Files and Devices

Reference Guide

Revision 1.0

Chapter 1 - Introduction

This guide is written for experienced programmers. It is a reference that describes the dL4 programmer's Input / Output interface between the dL4 programming language and external devices, files and database systems.

Information concerning individual statements and functions can be found in the dL4 Language Reference Guide.

This guide is divided into topical sections which describe the concepts and various types of supported I/O objects and operations within dL4.

Syntax Notations

The following partial list of notations are used herein to describe syntax. For a complete list of all such notations, please refer to the dL4 Language Reference Guide:

NOTATION	STANDS FOR	MEANING
chan.expr	Channel expression	An expression that combines a channel number followed by three optional numeric parameters, commonly indicating a record number, a field position, and a timeout value.
chan.no	Channel number	An integer value, between 0 and 99 inclusive, preceded by #, that the program uses for a logical connection between a BASIC program and a file.
crt.expr	CRT expression	An expression used for cursor positioning, e.g. @x,y.
expr	Expression	A valid series of constants, variables, functions, and operators to define a desired computation.
filename	Filename	A string literal or expression containing a name which is optionally preceded by a relative or absolute directory pathname.
file.spec.items	File specification, items	A file specification expressed as a list of items.
file.spec.str	File specification, string	A file specification expressed as a string expression.
num.expr	Numeric expression	An expression yielding a number.
str.expr	String expression	An expression yielding a string value or a string variable.
str.lit	String literal	A quoted sequence of characters, e.g. "string".
struct.name	Structure Name	The name of a pre-defined, fixed grouping of variables defined at compile-time.
var.list	List of variables or expressions	An arbitrary number of comma separated variables of any dL4 data types.
var.mat	Matrix Variable	Any numeric matrix variable name.
var.name	Variable Name	A variable name.
num.var	Numeric variable	A variable of numeric data type.
str.var	String variable	A variable of string data type.
struct.var	Structure variable	A variable of structure data type.

Typographical Conventions

This guide uses the following typographic conventions:

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

Channel I/O

I/O Using Channels

All Input and Output (I/O) between dL4 and external devices, files, or databases is performed by first establishing a logical, numbered, connection to a target object and thereafter directing commands and I/O operations through that numbered connection. Such numbered connections are called *channels* and are further defined as any numeric expression which, when truncated to an integer results in a positive value in the range 0 to 99.

In order to perform commands or direct Input or Output operations through a *channel*, a connection must first be established between dL4 and the deisred object. Connections are typically initiated by a request to either **OPEN** an existing, or **BUILD** a new object on a specific *channel*.

The Unification of I/O Via Channels

From the point of view of a dL4 program, **all** I/O with the outside world occurs via an operation of some kind to an open *channel*. Even the seminal statement:

```
Print X
```

is literally interpreted as:

```
Print #OUT;X
```

where #OUT is substituted with the number of the "default output" channel, which is typically a hidden (i.e. > 99) channel number referring to the terminal or window. The function of **PRINT** itself can be described in terms of even lower-level operations, so that the above example is nearly equivalent to:

```
BUFFER$ = X
Write #OUT;BUFFER$
```

Conversely, **INPUT X** is (nearly) equivalent to:

```
Read #IN;BUFFER$
X = BUFFER$
```

where #IN is the default input channel.

Record Locking

Locking is a feature implemented by many dL4 drivers to provide programmers an ability to block access to all or part of an object opened on a *channel*. Used primarily during updates to shared information, locking ensures that information is not erroneously updated by more than one user at a time.

Locking is essential in applications where two or more users may attempt to update the same information simultaneously. For example, one user in the Receiving Department might be performing a Purchase Order receipt, adding to inventory, while another in the Order Department is pulling inventory to fill an order. Applications must be written to ensure that all such updating operations are exclusive so that no information or transactions are lost. By employing Locking, whenever two or more users (or processes) attempt to

access the same information, the first is granted access, while additional requests are suspended (or an error is given) until the information is unlocked by the first user and made available.

For example, the first user is updating stock received into inventory. The part number is entered and its associated *record* is read and locked. The second user entering that part number for an order is suspended. The first user enters the amount received and the *record* is updated and unlocked. The second user now continues, reading the updated inventory record. In most cases, the transaction occurs so quickly, that other users are not aware that they are suspended. This assumes, of course, that the first user didn't leave the *record* locked indefinitely. A *timeout* feature provides programmer control over how long to wait for locked information.

A *deadly embrace* may occur when two or more users are attempting to access information which is locked by the other. Both users wait indefinitely for the other to unlock the information. For example, user 1 has locked the ABC Company customer information and is attempting to read the parts file information for wool carpet. Meanwhile, user 2 has already locked wool carpet and tries to read ABC Company. Each waits indefinitely for the other. Some systems return a system error when a *deadly embrace* is detected.

You can avoid infinite suspension of a program by specifying a *time-out* or period of time (in tenth-seconds) to wait for a locked information. If, after that amount of time the information is still locked, an error is generated to the program so that it may decide upon the next course of action.

Channel Operations, Expressions and Parameters

Once a connection is established with either **OPEN** or **BUILD**, programs direct operations on a *channel* by specifying:

- What: Specify a channel operation to perform, such as reading or writing data
- Where: Specify the *channel expression* which identifies the *channel number* and *parameters*
- With: Specify a list of arguments for the operation

Channel Operations

The following list of dL4 statements are used to perform operations through channels. The exact syntax and function(s) performed by the statement are documented in the dL4 Language Reference Guide. The purpose of documenting these statements herein is to facilitate a discussion on their behavior on an object by object basis.

ADD #	Add data to a channel. This low-level operation adds information to the driver.
ADD INDEX #	Add a new Index.
ADD RECORD #	Add a new record.
BOX #	Draw a rectangle or square.
BUILD #	Build a new object.
CHANNEL *op* #	Perform custom operation *op* on an opened channel.
CLEAR #	Clear an open channel - If the channel is opened to a newly built file, delete the file, otherwise the Clear operation is identical to a Close.
CLOSE #	Close an open channel - If the channel is opened to a newly built file, close the file making it permanent.
DEFINE RECORD #	Establish a record definition.
DELETE INDEX #	Delete an index from a Full-ISAM file.

DELETE RECORD #	Delete an existing record.
DUPLICATE	Copy a file.
EOPEN #	Exclusively open a file for single-user access if supported by the driver and operating system.
GET #	Obtain driver-specific parameters from channel.
INPUT #	Perform an Input statement from a channel.
KILL	Delete file(s).
MAP #	This low-level operation logically maps information on a channel.
MAP RECORD #	Logically Map a physical record layout to a structure
MAT INPUT #	Assign keyboard/file input to a matrix.
MAT PRINT #	Print contents of an array or matrix.
MAT RDLOCK #	Read an array, matrix, or string with locking.
MAT READ #	Read {lock} a matrix / binary string.
MAT WRITE #	Write {lock} a matrix/binary string.
MAT WRLOCK #	Write an array, matrix, or string with locking.
MODIFY	Change filename or a file's attributes.
MOVE #	Move a window.
OPEN #	Open an existing file for reading and writing or open a driver.
PRINT #	Redirect normal PRINT format to a channel.
RDLOCK #	Read and lock a record.
READ #	Read {lock} data from a channel.
READ RECORD #	Read entire structure and update indices.
RECV	Receive a message.
REWIND #	Reset the channel to the first record and byte.
ROPEN #	Open a file for Read-only, ignore locks.
SEARCH #	Maintain the index portion of a file.
SEND	Transmit a message to another port.
SET #	Read and write driver-specific parameters on a channel.
SETFP #	Set the file position for sequential transfers.
SIGNAL	Transmit/Receive a message.
SIZE #	Select size of window in columns and rows.
TRACE #	Enable statement trace debugging.
UNLOCK #	Unlock any locked record on a channel.
WINDOW	Maintain a window.
WOPEN #	Open a file/device for write-only.
WRITE #	Write {lock} data to a channel.
WRITE RECORD#	Unlock any locked record on a channel.
WRLOCK #	Write and lock a record.

Channel Expression - *#chn.expr*

A *channel expression, chn.expr,* consists of the **#** character followed by a *channel number* and zero to three optional numeric parameters, all separated by comma. The three optional numeric parameters usually indicate a record number, a field position, and a record-lock time-out value. However, it is possible for these parameters to indicate something else as the meaning of these parameters are dependent upon the object being operated upon. A *channel expression* always terminates with a semi-colon.

SYNOPSIS

STATEMENT #*channel* {,*parameter1* {,*parameter2* {,*parameter3*}}}**;***expr.list* {;}

DESCRIPTION

STATEMENT specifies any dL4 BASIC statement that performs an operation to an opened channel.

channel is any *num.expr* which, after evaluation, is truncated to an integer and used to select one of 100 possible open channels. The *channel* must be in the range 0 to 99. Hidden *channels*, outside of this range, are reserved for system use.

The optional *parameter1* is any *num.expr* which, after evaluation, is truncated to an integer. *parameter1* is typically used to select a starting *record* number for an operation.

The optional *parameter2* is any *num.expr* which, after evaluation, is truncated to an integer. *parameter2* is typically used to specify the *item number* (*field number*) or *byte-displacement* in a *record* for an operation.

The optional *parameter3* expression is any *num.var* which, after evaluation, is truncated to an integer. *parameter3* is typically used as a *timeout*, meaning the maximum time (in tenth-seconds) to wait for a selected *record* to become available and unlocked. If, after the specified *time-out* the *record* is still locked, the error message "Error in statement *stm.no*/ Record is locked" is returned to the program and an error number 123 is returned by **SPC**(8). If the *time-out* is (-1) or omitted, retry continues indefinitely. Any *time-out* is terminated immediately upon the *record* becoming available.

Occasionally, it may be necessary to supply *parameter2* without specifically supplying a value for the preceding parameter. Negative values typically specify a 'default' condition for such *parameters*. For example, the channel expression #5,-1,5; supplied to a **READ** statement might result in a read operation on channel 5 to the next record (-1) and the fifth item of that record. Any use of negative parameters, as well as the requirement of how many parameters are required, is a function of the **STATEMENT** and the type of object opened on the *channel*. The meaning of a negative value used as a placeholder is typically:

PARAMETER	ACTION
-1	The record selected is the next record after the one used in the last operation the channel.
-2	The record selected is the same record that was used in the last operation the channel.
-3	The record selected is the previous record to the one used in the last operation the channel.

This document details the use of individual statements and their function through an opened channel to all of the standard dL4 embedded objects.

Arguments and Parameters

A list of arguments, parameters, expressions, variables, etc., may follow the *chn.expr*, if the statement syntax permits or requires arguments.

The *expr.list* may contain a list of variables or expressions for the operation. However, if the channel operation is of a type which reads information from the channel, it is generally illegal to specify expressions. Data may only be read into variables.

If the **OPTION** statement "**OPTION FILE ACCESS RAW**" is in effect, an optional trailing **;** means nothing and is ignored. But if the default **OPTION** of "**OPTION FILE ASSESS STANDARD**" is in effect, an optional trailing **;** may be used with some statements, such as **READ** or **WRITE**, to control the

locking or unlocking of the current information being processed. For those statements which support a locking feature, termination with a semi-colon results in the unlocking of any current *record*, otherwise the *record* remains locked.

To maintain a record lock after reading information, simply omit the optional '**;**' at the end of a statement. To write part of a record and retain the record lock, omit the optional '**;**' at the end of a statement. To perform a read or write and unlock the record, include a trailing '**;**'. To unlock any previously locked information on a channel without performing a transfer, issue the statement:

```
UNLOCK #channel;
```

or

```
WRITE #channel;;
```

The *channel* may be the only parameter if the statement requires no additional parameters, or if it is followed by a semi-colon, i.e. **PRINT #3;**. Additional numeric parameters are parsed until the first semi-colon is seen. An error occurs whenever more than (4) parameters are processed prior to observing a semi-colon terminator.

Channel Functions and Operations

Various parameters of an open file or device may be obtained with Channel Functions. The argument ***xnn*** must specify an operation (***x***) as listed in the tables and an open channel number (***nn***).

String Functions

CHF$(xnn)	Operation Performed
1	Return Open Modes Selected. ("RWLE").
6	Return Driver Class
7	Return Driver title
8	Return Filename, portable if possible, based upon current working directory.
9	Return File owner
10	Return File group
11	Return File protection

Numeric Functions

CHF(xnn)	Operation Performed
0	Return Current file size in records of 512-bytes
1	Return Current record within the file
2	Return Current byte position within the file
3	Return Record length in words (See File Unit Option)
4	Return File size in bytes
5	Return Record Length in bytes always returns 512
6	Return Header size always returns 0
7	Return error
8	Return error
9	Return File owner in numeric form - error if the information cannot be expressed as a numeric value
10	Return File group in numeric form - error if the information cannot be expressed as a numeric value
11	Return File protection or permissions, expressed in numeric form - error if the information cannot be expressed as a numeric value

dL4 Drivers and Classes

For each and every connection between an imbedded or external object and a *channel*, dL4 assigns a specific internal program, written in C and called a *driver*, to have authority and responsibility for all operations directed through that *channel*. These *drivers* are, in fact, interfaces between dL4 and external

objects to which dL4 may communicate. The implementation of a *driver* defines the operations a program may perform on any specific object.

Some examples of *drivers* include the Text File and Window Driver. The Text File driver is a relatively simple driver, responsible for reading and writing ASCII characters to simple, unformatted files, devices or pipes. However, the Window Driver is quite complex - its responsibilities include drawing complex output and maintaining colors and multiple 'windows' on a dumb terminal.

Because there will likely be one or more objects which behave more or less identically, *drivers* are further said to belong to a *class*. The term *class*, when applied to a dL4 driver, is meant to describe a pre-defined interface and expected behavior of a channel *driver* together with the object it drives, whether that object is a file or a device or some other object, either physical or abstract. Every dL4 *driver* identifies itself as belonging to a particular class, represented by a simple ASCII string, such as "Text-File". Each object follows a set of rules, e.g., a specific Text file behaves in accordance with the rules governing Text files in general, further defined by any unique rules imposed by a specific type of Text File, such as a Unix Text file, DOS Text File, Macintosh Text File, IRIS Text File, etc.

Therefore, the driver implements a class-defined object according to a predefined set of rules. For example, consider a classification of Text files. All the printers in this class follow certain rules and exhibit certain behaviors and the dL4 programmer can operate on them. "All Unix Text files" might be an implementation of this class, so one or more Unix Text file drivers are included in the Unix text file class.

However, Macintosh text files, Unix Text Files and DOS Text files differ on how they represent an end-of-line-break. The dL4 programmer does not need to be concerned with whether to write a carriage return, linefeed or both; the real issue is that an end-of-line-break is to be inserted. The class rules for a text file specify that a **PRINT** statement which is not terminated by a **;** results in an end-of-line-break being inserted in the file. The programmer can then code with this model in mind. The actual text-file class driver, Dos, Mac, or Unix handles the actual writing of the characters. Correspondingly, when reading a text file, the **READ** or **INPUT** statements have rules when operating on a file in the Text-File class. The programmer codes to that standard and does not have to worry about the source material.

UniBasic programmers have essentially worked with the idea of driver-classes for years. For example, it has long been understood that in the statement:

```
Read #C,R,I;A$;
```

the parameter I is interpreted as a *field number* if the channel #C is linked to a Formatted File, but as a *byte displacement* when channel #C is a Contiguous File. Furthermore, it is known that if #C is a Contiguous File, then the fact that the statement is **READ** causes reading to continue through null bytes.

dL4 recognizes the sum of these traditional characteristics of Contiguous (and Indexed-Contiguous) files as a class of drivers, known as "Indexed-Contiguous". These characteristics are attributed by dL4 to a class, not to a driver or the object itself. This attribution allows dL4 to support more than one type of "Indexed-Contiguous" driver. In fact, there are currently three Indexed-Contiguous drivers: the UniBasic Indexed-Contiguous, the Portable Indexed-Contiguous, and the Full ISAM Bridge.

The dL4 BASIC Interpreter purposely remains ignorant about the actual effect or behavior of most channel I/O statements. Therefore, in:

```
Read #C,R,I;A$;
```

the Interpreter simply collects the arguments R, I, and A$ -- then passes them to the driver controlling channel #C, requesting that the driver perform a **READ** operation. What this operation does is unknown to the Interpreter, and remains under the complete control of the driver. Hence it may cause the reading of bytes from a file, or it may cause megabytes to be FTP'ed from Switzerland. Whatever its action is, the BASIC programmer knows what class of object is open on channel #C, and what effect a **READ** from that object has.

