

A Quick DCL Course

This chapter will introduce you to the elements that make up **DCL**'s implementation of Basic. If you have had Basic programming experience, we recommend that you peruse this chapter as well as Chapter 6, "Desktop Control Language Characteristics," as different implementations of Basic may vary. If you are new to Basic programming, this chapter will introduce you to the fundamentals of the language. You may also want to purchase a Basic programming book from the many that are currently available on the market.

For information about related commands, see Chapter 7, "Related Commands." For information about each command, see Chapter 8, "Command Reference."

A Simple DCL Script

Here is a simple **DCL** script:

```
sub main()  
    msgbox "Hello World"  
end sub
```

Every **DCL** script contains a subroutine called `main`. A subroutine is a set of commands that perform a specific task. Simple Basic programs often have only one subroutine. More complex Basic programs often have more than one subroutine.

The commands `sub main` and `end sub` define the beginning and end of the subroutine. The subroutine contains only one statement `msgbox "Hello World"`.

Running the Script

To run this script:

1. Start the **DCL Editor** (DCLEEDIT.EXE). If you allowed the installation to create a NetTools program group, choose the DCL Editor icon.

2. Type the second line of the above script. The first and third lines are already entered for you.
3. Choose the Start Script command from the Run menu.

When you run this script, the following message box is displayed:



Hello World message box

Saving the Script

To save the script:

1. Choose the Save command from the **DCL Editor** File menu.
2. In the File Save As dialog box, enter **hello** in the File Name box.
3. Choose OK.

A text (ASCII) file named HELLO.DCL contains your script. The default extension for all scripts is .DCL.

Comments

A comment is a note you write in a script. It is not executed with your script. Its sole purpose is to provide information about the script. Comments are very useful when you have to modify a script.

A comment begins with a single quote (') and continues for the rest of the line. The following script contains comments:

```
sub main()  
' Script Name: Hello  
' Written: 3/1/94  
' Author: J. Doe  
  
    msgbox "Hello World"    'display message box  
end sub
```

Notice that a comment can occupy the same line as a **DCL** command (if it follows the command).

Variables

In the previous script, we provided literal text (the actual letters, digits, and special characters) to be displayed in the message box. Literal text is enclosed in double quotation marks (""), for example, "Hello World". Although useful, literal text has its limitations. For example, what if we wanted to say "Hello Everybody" instead of "Hello World." We would have to write a new script.

Variables let you change the data displayed by and used in your scripts. Put simply, a variable is a name with which you associate data—a string of characters or a number. Variable names can be up to 40 characters long. They must start with a letter and can contain letters and digits. (Specific special characters are used to indicate the type of data contained in variables.)

Data Types

The most commonly used types of data in scripts are strings and integers, which are described in the following paragraphs. **DCL** supports many data types, allowing it to accommodate almost any programming situation. These data types include integers, long integers, strings, dialogs, single precision floats, and double precision floats. The characteristics of these data types are described in detail under “Data Types” in Chapter 6, “Desktop Control Language Characteristics.”

Strings

A string is a succession of letters, digits, spaces, and special characters. Here are some example strings:

```
"Hello"
"Hello "
"Hello World"
"Testing 1 2 3"
""           (Null string)
```

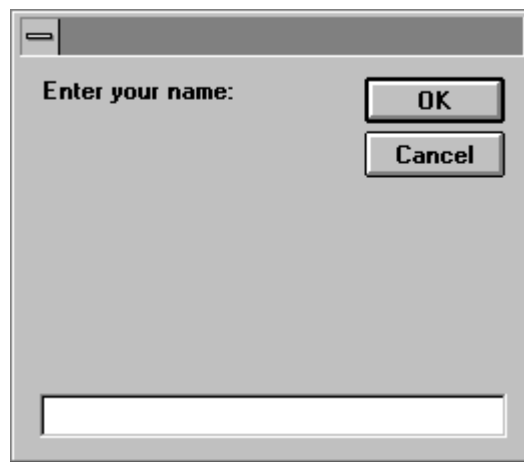
A string variable is simply a name representing a string. The actual string of characters represented by the variable may change. This script makes use of a string variable:

```
sub main()
    name$ = InputBox$("Enter your name:")
    msgbox "Hello " + name$
end sub
```

`InputBox$` is a function (a special type of command we'll discuss later) that displays a dialog box that prompts the user to enter his or her name. The name the user enters is stored in a string variable called `name$`. The contents of the string variable `name$` might be "Jack" or "Jane" or "Chris". The dollar (\$) symbol at the end of a variable name indicates that the variable is a string.

