

DCL Characteristics

This chapter provides an explanation of the language characteristics of **DCL**.

General Characteristics

- **DCL** is compiled, not interpreted.
- A **DCL** script consists of any number of user-defined functions, user-defined subroutines, and external DLL declarations.
- The compiler and runtime operate under Windows 3.1 and Windows for Workgroups 3.11 or later.
- A **DCL** script is an ASCII stream of text with a NULL terminator.
- **DCL** scripts are completely self contained. Variables have a scope local to the function or subroutine in which they are declared. Further, any referenced functions or subroutines must occur within the same script.
- Line numbers are not supported. Labels are used instead.
- Quotes can be embedded within constant string expressions using two consecutive quotes, such as "John said, ""hello"", to Jane".
- Numeric constant folding is supported. For example, the statement `i=10+23` is reduced before compiling to `i=33`.
- Constant expressions involving strings are reduced at compile time, including calls to the function `CHR$()` where the passed parameter is a constant. For example, the statement `s$ = "Hello" + chr$(13) + chr$(10) + "world"` is reduced to the expression `s$ = "Hello\r\nworld"`.
- The runtime is reentrant. Thus, two DCL scripts can execute concurrently.

Script Execution

Execution of a **DCL** script begins with a subroutine with the predefined name `Main`.

Structures

User-defined structures are not supported. **DCL** does support dialog structures.

Compiler

- All built-in statements, functions, and constants are implicitly declared and compiled in a resource file. Thus, there are no include files required for compiling.
 - Compiler metacommands are not supported.
 - The compiler is reentrant. Thus, two DCL scripts can be compiled simultaneously.
-

Variables

- **DCL** supports integers, longs, shorts, strings, dialogs, single precision floats, and double precision floats. See “Data Types” in this chapter for more information.
- Variables declared (explicitly or implicitly) within a subroutine are local to that subroutine.
- Variables that are not explicitly given a type (with the `dim` statement or using a type specifier) are given the type `integer`.
- All declared variables are assigned an initial value of 0. For strings, 0 equates to "" (or NULL).
- Internal type conversions are performed automatically between any two numeric quantities. Thus, the script author can freely assign numeric quantities without regard to type conversions. However, it is possible for an overflow error to occur when converting from larger to smaller types. This occurs when the larger type contains a numeric quantity that cannot be represented by the smaller type.

- For example, the following code will produce a runtime error:

```
dim amount as long
dim quantity as integer
amount = 400123 'assign a value out of range for int
quantity = amount 'attempt to assign to integer
```

Like many runtime errors, the overflow error is trappable.

- Loss of precision is not a runtime error.
- The declaration modifiers `STATIC`, `GLOBAL`, and `SHARED` are not supported.

Expression Evaluation

DCL allows expressions to involve data of different types. When this occurs, the two arguments are converted to be of the same type by promoting the “smaller” of the two data types. For example, **DCL** will promote the value of `i%` to a `double` in the following expression:

```
result# = i% * d#
```

When evaluating an expression, **DCL** also looks at the resultant type. During evaluation, the data type of each operand of each subexpression is compared with the data type of the result. Each operand will then be promoted to be the same as the “largest” of the two operands and the result. For example, the following expression results in the value `2.5`, even though the division appears to involve two integer values:

```
d# = 10 / 4 'd# becomes 2.5
```

The expression evaluator realizes that the result of the expression is being assigned to a `double`, and promotes the two integer operands of the division to be of the same type as the result. After the promotion, the division is performed. If this promotion did not occur, then the above expression would result in `2`, which would be incorrect.

Subroutines & Functions

- Subroutines and functions are available to any other subroutine or function within the same script.

- Arguments to functions and subroutines are passed automatically by reference. Passing by value is accomplished with the BYVAL keyword:

```
sub foo(byval a as integer)
:
end sub
sub main
d% = 10
foo d      'pass d by value
end sub
```

If a subroutine declares parameters by reference, then the caller can force a parameter to be passed by value by enclosing the parameter with parentheses:

```
sub foo(a as integer)
:
end sub
sub main
d% = 10
foo(d) 'force 'd' to be passed by value
end sub
```

- Forward references are not allowed. The body of a referenced subroutine or function must appear before its reference in the script.
- Recursive statements and functions are supported.

Error Handling

- **DCL** supports error handling in a manner that conforms to the Visual Basic error handling model. Individual error numbers may be different.
- `ON ERROR` traps are valid only during the execution of the current subroutine or function.
 - Error traps are saved and restored within a user-defined subroutine or function.

DCL errors fall into three categories:

- Trappable error numbers are between 10 and 1000.
- Non-trappable error numbers are between 0 and 10. Internal errors and out of memory errors are all non-trappable.
- Application specific error numbers are greater than or equal to 1000.