Benefits of Using dL4 Classes and Drivers

dL4 classes make applications development modular. When an application for a specific class is created, this application runs on all the drivers contained by that class. The programmer writing code for a particular class is writing code for all members of that class. The modularity comes in being able to fit an application into any software context where an application of that class is required.

In contrast, sometimes an application designer knows and desires to exploit some specific features, available on a specific driver implementation within a class. In such cases, the programmer is willing to sacrifice class portability and may explicitly do so within the application. The application becomes then self-documenting, because the programmer specifies the "FoxPro Full-ISAM" driver, rather than a more generic "Full-ISAM" class. The programmer will receive an error if the specific driver is not available.

When programming to the specification of a class, rather than a specific driver of a class, the programmer defers defining the default objects with which the application communicates. For example, the programmer simply requires that an end-user installation include a Full-ISAM database engine and Fax server, rather than specifying a specific brand name database and fax server. As market and customer needs change, new technologies may be substituted without any redesign or reprogramming.

Again, in contrast, if the programmer specifies the Microsoft SQL Server Full-ISAM driver, the application now requires that specific component be present, thereby restricting installations to NT and Windows 95 platforms. While it may be possible to later change all hard-coded occurrences of the Microsoft SQL Server to, say FoxPro, the application likely exploits special features only available on the Microsoft SQL Server. Careful planning and long-term objectives must be weighed when deciding to code to a specific model.

More on Record Locking

The dL4 interpreter, and its supporting drivers, implement a robust locking system designed to eliminate problems inherent in multi-user transaction processing systems. To accomplish this, dL4 drivers place a lock on the *record* being read or written at the start of a given statement. The operation or data transfer is performed in its entirety and the lock is only then removed if the statement terminates with a semi-colon. This method ensures that only one user is transferring the information, in its entirety, at one time. In those cases where the programmer is not actually locking the information, the operation completes so quickly that it has no apparent effect on other users who may also be requesting the same information.

To implement Record Locking, dL4 relies on the underlying operating system to place and maintain locks in a manner consistent with networks and other applications. The differing of such locking implementations across platforms leads to confusion only when the programmer designs with a specific operating system model in mind. The following discussion of locking is the expected dL4 behavior across all operating systems, platforms and networks supported.

Operating System Support for Record Locking

dL4 relies on the Operating System to properly implement local and remote file access, such as Open, Close, position within a file, Read, Write and Lock. These are standard system calls available on all operating systems. DL4 does not use any special network system calls. The Operating System must be properly configured.

The dL4 system is oblivious to any third-party software which is inserted between the Operating System and a target object. Any such software installed must provide a full and robust implementation and behave according to the published interface documented by the Operating System API and the network service it is supplying. dL4 may not work with third-party network software that improperly implements locks.

For example, dL4 may request the Operating System to Open a file for access by a user program. A call is made to the NT Operating System to **OPEN** the file. While dL4 does not know where the file is, or what internal NT or third-party software is involved in the location and opening process, dL4 does expect only a finite number of responses from NT and acts accordingly. If, as a result of information supplied or returned by third-party software, NT returns the incorrect, or some other condition, dL4 will likely fail to operate correctly.

Implementations of Locking

Under both Posix and Win32 platforms, record locking is implemented by attempting to place a write lock on the selected record at the beginning of any read or write operation. If another process, or user, has already locked the record, the operating system rejects the request to lock the record and the operation cannot be performed. If the driver supports a record-lock-retry feature, the value specified as *parameter3* in the *chn.expr* is typically used to control the retry.

0 Generate an immediate error if the record is locked.

-1 Wait indefinitely until the record is unlocked. This is the usual default condition whenever *parameter3* is missing from a *chn.expr*, and the driver supports record-lock-retry.

+n Wait until the record is unlocked, up to a maximum of n tenth-seconds. For example, a value of 150 would retry every few seconds for a maximum of 15 seconds. If the record was still locked, an error would be generated.

If no other process has already locked the record, the lock is placed and the read or write operation is performed. Upon completion, the record remains locked unless the statement is terminated with a ';'. When a file is accessed as a RAW file with the statement **OPTION FILE ACCESS RAW**, the '**;**' terminating feature is disabled.

NOTE: Any locked *record* on a channel is automatically removed on any of the following:
Closing the channel
Trailing semi-colon on the last operation unless OPTION FILE ACCESS RAW is in effect
Access to the same *record* without again locking
Attempted access to any other *record*.
Unlocking the channel using Write #c;; or Unlock #c;

Reading through Record Locks

Some drivers support the specification of <*attributes*> when the *channel* is first Opened. The optional "L" attribute, supported by some drivers, requests the driver to avoid placing or checking Record Locks prior and after Read or Write operations. Because of the obvious problems introduced by an ability to disable record locking, specification of the L attribute usually requires the inclusion of the "W" attribute to disable writing to the channel. The **ROPEN** statement uses the "L" and "W" attributes when it opens a file.

When operating with the"L" attribute, the driver will attempt to read data from a channel without checking or placing a lock on the selected record. The actual behavior of such an operation is driver and system dependent. Programmers must take note that Posix systems, which employ advisory locking, will typically allow Reads through locked records, while Win32 systems, which use mandatory locking, may block a Read operation. It is therefore recommended that you utilize the *timeout* facility and not expect the ability to read-past-locks on all operating systems.

When opening the same file on multiple channels, in either the same program or a Swap process never place two simultaneous locks on the same record. To do so will likely cause the operating system to combine

them and treat them as a single lock. When one program unlocks the record on the one channel, the operating system will likely unlock the record on the other.

Only a single *record* may be locked on any given channel. If you need to lock several records at once, you must open the file on separate channels.

You may not lock the same *record* on two or more channels simultaneously. Any attempt by a program to place a second lock on a record already locked on another channel by that program or its SWAPped ancestor is an operating system dependent operation. When opening the same file on multiple channels (with or without SWAPping), a program should never try to place two simultaneous locks on the same record. Opening and then reading from a second channel is safe if locking was disabled on the second channel with a statement such as:

```
OPEN #c,"<WL>filename"
```

dL4 Character Sets

One of the unique design characteristics of dL4 is its independence from the largely restrictive, traditional 7-bit ASCII and 8-bit character sets. Instead, dL4 considers character sets as one of the properties of the embedded and external objects with which it interfaces. The implementation is both simple and elegant and allows dL4 to operate upon virtually any character and symbol.

All data internal to dL4, whether encapsulated within a program or manipulated as data, is represented using the industry-standard Unicode 16-bit character set. This character set is more thoroughly defined in the following pages, however it is a set capable of representing the worlds language and symbol characters.

Whenever data must be exchanged between dL4 and the outside world, it is converted to and/or from Unicode and the usually more restrictive character set recognized by the object. Whereas, any 8-bit character can always be converted to and from Unicode, obviously not all 65,536 Unicode characters can be represented by a single 8-bit character. Therefore, an error is generated when attempting to convert (write) data to an object if the particular object does not support that character.

In most computer environments, an 8-bit character set (256-characters) is available to the programmer to represent a set of characters. In the USA and Europe, this set is usually the traditional alphabetic, numeric and symbolic characters, along with some special-use local characters, which can be represented on a variety of objects, such as terminals, printers and databases. As one switches between various European character sets for example, country dependent characters are substituted within the 8-bit model. Given the limited number of characters available, special-use characters maybe ‘forced’ into a set by manufacturers to cater to local populations. Without a standard, conversions may be required to ensure that a given character is represented identically on a terminal screen and printer.

Taken further, every dL4 class and/or driver supports one or more character sets. These are the defined set of characters which may be correctly represented by or within an object. In order to retain the integrity of data within an object supported by a dL4 driver, it must remain knowledgeable about that character set and perform any conversions consistently and, without error. A driver which improperly maintains an objects character set, is improperly implementing that object.

Some drivers supported by dL4 are capable of supporting a number of different character sets by actually recording character set information within the object. For example, a Portable Contiguous File may contain virtually any standard or custom character set because this object is unique to dL4 and we have the opportunity to write the rules concerning its behavior. Changing or extending the rules (or character sets) will have no impact on other software because all other access to this data is through a DCI gateway software product, or development system and API supplied with dL4.

However, the FoxPro database system supports only the specific ANSI character set, meaning that only characters within that set may be correctly read and written to a proper FoxPro database. Whereas a modified dL4 driver could allow the reading or writing of additional characters, doing so would likely invalidate the file as a FoxPro database file. More specifically, other applications would not understand the now ‘custom’ character set being imposed on the FoxPro database by dL4.

Intrinsic Character Sets

dL4 includes a number of built-in character set translations. As described, all data internal to dL4 is stored and operated upon using the Unicode character set. Conversion tables facilitate the translation between Unicode and those character sets typically used for terminals, printers, databases and flat files. The standard dL4 character sets are:

- "ASCII" (synonyms "US-ASCII" and "ISO 646") which is standard 7-bit US ASCII. Identical to Unicode 0x0000 - 0x007f.

- "ANSI" (synonyms "ANSI Latin 1" and "ISO 8859-1") which is ASCII plus the Latin 1 character set. Identical to Unicode 0x0000 - 0x00ff
- "IRIS" (synonym "IRIS-ASCII") is IRIS ASCII (high bit set) plus the IRIS mnemonic characters. In this set, standard 7-bit ASCII characters are represented by adding 0x0080 to each character, therefore placing printable ASCII characters in the range 0x0080-0x00ff. The values 0x0000-0x007f are used to represent a list of IRIS terminal and screen mnemonics.
- "uniBasic" (synonym "uniBasic-ASCII") is UniBasic ASCII (high bit zero) plus the UniBasic mnemonic characters. In this set, standard 7-bit ASCII characters are in the range 0x0000-0x007f. The values 0x0080-0x00ff are used to represent a list of UniBasic terminal and screen mnemonics.
- "IBM Code Page 437" is IBM code page 437.
- "IBM Code Page 850" is IBM code page 850.
- "Windows" (synonym "Windows Code Page 1252") is a Microsoft extended version of ANSI with some extra characters.
- "EBCDIC" (synonym "EBCDIC 037") is one popular form of EBCDIC.
- "UTF-8" is a multi-byte encoding of Unicode.

In addition to Intrinsic Character Sets, dL4 supports user-defined character sets. Users may dynamically add and remove character set translation tables under application program control. This feature allows users to establish and map their own custom set of Unicode characters into a 8-bit character mapping table. This feature is helpful when it is necessary, for example, to replace or add a local currency character, mnemonics or graphic characters to a character set. In order to be able to write and read custom character sets to a driver, that driver must support multiple character sets. For more information, refer to the dL4 Language Reference Guide **CALL CustomCharacterSet ( )**, and the Portable Contiguous documentation in this document.

Unicode Character Set

The first edition of the Unicode Standard contains over 28,000 characters from the world's scripts. These characters are more than sufficient for modern communication, as well as classical forms of languages such as Greek, Hebrew, Latin, Pali, Sanskrit and literary Chinese. Over 20,000 unique characters defined by national and industry standards of China, Japan, Korea, and Taiwan are included. The Unicode standard also includes math operators and technical symbols, geometric shapes and dingbats.

Unicode is used internally for all text processing in dL4. Externally, the various drivers at the I/O level perform any necessary translation to the appropriate 8-bit character set for a given file or device. Not all hardware devices are capable of displaying or printing the full complement of Unicode characters. The techniques used to handle unrenderable characters vary in different drivers or contexts, for example, consider the case:

```
Print "© 1997 Dynamic Concepts, Inc." ! If renderable
Print "\251\ 1997 Dynamic Concepts, Inc." ! If not
```

The copyright symbol is 251_8 in Unicode. In other words, what constitutes "non-printable" characters is completely file-/device-dependent, and not determined until I/O time.

Note that while this Print operation may be perfectly valid to a terminal, window or printer, the © symbol has no conversion or position within the older legacy IRIS and UniBasic character sets.

Supported Drivers

A *driver* is a program, embedded within dL4, to provide for interaction and communication between an application program and an embedded or external object, such as a file, database or device.

Default dL4 configurations include a number of driver classes and implementations which provide the programmer with the ability to interface with a number of standard objects. Some driver classes offer more than one driver, giving the programmer a choice. By designing programs according to the rules of a driver class, dL4 facilitates a high degree of portability. Applications may later communicate with other drivers in the same class with little or no reprogramming required.

The following Classes of drivers are supported by dL4:

Class	Drivers and/or Comments
Window	Default Window, Phantom Window, Terminal Window, Default Terminal Translation, Generic Terminal Translation, Win32 Window Terminal Translation
Text	Text, Unix Text, DOS Text, Macintosh Text, Pipe to Command, Pipe from Command
Profile	Profile Driver is used to process profile files, such as Terminal drivers and Bridge files
Index	c-tree Index. Manages the Indexed portion of an Indexed-Contiguous file
Formatted	Portable Formatted, UniBasic Formatted
Contiguous	Portable Contiguous, UniBasic Contiguous
Indexed-Contiguous	Portable Indexed-Contiguous, UniBasic Indexed-Contiguous, Full-ISAM Bridge
Full-ISAM	FoxPro Full-ISAM, Restricted FoxPro Full-ISAM, Microsoft SQL Server Full-ISAM.
Raw	Raw File, Raw Regular File. This lowest-level driver may be used by the programmer and is used by other drivers to access a file. No conversions of data are performed when accessing data through the Raw driver.
Directory	POSIX Directory, Directory. Directory Driver provides access to the filenames stored within a directory and supports the creation of a directory.

NOTE: In addition to the above driver list, dL4 includes the following internal drivers which, although not of interest to the BASIC programmer, provide operating system functionality to the interpreter.

AutoSelect	POSIX AutoSelect, Win32 AutoSelect. AutoSelect determines what driver should be assigned to a *channel* to manage an opened object when the application does not explicitly select the driver class or driver.
Driver List	Intrinsic Driver List. This driver provides directory-like access to the list of drivers. The SCOPE **DRIVERS** command reads driver information from this driver.
System	POSIX System, Win32 System. Returns and manages operating system specific information for dL4.

Port Communication	System V Message Queue Communication, Win32 WM_COPYDATA Communication. Implements and manages interprocess communications between dL4 applications.
Program	BASIC Program. Used to load and link Basic Programs.
Shared Memory	System V Shared Memory.

Driver Auto Selection Mechanism

Whenever Opening or Building an object on a channel, dL4 first assigns a specific driver responsibility for all operations performed on the channel. The method in which the driver selection is made is governed by the following rules.

- If the **OPEN** or **BUILD** statement specifies an **AS** clause which specifies a driver class, such as "Full-ISAM", the default Full-Isam driver is selected. The default is the first Full-Isam driver in the driver list provided by the SCOPE **DRIVER** command.
- If the **OPEN** or **BUILD** statement specifies an **AS** clause which specifies a specific driver such as "FoxPro Full-ISAM" or "Raw File", that specific driver is selected. An error is issued if the specific driver is not available. The **AS** clause is useful when the programmer wishes to override the auto-selection mechanism and is required when creating new types of files.
- The driver is selected according to the name, attributes, or contents of the file. The auto-selection process is capable of determining many of the built-in driver types, such as UniBasic, Contiguous, Formatted, Indexed and FoxPro Full-Isam.
- If the operation is to create a new object using **BUILD**and an **AS** clause isn't specified, then the value of *file.spec* (such as the presence of a record length) is used to select the class.
- If the operation is to open an existing object and no **AS** clause is specified and the auto-selection process cannot determine which driver to select, the file is opened as a Text file.

Introduction to Files

A *flat-file* or datafile is an area or areas of the disk, provided by the file system, to be considered a single data storage entity. *Flat-files* allow data to be stored and retrieved by programs, and retain their data indefinitely.

A *table* is an organized *flat-file* which provides a method of storage and retrieval of information in a more precise manner. Tables are similar to spreadsheets, made up of rows and columns of information.

A *database* is a group of one or more *tables* of information which are linked together based upon like information. An employee master table and employee history table might be part of a Payroll database.

A *device* is a physical external storage medium such as a hard-copy printer, magnetic tape, or terminal screen.

A *window* is a logical device upon which a user may draw and represent information.