The `msgbox` statement displays a string that is made up of the literal "Hello " and the contents of the string variable `name$`. The two strings are combined (concatenated) by the '+' operator.

Here are the dialog boxes generated by the script.



Input box

Assume that the user types 'Chris' and choose OK. The following message is displayed:



Hello Chris message box

Integers

Integers are whole numbers between -32,768 and 32,767. Long Integers are whole numbers between -2,147,483,648 and 2,147,483,647. The following script uses integers:

```
sub main()
    SysRes%=SystemFreeResources
    FreeMem&=SystemFreeMemory
end sub
```

The `SystemFreeResources` function returns an integer between 0 and 100, indicating the percentage of free system resources. This value is stored in the integer variable `SysRes%`. An integer variable is indicated by the `%` symbol at the end of the name. (The percent symbol in an integer variable name has nothing to do with percentages.)

The `SystemFreeMemory` function returns a long integer representing the amount of free memory. The value is stored in the long integer variable `FreeMem&`. The `&` symbol at the end of a variable name indicates a long integer.

Variable names are not case-sensitive. **DCL** does not distinguish between upper- and lowercase letters in a variable name. However, you can use capitalization to make your variable names more readable. A common practice is to use a capital letter to indicate the beginning of a word. For example, `FreeMem&` is much more readable than `freemem&`.

In its current state, this script produces no visible output. We will expand it in a later section.

Constants

A constant is a value that does not change such as 0, 1, 2, 4, etc. To help make your scripts more readable, **DCL** provides symbolic constants, text strings that represent numeric values. For example the constant `FALSE` equals 0 and `TRUE` equals -1. You do not need to know the numeric value of a constant in order to use it.

The following script illustrates the use of symbolic constants:

```
sub main()
    SystemMouseTrails TRUE
    msgbox "Move the Mouse Around Now"
    SystemMouseTrails FALSE
end sub
```

The values `TRUE` and `FALSE` are used to turn the display of mouse trails (ghost images following the mouse as it moves) on and off.

Several **DCL** commands make use of symbolic constants. By convention, constant names are in uppercase. If a symbolic constant is provided for a command, it is indicated in the description of the command in Chapter 8, “Command Reference.”

User-Defined Constants

The `Const` command lets you define your own constants.

The following script converts from Fahrenheit to Centigrade. The formula for converting from Fahrenheit to Centigrade is:

$$\text{Centigrade} = ((\text{Fahrenheit} - 32) * 5) / 9$$

The `*` is the symbol for multiplication. The `/` represents division.

The script performs the conversion, using the constant `FREEZING` instead of the literal 32.

Constants must be defined before the `main` subroutine is executed.

In this script, the `val` function converts a string to a number, the `str$` function converts a number to a string.

```
Const FREEZING = 32
sub main()
    ftemp$ = InputBox$("Enter a Fahrenheit temperature.")
    fahrenheit% = val(ftemp$)
    centigrade% = ((fahrenheit% - FREEZING) * 5) / 9
    ctemp$ = str$(centigrade%)
    msgbox "Centigrade temperature:" + ctemp$
end sub
```

When you define a constant, you should check the list of constants that have been predefined by **DCL** and avoid using these names. The **DCL** constants are listed under “Constants” in Chapter 6, “Desktop Control Language Characteristics.”

Operators

The following operators are used by **DCL**:

Operator (Precedence: highest to lowest)	Description
()	parentheses
^	exponentiation
-	unary minus
/, *	multiplication and division
\	integer division

Continued...

