenames on Unix and uppercase filenames on Windows. IMS allowed mixed case filenames. You can either rename existing files to the proper case or you can the LUMAP and DL4LUST parameters to specify the mixedcase or case= options.

You may need to set DL4DEFLU environment variable to replicate IMS DEFLU

Terminal ESCape sequences will work with dL4 but will not be understood by dL4Term.

The basic mode **CONVERT** command and scope mode **loadsave** command are available for converting individual programs. **Loadsave** is a command line interface, saves the file after compiling and has more options implemented than the **CONVERT** command.

convert

Synopsis

Convert UniBasic statements from a text file.

Syntax

CONVERT *textfilename* {, *alternate profile*}

Parameters

textfilename is the name of any ASCII text file that contains UniBasic program statements.

The optional *alternate profile* directs **CONVERT** to that file for conversion information. If this profile is not supplied, dL4 assumes that your conversion profile is stored within the file *convert.prf*. If the alternate profile is not supplied, you should obtain it from your installation file or tape.

Remarks

CONVERT is used when you convert your UniBasic statements from text files.

CONVERT is similar to **LOAD**, except that certain syntax conversions are automatically performed by **CONVERT** to assist in migrating programs from UniBasic, IRIS, or BITS to dL4. The **CONVERT** command converts a whole file at a time, statement by statement.

In addition to handling syntactical changes, **CONVERT** utilizes the file named *convert.prf*, or any alternate profile selected to assist in the migration of User Calls.

Difference between **CONVERT** and **LOADSAVE** is **LOADSAVE** is a scope command which also saves the code compiled and has more options implemented.

CONVERT performs the following functions automatically:

- **INDEX** #c is changed to **SEARCH** #c
- % operator is changed to **MOD**
- **CREATE** is changed to **BUILD**
- UniBasic Multi-LET with ‘,’ separators is converted to ‘;’ separator
- Inserts spaces for missing space separators in mnemonic strings – ‘**CSBU**’ is converted to ‘**CS BU**’
- Keyword collisions are corrected by appending ‘_’ to a symbol
- Characters in quoted strings are converted to Unicode characters
- **CHN** is converted to **CHF**
- Missing parentheses around function arguments are automatically added
- **ERM** is converted to **ERM\$**
- **MSF** is converted to **MSF\$**
- **STR** is converted to **STR\$**

- **REM** is not required to be followed by a space
- **RESTORE** is changed to **RESTOR**

By utilizing the conversion profile, User Calls are remapped from the pre-dL4 forms:

CALL NN, parameters or **CALL** \$NAME, parameters to the form:

Call procedure-name (arguments)

CONVERT inserts the appropriate **DECLARE** statements.

Examples

```
CONVERT ar.text arprofile
```

loadsave encodes BASIC source code from a text file into BASIC object code which is saved as an executable dL4 program. **loadsave** enables you to develop applications outside the dL4 Command Line-oriented IDE environment.

loadsave

Synopsis

Load and save a BASIC program.

Syntax

loadsave {*option switches*} *source file* -o *object file*

Parameters

option switches are optional command line options to run.

source file is a required text filename containing valid BASIC program code.

object file is the required output filename where the final encoded BASIC object code is saved.

Remarks

Option switches associated with **loadsave** are:

-h or -?	Output this help.
-e	Do not display the program source line of an error.
-l <i>n</i>	create an OSN protected program. The value " <i>n</i> " is the number of a master OSN as listed by the SCOPE OEM command.
-n <i>linenumber</i>	to specify the starting line number when using source files without line numbers. By using a starting line number other than the default of 1, an open range of line numbers can be left for use by conversion profiles that insert declarations and other header lines.
-ro	Output a run-only program (implies -s).
-s	Strip all remarks.
-o outfile	Specifies the output file for the compiled program.
-O outfile	Specifies the output file for the compiled program, capital O produces output file even if errors are detected during compilation.
-C outfile	<p>Specifies to do a source-to-source conversion, resulting in a converted program text file. The conversion profile can contain an "[OutputFormat]" section with lines such "Indentation=<i>n</i>", "LeftMargin=<i>n</i>", and "TabSpacing=<i>n</i>" to control the formatting of the output lines. The "TabSpacing" value specifies that a tab character should replace each occurrence of "<i>n</i>" leading spaces. These values can also be specified on the loadsave command line by using the "-i <i>n,m</i>" and "-t <i>n</i>" options.</p> <p>Note: Source-to-source will not report GOTO/GOSUB errors or linkage errors.</p>
-c profile	If you are converting from other versions of BASIC, you may need to use this option to convert older programs. ' <i>profile</i> ' is the name of a 'conversion profile' used to control the conversion

-u	Check program for undeclared variables (lists any numeric or string variables not dimensioned)
-v	Output the version number of loadsave .
-w	enable warning messages, ie string variables without DIM statements
-L	convert all line number references to labels

loadsave loads a BASIC program from a text file and saves it as a BASIC program file.

