Users Guide
For
Creative BASIC

September 2007

Commands, Statements and Functions

Creative BASIC is a language for computers just as English or Spanish is a language for humans. While we understand thousands of written and spoken words, Creative BASIC only understands 250 or so 'keywords'. These words make up the heart of the language and consist of commands, statements and functions.

A command is a single word with no parameters. The word 'parameter' may be unfamiliar to you and means 'input' or extra information. We have all used commands before, when training our dogs we say 'sit' and the dog understands us.

Statements and functions are commands that need extra information to be understood by Creative BASIC. This extra information is conveyed in the form of parameters. If we continue the dog analogy think of the statement 'Get ball', the keyword is 'Get' and the parameter is 'Ball'.

A function is a statement that returns information to you. If the dog obeys the statement 'Get ball' then we now have a function as he returns the ball and waits for the next command.

If we have a very smart dog, we may be able to give him two parameters in our statement as in 'Get red ball'. In Creative BASIC multiple parameters are separated with the comma ',' . For example:

```
GET red,ball
```

Or if used as a function:

```
success = GET(red,ball)
```

Please keep in mind that these are not real functions and are only used here for the purpose of discussion.

In the first example, 'GET' was used as a statement and did not let us know if the ball was actually returned. In the second example, we used 'GET' as a function and added parenthesis around the parameters. This is known as 'syntax' or the rules of the language.

Rule 1: A statement *should not* have parenthesis around its parameters

Rule 2: Functions ***Must*** have parenthesis around its parameters

In the previous section on using the editor we had you type in a short program:

```
OPENCONSOLE
PRINT "Hello World"
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

This program consists entirely of commands and statements with one exception. The keyword INKEY$ is actually a function that takes no parameters. Since there are no parameters there is no need for parenthesis and we have another syntax rule:

Rule 3: Functions with no parameters *should not* have parenthesis. An exception to this rule is when calling a DLL function that has no parameters. In this case you must use the parenthesis. Example: function( )

You may have noticed that we keep saying *should* and *should not*. This is because there are always exceptions to the rules and these will be noted as we progress.

Special Characters

Creative BASIC supports a few special shortcuts worth mention. The first is the line separator, the colon ':'. Using the colon, you can put more than one program statement on the same editor line. Example:

```
OPENCONSOLE : PRINT "Hello World"
```

You cannot use the colon to start a new line if the line begins DECLARE. The colon line separator may be used anywhere else.

To add comments to your program use the single quote ` at the beginning of any line

'This is a comment and wont be executed

Comments will show up as green text in the editor and are ignored by Creative BASIC. Comments are great for documenting your code so you can more easily understand what you wrote when you come back to it after a long period of time.

In the next section, we will learn about variables, a very important part of any computer language.

Variables

A variable is a temporary storage location for information in your program. Think of it as 'memory' for your information.

All variables are referenced by a name that you choose and can be any combination of letters and numbers. A variable name must begin with a letter. Consider the following example:

```
OPENCONSOLE
MyNumber = 1
PRINT MyNumber
PRINT "Press Any Key to Continue"
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

The second line of the program *defines* a variable named 'MyNumber' and stores in it the value of 1. The third line uses MyNumber as a parameter and prints '1' in the console window.

Type this program in yourself and run it. Creative BASIC understands that the word 'MyNumber' is a variable and displays the contents of the variable when asked to 'print' it.

Now change the second line to read:

```
MYNUMBER = 1
```

Do you think the program will still run correctly? Go ahead and try it. Variable names are not case sensitive in Creative BASIC so 'MyNumber' is the same as 'MYNUMBER' as well as 'mynumber'. Now that we have a basic understanding here are a few rules to remember:

Rule 4: Variable names must begin with a letter.

Rule 5: Variable names are not case sensitive.

Rule 6: Variable names can be a maximum of 30 characters long.

Our program above has one variable of type integer. This means that the variable can only store whole numbers with no decimal. What if you wanted to store a decimal value? Read on to find out how.