Files and *Tables* may be further divided into rows or *records* and columns or *fields*.

A *record* is a row of information in a table, often a fixed number of characters taken together which form a logical 'row' of information. A customer record, for example, might store a clients name, address, billing information and other pertinent information about that customer.

Some file types require the programmer to supply a *Record number* usually origin zero, meaning a file with five records has record numbers 0, 1, 2, 3, and 4. Other file types permit operations only on the current, previous or next *record* and a physical number is not required.

A *field* is an individual column or logical 'piece' of information within a *record*. A customer record, for example, might specify a field to hold a customer name in string format with a maximum length of *n* characters. The type of file determines how a *field* may be referenced. Some file types leave all *field* management to the programmer, in which case specific character positions of a record are reserved by the programs to hold the *field*. All management of the data type, truncation, padding and size must be handled by each and every program operating on the file to maintain data integrity. Programming complexities are apparent whenever a *field* requires changes to its length, type, etc. Other types of files provide a numbering or field naming mechanism to simplify field management. These file types, including database files, perform the necessary location of the data within the physical record and may even handle padding, case conversion, truncation and data type maintenance, and conversion automatically.

Some file types require the programmer to supply a *field number* or *byte-displacement*, usually origin zero, to specify the starting point within a record to transfer data. dL4 statements allow access to specific *fields* within a record by supplying the associated *field number* or a *byte displacement*. For some tables, a logical mapping of *field names* is accomplished by assigning a logical number to each named field.

A fixed-length-record is a type of record whereby each and every record of a file is identical in length.

A variable-length-record is a type of record whereby each and every record is possibly of a different length. Text files are perfectly suited to a variable-length approach and terminate each record with a unique character, such as a new-line or carriage return. In a text file, each line could be considered a record. Variable-length-record files are usually more difficult to manage as databases, since it is often necessary to read through them sequentially. Because records are not fixed in length, record 11 cannot be located until one reads through the first 10.

A fixed-format record is a type of record whose organization is identical in every position of the file. Used extensively in databases, each record contains the same 'fields' as the others. Every customer record has an 'address' field. This 'address' field is the third field of the record.

There exist many combinations of the above *record* and *field* types within the file structures supported by dL4. The common file classes are:

- Text Files are variable length record files, separated by a line-break character, with a single string field in each record.
- Formatted Item files are fixed-length, fixed-format, numbered *fields* with truncation and padding on the field level.
- Contiguous files are fixed-length, variable format record formats. All formatting of the record is left to the programmer. These files are well suited for databases where data diversity is required and the programmer has needs or wants to maintain the data.
- Indexed Contiguous files are Contiguous files with a programmer maintained variable number of Keys and indices. All record and key management, such as inserting and deleting are left to the programmer. Indexed Contiguous files offer the programmer fast lookup of records by key value and a high degree of control over the entire format and content of the records, fields and keys.
- Full-ISAM database files are fixed-length, fixed-format, named or numbered fields with truncation and padding on the field level. Keyed access is provided as well as maintained automatically whenever records are added or deleted. The number and type of keys, made up of field elements, is defined when the file is created. Full-ISAM files are easier to program and less error prone than Indexed-Contiguous files.

Text File Class

A Text object is a stream of 8-bit characters terminated by a zero-byte or the physical end-of-data. Text drivers are used to communicate with text objects, such as character terminals, printers and text files. Text files are files which contain characters, usually in human readable form, in a variety of character sets, such

as ASCII, ANSI, EBCDIC, etc. Word processing files, although they contain text data, include formatting information and are not a true example of a text file.

For purposes of random access, Text Files are assumed to have a record length of 512 bytes. Data begins in the first byte of the file and there is no special header of information imposed by most operating systems. Text files are good examples of variable length record files - each record of information is a line of text are separated by an operating system specific line-separation character.

Note: When Text Files are created, the driver typically stores data in native format to ensure compatibility with other local text editors, word processors and other programs. It is important to note that the local format is determined by the version of dL4 that you are running. That is, running the Windows version results in default Windows-style text files - even if those files are being created on a Unix server. Be sure to include an **AS** clause when specific formatting behavior is required.

Text objects are typically accessed sequentially. When data is written to a Text driver, an operating-system specific new-line identifier is written after each record. On Unix systems, this is a line-feed. Windows systems utilize both a carriage return and a line-feed. Macintosh systems use only a carriage return.

For programming convenience, a separate column counter is maintained for a channel connected to a text driver. Printable characters increment the column; new-line or form-feed operations reset the counter to zero. When **TAB** functions are used to the open channel (ie writing to a *device* such as a printer), the separate column counter forces the system to treat the local and channel devices exclusively.

When writing short to an existing file, i.e. in the middle of a file, a zero byte terminator is maintained at the end of each write operation. In such cases, a zero byte is written and the file pointer is decremented so that each subsequent write operation overwrites the trailing zero byte, and appends a new zero-byte at the end-of-file. When a new file is being created, or an existing file is being extended, a zero-byte is not required or added at the physical end-of-file.

Note: When data is being read from a Text File, End-of-File is signified by the occurrence of a zero byte, regardless of whether data exists beyond the zero-byte. Further Read operations will return null data.

Special Options with Text Files

Option File Access Raw

The File Access Raw option causes the text driver to generate an End-of-File error, rather than simply returning a Null string on a read operation. This option also generates the error during random access when positioning or reading at or beyond the end-of-file.

Types of Text Files

For compatibility purposes, the standard character-set for all text files is the UniBasic character set. This default permits the writing and reading of special characters, control characters and UniBasic mnemonics to a text driver. To create or access other types of character sets, the Charset option must be supplied.

The default driver selected when no **AS** clause is specified is determined by the platform version of dL4 being executed.

- Unix Text driver uses the single new-line character to signify an end-of-line.
- DOS Text driver uses a carriage-return new-line pair to signify an end-of-line.

- Macintosh Text driver uses a single carriage-return character to signify an end-of-line.
- Pipe to Command is used to write text to pipes, such as printers, driven through a program or script. The platform determines the handling of the end-of-line character according to the above rules.
- Pipe from Command is used to read data generated by another process through a pipe. The platform determines the handling of the end-of-line character according to the above rules.

Creating Text Files

BUILD

Create a new Text File. The creation of a text file may be specified using the *file.spec* parameters, however the number of records and record item length are ignored. Even though a text file has no header per se, the (option item) *charset* may be specified during open to select the type of character translation to invoke. If no *charset* option is specified, the native UniBasic character set is used.

To specify that **BUILD** is to create a text file, a **+** must precede the *file.spec*, or the **BUILD** statement must contain an **AS** clause.

```
Build #channel, + “file.spec”
Build #channel, “file.spec” AS “Text”
```

The **AS** clause may specify the default **Text**, or may specify one of the other specific text drivers, such as **Macintosh Text**.

Opening and Closing Text Files

CLOSE

Close the associated channel. If the file was newly created, make it permanent.

CLEAR

Clear the associated channel. If the file was newly created, delete the file.

OPEN, **EOPEN**, **ROPEN**, **WOPEN**

OPEN opens an existing text file. Supplemental Open options L, W, R and E are supported on a limited basis. The *charset* option may be specified to select a specific character translation for the file. Although no auto-selection process is provided for character sets, the **AS** clause selects a specific driver which, to a limited extent, specifies some translation of the handling of end-of-line. For further information on the specific drivers within the Text file class, see below.

EOPEN is equivalent to **OPEN** with <E> specified as part of the *file.spec.str* and opens the file for exclusive access. Exclusive Open is only supported when the underlying Operating System provides mechanisms for exclusive access. An error is generated if the support is not available under the host operating system.

ROPEN is equivalent to **OPEN** with <WL> specified as part of the *file.spec.str* and opens the file for read-only access.

WOPEN is equivalent to **OPEN** with <R> specified as part of the *file.spec.str* and opens the file for write-only access.

The special *pipe* drivers may be opened explicitly, or through the inclusion of a single or double $ within the *filename* portion of the *file.spec*. Use the single or double $, such as $PRINTER or $$ls -l respectively, to open a write-to or read-from pipe

```
Open #channel, “$file.spec”
Open #channel, “file.spec” AS “Pipe From Command”
```

Positioning Within Text Files

A *chn.expr* for Text drivers supports the specification of all three optional parameters, in the general form:

#channel { , record {, byte-displacement {, timeout }}} ;

The optional *record* specifies a block of 512-bytes within a file. A *record* cannot be specified for a Pipe From or Pipe To text driver. The following table illustrates the acceptable values when specifying a *record*. For positioning purposes, all Text drivers assume that the object has a record-length of 512-bytes.

An optional *byte-displacement* specifies the starting byte position within a 512-byte block of data. A *byte-displacement* cannot be specified for a Pipe From or Pipe To text driver

Positioning within a text file is performed by taking the result of (*record * 512) + byte-displacement* and using that to specify a specific byte within the file. For example, the values *record=1, byte-displacement=4* selects byte (512*1+4) or byte 516 within the file.

Typically, text files are accessed sequentially and positioning within a text file is not an issue. When it is desirable to position within such a file, it is usually required to read through the file in order to obtain current position information for later use with a request to reposition. This is especially important whenever a multi-byte character set, such as UTF-8 is used. A given *byte-displacement* is **not** a character position in all cases.

Record number	**Action Performed**
omitted	Continue to access sequentially from the last operation. Following Open, access the first character.
-1	Same as omitted. Continue the access sequentially.
-2	Special Read operation provides an ability to Read a specific number of characters and not terminate on a newline.
-3	Illegal.
value	Set the position to record *value*. Taken with the *byte-displacement*, reposition the file. If no *byte-displacement* is specified, or it's value is -1, assume a zero *byte-displacement*.

Byte Displacement	**Action Performed**
omitted	Continue to access the text file from the current byte-displacement. Sequential Access.
-1	Same as omitted. Sequential Access.
value	No meaning if param1 is -1. Taken as the byte-displacement if param1 is a value

The optional *timeout* is typically used when accessing devices, such as keyboards or com ports, to provide for a period of time to wait for data.

Record Locking is not supported by Text drivers, however a *timeout* may be included for those cases where an application, other than dL4 has placed locks on text data or a text device.

Rewind #c;

Reset the text file to the first character position within the file.

Setfp #c,p1,p2;

Set the file pointers to record p1 and byte displacement p2. Text files are assumed to have 512-byte records for positioning purposes. Therefore, any given position, expressed as record p1, byte-displacement p2 is computed as the byte-position within the file at p1*512+p2.

All positioning assumes an 8-bit character set. If the text file contains a multi-byte character set, such as UTF-8, arbitrary or computed file positioning with Setfp is impractical. In such cases, you can position

within the file to a specific character only when you are sure that position represents the start of a multi-byte character sequence.

```
Setfp #c,Chf(400+c)/512, Chf(400+c) Mod 512
! Position to end of file (used to append to a text file.
```

Record Locking with Text Files

Record Locking is not supported by any dL4 text driver and a programmer may issue the following statements without error.

Unlock #c;

Write #c;;

It is important to note that dL4 neither supports nor enforces locked text records. This means that should another third-party application place locks on a text file, you may still be able to read or write to the file or device. The success of such an operation is dependent upon the method of locking in place within the operating system. Advisory locking systems, such as Unix, will not prevent you from reading or writing to the device or file, whereas a Windows system will block all operations to a locked resource.

If you plan on accessing text objects which might be accessed (locked) simultaneously by other applications, care should be taken and the *timeout* option should be used to provide for recovery from a lockout condition

Reading and Writing Data with Text Files

Input and Mat Input

Re-direct the normal Input statement to read characters or a line of data from the channel. The specific text file driver will always convert any operating system specific line termination to a single carriage return, as if the data were entered from the keyboard and the return key pressed. Therefore, as in keyboard input, a null string represents a blank line within the text stream.

An End-of-file error is generated when all text has been read from the file.

Print and Mat Print

Following the rules of the Print and Mat Print statements, the arguments are prepared for display by converting them to printable characters and then re-directing ‘lines’ of output to the channel. The end-of-line character(s) appended by Print and Mat Print statements are converted by the driver to the format required by the specific type of text driver.

Read and Mat Read

Reading characters from the channel is performed by the Read and Mat Read statements. Both statements operate identically on Text drivers.

A read statement must specify a *str.var*, and terminates on the transfer of a null byte, end-of-file, end-of-line (return), or when the {specified or dimensioned} length of the variable has been read. If the read terminates on an end-of-line character, a carriage return (‘CR’ or \015\) character is placed in the string as the last character read.

An error is generated if you attempt specify a *num.var* or *expr*.

Multiple arguments may be specified, and the read operation is performed for each. If the argument is a string array, a Read operation is performed for each element of the array. A structure variable (consisting of all string members) is treated like a list of individual arguments.

You may read through any end-of-line characters and instead read a specified number of subscripted characters, or the Dimensioned size if no subscripts are supplied, by specifying a *record* of -2. In this case,

all end-of-line characters are converted and represented as single carriage returns (CR \015\) within the *str.var*.

Write and Mat Write

Writing characters to the channel is performed by the Write and Mat Write statements. Both statements operate identically on Text drivers.

A Write statement must specify a *str.var*, and terminates upon the transfer of the {specified or dimensioned} length of the variable. Whenever writing in the middle of an existing file, a null byte is written following each argument, unless the argument was subscripted. Subsequent write operations overlay the null byte, unless the file is being expanded. This process ensures that the driver can maintain a future 'end-of-file' position whenever re-writing over an existing file.

An error is generated if you attempt specify a *num.var* or *num.expr*, however, a *str.expr* may be specified. To write a single null byte to the file, the *str.expr* "" may be specified.

Multiple arguments may be specified, and the write operation is performed for each. If the argument is a string array, a write operation is performed for each element of the array. A structure variable (consisting of all string members) is treated like a list of individual arguments.

Table of Text Driver Options Supported

A driver related statement to create or define the channel may specify optional items as indicated by italics in the following general form:

`STATEMENT #chan;"<protection>(option)$cost[record]filename" As "driver"`

Statements	Optional Items	WIN32	UNIX
BUILD		Y	Y
	<Protection Item>		
	(Option Items)		
	(charset=UTF-8)	N - Syntax Error	N - Syntax Error
	$Cost Item		
	$9.99	Y (ignored)	Y (ignored)
	[Record Item]		
	[100:80]	N - Syntax Error	N - Syntax Error
OPEN		Y	Y
	<Protection Item>		
	< R >	Y	Y
	< W >	Y	Y
	< E >	Y	N - mode not supported
	< L >	Y	N - mode not supported
	(Option Items)		
	(charset=UTF-8)	Y	Y
	$Cost Item		
	$9.99	Y (ignored)	Y (ignored)
	[Record Item]		
	[100:80]	Y	Y
EOPEN		Y	N
	<Protection Item>		
	< R >	Y	N/A
	< W >	Y	N/A
	< E >	Y	N/A
	< L >	Y	N/A
	(Option Items)		
	(charset=UTF-8)	Y	N/A
	$Cost Item		
	$9.99	Y	N/A
	[Record Item]		

	[100:80]	Y	N/A
ROPEN		Y	Y
	<Protection Item>		
	< R >	N - mode not supported	Y
	< W >	Y	Y
	< E >	Y	N - mode not supported
	< L >	Y	Y
	(Option Items)		
	(charset=UTF-8)	Y	Y
	$Cost Item		
	$9.99	Y	Y
	[Record Item]		
	[100:80]	Y	Y
WOPEN		Y	Y
	<Protection Item>		
	< R >	Y	Y
	< W >	N - mode not supported	Y
	< E >	Y	N - mode not supported
	< L >	Y	N - mode not supported
	(Option Items)		
	(charset=UTF-8)	Y	Y
	$Cost Item		
	$9.99	Y	Y
	[Record Item]		
	[100:80]	Y	Y

A driver related statement to access the channel may specify optional items as indicated by italics in the following general form:

```
STATEMENT #chan,parameter1,parameter2,parameter3;passed_parameters;
```

Statements	Parameter1	Parameter2	Parameter3
READ INPUT MAT READ RDLOCK	record number	byte displacement	timeout
WRITE PRINT WRLOCK	record number	byte displacement	timeout

Pipe Drivers

Pipe drivers are typically used to provide an interface to a device such as a printer. But they are much more than a printer driver. Pipe drivers may be used to call operating system specific utilities to perform operations that may be cumbersome for dL4 or other language programs. They must be executable programs or scripts and reside in a directory specified by the **PATH**. An error is generated if the commands they call do not exist or are not in the **PATH** if given as a relative filename.