The -ro option creates a Run-only file which cannot be listed.

If the source file contains an error or does not exist, the object file is neither saved nor created. The object file is created only if the entire encoding process succeeds. If the object file already exists, it is overwritten.

Examples

```
loadsave {-s} {-c profile} -o outfile srcfile
loadsave -{vh?}
```

- o A simple include file feature has been added to the LOADSAVE utility.
The INCLUDE statement is recognized only by LOADSAVE and reads source text from a specified file into the program following the INCLUDE statement. The environment variable INCSTRING can be used to specify a space separated list of directories that should be searched when opening an include file. If lines in an include file use line numbers, the lines will be inserted at the specified lines replacing any previously loaded lines with those line numbers. The INCLUDE statement can have both a line number and a label. Example:

Include "filename"

Sample of convert.prf file

(can be found in /usr/lib/dl4/tools directory. Convbits.prf is provided as a sample BITS Oriented Profile.)

```
; Sample IRIS Oriented Conversion Profile for dL4
;
; A 'conversion profile' is referred to by the dL4 encoder when converting
; programs from uniBasic/BITS/IRIS, e.g. with SCOPE's CONVERT command. Its
; primary purpose is for mapping older user CALL statements into procedure
; calls in dL4.
;
; I. CALLs
;
;   The former CALL syntax, i.e.:
;
;       CALL ##,args
;   or  CALL $NAME,args
;
;   is illegal under dL4, having been replaced by the single syntax:
;
;       Call Name(args)
;
;   The "Name" refers to a subprogram procedure which may be either:
;
;       1. External (written in BASIC and saved in another program file).
;       2. Intrinsic (written in C and linked into the interpreter like old
;          CALLs).
;
;   The converter changes an old CALL statement by parsing the CALL
;   specification (number, $name, or even an expression) and searching for
;   a matching entry from this file. If not found, a unique call name is
;   substituted based on the CALL expression prefaced by "Undef".
;
;   All converted CALLs cause appropriate DECLARE and EXTERNAL LIB statements
;   to be added to the program, based on the info in this file. Intrinsic
;   CALLs cause a line such as:
;
;       Declare Intrinsic Sub TrxCo,Logic
;
;   to be added. External CALLs cause, e.g.:
;
;       Declare External Sub Time,FindF
;       External Lib "OLDCALLS.LIB"
;
;   both to be added.
;
; II. Program OPTIONS
;
;   In most cases where IRIS BASIC and BITS BASIC differed, dL4 uses IRIS
;   behavior as the default. For example, the IF (without ELSE) statement is
```

```
; line oriented in both IRIS and dL4, but statement oriented in BITS BASIC.
; This default behavior can be changed by adding the following OPTION
; statement to a program converted from BITS:
;
;   OPTION IF BY STATEMENTS
;
; The "[Header]" section of the conversion profile supports the need to add
; OPTION statements by inserting OPTION statements as the program is
; converted.
```

```
[Standard]
; These are UniBasic predefined BASIC functions.
ABS=ABS
ASC=ASC
CHN=CHN
Etc.....
```

```
[ExternalLibs]
OLDCALLS.LIB
MISSING.LIB
```

```
;add your library(s) here, ie
CUSTOM.LIB
```

```
;then add a section to list your calls, ie
[CUSTOM.LIB]
GETPART=$GETPART
```

```
[OLDCALLS.LIB]
; These are the legacy CALLs provided in version 1.7
; of "OLDCALLS.BAS".
DynWind=$WINDOW
Monitor=$MONITOR(Port,Status[],Dir$,Term$,Type$,ChanNum[],Fnms$,E),125
```

```
[MISSING.LIB]
; These CALLs are not provided by DCI, but are defined so they
; can be automatically converted to a name (i.e. there is no
; such library as MISSING.LIB, but the converter will change the
; names nonetheless).
```

```
[Intrinsic]
; These are the legacy CALLs provided in version 5.1
; of the dL4 interpreter.
ASC2EBCDIC=53
AToE=77,$ATOE
AvPort=$AVPORT
Etc.....
```

```
[Header]
; If IRIS programs to be converted use zero as a string subscript, the
; OPTION header line below should be uncommented by removing the leading
; semicolon and spaces:
;
;   Line=OPTION STRING SUBSCRIPTS IRIS
```

```
; If programs to be converted expect HAGEN string behavior (as in
; UniBasic with the HAGEN environment variable set), the header line
; should be uncommented by removing the leading semicolon and spaces:
```

```
; Line=OPTION DEFAULT STRINGS HAGEN
Line=Option Default Dialect IRIS1
```