Variable Types

A variable is said to have a specific type. Creative BASIC will automatically assign a type to a variable based on your program statements. For example, if you change line two in the previous program to read:

```
MyNumber = 1.0
```

Then Creative BASIC will give 'MyNumber' the type of *float* and you can store any decimal number in MyNumber. The best method of telling Creative BASIC what type your variable you want is to use the DEF statement that is short for 'DEFine'. When you use the DEF statement, you predefine the variable for use as a specific type. Example:

```
OPENCONSOLE
DEF MyNumber AS FLOAT
MyNumber = 13.5
PRINT MyNumber
PRINT "Press Any Key to Continue"
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

The syntax for the DEF statement looks like this:

DEF name{[x{,y{,z}}]}{,name...} AS type

Creative BASIC also supports the colon ':' as a shortcut to the AS keyword and this will be used throughout this user's guide

DEF name{[x{,y{,z}}]}{,name...}: type

The parameters in the first set of braces '{ }' are optional and will be discussed in the section on arrays. In its simplest form the syntax for the DEF statement is:

DEF name{,name..} : type

The optional second name allows you to define more than one variable of the same type at once. The braces are not actually typed in the editor and should be omitted. For example:

DEF MyNumber, TotalPrice : FLOAT

Defines two variables as type *float.* This is a shortcut to defining multiple variables of the same type on the same line.

DIM is synonymous for DEF and can be used interchangeably.

Up until now we have been discussing only numeric variable types. The next variable type to discuss is the *string* variable. String variables store text and get their name from the term 'a *string* of characters'.

You can define string variables in the same way as numeric variables, either with a direct assignment or by using the DEF statement. Examples:

Name$ = "John Doe"

or

DEF Name$:STRING

The dollar sign on the end of the variable name is used as a convention and is not normally required. Using the '$' makes it easier to differentiate variable types when reading your program and we will use it throughout this manual. Think of the '$' as meaning 'string'.

Notice that literal strings are enclosed in double quotes. This tells Creative BASIC that the characters are to be treated as text and not a command or variable.

Rule 7: All literal strings must be enclosed in double quotes.

Here is an example program you can try to show the use of string variables and string literals. Don't worry if you do not understand completely yet as the use of string variables will become clearer as we progress.

```
OPENCONSOLE
DEF MyNumber:FLOAT
DEF MyName$:STRING
DEF MyAge:INT
MyNumber = 13.5
MyName$ = "John Doe"
MyAge = 33
PRINT MyNumber,":",MyName$,":",MyAge
PRINT "Press Any Key to Continue"
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

We have covered the three basic variable types you will be using in most of your programs, INT FLOAT and STRING. Creative BASIC has many other built in types as well and we will be covering them in other sections. Refer to figure 11.1 for a list of built in types supported by Creative BASIC.

Variable Assignments

Variables are assigned values either directly with the '=' sign or indirectly by a function or statement. Variables can be assigned the contents of other variables as long as they are the same type or a conversion can be calculated by Creative BASIC. For example, an integer variable can be assigned the contents of a float variable but will loose its fractional value:

```
DEF WorkDays:FLOAT
DEF Days:INT
WorkDays = 13.5
Days = WorkDays
PRINT Days
```

Will print the number '13' as the fractional value of '.5' was lost in the conversion of FLOAT to INT.

Figure 11.1

TYPE	AUTO	USAGE	DIRECT ASSIGN	SIZE IN BYTES
WORD	NO	Integer up to 65535	YES	2
INT	YES	Signed integer numbers	YES	4
UINT	NO	Unsigned integer numbers	YES	4
INT64	NO	Signed large integer	YES	8
UINT64	NO	Unsigned large integer	YES	8
FLOAT	YES	Single precision numbers	YES	4
DOUBLE	NO	Double precision numbers	YES	8
STRING	YES	Text up to 254 characters	YES	255
ISTRING	NO	Text from 1 to 65,535 characters.	YES	1 to 65536
CHAR	NO	Single character of text or any number from 0 to 255	YES	1
WINDOW	NO	Placeholder for a window	NO	N/A
DIALOG	NO	Placeholder for dialogs	NO	N/A
FILE	NO	ASCII file I/O	NO	N/A
BFILE	NO	Binary file I/O	NO	N/A
MEMORY	NO	Allocated memory	YES	N/A
POINTER	NO	Variable reference	YES	N/A
USER	NO	TYPE statement	YES*	Sum of elements

*Identical user types can be directly copied by assigning.