Types of Pipe Drivers

An output pipe is opened by preceeding the filename with a single '$' in the *file.spec.str* of the **OPEN** statement. The **AS** clause of the **OPEN** statement may also be used. For example:

```
Open #chan,"file.spec.str" As "Pipe to Command"
```

An input pipe is opened by preceeding the filename with '$$' in the *file.spec.str* of the **OPEN** statement. The **AS** clause of the **OPEN** statement may also be used. For example:

```
Open #chan,"file.spec.str" As "Pipe from Command"
```

dL4 does not support bi-directional pipes.

Creating a Pipe Driver

On a UNIX system the pipe program may be a shell script or an executable program with parameters, such as "ls -l". The shell script may have an options line interpreted by the dL4 pipe driver. To use pipe driver options the first line of the script file must begin with "`# dL4opts=`". Options use the format "*`keyword=value`*". The supported keywords are "`charset`" and "`lock`". The "`lock`" value is a filename, for example:

```
# dL4opts=charset=utf-8,lock=/tmp/lpt2.lk
```

This options line specifies that the pipe driver should send all output to script file in the UTF-8 character set. It also specifies the filename of the lock file to be created.

Subsequent lines will contain the commands and parameters necessary to process the data.

The shell script file must have its permissions set to make it executable.

A Windows system pipe program may be an executable program (.exe or .com) or a script must be a batch file with a filename extension of ".bat". If an options line is required, it must be the first line and begin with "rem dL4opts=". Options use the format "*`keyword=value`*". The supported keywords are "`output`", "`translate`", "`charset`", and "`lock`".

Subsequent lines will contain the commands and parameters necessary to process the data. Following is an example of an input shell script to pipe a directory list to a dL4 program:

```
# dL4opts=charset=utf-8
ls -l
```

Another example of an output shell script might be one to sort dL4 output to a file. For example:

```
# dL4opts=charset=utf-8
sort -o sorted
```

takes the output of dL4, sorts it, and puts it in a file called sorted.

To use direct printing, the printer script must specify an "`output`" option to select the output device and a "`translate`" option to select a printer definition file path. For example, the following script outputs to the device LPT1 after translating output using the printer definition file "c:\dl4\printers\hplj":

```
rem dL4opts=output=LPT1,translate=c:\dl4\printers\hplj,lock=true
```

A printer script using direct output should consist of a single options line; all subsequent lines will be ignored. A direct output script always uses the Unicode character set and ignores any "`charset`" options.

Opening and Closing Pipe Drivers

The **OPEN, EOPEN, ROPEN,** and **WOPEN** statements may be used to associated a channel with a pipe driver. The general form is:

```
Open #chan,"$file.spec.str"
```

or

```
Open #chan,"file.spec.str" As "Pipe From Command"
```

for an output pipe and is:

```
Open #chan,"$$file.spec.str"
```

or

```
Open #chan,"file.spec.str" As "Pipe To Command"
```

for an input pipe. A character set option item may be specified as part of the *file.spec.str* of the **OPEN** statement, but a character option in the pipe driver script will take precedence. The default is the UniBasic ASCII character set. Record items,[100:10], and permissions items, <A>, are accepted in the *file.spec.str*, but not a cost item, $cost.

The **CLOSE** and **CLEAR** statements may be used to close the associated channel.

Locking with Pipe Drivers

The pipe drivers do not support locking records, but a script or device being accessed by the pipe driver script may be locked by including a lock option in the first line of the script. For UNIX the lock option must specify the absolute filename of a lock file. The syntax of the lock option is:

```
# dL4opts=lock=/tmp/lpt1.lk
```

For Windows the lock option is expressed as a boolean value, for example:

```
rem dL4opts=lock=true
```

The lock prevents concurrent opens of the batch script file and thereby allows only one process to execute the script at a time. The default lock option for both Windows and UNIX is none.

Reading and Writing with Pipe Drivers

The **READ, RDLOCK,** and **INPUT** statements may be used to receive data from an input pipe driver. The *record*, *byte-displacement*, and *timeout* entries are not accepted. An example, using the input pipe shell script described above to read a directory:

File "pipein" listing:

```
# dL4opts=charset=utf-8
ls -l
```

dL4 program using input pipe shell script "pipein":

```
Dim a$[512]
Open #2,"$$pipein"
Do
  Read #2;a$
  Print a$
Loop Until a$ = ""
Close #2
End
```

The **WRITE, WRLOCK,** and **PRINT** statements may be used to send data to an output pipe driver. The *record*, *byte-displacement*, and *timeout* entries are not accepted. An example, using the output pipe shell script described above to sort data to a file:

File "pipeout" listing:

```
# dL4opts=charset=utf-8
sort -o sorted
```

dL4 program using output pipe shell script "pipeout":

```
Data "s","z","a","d","t"
Dim a$[5]
Open #1,"pipeout" As "Pipe to Command"
For i = 1 To 5
  Read a$
  Print a$
```

```
  Print #1;a$ + "\12\"
Next I
Close #1
End
```

Profile Class

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**.

Types of Profile Drivers

The Profile class has the Profile driver.

Creating Profile Files

Profile files are just text files and may be created with the Text file driver or any text editor. The only thing special about Profile Files is the layout of the data as described above.

Opening and Closing Profile Files

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

```
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.

Use the **CLOSE** statement to close the channel associated with the Profile file.

```
CLOSE #chan
```

Where *chan* is the valid channel number on which the Profile file is OPENed.

Positioning within Profile Files

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

```
SEARCH #chan, str.var
SEARCH #chan, str.var1, str.var2, str.var3
```

The first form is used to perform a search for a specific *Section Name*. *str.var* contains the named *Section Name* to locate. The search operation is case insensitive. If *str.var* is null, a search is performed forward to the next section. *str.var* 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. *str.var1* contains the named section, *str.var2* the *left* label, and *str.var3* 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.

Record Locking with Profile Files

The Profile driver does not support any locking mechanism, due to the read-only implementation.

Reading Profile Files

Profile files are designed to be read-only. The current implementation does not support writing to the file.

```
READ #chan,record,item;str.var {,str.var}
```

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 = .

str.var 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.

```
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
```

Formatted Files Class

The Formatted File drivers support a type of data file or *table* consisting of identically defined, fixed-length *records*. Formatted files are well suited to store application template information and information which is user-specific or temporary in nature.

While usable in some database applications, Formatted File drivers provide random access only by record number and offer no high speed lookup or searching capabilities. Instead, they rely on the application having its own high-speed mechanism to categorize and locate specific records within the file. In fact, Formatted File drivers are similar to, albeit low-level versions of, Full-ISAM Database drivers. They provide high speed access for applications which only require access to numbered *records* and *fields* and require no high-speed search capabilities or record and field mapping.

Formatted Files are frequently used to store information based on such a unique *record* number, such as a user account number, terminal device *port number*, or some other session identifier, such as a *process id*.

Each *record* of a Formatted File is identically formatted, that is it contains the same type and number of *fields*. The format is initialized through creation and the writing of the first *record* and is then maintained for the duration of the file's existence. Records are numbered, starting at zero. During the creation process, the writing of data to record zero in progressive item or *field* numbers results in the writing of data and the definition of the record format.

Formatted File Drivers provide automatic truncation and padding when data is written whose length does not exactly match the field definition. Conversion services are not supported by the driver - it is the programmers responsibility to correctly differentiate between string, numeric and binary information. The driver will, however, perform numeric type conversions for some precisions of numerics supported.

Each item or *field* is defined to be one of the standard dL4 data types, including string, numeric, date or binary.

Special Options with Formatted Files

There are no special **OPTION** Statement directives which alter the behavior of Formatted files.

Types of Formatted File Drivers

Standard Unix versions of dL4 include two different Formatted File drivers, UniBasic and Portable. However, on Windows systems only the Portable driver is provided. Because UniBasic was never available on Windows, no Formatted files were ever created on that platform and therefore, a dL4 program running on Windows should not have a need to read or write to a local UniBasic Formatted file. If such a need arises, UniBasic version 6 and higher provides for creating Universal files, which are platform independent. Existing files may be converted to Universal files if they meet certain requirements.

The UniBasic version of this driver is provided on those platforms on which a previous version of UniBasic was released. The primary purpose for including this version is to provide for sharing of files between UniBasic and dL4, as well as eliminate the need to convert such files to dL4 portable format.

UniBasic Formatted File Driver

The UniBasic Formatted driver supplied on any given dL4 platform is limited to operating on UniBasic Formatted files created on that same platform. For example, a dL4 system supplied for platform 99 (SCO Unix), is capable of creating, reading and writing SCO UniBasic Formatted Files. Since UniBasic files are not portable, the UniBasic Formatted driver on SCO cannot, by default, be used to read or create UniBasic Formatted files for AIX.

While the current offering does not include such drivers, the dL4 development system does permit the creation of specific UniBasic Formatted drivers which are platform specific. For example, a Windows or SCO version of dL4 could include other Formatted drivers, such as UniBasic AIX Formatted File. In the current release of dL4, no such additional drivers are included, however the source code for the UniBasic Formatted driver is included to facilitate later or customer addition of such drivers.

While the basic access methods are identical for both dL4 and UniBasic, existing and newly created UniBasic Formatted files must remain compatible with UniBasic and therefore do not allow the reading or writing of newer dL4 data types. UniBasic Formatted files support String, Numeric and Binary types. The Binary type is forced by using the **MAT** statement to initially write a numeric array, numeric variable or string variable.

The record length in a UniBasic Formatted file can be up to 65534 bytes in length, with each record containing from 1 to 128 items or *fields*.

A null record is returned when access is made to a record below the maximum record number, but not physically in the file.

The UniBasic Formatted File driver in dL4 does not recognize any settings defined in the UniBasic environment variable **PREALLOCATE**, namely:

- During expansion of the file, all intervening records are written (with zero bytes) from the file's current physical size up to and including the record being accessed. For example, writing record zero followed by record 1000 causes all intervening (1-999) records to be written. This may take several seconds or minutes.
- No special Record-Not-Written error is returned when a record contains all null data. Attempts to read records which are less than the physical last record number, but not logically within the file results in null records being returned.

All **MAT** operations must be to Binary field types and specify string variables only.

Portable Formatted File Driver

By default, all newly created files will be dL4 Portable Formatted. Portable Formatted files do not have limits on record size or the number of items or fields.

Universal Files

Universal Data files are IRIS BCD style Formatted files which are platform independent. The files are accessible across all Unix platforms. In addition, they are usable on a Windows system with version 3.0 and higher of dL4 for Windows. Packed data should be avoided for maximum platform independence.

Creating Formatted Files

Formatted ITEM files are created using the **BUILD** statement.

To create a Formatted Item file within an application, write to *record* zero a list of variables to sequential item numbers. The type and **DIM** of each variable is recorded in the format map. When a numeric variable is written, its precision is also stored in the format map. When a string variable is written, its **DIM**ensioned size is incremented and then rounded up to an even number of bytes. If a **MAT** operation is performed, the items are created using the actual **DIM**ensioned size. Strings are rounded up (not incremented first), and numerics occupy the entire size of the specified variable, array or matrix. The actual data within the variables is also written to the record after the item is defined in the format map.

An error is generated if items are written in other than sequential item number order starting at 0. Once an item is defined, its type, precision or length may not be changed.

Opening and Closing Formatted Files

CLOSE

Close the associated channel. If the file was newly created, make it permanent.

CLEAR

Clear the associated channel. If the file was newly created, delete it.

OPEN, EOPEN, ROPEN, WOPEN

Open an existing file. Supplemental Open options L, W, R and E are supported on a limited basis. The *charset* option may be specified to select a specific character translation for the file. Although no auto-selection process is provided for character sets, the **AS** clause selects a specific driver which, to a limited extent, specifies some translation of the handling of end-of-line.

Table of Formatted Driver Options Supported

A driver related statement to create or define the channel may specify optional items as indicated by italics in the following general form:

`STATEMENT #chan;"<protection>(option)$cost[record]filename" As "driver"`

Statements	Optional Items	UNIX	WIN32
OPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	Y	Y
	< E >	Y	Y
	< L >	N - mode not supported	N - mode not supported
EOPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N-Syntax Error
	< W >	Y	Y
	< E >	Y	Y
	< L >	N - mode not supported	N - mode not supported
ROPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - mode not supported
	< W >	Y	Y
	< E >	Y	Y
	< L >	Y	Y
WOPEN		N	N
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	N - Syntax Error	N - mode not supported
	< E >	N - Syntax Error	N - Syntax Error
	< L >	N - mode not supported	N - mode not supported

Positioning with Formatted Files

A Formatted file will return as its number of records (**CHF**/**CHN** functions), the first record not contained within the file. If your files grow dynamically using this function, no empty records exist in the file. If you **READ** a record beyond the current number of records in the file, an error is generated (Illegal Record or

End-of-file). When you **WRITE** a record beyond the current number of records, the file is expanded automatically.

Reading and Writing Data with Formatted Files

Formatted files are accessed by supplying the *record* and *field number*. Access cannot cross a logical record boundary.

When transferring data to a Formatted Item file, the *record*, and *item number* are used to specify the starting point for the transfer. All items in the *var list* are transferred, and each must match the pre-defined record layout in the format map.

If an Item is defined as string, only a *str.var* may be transferred. If the Item is numeric or date, a conversion is performed when the variable precision does not match the item's definition. Data is converted to the precision of the destination; *var* when reading, *item* when writing. An error occurs if the destination precision is too small to hold the numeric value.

Binary items are accessible using **MAT** statements. You can transfer any *str.var*, *mat.var* or *array.var* into a binary field. No conversion is performed. Care must be exercised to ensure that numeric data is transferred into variables of the same precision used when written or the resulting data will be in-distinguishable to the application..

The following table illustrates the optional use of the supplied *record*.

RECORD	ACTION PERFORMED
omitted	The *record* number used for the last access to this channel is incremented and used to select the *record*. This mode reads sequential records of a file.
-1	Performs identically to 'omitted' except that it serves as a place holder so that a *byte displacement* may be specified.
-2	The *record* is reset to the same *record* number used during the last access to this channel. This accesses the same record.

Contiguous Data Files Class

Contiguous files utilize a fixed-length *record*, specified during creation. Each *record* contains the identical number of bytes. The total number of records to be within the file is stored within the file's header during creation. Contiguous files are very similar to the BITS Tree-Structured Data files.

dL4 is compatible with applications designed to use Contiguous data files, even though the Unix systems do not support Contiguous files in the traditional internal sense.

Special Options with Contiguous Data Files

OPTION FILE ACCESS RAW changes alignment rules to match the BITS rules.

Types of Contiguous File Drivers

Standard Unix versions of dL4 include two different Contiguous File drivers, UniBasic Contiguous and Portable Contiguous. However, on Windows systems only the Portable driver is provided. Because UniBasic was never available on Windows, no Contiguous files were ever created on that platform and

therefore, a dL4 program running on Windows should not have a need to read or write to a local UniBasic Contiguous file. If such a need arises, UniBasic version 6 and higher provides for creating Universal files, which are platform independent. Existing files may be converted to Universal files if they meet certain requirements.

The UniBasic version of this driver is provided on those platforms on which a previous version of UniBasic was released. The primary purpose for including this version is to provide for sharing of files between UniBasic and dL4, as well as eliminate the need to convert such files to dL4 portable format.

Creating Contiguous Data Files

Contiguous files are created with the **BUILD** statement. A Contiguous file may have any number of records and there is no maximum record length.

Opening and Closing Contiguous Data Files

CLOSE

Close the associated channel. If the file was newly created, make it permanent.

CLEAR

Clear the associated channel. If the file was newly created, delete it.

OPEN, EOPEN, ROPEN, WOPEN

Open an existing file. Supplemental Open options L, W, R and E are supported on a limited basis. The *charset* option may be specified to select a specific character translation for the file. Although no auto-selection process is provided for character sets, the **AS** clause selects a specific driver which, to a limited extent, specifies some translation of the handling of end-of-line.

Table of Contiguous Driver Options Supported

A driver related statement to create or define the channel may specify optional items as indicated by italics in the following general form:

`STATEMENT #chan;"<protection>(option)$cost[record]filename" As "driver"`

Statements	Optional Items	UNIX	WIN32
OPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	Y	Y
	< E >	Y	Y
	< L >	N - mode not supported	N - mode not supported
EOPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N-Syntax Error
	< W >	Y	Y
	< E >	Y	Y
	< L >	N - mode not supported	N - mode not supported
ROPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	Y	Y
	< E >	Y	Y

	< L >	Y	Y
WOPEN		N	N
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	N - Syntax Error	N - Syntax Error
	< E >	N - Syntax Error	N - Syntax Error
	< L >	N - Syntax Error	N - Syntax Error

Positioning with Contiguous Data Files

As its number of records (**CHF/CHN** functions), a Contiguous file returns the greater value of its current physical size, or the size in records specified during creation.

Contiguous files are accessed by supplying the *record* and *byte displacement*. Access may cross a logical record boundary. Care must be taken to ensure that your transfers are within the specified *record* or data in subsequent records may be damaged.

Reading and Writing Data with Contiguous Files

Access to any *record* within the valid **CHF/CHN** range with either **READ** or **WRITE** statements is permitted. If the record is beyond the current physical size, the file is extended. To expand a Contiguous file, simply write to any record higher than the current size.

During expansion of the file, all intervening records are written (with zero bytes) from the current physical size up to and including the new record. This automatic filling in of records is to prevent Unix from reporting the file as sparse (i.e., containing gaps). Sparse files are usually considered corrupted when the file system is checked, although they are valid.

When transferring data to a Contiguous file, the *record*, and *byte displacement* are used to specify the starting point for the transfer. All items in the *var list* are transferred sequentially. The following table illustrates the optional use of the supplied *record*.

RECORD	ACTION PERFORMED
omitted	The record number used for the last access to this channel is incremented and used to select the record. This mode reads sequential records of a file.
-1	The record number used for the last access to this channel is incremented and used to select the record . This mode permits the selection of a new byte displacement within the incremented record.
-2	The record is reset to the same record number used during the last access to this channel. This accesses the same record.

Indexed Contiguous Data Files Class

An Indexed Data File is any Contiguous Data File which is defined to contain a companion ISAM Key file. Access to data records is identical to a standard Contiguous Data File. The companion ISAM (Indexed Sequential Access Method) file holds keys and pointers to data within the Contiguous Data File. The use of an Indexed file allows an application to rapidly locate data in a large database. Even when a file contains several hundred thousand data records, a specific record can be located instantly.

Indexed files, consisting of optional data records and keys, are maintained by the application program. When new data is to be added to the file, you request a new record. Automatically, the system expands the

file if there are no unused records. After writing your new data to the supplied record of the file, you insert a *key*, that is a unique piece of information tagged to the new record. The *key* could be a customer name, number; any unique information about the record. Later, you retrieve the record by simply asking for the record that contains the key.

Each file can have from 1 to 62 separate indices, and each index may have a different sized *key* (up to 122-bytes). This allows multiple *keys* (e.g. name, account number) to access the same data. Each different index provides a different way to locate a *record*.

Any given *record* may be located by its specific *key*. When the entire *key* is not available, a group of *records* matching a partial *key* may be displayed for final selection under program control.

Data records may be read from the file sequentially (in key order), forward or backward for as many different indices as are in the file. For example, a file keyed by customer name and number could produce a sorted (ascending or descending) report by those fields without any resorting.

When information is no longer needed in a file, the user application deletes the keys and returns the record to the system for later reuse before extending the file.

Indexed Files are not required to contain data records. A Contiguous Data File is always present with a single data record, but may be unused. This allows indices to exist separately from the data referenced, or to build key-only files into existing data bases.

The index compression algorithm has the following benefits:

- Unused space in an index is kept to a minimum. When an index block becomes empty, it is placed on the delete list. It therefore be can be reused elsewhere in the index when required.
- An index that has keys systematically added to the end and deleted from the beginning does not require the file to grow continuously.
- Since overall index size is reduced, overall access performance to the index is proportionally increased, with very large indices benefiting the most.

Indexed Data Files are maintained within 2 separate Unix files. These are a standard Contiguous Data File utilizing a lower-case name (as built), and the ISAM (key) portion in a companion file. The companion file will be a file with the same name using upper-case characters (i.e. payroll and PAYROLL) if it is a UniBasic file or a file with the same name with an extension of .idx for dL4 files.

Types of Index Contiguous File Drivers

The drivers supplied in the Contiguous class are:

- Portable Contiguous driver. Standard Unix versions of dL4 include two different Index Contiguous File drivers, UniBasic Indexed-Contiguous and Portable Indexed-Contiguous. However, on Windows systems only the Portable driver is provided. Because UniBasic was never available on Windows, no Index Contiguous files were ever created on that platform and therefore, a dL4 program running on Windows should not have a need to read or write to a local UniBasic Index Contiguous file. If such a need arises, UniBasic version 6 and higher provides for creating Universal files, which are platform independent. Existing files may be converted to Universal files if they meet certain requirements.
- UniBasic Contiguous driver. The UniBasic version of this driver is provided on those platforms on which a previous version of UniBasic was released. The primary purpose for including this version is to provide for sharing of files between UniBasic and dL4, as well as eliminate the need to convert such files to dL4 portable format.
- Full_ISAM Bridge driver. The Full-ISAM Bridge driver is available on both UNIX and Windows platforms. This driver allows existing programs that were written to access Indexed Contiguous files to also access Full-ISAM files. This allows developers to gradually migrate to the Full-ISAM files and be transparent to the existing application programs.

Creating Indexed Files

Indexed files are created using the **BUILD** and **SEARCH** statements. They are initially created with a single data record. The actual number of records supplied to the statement is stored in the file header.