[Settings]

```
; If the programs to be converted expect the CHF(8xx) function to return
; absolute paths, the line below can be uncommented to converted CHF(8xx)
; CHF$(13xx) which always returns absolute paths.
```

```
; ConvertCHF800To1300=True
```

```
; When programs are converted, lines are inserted at the lowest available
; line number to add needed DECLARE, EXTERNAL LIB, or OPTION statements.
; Errors may occur if too many low valued line numbers are already in
; use (for example, a program that has lines 1, 2, 3, . . .). The line
; below can be uncommented to renumber the program to start at line 100
; before the new lines are inserted.
```

```
; Renumber=100,10
```

```
; If the programs to be converted contain GOTOs and GOSUBs to non-existent
; line numbers, the line
```

```
; "ConvertUndefinedLineRefs=True"
```

```
; automatically converts the line numbers to labels
; ("Unnnn" where "nnnn" is the original line number) and append lines
; ("Unnnn: ERROR 6") to the end of the program to define those labels.
; This permits the program to run without fixing the incorrect GOTO or GOSUB
; statements.
```

```
; Comment this line to disable this setting.
```

```
ConvertUndefinedLineRefs=True
```

```
ReportUndefinedProcedures=True
```

```
;Convert IRIS programs or programs that used UniBasic in IRIS mode
```

```
Language=IRIS
```

[Edit]

```
; This section defines text edit commands to be performed on each source
; line before it is converted to dL4 syntax. The entries in the "[Edit]"
; section consist of pairs of lines where the first line defines what is
; to be replaced and the second line defines the new text. The lines
```

```
; OldText=Hello
; New=GoodBye
```

```
; would replace each occurrence of "Hello" in a source line with "GoodBye"
```

```
; The keyword in the first line defines how the search will be performed:
```

```
;
; OldText=      finds exact matches
; OldTextCI=    finds case-insensitive matches
; OldTokens=    ignores case, spaces, string values, DATA, and comments
; OldString=    searches only in string literals ("abc")
; OldMnemonic=  searches only in mnemonic literals ('CS')
; OldData=      searches only in DATA statement values
; OldComment=   searches only in comment text
```

Changes to Profile file

Here is a sampling of typical changes to be made to a .profile file when launching dL4 instead of Unibasic. (Configuration of environment variables could be done using the ub2dl4 utility or by editing the .profile manually.)

Duplicate LUST variable for LIBSTRING except replace : with space and change directory where dl4 programs instead of ub programs reside, i.e. /progs to /dl4progs and surround entire string with quotes as follows :

```
LUST=:$HOME:$HOME/progs:$HOME/files
```

```
LIBSTRING="$HOME $HOME/dl4progs $HOME/files"  
export LIBSTRING
```

LUMAP should be as follows to find data files:

```
LUMAP="SYS=$HOME/sys FILES=$HOME/files"  
export LUMAP
```

If your application is dependent on searching for the correct data files based on LUST, **instead of configuring LUMAP and LIBSTRING** you can define DL4LUST similar to the existing LUST variable as follows :

```
LUST=:$HOME:$HOME/progs:$HOME/files  
DL4LUST="$HOME $HOME/dl4progs $HOME/files"  
export DL4LUST
```

Optionally, you can leave your LUST setting as is and do the following to replicate for DL4LUST:

```
DL4LUST="(uselust)"  
export DL4LUST
```

You might want to add /usr/lib/dl4/tools to your LIBSTRING or DL4LUST to easily locate dL4 utilities. You might also duplicate DEFLU to DL4DEFLU for a default directory.

add the following to PATH :

```
PATH=$PATH:/usr/lib/dl4/printers    (and copy printer drivers!)
```

add the TERMDIR environment variable :

```
TERMDIR=/usr/lib/dl4/term  
export TERMDIR
```

File search environment variables

LIBSTRING : Like Unibasic LUST or IMS LUPATH, but applies to programs only, not data files.