Operator (Precedence: highest to lowest)	Description
mod	modulo
+, -	addition and subtraction
=, <, >, <=, >=	relational
not	logical negation
and	and
or	or
xor	exclusive or

Subroutines

A subroutine is a separate block of **DCL** code (in the same script) that is called and executed as a unit. Short scripts usually do not require subroutines. When you write a long script, it is helpful to break tasks down into subroutines. The following code illustrates a subroutine:

```
sub HelloProcessing()
    name$ = InputBox$("Enter your name:")
    msgbox "Hello " + name$
end sub
sub main()
    HelloProcessing

    'Rest of Script

end sub
```

In this example the subroutine `HelloProcessing` greets the user and gets his or her name. The block of code contained in the subroutine begins with the command `sub` <NameOfSubroutine> and ends with the command `end sub`. To call a subroutine, you enter the name of the subroutine as you would any other command. The “command” `HelloProcessing` causes the `HelloProcessing` subroutine to be executed. Note that the subroutine must be defined before the call to the subroutine.

For further information about subroutines, see `Sub . . . End Sub` in Chapter 8, “Command Reference.”

Functions

Functions are like subroutines. Unlike subroutines, however, functions return values (integers, strings, etc.). `InputBox$` is a **DCL** function that returns a string. When you call a **DCL** function, you use the returned value in an expression, and you must provide any data to be manipulated by the function. The term for this data is arguments or parameters. The general format for a function call is:

```
VariableName = Function(Parameter1,Parameter2,etc.)
```

For example:

```
name$ = InputBox$("Enter your name:")
```

In this code sample, `name$` is a string variable that contains the string value returned by `InputBox$`. The argument "Enter your name:" is passed to the function and displayed in the dialog box generated by `InputBox$`.

The `len` function returns an integer representing the length of a string. For example:

```
StringLength%=len(name$)
```

In this example, the `len` function puts the length of the string `name$` in the integer variable `StringLength%`.

User-Defined Functions

You can write your own functions. Functions are useful for frequent tasks that return a value. The following script uses a function to convert from Fahrenheit to Centigrade.

```
Const FREEZING = 32
function FarToCent%(f%)
    FarToCent% = ((f% - FREEZING) * 5) / 9
end function
sub main()
    ftemp$ = InputBox$("Enter a Fahrenheit temperature.")
    fahrenheit% = val(ftemp$)
    centigrade% = FarToCent%(fahrenheit%)
    ctemp$ = str$(centigrade%)
    msgbox "Centigrade temperature:" + ctemp$
end sub
```

The function block is defined by the function and end function statements.

The `function` statement contains the function name, return type, and parameters passed to the function. The function name (`FarToCent%`) ends in a character that indicates the type of value the function returns. The function in our script returns an integer (denoted by the `%` symbol). One parameter (`f%`) is passed to the function.

The value to be returned is assigned to the function name (in our script, `FarToCent%`).

The processing of the function occurs as follows:

1. In the main subroutine, `FarToCent% (fahrenheit%)` calls the `FarToCent%` function and passes it the `fahrenheit%` parameter.
2. The `FarToCent%` function receives the integer parameter as `f%` and uses it to represent the Fahrenheit temperature in the conversion to Centigrade.
3. The `FarToCent%` function performs the conversion and returns the result.
4. The returned value is put in the `Centigrade%` variable in the main subroutine.

Passing Parameters by Reference and by Value

As a default, parameters are passed to functions by *reference*. If you are new to programming, this means that the function-side parameters refer to the same variable as the calling-statement-side parameters. In other words, `fahrenheit%` and `f%` refer to the same variable. If you were to change `f%` in the `FarToCent%` function `fahrenheit%` in the main subroutine would also be changed.