String variables can only be assigned by another string variable, a character variable a string literal or a function that returns a string. Any attempt to assign a string variable with a numeric value will result in an 'Illegal Assignment' error. Examples:

```
DEF Name$,Name2$:STRING
DEF Initial:CHAR
Name$ = "John Doe"
Name2$ ="Julie Doe"
Initial = "A"
Name$ = Name2$
REM this next line produces an error
```

Name2$ = 100

Referring again to figure 11.1 you will see that some variable types are not directly assignable and are only assigned a value from a function. Statements trying to use these variable types on the left side of an '=' sign will generate an 'Illegal Assignment' error.

The second column marked AUTO specifies which variable types do not need to be predefined with the DEF statement. These variable types can be defined just by assigning them to a value. Examples:

REM define an integer

A = 1

REM define a float

B = 1.1

REM define a string..Since not predefined we need to use the $

C$ = "Text"

Turning off auto define

While it is convenient to let Creative BASIC define your variables automatically, it is sometimes desirable to turn this feature off. This can aid in finding errors that would normally be difficult to locate. The Creative BASIC statement AUTODEFINE handles this for you. The syntax of AUTODEFINE is

AUTODEFINE "ON" | "OFF"

For example:
AUTODEFINE "OFF"
A = 1

Will generate the error message "Undefined variable on line 2 'A' not found" indicating that the DEF statement was never used to define the variable.

Arrays

An array is a collection of data of the same type. Arrays are referenced by their variable name and an index. Think of a box of cookies. Each row of cookies can be thought of as an array. The syntax of an array is:

DEF variable[items {, items, items}]:type

Example fragment:

DEF age[20]:INT
age[0] = 40
age[1] = 32

As noted in the above example you access an array using an 'index' into the array. Indices start at 0 so the maximum index will be one less than the value specified in the DEF statement. To create a multidimensional array use the comma to separate dimensions.

DEF myarray[20,20]:INT
myarray[0,1] = 5

Arrays can also be defined dynamically when the size of the array is to be determined at run time. In this case a variable may be used to define the size of the array.

DEF sizex,sizey:INT
... Calculate the sizes
DEF dynamic[sizex,sizey]:INT

Creative BASIC supports arrays up to three dimensions.

Initializing the array

An array may be initialized by using a list of data parameters. Each line can have up to 25 data elements and the array can be initialized starting with a certain element. For example:

```
DEF myarray[10]:INT
'initialize the first 7 elements
myarray = 1,10,23,2,4,16,27
'initialize the last 3 elements
myarray[7] = 5,18,42
```

The ISTRING type

ISTRING is an advanced STRING type that can be defined and accessed like an array. ISTRING can then be used anywhere a normal string would be expected. The maximum length of an ISTRING is 65535 characters.

Note: An ISTRING always allocates 1 byte extra for the terminating NULL character. So if you need an exact length for passing to a DLL or Windows function then subtract one from the defined size.

Example:

```
OPENCONSOLE
DEF name[30]:ISTRING
name = "john Smith"
name[0] = "J"
PRINT name
PRINT "Press any key to close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

The TYPE statement

The TYPE statement begins defining a user variable type. The new type will contain all of the variables between TYPE and ENDTYPE. Accessing the variables in a user type is done with the dot operator `.`. Any variable type can be defined between TYPE and ENDTYPE. Syntax of TYPE:

```
TYPE name {,pack}
DEF ...
{DEF ...}
ENDTYPE
```

Once a user variable type is defined you can create variables of that type using the DEF statement.

Example fragment:

```
TYPE phonerecord
DEF Name:STRING
DEF Age:INT
DEF Phone[20]:ISTRING
ENDTYPE

DEF Rec:phonerecord

Rec.Name = "Joe Smith"
Rec.Age = 35
Rec.Phone = "555-555-1212"
...
```

Typed variables provide a convenient way of accessing groups of related information. The optional pack parameter specifies how this variable will be sent to DLL functions.

User types can be copied as long as the typed are identical.

Example:

```
TYPE mytype
DEF Name:STRING
DEF Age:INT
ENDTYPE

DEF t1,t2:mytype

t1.Name = "John Doe"
t1.Age = 30

t2 = t1
PRINT t2.Name
```

Since the types are identical, Creative BASIC copies the contents of t1 into t2.

The Console Window

In all of our previous examples we have used the console window to show our programs output. The console window is similar to the MSDOS™ window built in to Windows. The console window can be used for text output only. Graphics are not supported for console mode programs.

To open a console window in your program use the OPENCONSOLE command. When you are done with the console window issue a CLOSECONSOLE command. If the console window is already open the OPENCONSOLE command does nothing.

Only one console window can be open in your program and it can only be closed with the CLOSECONSOLE command.

Displaying Text

To display text in the console window use the PRINT statement. The print statement has the following syntax.

PRINT {window,}param1{,param2...}{,}

The first optional parameter is to output text in a regular window and will be discussed in the section on using windows.

The optional trailing comma tells Creative BASIC not to issue a carriage return at the end of the line. This allows using multiple PRINT statements to work with the same line of text.

A print statement with no parameters will move the console window cursor to the next line. Up to 25 parameters may be specified in a print statement. The following program demonstrates PRINT statement usage.

```
OPENCONSOLE
DEF name$,address$,city$,state$,zip$:STRING
DEF income:FLOAT
name$ = "John Doe"
address$ = "123 Anywhere St"
city$ = "Middle Town"
state$ = "WI"
zip$ = "55555"
income = 38.5
PRINT name$
PRINT address$
PRINT city$," ",state$," ",zip$
PRINT
PRINT "Median Income:",
PRINT income
PRINT
PRINT "Press Any Key To Close"
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

Changing Colors

To display your text in different colors in the console window use the COLOR statement. The COLOR statement has the syntax of:

COLOR foreground,background

Where background and foreground are positive numbers from 0 to 15. Refer to figure 16.1 for the colors supported by the console window.

In the previous example, insert the statement:

COLOR 14,1

Right before all the PRINT statements to see yellow text on a blue background.

Positioning Text

You can position your text anywhere in the console window by using the LOCATE statement. The LOCATE statement has the syntax of:

LOCATE y,x

where y is the vertical character position and x is the horizontal character position. The origin is the upper left corner at character position 1,1. This short example demonstrates the LOCATE statement:

```
OPENCONSOLE
LOCATE 10,1
PRINT "Position 10,1"
LOCATE 10,50
PRINT "Position 10,50"
LOCATE 1,1
PRINT "Position 1,1"
LOCATE 1,50
PRINT "Position 1,50"
LOCATE 12,1
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

figure 16.1

COLOR number	Color Produced
0	BLACK
1	BLUE
2	GREEN
3	CYAN
4	RED
5	MAGENTA
6	BROWN
7	WHITE
8	GRAY
9	LIGHT BLUE
10	LIGHT GREEN
11	LIGHT CYAN
12	LIGHT RED
13	LIGHT MAGENTA
14	YELLOW
15	HIGH INTENSITY WHITE

Clearing The Window

To clear the console window use the command CLS. CLS removes all text and color from the console window. Example:

```
OPENCONSOLE
PRINT "CCCCCCCCCCCCCCCCCCCC"
PRINT "XXXXXXXXXXXXXXXXXX"
PRINT "Press any key to clear this window"
DO:UNTIL INKEY$ <> ""
CLS
LOCATE 12,1
PRINT"Now press any key to close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

Getting Input

So far, we have concentrated on displaying and manipulating text in the console window. For a program to be interactive, we need some way to get data from the user into our program. Creative BASIC provides one statement for the console window INPUT and one function INKEY$.