```
BUILD #chan, "(countused=true)[10,40]filename" AS "Indexed-Contiguous"
```

The 'countused' option enables records-in-use counting. By default, a count of the number of records allocated is not maintained by the driver. This default results in increased performance.

```
SEARCH #chan, 0, 0; "filename", recnum, status
```

Note: A UniBasic ISAM file is made up of (2) separate files; the lower-case filename holds the data portion and an uppercase filename is created to hold the ISAM portion. Filenames that do not contain at least one letter cannot be used for ISAM data files.

During initial creation, you may specify the type of B-Tree balancing to apply to each index. Proper selection increases performance and minimizes the disk space required to hold keys. The default is to assume random key insertions into each index. This results in a well balanced tree-structure with nodes split when half full. If your insertions into a specific directory are sequential (ascending or descending), you may change this parameter to suit your application. An example of a sequential index is an order/invoice number file keyed by an increasing (decreasing) number or date. By setting the proper parameter, as much as 25% performance and a 50% reduction in disk space may be realized; See SEARCH Modes in the dL4 Language Reference Guide.

The **ISAMSECT** environment variable could be used with UniBasic to control the number of c-tree levels. If undefined UniBasic used a default of 4. This parameter is defined as a constant of 8 for the dL4 driver.

When allocating new records, the system first checks for any deleted records that can be reused. If found, they are used first. When no deleted records exist in the file, the file is expanded by one record.

Similarly, when the ISAM portion of the file is full, it is expanded by 512 bytes.

To maintain a dynamically expandable file structure, c-tree maintains a linked list of deleted records in the data portion of a UniBasic file. When records are returned to the system, c-tree checks that you have not returned the same record twice in a row. It does not normally check to see if you have returned the record in a previous operation. It is therefore possible to corrupt the Deleted Record Chain if you arbitrarily return records not actually allocated.

Deleted records in a UniBasic file are flagged with a single-byte *delete-character* (ff hex, $377_{8)}$. Next, a 4-byte pointer is written linking deleted records together into a *delete-list*. The top of the *delete-list* is maintained in the header. It is possible to corrupt this pointer system if you perform a **WRITE #** operation to a *record* following its release as a free record. Many applications write their own delete-flag into unused records. If your applications require this capability, set the environment variable **ISAMOFFSET** to a byte location other than zero (default) such that c-tree has 5 contiguous bytes available for *delete-list* maintenance.

C-tree requires internal arrays of data to maintain fast key operations such as search next. For each Indexed file your application opens, one array element is required for the data portion of the file, and one element for each Index in the file. A typical application opening 10 files with an average of 3 indices requires (3 + 1) * 10 or 40 positions. If your application errors trying to **OPEN** too many ISAM files, change the default value of the environment variable **ISAMFILES**.

Indexed files dynamically expand to meet the requirements of your application.

Opening and Closing Indexed Data Files

CLOSE

Close the associated channel. If the file was newly created, make it permanent.

CLEAR

Clear the associated channel. If the file was newly created, delete it.

OPEN, EOPEN, ROPEN, WOPEN

Open an existing file. Supplemental Open options L, W, R and E are supported on a limited basis. The *charset* option may be specified to select a specific character translation for the file. Although no auto-selection process is provided for character sets, the **AS** clause selects a specific driver which, to a limited extent, specifies some translation of the handling of end-of-line.

Table of Indexed Contiguous Driver Options Supported

A driver related statement to create or define the channel may specify optional items as indicated by italics in the following general form:

`STATEMENT #chan;"<protection>(option)$cost[record]filename" As "driver"`

Statements	Optional Items	UNIX	WIN32
OPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	Y	Y
	< E >	N - Syntax Error	N - Syntax Error
	< L >	N - Syntax Error	N - Syntax Error
EOPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	Y	Y
	< E >	N - Syntax Error	N - Syntax Error
	< L >	N - Syntax Error	N - Syntax Error
ROPEN		Y	Y
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	Y	Y
	< E >	N - Syntax Error	N - Syntax Error
	< L >	N - Syntax Error	N - Syntax Error
WOPEN		N	N
	<Protection Item>		
	< R >	N - Syntax Error	N - Syntax Error
	< W >	N - Syntax Error	N - Syntax Error
	< E >	N - Syntax Error	N - Syntax Error
	< L >	N - Syntax Error	N - Syntax Error

Accessing an Indexed Data File

An Indexed File is accessed using the **SEARCH #**statement. The parameters select the operation mode, index to operate upon, and data values passed both ways.

`Search #channel, mode, index; key.var, record.var, status.var`

channel is any *num.expr* which, after evaluation is truncated to an integer and used to specify an opened channel currently linked to an Indexed Data file.

mode is any *num.expr* which, after evaluation is truncated to an integer and used to specify a mode of operation for the statement. The following pages provide a detailed list of *mode* operations.

index is any *num.expr* which, after evaluation is truncated to an integer and used to specify an Index or Directory (list of keys) for the operation.

key.var is any DIMensioned *str.var* which must be DIMensioned large enough to hold the key being operated upon. An error is generated on search type operations if a key from the file cannot fit into the supplied *str.var*.

record.var is any *num.var* and contains (or returns) a value for the statement *mode*.

status.var is any *num.var* used to return a status (exception) value to the program. Generally, a zero indicates a successful operation; non-zero for an exception error. When issuing *mode* 1 functions, the *status.var* is set before the statement to select the miscellaneous information to be returned.

Mode 0 - Index Definition

Generally, Indexed Files are created and structured using the **BUILD** statement. **SEARCH** *mode* 0 is used to create an Indexed File during program execution.

Each index in the file is defined using a *mode* 0 statement specifying the *key* length. Indices must be defined in sequential order, beginning with 1, up to a maximum of 62. The index is selected with the *index* expression.

The *record.var* defines the key length (2-122 bytes) of the selected *index* in words unless **OPTION FILE UNIT IS BYTES** is used.

status.var is set upon completion as follows:

0 Operation successful.

4 File is not a data file (type Data or Contiguous).

6 Selected index number is out of sequence.

8 File already indexed (May not be changed once defined).

9 Illegal parameter specified. Key length can be 2-122 bytes.

10 Too many indices specified. Maximum is 62.

To create an Indexed File with two indices of key lengths (bytes) of 6 and 24 requires two *mode* 0 statements. The first to *index* 1 with *record.var* containing 6; the second to *index* 2 with *record.var* equal to 24.

As each *index* is defined, a *mode* 8 may be issued to the same *index* with *record.var* set to 0 for random insertions, 1 for increasing keys, and 2 for decreasing keys. If this step is omitted, random insertions are performed.

The data portion of an Indexed File begins with data *record* one. To force the first data record to be other than one, issue a *mode* 1, with *record.var* set to the desired first record number and *status.var* set to 6. Setting a First Real Data Record other than zero does not occupy space within a file. The system simply stores a starting record constant which is added or subtracted from all file operations. If the First Real Record is set to 200, then logical record 200 equals physical record 0; 210 record 10, and so on. This feature is included for compatibility when moving existing data files from a live IRIS system in order to keep the record numbers and key pointers consistent.

Once all indices have been defined, the file structure must be locked. This is accomplished by issuing a *mode* 0 statement with *index* equal to 0 and *record.var* set to the desired number of data records.

Once all indices are structured according to the information supplied, the file is available for key insertion, record allocation and other operations.

No further mode 0 statements may ever be issued to this file without an exception status occurring.

Mode 1—Miscellaneous Index Information

SEARCH *mode* 1 is used to access structure information about an open Indexed File. When the *index* expression is non-zero, the *key length* of the selected index is returned in *record.var*. This length is expressed in words unless **OPTION FILE UNIT IS BYTES** is in effect.

Specify *index* zero and set *status.var* to select one of the functions listed below. The value (if any) yielded by the function is returned in *record.var*.

0 Return in *record.var* the First Real Data Record as defined during creation.

1 If built with (countused=true) option, return in *record.var* the available record count. This is computed by taking the current size of the file and subtracting the actual number of active records. Otherwise return in *record.var* the number of records in the file.

2 Allocate a new record in the file returning its value in *record.var*. Possible exception status:

3 = No free records remaining and insufficient disk space prevents expansion of the file.

3 De-allocate (return) a record to the file. Available record count is incremented, active records is decremented. *record.var* supplies the record number to mark as 'available'. Possible exception status:

1 = Record number already de-allocated. If you attempt to return the same record twice in a row, this condition is returned.

4 Return in *record.var* the number of physical records within the file. Does not include the addition of the First Real Data Record value.

5 Same as mode 4.

6 Set the First Real Data Record to the value supplied in *record.var*. This function is used by the Conversion Programs, and whenever having a record zero is undesirable. This option may only be set prior to freezing the structure with mode 0.

7 If built with (countused=true) option, return the current (actual) number of records in use within the file in *record.var*. This number is maintained as records are allocated and de-allocated (See 2 and 3 above). Otherwise return a -1 in *record.var*.

Mode 2—Search for a Specific Key

SEARCH *mode* 2 is used to search an index for an exact match to the supplied *key.var*. If found, *record.var* receives the data record number associated with the *key*, and the *status.var* is set to zero. If no match is found, *record.var* is unchanged and *status.var* is set to one.

A match is indicated when the supplied *key.var* is equal to an entry in the index up to the end of *key*, even if the entry in the file is longer. When the entry is longer, its value is returned in *key.var*.

For example, a search for *key* **ABC** produces a match with the first entry whose first three characters are **ABC**. If the first such entry is **ABC Company East**, then a match is indicated, *key.var* is set to contain **ABC Company East**, *record.var* is set to the associated record number, and *status.var* is set to zero. A match is not produced if the entry in the index is shorter than the key supplied. For example, the entry **AB** is not considered a match.

Note: The actual keys are case-sensitive. This means that "ABC" does not equal "abc."

Mode 3—Search for the Next Highest Key

SEARCH *mode* 3 is used to access data records alphabetically, or to search forward from a selected point in the index. The selected *index* is searched for the first entry logically greater than the supplied *key.var*. If found, *record.var* receives the data record number associated with the *key*, and *status.var* is set to zero. When no more entries are found, *record.var* is unchanged and the *status.var* is set to two (End of Index).

For example, a search with *key* **ABC** returns the first entry logically exceeding **ABC**, such as **ABC Company East**. Subsequent *mode* 3 searches using the same *key* might yield entries such as **ABC Company West**, **Dynamic Concepts**, and **Dynamic Conversions**.

To search an entire *index*, start by setting *key.var* to a null string, and perform *mode* 3 commands until *status.var* is set to 2.

Note that a *mode* 3 search yields the first entry greater than *key*; a *mode* 3 with the *key* **ABC** does not return **ABC** itself if it exists. It is best to perform a *mode* 2 search first when you want to include the starting *key* in your search.

Mode 4—Insert a New Key into an Index

SEARCH mode 4 insert new *keys* into an index. The selected *index* is first searched for an entry exactly matching *key.var*. If found, *record.var* is set to the record number associated with the *key* and *status.var* is set to one.

If no match is found, and sufficient space exists within the selected *index*, *key.var* is inserted in the index using the record number supplied in *record.var* as a pointer to the data record. Successful insertion is indicated by a zero in the *status.var* . If no space exists within the selected index, the *status.var* is set to two (End of Index).

Mode 5—Delete an Existing Key from an Index

SEARCH mode 5 deletes existing entries from an index. The selected *index* is searched for an entry exactly matching *key.var*. If found, the *key* is removed from the index, *record.var* is set to the record number associated with the *key* and the *status.var* is set to zero (successful deletion).

If the exact entry is not found, the *record.var* is unchanged and *status.var* is set to one.

Following successful deletion of a *key*, the *record* should be returned for re-use using *mode* 1 with *status.var* set to 3.

Mode 6—Search for a Previous Lower Key

SEARCH *mode* 6 is used to access data records in descending order, or backward from a selected point in the index. The selected *index* is searched for the first entry logically less than the supplied *key.var*. If found, *record.var* receives the data record number associated with *key*, and *status.var* is set to zero. If not found, *record.var* is unchanged and *status.var* is set to two (End of Index).

For example, a search with the *key* **XYZ** returns the first key found logically less than **XYZ**, such as **Solution Systems**. Subsequent *mode* 6 searches using the same *key* might yield keys such as **Solution Concepts, Resources International**, etc.

Note that a *mode* 6 search yields the first entry less than *key.var*, so a *mode* 6 executed with **XYZ** will not yield the **XYZ** itself if it exists. It is best to perform a mode 2 search first when it is desirable to include the starting *key* in your search.

To search an entire *index*, start by setting *key* to "\377\", and perform *mode* 6 commands until 2 is returned in *status.var* .

Mode 7—Reorganize Index

SEARCH *mode* 7 provides for compatibility with older IRIS applications performing an index reorganization. This mode is a non-operation and always returns a *status.var* of zero indicating success and allowing the older program to run without error.

Mode 8—Specify B-Tree Insertion Algorithm

SEARCH *mode* 8 retrieves or changes the B-Tree insertion algorithm for an index. If *record.var* is greater or equal to zero, it's value is truncated to an integer and used to select the new insertion method for *index*. If successful, the file's header is changed, and *status.var* is set to zero. If the *record.var* is outside the accepted range, *status.var* is set to one, and no change is made.

If *record.var* is any negative value, the current insertion algorithm used for *index* is returned in *record.var* and *status.var* is set to zero.

Value	Type of Insertion Algorithm Invoked
0	Default. Selects random insertions and is used when keys in the index are inserted in any order.
1	Selects increasing insertions and is used when each key inserted in the index is greater than the previous insertion. Types of keys in this category include sequential order numbers or date keys.
2	Selects decreasing insertions and is used when each key in the index is less than the previous insertion.

Changes are stored in the file's header and become effective immediately for the user storing the change. Other users must first **CLOSE** and **OPEN** the file before the change takes effect.

By default, files are created for random insertions. Random insertions split B-Tree nodes when they are half full. This provides a better balancing and room for future insertions.

When sequential keys are inserted (ascending or descending), the nodes should be split only when full. Extra space is not required for later insertions between sequential key values.

The benefits of adding a *mode* 8 to your Application code include saving up to 50% on disk space; 25% increase in performance on insertions, deletions and searches.

Indexed File Errors & Recovery

If you accidentally delete the ISAM portion of an Indexed file, you can rebuild the file by the following steps.

1 Create a new Indexed file with a different name using the same parameters for number of Indices and Key Lengths.

2 Write a small program to rebuild and insert the keys into the new temporary file. Only insert keys and records, do not copy the existing data.

3 Use the Unix mv command or DOS rename to rename the new temporary files ISAM portion as the old files ISAM file, ie: mv TEMPFILE MYFILE or rename tempfile.idx myfile.idx. This command must be performed at the shell. Do not use any utilities designed to operate on both the lower and upper case portions of UNIX ISAM files or DOS files with idx and dbf extensions.

If an error is encountered during ISAM file access, an exception (V2=5) status may be returned. Check to see if your string **DIM** is at least the size of the Key.

Full-ISAM 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.

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.

Each directory must have one and only one key per record.

Data fields must be numeric or character.

Other types (packed) not supported.

Character fields cannot have significant data past first null.

Accessing Emulated Indexed-Contiguous Files

Cannot **BUILD** a full-ISAM file using the Bridge driver; A normal indexed-contiguous file is always built 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.

Record numbers are not physical within the Full-ISAM file:

- Valid within a single transaction.
- May vary the next time the same record is accessed.

Bridge Profile - a "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 stored under the same name as the emulated Indexed File.

Application opens a profile believing it to be an Indexed file.

Profile contains the filename, and optional driver, of underlying Full-ISAM database file.