LUMAP : maps a relative path to an absolute path for both program and data files. Useful for mapping 5/filename to /usr/accounting/filename. It is NOT a list of search paths.

The map "LPT1=C:\dL4\Printer1.bat" makes it possible to use "\$LPT1" as a printer name even though "LPT1" is a reserved filename under Windows.

DL4LUST : Like Unibasic LUST, and applies to both programs and data files. If LIBSTRING is defined, LIBSTRING is used first for searching programs. (IMS Basic users, explicitly specify the current directory as the first directory in DL4LUST to replicate LUPATH functionality.)

DL4DEFLU : Like IMS DEFLU, if defined then a file search will try each of the directories in DL4LUST first without and then with the value of DL4DEFLU appended. Not recommended unless porting from IMS Basic.

Assume

DL4LUST is equal to ". /usr/data test"

DL4DEFLU is equal to "5"

the current directory is "/home/fred/"

then the statement

OPEN #1,"file"

would try to open "file" using the following paths and in the following order:

```
./file
./5/file
/usr/data/file
/usr/data/5/file
/home/fred/test/file
/home/fred/test/5/file
```

More information on LIBSTRING, LUMAP, DL4DEFLU and DL4LUST are in the readme.txt for dL4.

DL4DRIVERS : allows configuration of dL4 driver selections. There are currently two DL4DRIVERS options, "Universal" and "ANSI Text". Setting DL4DRIVERS to "Universal" causes dL4 to create Universal Indexed-Contiguous or Formatted files by default.

Setting DL4DRIVERS to "ANSI Text" causes dL4 to create and read text files using the ANSI (ISO 8859-1) character set instead of the UniBasic character set. Options in the DL4DRIVERS parameter are case-insensitive and multiple options can be separated by commas.

Examples:

DL4DRIVERS="Universal"

DL4DRIVERS="universal,ansi text"

dL4

Product Training

Install

(Refer to dL4Term Reference Guide)

(Refer to dL4 Installation & Configuration Guide for Unix)

(Refer to dL4 Installation & Configuration Guide for Windows)

Shared Cache

A single memory image of a program or library can be shared between different processes and users by using a shared program cache. This feature, available only on Unix systems, can greatly reduce the amount of memory needed to support multiple users accessing large dL4 programs. Using the cache is largely transparent to both users and applications. Cached program files are accessed using normal program paths and obey the normal rules for lookup and access permission. Programs can be modified and re-SAVEd while the cache is active without disrupting other users. Any users executing the older version of the program from the cache will continue to execute that older version while new users will invoke the most current version.

Using the program cache does not require any programming changes in applications.

The program cache is enabled and configured using the new environment variable DL4CACHE. The value of DL4CACHE is a file specification of the form:

`"<access-permissions> [size] name"`
where:

"<access-permissions>" is a standard dL4 file access option such as "<644>" or "<W>". If ?[size]? is specified, then the permissions will be treated as BUILD permissions, otherwise they will be used as OPEN permissions. "<access-permissions>" is an optional value.

"[size]" specifies the size of the program cache as a number of records and a record length in bytes similar to that used for contiguous files. For example, "[256:1024]" specifies a 256 kb cache. Note that a large cache will only consume virtual memory and does not reserve physical memory. If a "[size]" value is specified, the cache will be created if it does not exist. If "[size]" is not specified and the cache does not exist, then no cache will be used.

"name" is the name of the Unix semaphore and shared memory resources used by the program cache. Any decimal ("nnn"), octal ("Onnn"), or hexadecimal ("0xnxxx") format name will directly converted to a Unix resource id value (as displayed by the Unix "ipcs" utility). Any other name will be used as a seed to generate a pseudo-random resource name. The standard cache name of "0xdddc0500" should be used unless this value conflicts with other applications or it is desired to maintain several different caches for different groups of users. "name" is a required value.

Example:

```
DL4CACHE="<666> [2048:4096] 0xdddc0500" export DL4CACHE
```

If the value of DL4CACHE is illegal or if the cache cannot be accessed, caching will be disabled. The status of the cache can be determined by using mode 0 of the ProgramCache() intrinsic as shown in the "List entries in cache" example shown in the ProgramCache() description later in this paper.

In order to use a shared program cache, the operating system must be configured to support both shared memory and semaphores. On many Unix systems, the default maximum size for shared memory will need to be increased. Please see your operating system documentation for instructions on how to configure shared memory and semaphores.