The INPUT statement allows your program to receive information from the user directly into any assignable variable type. The syntax of the INPUT statement looks like this:

INPUT {”prompt”,}variable

The first optional parameter is a literal string used as the prompt for the input statement. The INPUT statement keeps collecting user key presses until the <ENTER> key is pressed at which time the input is stored in the variable. Example:

```
OPENCONSOLE
CLS
DEF name$:STRING
INPUT "Enter your name:",name$
PRINT "Hello ",name$," Nice to meet you"
PRINT name$,"Please press any key to close!"
DO: UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

The INKEY$ function waits for the user of your program to press one key and returns the character pressed. This return value can be assigned to either a STRING or CHAR variable. The syntax of the INKEY$ function is:

STRING|CHAR = INKEY$ { (raw) }

The optional raw parameter if equal to 1 tells INKEY$ to return virtual key codes. It is important to note that before your program executes the INKEY$ function that the user of your program may have typed a number of keys. Always check the return value for the expected results.

To demonstrate the INKEY$ function type in this program and run it. There are a few statements you may not recognize yet but will be covered soon.

```
OPENCONSOLE
DEF key$:STRING
LOCATE 1,1
PRINT "INKEY$ demonstration. ",
PRINT "Press <ENTER> to end program"
PRINT "Press any other keys to display them"
DO
key$ = INKEY$
IF Key$ <> ""
LOCATE 10,1
```

```
COLOR 14,1
PRINT "You Pressed >",
COLOR 2,0
PRINT key$,
ENDIF
UNTIL key$ = CHR$(13)
CLOSECONSOLE
END
```

The INKEY$ statement is great for processing single key choices from a list of options.

In the next section we will learn about Creative BASIC's math operators and syntax. Also we will cover all the built in trigonometric functions available to your program.

Operators and Math

Creative BASIC has a complete range of math operators and functions for your program to use. The following table list Creative BASIC's operators and their meanings.

figure 21.1

Operator	Function
+	Addition, String Concatenation
-	Subtraction
*	Multiplication
/	Division
&	Bit wise and logical AND
|	Bit wise and logical OR
||	Exclusive OR (XOR)
%	Modulus. The remainder of an integer division
^	Power of
Conditional Operators	Meaning
>	Greater Than
<	Less Than
<>,~	Not Equal to
=	Equal To
>=	Greater Than or Equal To
<=	Less Than or Equal To

This following short example will demonstrate the use of math operators.

```
OPENCONSOLE
DEF num,num2:FLOAT
num = (4 * 5 + 2) / 3
PRINT "(4 * 5 + 2) / 3 = ",num
num2 = num * 10
PRINT "Times 10 = ",num2
PRINT "4 raised to the power of 3 = ", 4 ^ 3
PRINT
PRINT "Press any key to close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

String Concatenation

The '+' sign also allows concatenation or appending of many strings together. This is the only math operator to work with strings. For example, suppose we had four string variables and we wanted their contents 'appended' together to form one long string:

```
OPENCONSOLE
DEF a$,b$,c$,d$,result$:STRING
a$ = "this "
b$ = "is "
c$ = "a "
d$ = "long string"
result$ = a$ + b$ + c$ + d$
PRINT result$
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

Would print "this is a long string" to the console window.

Math Functions

Creative BASIC includes math functions for trigonometric, intrinsic and random number generation. Figure 21.2 lists the functions and their return values. Each of these functions takes 1 parameter, a numeric expression or variable, and returns a numeric result. All trigonometric functions take their parameters in radians and return their results in radians. Example of using math functions:

```
OPENCONSOLE
DEF number:INT
PRINT "The sine of 1 = ",SIN(1)
PRINT "The cosine of 1 = ",COS(1)
number = INT(RND(50))
PRINT "A number between 1 and 50 = ",number
PRINT "Absolute Value of -50 = ",ABS(-50)
PRINT
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