```
Sample Bridge Profile File
[FullISAMBridge]
File=taxcodes
OpenAs=FoxPro Full-ISAM

[Record]
Field=TAX_LOCALE,0,15
Field=TAX_RATE,16,2%
Field=MTDTAXABLE,20,3%
Field=MTDNONTAX,26,3%
Field=MTDSLSTAX,32,3%
Field=QTDTAXABLE,38,3%
Field=QTDNONTAX,44,3%
Field=QTDSLSTAX,50,3%
Field=YTDTAXABLE,56,3%
Field=YTDNONTAX,62,3%
Field=YTDSLSTAX,68,3%
```

```
[Index1]
Name=TAXCODE
KeyPart=TAXCODE,0,8,"~","0123456789"
```

Index Files Class

The Index class driver may be used to manage the indexed portion of an Indexed-Contiguous file.

Types of Index Drivers

The Index class driver supported is c-tree Index. This is a low level driver used by various higher level drivers, such as in the Indexed-Contiguous class, to manage the index portion of the file. As such, the dL4 BASIC programmer will not have a need to use this driver.

Full-ISAM Database Files Class

Full-ISAM database files are designed to offer the developer an alternative to Indexed Contiguous files. Immediate benefits of Full-ISAM include:

- Providing a structured approach to data storage and retrieval.
- Access to a file is field-oriented using named fields.
- Indices are maintained automatically.
- 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 principles similar to 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.
- Full-ISAM files include a data dictionary which defines field names, types and sizes. Access to a given field is performed by specifying its 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.

```
! TESTREC defines the record structure as it will
! actually exist in the FoxPro file.

Def Struct TESTREC
      Member 1%,A   : Item "A"
```

```
      Member 1%,B   : Item "B"
      Member C$[10] : Item "C"
End Def

!Declare structure variables of the type TESTREC

Dim TheRec. As TESTREC, ReadRec. as TESTREC

! Assign values to the structure variable

TheRec.A = 19
TheRec.B = 23
TheRec.C$ = "record 1"

! Create the Full-ISAM file using the record definition

Build #1,"fi-file!" As "Full-ISAM"
Define Record #1;TheRec.
Close #1

! Add the record

Open #1,"fi-file" As "Full-ISAM"
Add Record #1;TheRec.
Close #1

! Read a record into a structure variable

Open #1,"fi-file" As "Full-ISAM"

! Set current record to beginning of file

Search > #1,1;

! Read the record into a structure variable

Read Record #1;ReadRec.
Print ReadRec.A
Print ReadRec.B
Print ReadRec.C$
Close #1
```

Using 'item' Designations in Structure Variables

Structure definitions may also include supplemental designations, typically recognized by various system file and device drivers. While most statements ignore these designations, the drivers might utilize such information for record and key definitions as well as file positioning.

Def Struct *tagname* {**: Item | Key** *structoption* }

Member *varname* {**: Item** *memberoption* **| Key** *memberoption* **| Decimals** *constant* }

tagname is the unique name tagged to this structure template.

Item and **Key** designations are synonomous. However, **Item** typically refers to record specifications, whereas **Key** refers to index information. **Decimals** is used to define the number of significant fractional decimal digits for Full-Isam database files.

structoption and *memberoption* must be either numeric or string constants. *constant* must be a numeric constant.

A colon (**:**) is used to separate **Item**, **Key**, and **Decimals** specifiers.

Multiple **Item** or **Key** specifications are concatenated with **+**.

When specifying an **Item** or **Key** designation on the structure definition itself, one is specifying information which relates to the entire structure definition. For example, one might name the structure such as **Item "Cust Record"**, or specify information pertaining to a key, such as **Key "ByZip"**.

When defining individual **Member**s within a structure, **Item** typically refers to data field information, whereas **Key** refers to index key-part definitions. When specifying **Item** or **Key** designations for individual members, one is specifying information which relates to that member only. For example, one might specify a byte-displacement, such as **Item 70**, or database field name, such as **Item "Addr"**.

Item and **Key** *structoption* and *memberoption* may be any of the following:

string	Name of record or key definition when included on the structure definition statement.
string	Name of database fieldname when included with a Member definition. Names a field within a record, or the field to use as a key part.
number	Undefined when specified with structure definition.
number	Byte displacement for Contiguous and Indexed data files
UNIQUE	Define a directory of Unique keys - when specified with the structure definition.
DUPLICATES	Define a directory of possibly duplicate keys - when specified with the structure definition.
ASCENDING	Define a directory stored in ascending order
DESCENDING	Define a directory stored in descending order
UPPERCASE	Uppercase a Member when specifying a key part.
DECIMALS x	Specify the number of fractional decimal digits for a member when specifying a key part or database field.

Map Record provides for the mapping of database fieldnames to your structure **Members**.

To access database files, the structure definition may define items using 'fieldnames', such as:

```
Def Struct Customer     Item "CustRecord" ! 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" : DECIMAL 2
End Def
```

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

```
Def Struct CustKey1   : KEY "NameCtyBal" + Unique + Descending
  Member Name$[25]    : KEY "Name" + Uppercase
  Member City$[25]    : KEY "City$"
  Member 3%,Balance   : KEY "CurrBal" + Descending : Decimals 2
End Def
```

Types of Full-ISAM File Drivers

The Full-ISAM class drivers available are:

- FoxPro Full-ISAM driver supported on UNIX and Windows platforms.
- Restricted FoxPro Full-ISAM driver supported on UNIX and Windows platforms.
- Microsoft SQL Server Full-ISAM driver supported only on the Windows platform.

Creating Full-ISAM Database Files

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.

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" : Decimals 2
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 until data has been written to the file. One way to define an index is 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 the next 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 + Descending
  Member Name$[25]    : KEY "Name" + Uppercase
  Member City$[25]    : KEY "City" + Uppercase
  Member 3%,Balance   : KEY "CurrBal"
End Def

Dim Key1. As CustKey1
Add Index #5,1;Key1. ! Directories must be defined in order
```

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

The member Name$ is an 25-character string from the data field with the same name. It is to be uppercased and stored in descending order. The field City$ is a 25-character string from the data filed with the same name. It is also to be uppercased and stored in descending 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 is supported or results in savings of disk space. In most cases, it is assumed that the file structure will reuse the empty portion of the file. 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.

Logically Mapping Full-ISAM Records & Indices

Often it is necessary to work with a subset of fields within a database or provide for later changes in the field content or order within the file. The **Map** statement allows a program to 'marry' a structure definition to the current file's data dictionary. It is recommended that applications use **Map Record** whenever a Full-ISAM file is opened so that the file field layout can be changed without effecting the program. Similary, **Map** should be used with index names to avoid dependence on particular index numbers. The general form is:

```
Map Record #channel As struct
```

```
Map #channel , index ; string expression
```

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, or named index within the database. *struct* members must have **Item** fieldname or directory name definitions.

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

string expression is any string which evaluates to a named directory within the file.

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.

Map defines the logical index or directory number used within the application. This statement allows a program to be written using a hard-coded directory number, which is then logically mapped to the physical directory number within the file.

The fieldnames given within the Customer structure are used to align each member to its current item number within the file. For example, if the field "Addr", which is item 1 in the structure, is currently item 4 in the physical record, a **Map Record** would cause the driver to perform the necessary item-number translation so that any further access to item 1 will actually access item 4.

This kind of dynamic record access not only insulates the application from certain modifications to the file structure, but also could be used by individual programs to limit record accesses to only those fields which are directly used. Depending on the format of the underlying record data (which is subject to the rules of the actual file being driven; FoxPro, etc.), this may circumvent unnecessary data conversion and thereby boost performance.

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 ; 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 current record to be deleted.

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>" of Indexed-Contiguous files. 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
  Member Name$[25]    : KEY "Name", Ascending, Uppercase
  Member City$[25]    : KEY "City$", Ascending, Uppercase
  Member 3%,Balance   : KEY "CurrBal" : Decimals 2
End Def

Dim Key. as CustKey1
Key.Name = "Acme" ; Key.City = "Toledo" ; Key.Balance = 0
Search  = #C, I; Key.   !Exact search
Search  > #C, I; Key.   !Search Greater
Search  < #C, I; Key.   !Search Less
Search  >= #C, I; Key.  !Search Greater or Equal
Search  <= #C, I; Key.  !Search Less than or Equal
Search  < #C,1;         !Position to last key of Index 1
Search  > #C,1;         !Position to first key of Index 1
```

If the **Search** succeeds, the current record position is set accordingly and the index used becomes the current index. Relative record access forward or backward is then performed using this index.

When used in conjunction with Full-ISAM files, the application would perform an initial **SEARCH** and read the current record. A loop, such as **WHILE** or **DO** can then used to read next or previous through the file.

When **SEARCH** is used with older-style indexed files, structure variables can still be used by defining a structure containing the traditional parameters supplied to a **SEARCH** statement. Only the modes =, >, < are supported for Indexed files.

```
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 = #1,4;K. \ IF K.V2 ... ! etc.
```

Managing Records within a Full-ISAM File

The management 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. Full ISAM files may only select one of the following:

-1 Read next record relative to the index ordering.

-2 Read current record.

-3 Read previous record relative to the index ordering.

item is any numeric expression which, after evaluation is truncated to an integer specifying the item number 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.

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   : Item "Customer Record"
  Member Number$[8]     : ITEM "Number"
  Member Name$[30]      : ITEM "Name"
  Member Addr$[30]      : ITEM "Addr"
  Member Balance. As DRCR     : ITEM "Balance"
  Member 1%,LastOrderNumb#    : ITEM "LastOrDate
End Def

Dim Customer. As Cust

Open #5, "Customers" "As Full-ISAM"
Map Record #5 As Cust
Map #5,1; "ByName"
Search > #5,1;
Read Record #5,-2; Customer.  !Read entire structure
Write Record #5,-2; Customer.
```

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.

FoxPro Full-ISAM Driver

The following parameters outline the capabilities of the FoxPro compatible Full-ISAM database driver supplied with dL4.

An index cannot be added unless the file is empty. Deleting an index is not supported.

```
!    MAXIMUM LENGTH OF FIELD NAME = 10 CHARACTERS
!    MAXIMUM NUMBER OF FIELDS PER RECORD = 128
!    MAXIMUM LENGTH OF A CHARACTER FIELD = 254 CHARACTERS
!    MAXIMUM NUMBER OF DIRECTORIES = 47
```

```
!    NUMBER OF DECIMAL PLACES IN NUMERIC FIELDS IS REQUIRED
!    RECORD NAME ITEM PARAMETER IS IGNORED IF SUPPLIED
!    BINARY FIELDS NOT DEFINABLE IN BASIC
!    KEY PART OPTIONS ALLOWED:  UPPERCASE (FOR STRING FIELDS)
!                               DECIMALS  (FOR NUMERIC FIELDS)
!    DIRECTORY OPTIONS ALLOWED: ASCENDING SEQUENCE (DEFAULT)
!                               DESCENDING SEQUENCE
!                               UNIQUE KEYS REQUIRED (DEFAULT)
!                               DUPLICATES ALLOWED
!    DEFAULT DIRECTORY NAMING CONVENTION:
!   'Keyxxx' WHERE XXX IS THE DIRECTORY NUMBER, IE KEY001 IS DIRECTORY 1
!   'DelRC64782' IS THE NAME OF A SPECIAL DIRECTORY USED TO MANAGE FREE
!    RECORDS
```

Note: When supplying names for FoxPro directories, the actual directory number is based upon the sorted order of named directories. This is true regardless of the order of definition, or the directory number specified during creation. When using named directories, use the Map statement within your applications to logically number a named directory.