The alternative to passing parameters by reference is to pass them by *value*. This means that the *contents* of the variable are passed to the function. The function-side parameter and the calling-statement-side parameter refer to different variables. To specify that a parameter is to be passed by value, precede the parameter in the function statement with `byval`. By adding `byval` to the `FarToCent%` function, you can cause `f%` and `fahrenheit%` to be entirely separate variables. You can change `f%` in the function without affecting `fahrenheit%` in the main subroutine.

```
Const FREEZING = 32
function FarToCent%(byval f%)
    FarToCent% = ((f% - FREEZING) * 5) / 9
end function
sub main()
    ftemp$ = InputBox$("Enter a Fahrenheit temperature.")
    fahrenheit% = val(ftemp$)
    centigrade% = FarToCent%(fahrenheit%)
    ctemp$ = str$(centigrade%)
    msgbox "Centigrade temperature:" + ctemp$
end sub
```

For a more in-depth discussion of functions, see the `Function...End Function` command in Chapter 8, “Command Reference.”

Arrays

An array is a series of variables of the same type all having the same name. You differentiate between the elements of the array by using an integer index. Assume that you have an array of strings called `Cities$`.

Cities\$	Index
"Atlanta"	0
"Chicago"	1
"Los Angeles"	2
"Montreal"	3
"New York"	4

Cities\$ array

`Cities$(0)` contains the string "Atlanta"; `Cities$(1)` contains "Chicago", etc. The form for referencing an element of an array is:

ArrayVariableName(Index)

Here is a sample script that uses an array of strings:

```
sub main()
  dim Performance$(5) as String
  Performance$(0)="Poor"
  Performance$(1)="Fair"
  Performance$(2)="Good"
  Performance$(3)="Excellent"
  Performance$(4)="Perfect"
  gradestr$=InputBox$("Enter a grade from 0 to 100.")
  grade#=val(gradestr$)
  plevel%=grade#\25
  msgbox "The grade is " + Performance$(plevel%)
end sub
```

The activities required to use an array are:

- Declare the array using the `dim` statement. This statement names the array, indicates the number of elements (and dimensions) and specifies the type of data to be contained in the array. The sample script defines an array named `Performance` of type `string`, containing 5 elements and one dimension.

- Load the array with values. This essentially amounts to assigning a value to each element in the array. The `Performance(n) = "value"` statements load the array. The number in parentheses is the index to the array. (Some DCL functions that use arrays load the initial values for you.) By default the first element of an array has 0 as its index. You can use the `Option Base` to set this value to 1, if desired.
- Reference the array as needed. In the subsequent code, an input box asks the user to enter a number from 0 to 100. This returns a string, which the `val` function converts into the long integer `grade#`. We perform integer division of `grade#` by 25 to get an integer between 0 and 4. (Integer division returns only the nonfractional part of the quotient.) This integer is `plevel%`, which serves as an index to select the correct element from the array. The selected string is then displayed in a message box.

For additional information about arrays, see the `dim` statement in Chapter 8, “Command Reference.”

Conditional Statements

Conditional statements are executed only if a particular condition is true. **DCL** supports the following conditional statements: `If...Then...Else` and `Select Case`. They allow your script to perform different tasks under different conditions.

If...Then...Else

The `if...then` statement performs a statement (or set of statements) if a condition is true. Otherwise the statement is skipped. Here is an example of the `if...then` statement.

```
sub main()
    SysRes%=SystemFreeResources
    if SysRes% < 30 then
        msgbox "Your available resources are low."
    end if
end sub
```

This script informs the user if available system resources fall below 30 percent. The `if` keyword specifies the condition that must be true; the `then` keyword signals the beginning of one or more statements to be executed if the condition is true. The terms `if` and `then` must be on the same line. The `if...then` statement ends with the `end if` keyword.

The `if...then...else` statement lets you specify activities for the script to perform whether the condition is true or false. The `else` keyword precedes one or more statements to be executed when the condition is false.

```
sub main()
  SysRes%=SystemFreeResources
  if SysRes% < 30 then
    msgbox "Your available resources are low."
  else
    msgbox "Your available resources are fine."
  end if
end sub
```

A more complex form of the `if...then...else` statement lets you accommodate multiple conditions. The following `if...then...elseif...else` statement varies the message depending on the percentage of available system resources.

```
sub main()
  SysRes%=SystemFreeResources
  if SysRes% < 30 then
    msgbox "Your available resources are low."
  elseif SysRes% < 20 then
    msgbox "Resources extremely low. Close some Applications now."
  elseif SysRes% < 10 then
    msgbox "Resources are DANGEROUSLY low. Close Applications now."
  else
    'everything's ok
    msgbox "Your available resources are fine."
  end if
end sub
```

In this manual the symbol \leftarrow indicates we had to float the rest of a **DCL** command to the next line. In **DCL** scripts, a command cannot be broken between lines. In some cases, the book size made it impossible to keep all of the code on the same line.