Figure 21.2

Function	Return Value
SIN(n)	Sine of n
COS(n)	Cosine of n
TAN(n)	Tangent of n
ASIN(n)	Arcsine of n
ACOS(n)	Arccosine of n
ATAN(n)	Arctangent of n
SINH(n)	Hyperbolic sine of n
COSH(n)	Hyperbolic cosine of n
TANH(n)	Hyperbolic tangent of n
LOG(n)	Natural logarithm of n
LOG10(n)	Base 10 logarithm of n
SQRT(n)	Square root of n
EXP(n)	Exponential value of n

ABS(n)	Absolute value of n (removes sign)
CEIL(n)	The smallest integer greater than or equal to n
FLOOR(n)	The largest integer that is less than or equal to n
RND(n)	A random number between 0 and n
INT(n)	The whole number portion of n
NOT(n)	Returns the ones compliment of n
SGN(n)	-1 if negative, 1 if positive and 0 if n is 0

Precedence of math operators from highest to lowest:
( )
- Unary Minus
^
/, *, %,||
+, -
<, >, <=, >=, <>, =
&, |

Operators and Strings

Creative BASIC allows comparison of strings in IF and SELECT statements. When comparing strings the allowed operators are:

= Test for equaltity
<> Test for inequality
< Test if one string is less than another alphabetically
> Test if one string is greater than another alphabetically
<= Test for less than or equal to
>= Test for greater than or equal to

The less than and greater than variants check each character position until a determination can be made. The first character always has greatest significance in the test.

Example Fragment:

```
...
A$ = "aaa"
B$ = "aab"
IF A$ < B$ THEN PRINT "Less Than" ELSE PRINT "Greater Than"
...
```

Would print "Less Than" since the third character in A$ is alphabetically less than the third character in B$

Controlling Precision

Creative BASIC defaults to 2 decimal places when displaying FLOAT and DOUBLE type with the PRINT statement. To control this use the SETPRECISION command. The syntax of SETPRECISION is:

SETPRECISION places

For example using SETPRECISION 16 will display 16 digits after the decimal place.
See also: The USING statement

Conditional Statements

Creative BASIC has two conditional statements, IF and SELECT. A conditional statement selects a section of your program to execute based on a condition that you choose.

IF Statement

The IF statement will probably be your most used conditional statement, and has the syntax of:

```
IF condition
……
ELSE
…..
ENDIF
```

The condition is any math, comparison or algebraic expression that results in a yes or no result. The ELSE statement is optional and represents the group of statements that will be executed if the condition is false.

An example of the IF statement:

```
OPENCONSOLE
DEF number:INT
INPUT "Pick a number ",number
IF number = 12
PRINT "You guessed correctly!"
ELSE
PRINT "Sorry wrong number"
ENDIF
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

In the previous math section, figure 21.2 listed the conditional operators that can be used in an IF statement. Comparisons can be performed on numbers, strings and variables.

The IF statement also supports a single line version using the THEN keyword. When used on a single line the ENDIF statement is not necessary. Single line IF statements allow only one action be executed when the condition is true and one following an ELSE.

Example of a single line IF statement:

```
OPENCONSOLE
DEF number:INT
INPUT "Pick a number ",number
IF number = 12 THEN PRINT "You guessed correctly!" ELSE PRINT "Sorry wrong number"
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

You can use multiple conditions in an IF statement by using the logical operators & and |. The conditions should be enclosed in parenthesis because the logical operators have higher precedence. Examples:

```
IF (a < 1) & (b > 3)
PRINT "TRUE!"
ENDIF
```

Will print TRUE! if a is less than 1 AND b is greater than 3.

```
IF (a < 1) | (b > 3)
PRINT "TRUE!"
```

```
ENDIF
```

Will print TRUE! if a is less than 1 OR b is greater than 3.