Microsoft SQL Server Full-ISAM Driver

The Microsoft SQL Server Full-ISAM driver is available only with dL4 for Windows. It provides a Full-ISAM interface to SQL Server tables, which appear to a dL4 BASIC program as a Full-ISAM file. The dL4 BASIC program makes requests using indices, although the SQL Server may or may not use indices.

The SQL table must have at least one index with unique keys and allow NULL values in all fields that are not used as keys.

The dL4 BASIC program can only do SQL queries, therefore it cannot:

- create nor delete tables.
- open database views.
- create nor delete databases.
- create nor delete indices.
- issue SQL statements.
- do database administrator functions.

The driver cannot be selected with the dL4 Auto Select Mechanism and does not support the record number, record size, or file size channel functions provided with the CHF statement.

Window Class

A "Window" class driver in dL4 attempts to implement an abstract object called a "window", whose capabilities are:

- A superset of those typically found in an ASCII terminal.
- A subset of those typically found in most Graphical User Interfaces (GUI).

The two objects mentioned, an ASCII terminal and a GUI window, have a great many similarities. The purpose of the "Window" class is to define an object which:

- can be reasonably implemented on both serial terminals and graphic displays.

- is largely upward-compatible with the expectations of software written to work with ASCII terminals under IRIS and uniBasic.

This allows most dL4 BASIC code to run unmodified in both terminal and GUI environments.

Dynamic Windows support is provided for UniBasic compatibility, but the dL4 BASIC programmer is encouraged to use the Window driver statements described in the following section of this manual, Controlling Windows from BASIC.

The Underlying Principles of a Window

A window object in dL4 includes three important components:

1 The "Canvas". This is a virtual, rectangular area onto which all drawing is done. For example, the output of a string to a window results in text being virtually "drawn" onto its canvas.

2 The "Display Region". This is a rectangular area, constrained within the boundaries of the canvas, that controls which portion of the canvas is displayed as the contents of the window on the physical device. If the display region is smaller than the canvas, then only that portion of the canvas contents are displayed. Moving such a display region in any direction results in "scrolling" through the canvas.

3 The "Output Region". This is a another rectangular area, constrained within the boundaries of the canvas, that serves as an output "mask". Drawing on the canvas can occur only within the bounds of the current output region. The remaining portions of the canvas are thus protected. Output regions are primarily used for compatibility with uniBasic applications using Dynamic Windows (i.e. the WINDOW statement).

An important property of these three rectangles is that they do NOT consist of a "grid" of character cells. Nothing in the design of a window limits it to textual display only, or to text of a fixed size. The canvas is considered a continuum, and the display and output regions can (conceptually) be resized by arbitrary increments.

In addition, a window has several other settings that affect its behavior:

- Title display on/off. A window always has a title string associated with it. This setting controls whether or not the title is displayed with the window.
- Wrapping mode on/off. Controls whether horizontal movement off the right/left edge of the output region wraps to the next/previous line.
- Scrolling mode on/off. Controls whether certain vertical motion off the bottom of the output region causes the contents of the region to scroll up by a line, or not.
- Hide mode on/off. Controls whether the window is visible on the display device on not.
- Echo mode on/off. Controls whether input characters are also output to the window canvas.
- Input pending mode on/off. Controls the behavior when reading from a window and the destination string is filled. With pending mode off, the input is terminated; with pending mode on, a terminator character must still be entered.

There are a large assortment of other mode states which a window driver may or may not support, most of which affect text output (underline mode, blink mode, etc.).

Differing Implementations

The fact that ASCII terminals naturally impose many restrictions on the abilities of a window is simply an issue of implementation on ASCII terminals, reflecting their own capabilities. The underlying model of a window still applies, even on terminals with very limited capabilities.

The matter is comparable to the situation where some terminals can display color and others can't. A program which requires certain capabilities may be limited to using devices which have such capabilities. For any given request to execute some feature, a window driver may decide to approximate the request, ignore the request, or refuse the request, depending on the actual capabilities of the implementation.

So, although the window model defines e.g. how a program can draw text in differing fonts and sizes or even simple graphics, it does not imply that all programs using such features will operate identically (or at all) on all possible window devices. As in the past, the application developer must consider his target system(s) and/or peripherals when deciding what features to use.

Types of Window Drivers

The Window class drivers available are:

- Default Window driver supported on UNIX and Windows platforms. This is the high level driver that dL4 BASIC programmers should use in all but special cases. It is the driver that is used when the **AS** clause specifies the Driver Class, i.e. **As "Window"**. This driver calls the following lower level drivers as necessary.
- Phantom Window driver supported on UNIX and Windows platforms. This driver is used in processes that are not associated with a terminal and keyboard for input and output.
- Terminal Window driver supported on UNIX and Windows platforms. This driver is used in processes that use a terminal and keyboard for input and output.
- Default Terminal Translation driver supported on UNIX and Windows platforms. This driver selects the driver (such a Generic Terminal Translation) used to support the physical terminal.
- Generic Terminal Translation driver supported on UNIX and Windows platforms. This low level driver controls the platform specific terminal.
- Win32 Window Terminal Translation driver supported only on the Windows platform. This low level driver controls the platform specific terminal.

Controlling Windows From BASIC

A driver related statement may specify parameters and optional items as indicated by italics in the following general form:

`STATEMENT #chan,`*`parameters`*`;`*`items`*`,variables`

Windows are controlled from BASIC through the following statements:

Statement	Parameters	Items	Summary
OPEN	Title$	none	String to use as the window title
	Style$		Keyword toggles - TITL,WRAP,SCRL,HIDE
	Width		Initial width in columns as per the font
	Heigth		Initial heigth in rows as per the font
	Parent		Makes child of the window on channel #parent
	StartX		Relative starting position within parent window
	StartY		Relative starting position within parent window
CLOSE	none	none	Removes window from display and close channel
CLEAR	none	none	Removes window from display and close channel
READ	param3	none	Provides timeout for input of string data
	param1		set to -1 as placeholder if using param3 (timeout)
	param2		set to -1 as placeholder if using param3 (timeout)
WRITE	none	none	Display begins at current position
ERASE	none	none	Clears output region

SIZE	param1		0 - display region; 1 - canvas; 2 - output region
		width,heigth	Size in current coordinate system
MOVE	param1		0 - child window; 1 - display region; 2 - output region
		width,heigth	New upper left corner position
CHANNEL	param1	none	11 - Display window 12 - Hide window
	param1		14 - Horizontal scroll 15 - Vertical scroll
		hamount	Positive - right; Negative - left
		vamount	Positive - up; Negative - down

These statements are described in more detail on the following pages.

OPEN

Synopsis

Creates a window according to the given parameters accessible on channel #C

Syntax

Open #*C*,{*title*$,*style*$,*width*,*height*,*parent*,*startX*,*startY*} **As** "Window"

Parameters

title$ is a str.expr that becomes the title of the window.

style$ is a str.expr that can contain keywords controlling the initial settings for the window.

width is a num.expr that is the window's initial width in columns.

height is a num.expr that is the window's initial height in rows.

parent is a num.expr that is the channel number of another open window.

startX and *startY* are num.exprs that are a child window's relative starting position within the parent window.

Remarks

title$ strings are usually restricted to those characters which are "listable" in the window itself. That is, if a window lists a character in \ooo\ notation, it probably won't be allowed in the window title.

The *style$* keywords are each four letters, not case-sensitive, and are separated by commas.

- TITL Set title display on. Default is off.
- WRAP Set wrapping mode on. Default is off.
- SCRL Set scrolling mode on. Default is off.
- HIDE Hide the window. Default is to show the window.

The *width* is in columns where, in a window, a column is defined as the average width of a text character in the window's current font.

height is the window's initial height in rows. In a window, a row is defined as the height of a text character in the window's current font.

parent is the channel number of another open window. The new window will be a "child" window, whose display is constrained within the boundaries of the parent, moves with the parent, etc. If not supplied, the window is an "independent" window, whose initial position is chosen by the window driver.

startX and *startY* are a child window's relative starting position within the parent window. The position coordinates are interpreted according to the parent window's current coordinate system.

CLOSE/CLEAR

Synopsis

Window object on channel #C is removed from display.

Syntax

Close *#C*

Clear *#C*

Parameters

None.

Remarks

For a window, **CLOSE** and **CLEAR** function identically. The window object is removed from display and the channel is closed. Other window areas obscured by the closed window are redrawn.

If the window contained child windows, those channels are closed first before closing the parent window.

READ

Synopsis

Input a character string from the window on channel #C at the current position.

Syntax

Read #*C*;*str.var*

Read #*C*{,-1,-1,*param3*};*str.var*

Parameters

str.var is a variable of string data type which returcs the input character string.

param3 is a numeric expression that specifies the timeout.

The -1 values preceding *param3* are place holders.

Remarks

Input a character string from the window at the current position. Input from a window makes it the "top" window and, if needed, adjusts the display region to make the cursor visible during input. If echo mode is on, the entered characters are displayed on the window. Input terminates when a termination character is received and placed in <string.var> or when the destination string is filled (if Input Pending mode is off).

The **INPUT** statement in dL4 is always executed in terms of **READ**, so all variations of **INPUT** (**INPUT LEN**, **INPUT TIM**, etc.) are supported for a window.

WRITE

Synopsis

Display a character string on the output region of the window on channel #C beginning at the current position.

Syntax

Write *#C;str.var*

Parameters

str.var is a variable of string data type which returcs the input character string.

Remarks

Display a character string on the output region of the window beginning at the current position. The cursor position is advanced beyond the last character output. Special output characters, such as non-textual, are supported as described in the section Special Output Characters.

The **PRINT** statement in dL4 is always executed in terms of **WRITE**, so all variations of **PRINT** (**PRINT USING**, etc.) are supported for a window.

ERASE

Synopsis

Erases the output region of the window on channel #C.

Syntax

Erase *#C*

Parameters

None.

Remarks

ERASE clears the output region of the window.

SIZE

Synopsis

Changes the size of the various window components on channel #C.

Syntax

Size *#C*,*param1*;*width*,*height*

Parameters

param1 is a numeric expression.

width is a numeric expression.

heigth is a numeric expression.

Remarks

The record number parameter (*parm1*) controls which component is being resized:

0 Display region (default if not supplied).

1 Canvas.

2 Output region.

The *width* and *height* arguments are interpreted according to the current coordinate system of the window. The default is text columns and rows.

MOVE

Synopsis

Moves the various components of a window on channel #C.

Syntax

Size *#C* {,*param1*}; *@width,height*

Parameters

param1 is a numeric expression.

width is a numeric expression.

heigth is a numeric expression.

Remarks

The record number parameter (*param1*) controls which component is being moved:

0 Window, relative to it's parent window (default if not supplied). Only a child window can be moved in this fashion.

1 Display region, relative to the canvas.

2 Output region, relative to the canvas.

The coordinate expression (*@width,heigth*) gives the new position of the upper-left corner of the component.

If 0 is selected, the coordinate expression is interpreted according to the current coordinate system of the parent window. If 1 or 2 are selected, the coordinate expression is interpreted according to the current coordinate system of the affected window.

CHANNEL:SHOW/HIDE

Synopsis

Show or hide a window on channel #C.

Syntax

Channel *param1*, *#C*

Parameters

param1 is a numeric expression.

Remarks

There is no dedicated statement in dL4 BASIC for the ShowWindow or HideWindow operations on a channel, therefore the general-purpose **CHANNEL** statement must be used. *param1* is a num.expr, the value of which determines the operation:

param1	**Operation**
11	Show the Window
12	Hide the Window

A window which is hidden can still be otherwise modified and accept output while remaining hidden. A read from a hidden window causes an implicit ShowWindow to occur.

CHANNEL:HSCROLL/VSCROLL

Synopsis

Scrolls the display region within the canvas of the window on channel #C.

Syntax

Channel *param1*, *#C*; *hamount*

Parameters

param1 is a numeric expression.

hamount is a numeric expression.

vamount is a numeric expression.

Remarks

There is no dedicated statement in dL4 BASIC for the HScroll or VScroll operations on a channel, therefore the general-purpose **CHANNEL** statement must be used. Scrolling is the equivalent of the **MOVE** command in mode 1: moving the display region relative to the canvas by the indicated amount. *param1* is a num.expr, the value of which determines the operation:

param1	**Operation**
14	Horizontal scroll the Window
15	Vertical scroll the Window

Positive values for *hamount* move to the right, and negative values move to the left.

Positive values for *vamount* move down, and negative values move up.

The movement amount is interpreted in units that correspond to the current coordinate system of the window.

Special Output Characters Defined for Windows

The set of characters which can be output to a window is defined as all characters which are either:

1 A "listable" character for that window, or

2 One of the supported "special" characters documented herein.

In addition, some window drivers may support only a subset of the special characters. For each special character, a window driver is obligated to either support the operation as defined here, or fail on output.

Special Output Characters Controlling I/O modes

'IOBE'	Begin input echo mode.
'IOEE'	End input echo mode.
'IOTE'	Toggle input echo mode.
'IOBD'	Enable destructive backspace mode.
'IOED'	Disable destructive backspace mode.
'IOBC'	Enable activate-on-control-character mode.
'IOEC'	Disable activate-on-control-character mode.
'IOB\'	Enable echo "\" on escape mode.
'IOE\'	Disable echo "\" on escape mode.
'IOBI'	Enable binary input mode.
'IOEI'	Disable binary input mode.
'IOCI'	Clear the input type-ahead buffer.
'IORS'	Reset all I/O modes to default state.

Special Output Characters Controlling the Cursor

Note: In all cases, the cursor cannot be positioned within a protected field.

'ML'	Move cursor left 1 column. If movement exceeds the left edge of the output region, the result depends on the status of wrapping mode. If wrapping mode is on, the cursor is moved to the last column of the previous row. If already at the top row, the bottom row is considered the previous row. If wrapping mode is off, no motion occurs.
'MR'	Move cursor right 1 column. If movement exceeds the right edge of the output region, the result depends on the status of wrapping mode. If wrapping mode is on, the cursor is moved to the first column of the next row. If already at the bottom row, the top row is considered the next row. If wrapping mode is off, no motion occurs.
'MU'	Move cursor up 1 row. If movement exceeds the top edge of the output region, the result depends on the status of wrapping mode. If wrapping mode is on, the cursor is moved to the same column of the bottom row. If wrapping mode is off, no motion occurs.
'MD'	Move cursor down 1 row. If movement exceeds the bottom edge of the output region, the result depends on the status of wrapping mode. If wrapping mode is on, the cursor is moved to the same column of the top row. If wrapping mode is off, no motion occurs.
'MH'	Move cursor to first column of first row.

'xMOVETO'	Move cursor to position x of current row. Also accessible in BASIC as "Tab(x)". The result of attempting to position outside the current output region is undefined.
'x,yMOVETO'	Move cursor to position x,y. Also accessible in BASIC as "@x,y;". The result of attempting to position outside the current output region is undefined.
'CR'	Move cursor to first column of next row. If movement exceeds the bottom edge of the output region, the result depends on the status of scrolling mode, format mode, and wrapping mode. If scrolling mode is on and format mode is off the output region is scrolled up by one row, else if wrapping mode is on the cursor is moved to the first column of the first row, else no motion occurs.
'LF'	Move cursor down to next row. If movement exceeds the bottom edge of the output region, the result depends on the status of scrolling mode, format mode, and wrapping mode. If scrolling mode is on and format mode is off the output region is scrolled up by one row, else if wrapping mode is on the cursor is moved to the same column of the first row, else no motion occurs.
'BK'	Move cursor to first column of current row.
'TF'	Move cursor right to next tab stop. If movement exceeds the right edge of the output region, the result depends on the status of wrapping mode. If wrapping mode is on, the cursor is moved to the first tab stop of the next row. If wrapping mode is off, no motion occurs.
'TB'	Move cursor left to previous tab stop. If movement exceeds the left edge of the output region, the result depends on the status of wrapping mode. If wrapping mode is on, the cursor is moved to the last tab stop of the previous row. If wrapping mode is off, no motion occurs.
'xALIGN'	Move cursor right to next horizontal boundary of x columns.
'K0'	Set cursor display off.
'K1'	Set cursor display to a blinking block.
'K2'	Set cursor display to a steady block.
'K3'	Set cursor display to a blinking underline.
'K4'	Set cursor display to a steady underline.
'BS'	Backspace. Equivalent to 'ML'.
'RI'	Reverse index. Equivalent to 'MU'.
'NEL'	Next line. Equivalent to 'CR'.
'LINESEP'	Line separator. Equivalent to 'CR'.
'PARASEP'	Paragraph separator. Equivalent to 'CR LF'.
'IND'	Index. Equivalent to 'LF'.
'HT'	Horizontal tab. Equivalent to 'TF'.