A program cache is created by the first user that enters dL4 with a DL4CACHE value that specifies a cache size and specifies a cache name that does not exist. Once created, a program cache persists until deleted by the ProgramCache() intrinsic (described later in this paper) or the operating system is reloaded. The program cache can also be deleted manually by using the Unix "ipcrm" utility to remove the shared memory and semaphore ids used by the cache.

Each user accesses the program cache in either the dynamic or the static mode. The cache mode is dynamic if a user has write access to the program cache and static if the user has read-only access to the program cache. Read and write access is controlled by the access permissions specified in the DL4CACHE environment variable (see above). Note that the mode is specific to the user and different users can be setup to use the cache in different modes.

If the user's cache mode is dynamic, all programs and libraries are entered automatically into the cache when they are used. If a new program or library is invoked and the cache is full, programs and libraries that have no current users will be deleted from the cache until sufficient space is available. If sufficient space cannot be freed, the new program or library will be loaded into the user's private memory. A typical DL4CACHE value for dynamic mode use is "<666> [16384:1024] 0xdddc0500". This provides a 16 megabyte cache with write access permitted to everyone. A larger or smaller cache can be used depending on the number and size of frequently used programs.

The cache is used in a static mode whenever a user lacks write access to the cache. In static mode, the user never enters programs or libraries into the cache. Programs and libraries are loaded into the user's private memory unless a copy of the program file has been loaded permanently into the program cache by a user in dynamic mode via the ProgramCache() intrinsic (described in a later section of this paper). Static mode has two very important advantages: it avoids thrashing and offers higher security. A cache used in static mode cannot be corrupted either accidentally or deliberately by a user.

Using a cache in static mode is more secure, but it is also more complex. The cache must be created and initialized in dynamic mode before the static mode users enter dL4. For example, suppose a system has two megabytes of frequently used dL4 libraries. At system startup time, a dL4 process would be run with a DL4CACHE value of "<644> [2500:1024] 0xdddc0500". The process would use the new ProgramCache() intrinsic (see below) to add each of the frequently used dL4 library programs to the cache. Other users would then be started in dL4 with a DL4CACHE value of "<W> 0xdddc0500" which provides read-only (static) access.

A standard intrinsic CALL, 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.

BASIC syntax:

Mode 0 - Read next entry in cache.

Call ProgramCache(0, errorcode, position, filename, usagecount)

Mode 1 - Load program into cache as a permanent entry.
 Call ProgramCache(1, errorcode, filename)
 Mode 2 - Delete cache when the current process exits.
 Call ProgramCache(2, errorcode)

Where:

errorcode - a numeric variable that 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).
 position - a numeric variable that determines which cache entry is read. "position" should be set to zero to read the first entry. Each mode 0 call will update the value of "position" so that the next call will read the next cache entry. The precision of "position" 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 "position" values of zero or those returned by the previous mode 0 call to ProgramCache().
 filename - a string variable or expression that will receive a program file path (mode 0) or supply a program file path (mode 1).
 usagecount - a numeric variable set to the number of users of the program. A usage count of -1 indicates that the program has been added to the cache as a permanent entry.

Example: 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 above for more information on dynamic and static cache modes.