SELECT Statement

The SELECT statement is an advanced conditional statement that allows you to test against many different combinations. The syntax of the select statement is:

```
SELECT variable|expression
CASE test1
...
CASE test2
...
CASE testn
...
DEFAULT
...
ENDSELECT
```

The SELECT statement allows unlimited ‘if equal to" conditions. The statements after the CASE statement will only be executed if the variable or expression is equal to the test condition. Example:

```
OPENCONSOLE
DEF Choice$:STRING
PRINT "Press some keys, Q to quit"
LABEL again
DO
Choice$ = INKEY$
UNTIL Choice$ <> ""
SELECT Choice$
CASE "A"
CASE "a"
PRINT "You pressed A!!"
CASE "Z"
CASE "z"
PRINT "Z is my favorite letter!"
CASE "Q"
CASE "q"
CLOSECONSOLE
END
DEFAULT
PRINT "You pressed the letter ",Choice$
ENDSELECT
GOTO again.
```

The DEFAULT condition is optional and will be executed if none of the CASE statements is true.

The SELECT statement can also be used to test multiple conditions and execute the first TRUE case. To do this use SELECT 1 as the opening statement and code each CASE statement as a condition. Example:

```
OPENCONSOLE
DEF Choice:INT
PRINT
LABEL again
INPUT "Enter a Number, 0 to end: ", Choice
IF Choice = 0
CLOSECONSOLE
END
ENDIF
SELECT 1
CASE (Choice > 10)
```

```
PRINT "number is greater than 10"
CASE (Choice < 10)
PRINT "Number is less than 10"
CASE (Choice = 10)
PRINT "Number is equal to 10"
ENDSELECT
PRINT
GOTO again.
```

In this example we test the choice against multiple conditions using only one SELECT statement. Your conditions can be as complex as needed, only the first TRUE condition will be executed so make sure the conditions are unique.

Loop Statements

Creative BASIC has three built in loop statements. Loop statements execute one or more lines of your program repeatedly until a condition is satisfied.

FOR Statement

The FOR statement executes lines of code until a counter variable reaches a specified number. The NEXT statement determines the end of the loop. The FOR statement has the syntax of:

```
FOR variable = start TO end {# skip}
...
NEXT variable
```

The optional skip number tells Creative BASIC to count by a certain number. One NEXT statement is needed for each FOR statement. To better demonstrate look at this example:

```
OPENCONSOLE
DEF counter:INT
DEF name$:STRING

INPUT "Enter your name ",name$

FOR counter = 1 TO 20 #2
PRINT "Hello ",name$
NEXT counter

PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

Will print a greeting 10 times. The '#2' tells Creative BASIC to count by twos until the counter variable is greater than or equal to 20. Once the counter has exceeded the end condition the loop ends. Start, end and skip could also be variables instead of constants. Besides the # sign Creative BASIC also recognizes the SKIP keyword.

DO and WHILE statements

The DO statement executes one or more statements in your program at least once and repeats them until the condition in the UNTIL statement is met. The DO statement has the syntax of:

```
DO
...
UNTIL condition
```

The WHILE statement executes one or more statements in your program as long as the condition is met. The end of the loop is determined by the ENDWHILE statement. The WHILE statement has the syntax of:

```
WHILE condition
...
ENDWHILE
```

If you examine the two statements, you will see that they are exact opposites of each other. The DO statement loops while a condition is false, the WHILE statement loops while a condition is true.

Examples:

```
OPENCONSOLE
DEF x:INT
x=20

DO
```

```
PRINT x
x=x-1
UNTIL x < 11


WHILE x < 21
PRINT x
x=x+1
ENDWHILE

PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

File Operations

File operations allow you to read and write data to permanent storage devices such as a hard drive, floppy or removable media. They can also be used to send data to peripherals attached to your computer, such as a printer.

OPENFILE function

The OPENFILE function initiates the connection between your program and the outside world. It has as syntax of:

Error =OPENFILE (FileVariable, filename, flags)

The filevariable must be either of type FILE or BFILE. The flags can either be "R" for reading, "W" for writing or "A" for appending to an existing file.

The function will return 0 if the file was successfully opened. Any other value means that there was a problem and you should not try reading or writing to the file.