For further information, see `If...Then...Else` in Chapter 8, “Command Reference.”

Select Case

The `select case` statement also handles multiple conditions.

Like the previous `if...then...elseif...else` statement, the following `select case` statement varies the message depending on the percentage of available system resources.

```

sub main()
  SysRes%=SystemFreeResources
  select case SysRes%
    case 21 to 30
      msgbox "Your available resources are low."
    case 11 to 20
      msgbox "Resources extremely low. Close some Applications now."
    case 0 to 10
      msgbox "Resources are DANGEROUSLY low. ←
        Close Applications now."
    case else
      'everything's ok
      msgbox "Your available resources are fine."
  end select
end sub

```

The `select case` statement begins with `select case <VariableName>` and ends with `end select`. The logic of the `select case` statement is to test each case in the `select case...end select` block. Each case statement specifies a condition. If the condition is true, the statement(s) associated with are executed. The `case else` statement is executed if none of the conditions tested are true. The breadth of expressions supported by the case statement make it a powerful tool for conditional processing. For further information, see `Select Case` in Chapter 8, “Command Reference.”

Loops

Loops are blocks of statements that perform the same action multiple times while a condition is true or until a condition is met. Each pass through a loop is called an iteration. **DCL** supports the following iterative statements: `For...Next`, `While...Wend`, and `Do...Loop`.

For...Next

A `For...Next` loop executes a group of statements a specified number of times. Here is a `For...Next` loop:

```

sub main()
  for x% = 1 to 5 step 1
    msgbox str$(x%)
  next x%
  msgbox "You're past the loop."
end sub

```

In the `For` statement you specify a:

- Counter variable, which in our example is `x%`.

- Beginning and ending value for the counter variable, in our example 1 and 5 respectively.
- Increment. This is provided by the `step` portion of the statement. The `step` can be omitted if the increment is 1.

The `Next` statement marks the end of the loop and increments the counter variable by the `step` value.

The statements inside the loop are executed repeatedly until the counter exceeds the ending value. In our example the loop is executed five times. When the counter exceeds the ending value, the script continues with the next statement after the `For . . . Next` loop.

You can use variables to represent starting and ending values and the increment. For example:

```
sub main()
  startval%=1
  endval%=10
  incr%=2
  for x% = startval% to endval% step incr%
    msgbox str$(x%)
  next x%
  msgbox "You're past the loop."
end sub
```

For more information, see `For . . . Next` in Chapter 8, “Command Reference.”

While...Wend

A `While . . . Wend` loop executes a group of statements repeatedly while a condition is true. Once the condition is false, the statement following the `Wend` statement is executed.

```
sub main()
  testval%=1
  while testval% < 6
    msgbox str$(testval%)
    testval% = testval% + 1
  wend
  msgbox "You're past the loop."
end sub
```

The `While` statement marks the beginning of the loop and specifies the test condition. In our example, the condition is that the variable `testval%` must be less than 6.

The `Wend` statement marks the end of the loop. The loop is executed repeatedly until `testval%` is equal to 6. For more information, see `While . . . Wend` in Chapter 8, “Command Reference.”

Do...Loop

The Do...Loop statement executes a group of statements while or until a condition is true. It has several forms. For further information see, `Do . . . Loop` in Chapter 8, “Command Reference.”

Where to Go from Here

The purpose of this chapter was to give you an initial exposure to **DCL**. To increase your familiarity with the language, we recommend that you examine the numerous sample scripts provided for the individual **DCL** commands in the online help or Chapter 8, “Command Reference.”