Error numbers will change in the next release of **DCL**.

DCL uses the Windows floating point emulator WIN87EM.DLL. Floating point exception errors are handled in a standard way using the SDK `__fpmath()` function. When the host application calls the compiler or the runtime, the floating point exception handler of the host application is saved and restored on exit.

Arrays

- Dynamic allocation of arrays is supported using both the `DIM` and the `REDIM` statements.
- Arrays can be declared to contain any fundamental data type.
- Arrays can have up to 60 dimensions.
- Array dimensions and size can be changed dynamically using the `REDIM` statement.
- Arrays are always passed by reference.

Data Types

Integer	Type Declaration Significant Digits Size Range	% 4 2 bytes (16 bits) $-32768 \leq X \leq 32767$
Long	Type Declaration Significant Digits Size Range	& 9 4 bytes (32 bits) $-2147483648 \leq X \leq 2147483647$
String	Type Declaration Significant Digits Size Range Note	\$ N/A 1 byte per character $0 \leq \text{LENGTH} < 32768$ All strings are variable length. Fixed length strings are not yet supported.

Single	Type Declaration Significant Digits Size Range	! 7 4 bytes (32 bits) Negative Values: -3.402823E38 to -1.401298E-45 Positive Values: 1.401298E-45 to 3.402823E38
Double	Type Declaration Significant Digits Size Range	# 15-16 8 bytes (64 bits) Negative Values: 1.797693134862315E308 to -4.94066E-324 Positive Values: 4.94066E-324 to 1.797693134862315E308

Operator Precedence

Operators in order of precedence (highest to lowest)	Description
()	parentheses
^	exponentiation
-	unary minus
/, *	multiplication and division
\	integer division
mod	modulo
+, -	addition and subtraction
=, <, >, <=, >=	relational
not	logical negation
and	and
or	or
xor	exclusive or

DCL Comments

Comments can be added to DCL code in the following manner:

- All text between a single quote and the end of the line is ignored:

```
MsgBox "hello" 'display a message box
```

- The REM statement causes the compiler to ignore the entire line:

```
REM This is a comment...
```

- **DCL** supports C-style multi-line comment blocks `/* . . . */`, as shown in the following example:

```
MsgBox "Before Comment"  
/* This stuff is all commented out.  
This line too will be ignored  
This is the last line of the comment */  
MsgBox "After Comment"
```

The C-style comments cannot be nested.

Constants

- Numerous symbolic constants that you can use with specific **DCL** commands are defined in the files ADSCON11.EBL and DCL.EBL. The constants defined for a particular **DCL** command are listed in Chapter 8, “Command Reference.” Two constants are defined by **DCL** itself—TRUE and FALSE.

- The following constants are predefined for use with **DCL**:

ATTR_ARCHIVE	NS_LOGGEDON
ATTR_DIRECTORY	PO_LANDSCAPE
ATTR_HIDDEN	PO_PORTRAIT
ATTR_NONE	TRUE
ATTR_NORMAL	TYPE_DOS
ATTR_READONLY	TYPE_WINDOWS
ATTR_SYSTEM	VK_LBUTTON
ATTR_VOLUME	VK_RBUTTON
ENV_BOTH	VS_VERSION_INFO
ENV_DOS	WS_MAXIMIZED
ENV_WINDOWS	WS_MINIMIZED
FALSE	WS_RESTORED
NS_ACTIVE	

DCL Limitations

- Each running script is limited to 64K of data. The data segment contains dynamic variables (such as strings and arrays), constants, an event jump table, and external DLL call information.
- Strings are limited to 32K in size (32767 bytes).
- Script source size is limited to 42K.
- Compiled code size is limited to 64K. The code segment contains the executable pcode.
- The maximum size of the symbol table (used by runtime errors and the debugger) is 64K.
- Arrays can have up to 60 dimensions.
- Variable names are limited to 40 characters.
- Labels are limited to 40 characters.
- A given subroutine or function can have up to 100 variables, including variables that are passed.
- The stack size for the runtime is 2048 bytes.
- The number of open DDE channels is unlimited (limited only by available memory and system resources).
- The number of open files is limited to 255, or the operating system limit, whichever is less.

- The number of characters within a string literal (a string enclosed within quotes) is 255 characters. (Strings can be concatenated using the plus (+) operator with the normal string limit of 32767 characters.)
- Number of nesting levels is limited by compiler memory.
- Text file input/output buffer size is 512 bytes.
- Queue playback buffer size is limited to 64K. With 10 bytes per event, this allows for 6553 events. This memory is buffered in blocks of 100 events (1000 bytes).
- Each `gosub` requires 2 bytes of the DCL runtime stack.