Special Output Characters Controlling Text Drawing

The description of each of the text drawing modes is as notable for what is said as for what is **not** said. For example, the description of 'BB' merely states that it begins blink mode. It does not specify what effect the setting of blink mode may have on any of the other text drawing modes. It is the nature of certain ASCII terminals that only one mode may be in effect at a time. A (1) in the table denotes a mnemonic that is not yet supported.

'BB'	Begin blink mode.
'EB'	End blink mode.

'BBOLD'	Begin bold mode.
'EBOLD'	End bold mode.
'BC'	Begin compressed mode. (1)
'EC'	End compressed mode. (1)
'BX'	Begin expanded mode. (1)
'EX'	End expanded mode. (1)
'BD'	Begin dimmed intensity mode.
'ED'	End dimmed intensity mode.
'BG'	Begin graphics mode. Graphics mode has no defined effect.
'EG'	End graphics mode. Graphics mode has no defined effect.
'BI'	Begin italic mode.
'EI'	End italic mode.
'BR'	Begin reverse video mode.
'ER'	End reverse video mode.
'BSO'	Begin strike-out mode.
'ESO'	End strike-out mode.
'BSUB'	Begin subscript mode. (1)
'ESUB'	End subscript mode. (1)
'BSUP'	Begin superscript mode. (1)
'ESUP'	End superscript mode. (1)
'BU'	Begin underline mode.
'EU'	End underline mode.
'BLACK'	Set foreground color to black.
'BL'	Set foreground color to blue.
'CY'	Set foreground color to cyan.
'GN'	Set foreground color to green.
'MA'	Set foreground color to magenta.
'RE'	Set foreground color to red.
'WH'	Set foreground color to white.
'YE'	Set foreground color to yellow.
'RESETCOLOR'	Restore default colors.

Special Output Characters Controlling Canvas Editing

'BP'	Begin protected field drawing. All areas of the canvas subsequently drawn with text or graphics are eligible to be "protected fields". Such areas actually become protected fields by entering format mode ('FM').
'EP'	End protected field drawing.
'FM'	Enter format mode. Format mode has the effect of inhibiting scrolling mode and causes all eligible areas to become protected fields. See also, 'DL' and 'IL' characters.

'FX'	Exit format mode.
'CE'	Clear all unprotected fields from cursor to end of output region.
'CL'	Clear all unprotected fields from cursor to end of row in the output region.
'CU'	Clear all unprotected fields in the output region.
'CS'	Clear entire output region. This also causes format mode to be disabled and all protected fields to revert to normal fields.
'DC'	Delete 1 character space beneath the cursor and insert empty space at the last column of the row or field (as delimited by a protected character). The intervening contents are shifted left by one column.
'IC'	Insert 1 space character beneath the cursor and discard 1 character space at the last column of the row or field (as delimited by a protected character). The intervening contents are shifted right by one column.
'DL'	Delete 1 row beneath the cursor and insert an empty row at the last row of the output region. The intervening contents are shifted up. If format mode is on, 'DL' is ignored.
'IL'	Insert 1 empty row beneath the cursor and discard the last row of the output region. The intervening contents are shifted down by one row. If format mode is on, 'IL' is ignored.

Special Output Characters for Graphic Drawing

'x,yLINETO'	Draw a line from the cursor to position x,y. The cursor is then positioned at x,y.
'x,yRECTTO'	Draw a rectangle from the cursor to position x,y. The cursor is then positioned at x,y..
'a,b,c,dRECT'	Draw a rectangle from position a,b to position c,d. The cursor is then positioned at c,d.

Miscellaneous Special Output Characters

'BEL'	Ring bell.
'RB'	Ring bell.
'XX'	Initialize hardware. The effect of this character is completely implementation-defined.

Form and Chart Drawing Characters

Note that the form and chart drawing characters 'G1', 'G2', etc. are absent from the above list of special characters. This is because in dL4 these mnemonics each have been assigned to the Unicode character value that directly corresponds to the equivalent character glyph. For example,

'G1' = 'U+250C' = "\22414\" = FORMS LIGHT DOWN AND RIGHT

in other words, the upper-left corner character in Unicode.

Whether or not this character is supported falls under rule 1 within Special Output Characters Defined for Windows above, i.e. it fully depends on whether or not it is present in the character set of the underlying device. A window class driver handles such characters no differently than it handles alphabetic text.

As a consequence of this fact, the "graphics mode" normally required to output such characters is superfluous in a dL4 window, therefore 'BG' and 'EG' are typically ignored. If the device requires a change of mode in order to display the character, the driver itself will manage it.

Special Output Characters Which Support Repeat Counts

Of the special output characters documented above, those listed below interpret any single parameter passed to them as a repeat count. For example:

```
Print 'MR'  ! Move right once
```

```
Print '5MR' ! Move right 5 times
```

The characters in this category are the window's entire set of listable characters plus the following:

'ML'	Move cursor left 1 column.
'MR'	Move cursor right 1 column.
'MU'	Move cursor up 1 row.
'MD'	Move cursor down 1 row.
'CR'	Move cursor to first column of next row.
'LF'	Move cursor down to next row.
'TF'	Move cursor right to next tab stop.
'TB'	Move cursor left to previous tab stop.
'BS'	Backspace.
'RI'	Reverse index.
'NEL'	Next line.
'LINESEP'	Line separator.
'PARASEP'	Paragraph separator.
'IND'	Index.
'HT'	Horizontal tab.
'DC'	Delete 1 character.
'IC'	Insert 1 character.
'DL'	Delete 1 row.
'IL'	Insert 1 row.
'BEL'	Ring bell.
'RB'	Ring bell.

Cursor Tracking Mode

dL4 does not support Cursor Tracking Mode, but the keyboard input translation can be configured to pass the cursor keys to the application as data with the value of the movement mnemonics ('ML', 'MR'). If this is done, the program will always be in Cursor Tracking Mode

Using Dynamic Windows

Each window behaves as a full screen of the dimensions specified. Data automatically wraps within the boundaries of the window and many of the mnemonics are supported. Cursor positioning is relative, such that position 0,0 is the first character of the window. Scrolling within a window is allowed.

Window *Zero* is the full screen before any windows are open

Raw Class

The Raw class may be used for low level access to files and devices. Internally, dL4 represents all data as 16-bit Unicode. When presenting data to the outside world, the rawfile drivers convert the Unicode as 8-bit binary data in the least significant bits (LSB) and the most significant bits (MSB) are all set to zero.

The raw drivers treat the data as binary and no conversions are performed, therefore character sets are not supported.

Types of Raw Drivers

The Raw class drivers available are:

- "Raw File" driver supported on the UNIX and Windows platforms.
- "Raw Regular File" supported on the UNIX and Windows platforms.

The two drivers differ in that "Raw File" (the default) will open both files and devices. The "Raw Regular File" driver will only open files.

Creating Files

The **BUILD** statement is used to create a file with the **Raw** class drivers. The *file.spec.str* items: option item, (charset=*character-set*), and cost item, $*cost*, are accepted but ignored. A protection item, <*A*>, is accepted and applied. A record length must be specified in the record item, [*num-of-records*:*record-len*], of the *file.spec.str*. The number-of-records expression of the record item is ignored. An exclamation point, (!), may be used at the end of the filename item to replace filename if it already exists. For example:

```
Build #channel,"[10:40]rawfile!" As "Raw Regular File"
```

Opening and Closing Files and Devices

The **OPEN** statement is used to Access a file with the **Raw** class drivers. The *file.spec.str* items: option item, (charset=*character-set*), and cost item, $*cost*, are accepted but ignored. A protection item, <*A*>, is accepted and applied. A record length must be specified in the record item, [*num-of-records*:*record-len*], of the *file.spec.str*. The number-of-records expression of the record item is ignored.

dL4 does not have the UniBasic statements **RDREL** and **WRREL** which read-from or write-to files in blocks of 512 bytes. This functionality may be achieved with the **READ** and **WRITE** statements if the file is opened with a record length of 512 specified. For example:

```
Open #channel,"[10:512]rawfile" As "Raw Regular File"
```

The **EOPEN** statement is supported in the WIN32 driver, but not in the UNIX driver.

The **ROPEN** and **WOPEN** statements are supported and used to open files in read-only and write-only modes respectively.

The **CLOSE** statement is used to close the file associated with the channel. If the file was newly created , make it permanent. If no channel is given, all open channels are closed.

The **CLEAR** statement is used to close the file associated with the channel. If the file was newly created , it is deleted. If no channel is given, all open channels are closed.

Positioning Within Files

A *chn.expr* for **Raw** drivers supports the specification of all three optional parameters, in the general form:

```
#channel { , record {, byte-displacement {,  timeout }}} ;
```

The optional *record* specifies a block of bytes, the number of which is specified with *record-length* in the **OPEN** statement, within a file. The record must be zero or greater. A *record* cannot be specified for a device.

An optional *byte-displacement* specifies the starting byte position within a block of data. It must be zero or greater. A *byte-displacement* cannot be specified for a device.

Positioning within a file is performed by taking the result of (*record * record-length) + byte-displacement* and using that to specify a specific byte within the file. For example, the values *record=1, record-length =20, byte-displacement=4* selects the (1*20+4) or the 25^{th} byte within the file.

The optional *timeout* is typically used when accessing devices, such as keyboards or com ports, to provide for a period of time to wait for data.

Record Locking is not provided automatically by Raw drivers and a *timeout* may be included for those cases where an application, other than dL4 has placed locks on a file or a device.

Record Locking with the Rawfile Driver

The **Raw** class drivers do not provide record locking on a **READ** or **WRITE** statement, but records can be locked using dL4 Channel statements.

```
Channel 21, #C,r,o,t;v  !Set shared lock for size of variable v
Channel 22, #C,r,o,t;v  !Set exclusive lock for size of variable v
Channel 23, #C,r,o,t;v  !Remove lock for size of variable v
```

Read and Writing with the Rawfile Driver

Use the **INPUT** or **READ** statements to receive data from the channel. There is no line termination conversion performed The *record, byte-displacement,* and *timeout* entries are accepted if zero or greater.

Use the **PRINT** or **WRITE** statements to output data to the channel. The *record, byte-displacement,* and *timeout* entries are accepted if zero or greater.

Channel Functions and Operations

String Functions

CHF$(x)	Operation Performed
1	Return Open Modes Selected. ("RWLE")
6	Driver Class
7	Driver title
8	Filename, portable if possible, based upon current working directory. If you change cwdir, this value changes

Numeric Functions

CHF(x)	Operation Performed
0	Current file size in records of 512-bytes
1	Current record within the file
2	Current byte position within the file
3	Record length in words (See File Unit Option)
4	File size in bytes
5	Record Length in bytes always returns 512
6	Header size always returns 0
7	error
8	error
9	File owner in numeric form - error if the information cannot be expressed as a numeric value
10	File group in numeric form- error if the information cannot be expressed as a numeric value
11	File protection or permissions, expressed in numeric form- error if the information cannot be expressed as a numeric value

Directory Class

The Directory class driver provides access to the filenames stored within a directory of the file system of the platform.

Types of Directory Drivers

The Directory class drivers available are:

- POSIX Directory driver supported on the UNIX platform.
- Directory driver supported on the Windows platform.

Accessing Directories

The Directory driver is used to create a new directory or open an existing directory. For example:

```
Build #1,"c:\\dl4train" As "directory"
Open #1,"c:\\dl4train" As "directory"
```

Reading and Writing with Directory Drivers

Writing to a directory is not allowed.

Sequential reading from the directory is performed and only filenames are returned. Random access is not available, but the record number parameter can be set to zero to rewind to the beginning. For example:

```
! Read the filenames in a directory
Dim FileName$[80]
Dim DirName$[48]
Input "Enter the directory name: ",DirName$
Print
Try
  Open #1,DirName$ As "directory"
Else
  Print "Could not open the ";DirName$;" directory. Error ";Spc(8)
  End
End Try

Read #1,0;FileName$           ! rewind to beginning

Do
  Read #1;FileName$
  Print FileName$
Loop Until FileName$ = ""
Close #1
End
```

Use the returned filename to open the file and the CHF or other functions to get specific file information.

```
! Read the filenames in a directory
Dim FileName$[80]
Dim DirName$[48]
Input "Enter the directory name: ",DirName$
Print
Try
  Open #1,DirName$ As "directory"
Else
  Print "Could not open the ";DirName$;" directory. Error ";Spc(8)
  End
End Try

Read #1,0;FileName$           ! rewind to beginning

Do
  Read #1;FileName$
  Open #2,DirName$ + "\\" + Filename$
  Print FileName$
  Print "Size: ";Chf(2)
  Print "Rec-Len: ";Chf(502)
  Print "Driver Class: ";Chf$(602)
  Print "Driver Title: ";Chf$(702)
  Print "File Name: ";Chf$(802)
  Close #2
Loop Until FileName$ = ""
Close #1
End
```

Driver List Class

The Driver List driver is used by SCOPE to provide a list when the **DRIVERS** command is issued.

Types of Driver List Drivers

The Intrinsic Driver List is the driver in this class. It is used by SCOPE, when the **DRIVERS** command is issued from the SCOPE Interactive Development Environment, to display a list of the driver classes and the

individual drivers available from those classes. This is the dL4 BASIC programmer's only interface to the Intrinsic Driver List driver.

System Class

The System driver returns and manages operating system specific information for dL4.

Types of System Drivers

The individual driver in the System Class is determined by the operating system. Under the UNIX OS it is POSIX System. For WIN32 it is WIN32 System. The dL4 BASIC programmer should never access this driver.

Port Communication Class

The Port Communication driver implements and manages interprocess communications between dL4 and applications.

Types of Port Communication Drivers

The individual driver in the Port Communication Class is determined by the operating system. Under the UNIX OS it is System V Message Queue Communication. For WIN32 it is WIN WM_COPYDATA Communication. The dL4 BASIC programmer should not access this driver.

Program Class

The BASIC Program driver is used to load and link Basic programs. The dL4 BASIC programmer should not access this driver.

Appendix A - Whether A Statement Is Used With A Driver-Class

The table on the following pages shows whether a given statement is used with the driver-classes. "Y" indicates that the statement is used with the indicated driver-class, while "n" indicates it is not.

For example, the **DEFINE RECORD** statement is used only with the Full-ISAM driver-class, and not with any of the others.

The driver-class names are abbreviated as follows:

ABBREVIATION	DRIVER-CLASS
As	Auto Select
Co	Contiguous
Di	Directory
Dl	Driver List
Fo	Formatted
FI	Format Index
In	Indexed
IC	Indexed Contiguous
PC	Port Communication
Pr	Profile
Pg	Program
Ra	Raw
Sy	System
Tx	Text
W	Windows

STATEMENT	As	Co	Di	Dl	Fo	FI	In	IC	PC	Pr	Pg	Ra	Sy	Tx	W
ADD	n	n	n	n	n	Y	n	n	n	n	n	n	n	n	n
ADD INDEX	n	n	n	n	n	Y	n	n	n	n	n	n	n	n	n
ADD RECORD	n	n	n	n	n	Y	n	n	n	n	n	n	n	n	n
BOX	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n
BUILD	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
CHANNEL															
CLEAR															
CLOSE															
DEFINE RECORD						Y									
DELETE INDEX						Y									
DELETE RECORD						Y									
DUPLICATE															
GET															
INPUT															
KILL															
MAP						Y									
MAP RECORD						Y									
MAT INPUT					Y										
MAT PRINT					Y										
MAT RDLOCK					Y										
MAT READ					Y										
MAT WRITE					Y										
MAT WRLOCK					Y										
MODIFY															
OPEN															
PRINT	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n
RDLOCK															
READ		Y			Y	Y						Y			
READ RECORD						Y									
RECV															
REWIND															
ROPEN															
SEARCH						Y									
SEND															
SET															
SETFP															
SIGNAL															
SIZE															
TRACE															
UNLOCK															
WINDOW															
WOPEN															
WRITE		Y			Y							Y			
WRITE RECORD						Y									
WRLOCK															

Index