Example: 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"
```

Note: If a cached program is edited and resaved, then the same program name is listed multiple times.

Example: 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.

dL4

Product Training

Tools

A “tools” directory is installed as part of the standard runtime installation. The default directory is /usr/lib/dl4/tools.

This directory contains the following utilities :

buildfi	- utility to create Full-ISAM files
buildxf	- utility to create Indexed-Contiguous files
checksum	- utility to calculate 32-bit CRC file checksums
convbits.prf	- sample conversion profile for BITS programs
convert.prf	- sample conversion profile for IRIS programs
pgmcache	- program cache utility
query	- utility to display file type and characteristics. New options have been added to the query utility. The "-p" option enables division of long displays into screen sized pages. The "-l", "-l=\$printer", "-l=path" options direct output to the "\$lpt" printer, "\$printer" printer, or the "path" text file respectively.
term	- utility to display port status or terminate programs. Enhanced to display user name and terminal name and if blocked by a record lock wait, it's state will be displayed as "Blkd" with the port number that is currently holding the locked record in parentheses. Enhanced to display the channels open on each port. The "F" option ("term all mf") shows each open channel number, the filename open on the channel, and, if supported, the current record number. The record number is followed by a letter showing the lock status of the record. The status letters are "U" (unlocked), "L" (locked), and "B" (blocked waiting for a record lock). Term has also been enhanced with a "B" option to display only ports that are blocked waiting for a record lock. For each blocked port, the utility finds and displays the port number of the program which is currently locking the desired record. The "P" option will list active ports in screen sized pages.
oldcalls.lib	- a dL4 library that implements CALL DYNWIND() and CALL MONITOR().

New as of version 5.1

UniBasic utilities have been converted to dL4. The user interface and functionality have been maintained to be identical to that of the original Unibasic utilities.

batch	- execute commands on a phantom port
bitsdir	- list directory contents (Unix only), same as Unibasic DIR command, which is like LIBR command, except it is interface compatible to BITS DIR command and output can be used with makecmnd.
change	- change filename or attributes
copy	- copy files
dokey	- access or modify Indexed-Contiguous files
format	- create formatted files
libr	- list directory contents (Unix and Windows compatible)
keymaint	- access or modify Indexed-Contiguous files (same as dokey)
make	- create multiple files with same attributes
makecmnd	- generate command files for BATCH or EXEC
makehuge	- utility to convert files to huge (> 2gb) format
mfdel	- delete a list of files

port	- display port status or evict ports (the “term” utility has extended dL4 options)
scan	- display file information
verindex	- utility to validate the index portions of Portable or Universal Indexed Contiguous files.
who	- displays information about your process

New as of version 5.2

ic2fi	- utility to convert indexed-contiguous files to full-isam or SQL files (deprecates ictofi) See end of File Drivers section for more info
-------	--

New as of version 5.3

testlock	- utility to test network file system support for record locking. Use the command “testlock -h” to display usage instructions.
Bitsterm	- utility identical in function to the uniBasic TERM utility.

Example of using BITS DIR, MAKECMND and EXEC to dump a group of programs to text

In dL4 scope :

1. Create a text file directory listing of the programs using the BITS DIR command. Be sure you are in the */programs* directory first.

BITS DIR directoryname of programs /L=filename of list @ /A T=B

BITS DIR */programs* /L=progdir! @ /A T=B

All T=B, type=basic files alphabetically in */programs* . List in file called 'progdir'

2. Use utility MAKECMND to create a text file of commands to dump the programs to text.

MAKECMND *file* USING *DIRfile*

MAKECMND dump2text! USING progdir

```
BASIC ?  
PRINT "?"  
DUMP ../textdir/?.txt!  
EXIT
```

3. Execute the command made in step 2 with EXEC command.

EXEC dump2text

This will dump all the programs to text files into */textdir* directory, with filenames ending with *.txt* (*/textdir* directory must already exist.)

dL4
Product Training
New Enhancements
dL4Term, Printers, Install

New for dL4 Term

- o When using **dL4Term with dL4 Unix**, it is now possible to open a printer on the user's PC on a channel. Opening the "Window Terminal Printer" driver opens the dL4 for Windows "Selected Page Printer" on the user's PC and allows the user to select a printer. The driver supports the same mnemonics and open options as the "Selected Page Printer" driver. The OPEN statement will return an error if the user cancels the printer selection.

The driver can be opened directly as in the statement:

```
OPEN #1,"filler" As "Window Terminal Printer"
```

or indirectly using a printer script:

```
OPEN #1,"$AUXPRINTER"
```

where "AUXPRINTER" is a script file found somewhere in the user's PATH on the Unix system. The following is an example of such a printer script:

```
# dl4opts=openas=Window Terminal Printer,options=landscape=t
```

The auxiliary printer mnemonics (such as 'BA' or 'EA') shouldn't be used when the "Windows Terminal Printer" driver is open. This feature requires using dL4Term 4.3.1.2 or later.

This is similar to using the \$SELECTLP driver in dL4 for Windows

This provides for same syntax as a Unix OPEN statement. Also the client PC does not need to be configured for auxiliary printing.

New to Printer drivers

- o Two printer scripts, `dfltp.bat` and `selectlp.bat`, are now supplied and installed with dL4 for Windows. The default Windows printer can be opened as `"$dfltp"`. A user selected Windows printer can be opened as `"$selectlp"` which will cause a printer dialog to be displayed. Examples:

```
OPEN #2,"$dfltp"  
OPEN #99,"$selectlp"
```

- o Two new open options have been added to the Windows Page Printer driver. The `"LPI=n"` option selects a default font size such that "n" lines per inch will be printed. The `"CPI=n"` option selects a default font size such that "n" characters per inch will be printed. The two options can be used together. The following example shows a printer script that uses 8 lines per inch, 10 characters per inch, with half inch horizontal and vertical margins:

```
rem dl4opts=openas=selected page printer,lpi=8,cpi=10,hmargin=36,vmargin=36  
rem  
rem dL4 selected by dialog printer script
```

- o A new printer mnemonic, `'nLANDSCAPE'`, has been implemented to select landscape mode if "n" is 1 and portrait mode if "n" is 0. The mnemonic should only be used at the beginning of a page.
- o A new mnemonic `'n LPI'` has been implemented to set the number of lines per inch when printing to the Page Printer driver.
- o A new mnemonic `'n CPI'` has been implemented to set the number of characters (columns) per inch when printing to the Page Printer driver. A second form, `'n,d CPI'` has been implemented to use the fraction "n/d" when setting the number of columns per inch.
- o A new mnemonic, `'MARGIN'`, has been implemented to set horizontal (`'w MARGIN'`) and vertical (`'w,h MARGIN'`) margins in the Page Printer driver. The arguments to the mnemonic are margins expressed in grid coordinate system units. For example, if the current coordinate grid is tenth inches (`'100GRIDENGLISH'`), then a half inch left margin can be set by `'5MARGIN'` or a half inch left margin combined with a one inch top/bottom margin can be set by `'5,10MARGIN'`.
- o A new open option, `"BINARY=<boolean>"`, has been added to the Page Printer drivers. This option causes all characters printed to a printer channel to be sent directly to the printer without any processing or translation. All characters sent in this mode must be between 1 and 255 decimal. This option is intended for applications that need complete printer control. Example:

Open #1,{", "binary=true"} As "Default Page Printer"

- o A new special output macro can be defined in the "[OutputMacros]" section of a terminal or printer definition file. The "Illegal" macro can be used to define a single character to be output whenever an illegal output character is output. Example:

```
[OutputMacros]  
Illegal=?
```

New to dL4 runtime parameters

- o A UniBasic compatible PORTS runtime parameter has been implemented to set a user's port number according to the user's terminal name. By default, dL4 for Unix tries to use the numeric portion of the terminal name as the port number. For example, a user logged in on /dev/tty14 will try to use port number 14 if it is available (and assuming the PORT runtime variable isn't set to explicitly select the port number). The PORTS runtime parameter makes it possible to select which port number is used with a terminal name. The PORTS value is a list of colon separated terminal names where the first terminal name is port 0, the second name is port 1, and so on. If the specified terminal name contains an asterisk ("*"), a wildcard match will be performed and, if a match occurs, the current port number plus any number from within the wildcard portion will be the port number. If a name begins with a pound sign("#"), the number following the pound sign is used to set the current port number for any subsequent terminal names. A name of "#any" will cause subsequent terminal names to use the highest available port number.
- o The PORT runtime parameter has been extended so that a value of "any" will cause the highest available port number to be used regardless of the terminal name.
- o A new runtime parameter, "MINPORT", has been defined to set the minimum port number to be used when automatically generating a port number.
- o A new runtime parameter, "AVAILREC", can be used to specify the value returned by SEARCH as the number the records available in indexed contiguous files. If the "AVAILREC" parameter is not set, the SEARCH statement will return, as it did in previous dL4 releases, the actual number of records available from the file free list or a minimum value of one.
- o A new runtime parameter, DL4STOPDUMP, has been implemented to control whether a program dump should be written when a program exits via a STOP statement. If the environment variable DL4STOPDUMP is defined as an absolute path, then any program that exits via a STOP will cause a program dump file to be written to the path. The path can use the macro variables and other features of the DL4PORTDUMP runtime parameter.
- o A local and private cache for dL4 programs has been implemented to improve program load performance, particularly when programs are loaded from remote networked file systems. The feature can be enabled by setting the environment variable DL4LOCALCACHE to the size of the desired local cache in bytes on the Windows client. The cache resides in process memory, is not shared with other users, and may increase process size. Programs in the local cache are identified by the absolute path of the program file. If a program file is modified while it is in the local cache, the old version in the local cache will continue to be used until the program is flushed from the cache by lack of use or a local SAVE command to the file. To

effectively use the local cache, programs should be in the first program search directory (for example, the first directory in the LIBSTRING environment variable).

The local cache is controlled with the pgmcache utility and CALL PROGRAMCACHE.

New ability to run dL4 programs as executable scripts

- o A new, optional program file format is now available for Unix systems to allow direct execution of dL4 program files from a shell command line. The new format starts each dL4 program file with the line `#!/string` where string is a program path. For example, the command:
save <755> (exec=/usr/bin/run) programname
would save the current program into the file "programname" as an executable script with an initial line of `#!/usr/bin/run`. The program file could then be executed directly at a shell command line by typing "programname". The program file would also be usable in dL4 SCOPE, in CALL/CHAIN/SPAWN statements, or in dL4 for Windows. The "exec" line can include options RUN such `'-t ""'`.

In addition to the "exec=" option, executable Unix scripts can be produced by using the "stdexec" or "netexec" options which are equivalent to "exec=/usr/bin/run" and "exec=/usr/bin/run -t".

Program files can be made executable on Windows systems by using a unique filename suffix such as ".dl4" and then associating that suffix with the required command line.

dL4 Product Training

OSN's

Protect your investment with OSN's

An OSN (OEM Security Number) based form of program protection is available in dL4. This protection mechanism is very similar to the SSN (Software Security Number) mechanism used by DCI and the PSAVE method in UniBasic.

In summary, the process works as such :

1. DCI supplies a Product Description Number, referred to as a PDN, for your product.
This is issued only once for a product (software package) and should be closely guarded.
2. When a new installation is configured, a hardware device license # or software license # is
assigned, along with an SSN number to activate licensed DCI products.
3. The developer creates an OSN number for each license # by running the *makeosn* utility
and entering the PDN and license #. (Running *makeosn* with a *-m* option creates both a
master and user OSN. A master OSN can list PSAVE'd files on the licensed machine.)
4. The OSN number is placed in the Unix */etc/DCI/osn* file or in the Windows Registry to
enable programs PSAVE'd with the related PDN.

A program saved with the PSAVE command can be loaded only on systems that have been authorized with a developer supplied OSN. Any attempt to run protected programs on an unauthorized system will cause an error 265, "Not licensed to load or create this program".

PSAVE protected programs can be modified and re-*SAVED* on any authorized system, but they can be listed only on systems which have been authorized with a master OSN. The ProgramDump intrinsic can be used in protected programs and any errors that occur while attempting to list a source line will be ignored (variables names can always be listed).

The dL4 PSAVE mechanism differs from UniBasic in two ways:

1. There is no *"-o"* startup option to add new OSNs. Instead, OSNs are added by creating and/or editing the text file */etc/DCI/osn*. (Or on Windows platforms, editing the Software | Dynamic Concepts | Passport | OSN registry entry. The registry entry name is your product name with the OSN number as the string value (REG_DZ).)
2. There is no *"-t"* startup option to add a temporary OSN. A temporary OSN is instead added by using the SCOPE *"OEM TEMP"* command which will prompt for an OSN that will be used only by the current SCOPE session.

OSNs are created with the *makeosn* utility that is supplied as part of Passport version 3.6 or later. A PDN (Product Description Number) is required to use *makeosn*. Please contact the Dynamic Concepts Sales department for information on how to obtain a PDN.

To support this protection method, two commands are available SCOPE: OEM and PSAVE.

The OEM command lists the currently authorized OSNs. "M" is shown next products where a Master OSN is present.

If the TEMP option is used ("OEM TEMP"), the OEM command will first prompt for a temporary OSN to be used only by the current SCOPE session.

The OEM command can be used in the SCOPE command, BASIC, and debug modes.

The PSAVE command is used to create OSN protected programs. The PSAVE command is identical to the SAVE command except for an optional OSN number that can precede the SAVE filename. For example, the command "PSAVE 2,menu" would save the current program as "menu" after protecting it to require the second OSN listed by the OEM command.

Protected programs can be created only if the specified OSN is a **master OSN**.

The PSAVE command is available in the SCOPE command and BASIC modes.

A new option, "-l n", has been added to the SCOPE SAVE command and to the LOADSAVE utility to create OSN protected programs. The value "n" is the number of a master OSN as listed by the SCOPE OSN command.

dL4

Product Training

Unix Source Control

Development with Source Control

Since dL4 Source programs are saved as standard text files standard source control utilities can be used. Also 'Make' scripts can be written to create a 'package' of programs.

Refer to Unix 'man' pages or other Unix reference material to familiarize yourself with how to utilize utilities such as SCCS, make and m4.

SCCS can be used to maintain source control.

A 'make' file can be created to manage an automated compiling of the application programs. The makefile is then processed by the make utility.

'm4' include commands can be used to insert pieces of code into a program before compiling.