CLOSEFILE statement

After you are finished reading or writing to a file you must close it to make sure that system resources are returned to Windows. If you leave too many files open at once you will receive an error message. The CLOSEFILE statement has a syntax of:

CLOSEFILE filevariable

Example:

```
OPENCONSOLE
DEF myfile:FILE
IF(OPENFILE(myfile,"C:\CBASICTEST.TXT","W") = 0)
WRITE myfile,"This is a test"
CLOSEFILE myfile
PRINT "File created successfully"
ELSE
PRINT "File could not be created"
ENDIF
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

The previous example attempts to open a file for writing in the C:\ directory called "CBASICTEST.TXT". After opening the file, it writes one line of text to the file and then closes the file.

WRITE function

In the example a new function was introduced, WRITE. WRITE is to file operations what PRINT is to the screen. WRITE returns 0 on success. It has the syntax of:

error = WRITE (filevariable, expression {,expression…})

Any expression, variable or constant that you can use with a PRINT statement will work with the WRITE function

READ function

Writing to files would not be of much use if we could not read what we wrote. The READ function is to file operations what INPUT is to the console. The READ function returns 0 on success and has the syntax of:

Error = READ (filevariable, variable {,variable…})

Example:

```
OPENCONSOLE
```

```
DEF myfile:FILE
DEF ln:STRING
IF(OPENFILE(myfile,"C:\CBASICTEST.TXT","R") = 0)
IF(READ(myfile,ln) = 0)
PRINT ln
ENDIF
CLOSEFILE myfile
PRINT "File read successfully"
ELSE
PRINT "File could not be opened"
ENDIF
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

If you run both examples you will see "this is a test" printed in the console window.

LEN function
The LEN function works with any variable type in Creative BASIC. For strings it returns the number of characters in the string. For files it returns the length of the open file in bytes. The syntax of LEN is:

Length = LEN (variable | file)

Binary Files

Reading and writing text to a file is fine for a word processor or editor. What if you wanted to create a file with the last 100 years of stock market data? You could use a standard FILE and it would work fine. The problem is that a standard FILE works with ASCII data, or text, and every number would be stored in human readable form. For example the number 3,000,000 would take 7 bytes of space in the file to store. A more efficient way would be to store the number the same way your computer does, in binary. The number 3,000,000 only takes 4 bytes in binary making the file much smaller.

To use binary files you define a variable of type BFILE and use the OPENFILE, CLOSEFILE, READ and WRITE functions just like you would with standard files.

Example:

```
OPENCONSOLE
DEF myfile:BFILE
DEF mynumber:INT
mynumber = 3000000
IF(OPENFILE(myfile,"C:\CBASICDATA.DAT","W") = 0)
WRITE myfile,mynumber
CLOSEFILE myfile
ENDIF
IF(OPENFILE(myfile,"C:\CBASICDATA.DAT","R") = 0)
IF(READ(myfile,mynumber) = 0)
PRINT "This is what I read: ",mynumber
ENDIF
CLOSEFILE myfile
ENDIF
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

If you were to look at that file with a disk, or file editor, you would see it contained exactly four bytes of data. A byte is the same size as a character or letter.

SEEK command

The SEEK command allows setting the file pointer to a position for reading or writing. SEEK also allows obtaining the current file

position. The syntax of SEEK is:

SEEK filevariable, position
position = SEEK(filevariable)

Position specified the zero-based byte offset to begin reading or writing data. The second for of seek returns the current position

Random Access Files

The READ and WRITE functions are known as sequential file operations. They read and write one piece of data at a time until the end of file is reached. For small files this is fine and very manageable. Lets consider, however, a very large file like an address book. If you have 200 names and addresses stored in a file and want to read the 125th name you would need to read 124 entries before you get to the one you want. Luckily, there is a better way.

GET and PUT statements

The GET and PUT statements operate on a file using a record number as a parameter. Instead of having to access the 125th entry by reading the entire file you can get to that entry directly. The syntax of GET and PUT is:

GET filevariable, record, variable

PUT filevariable, record, variable

File variable must be of type BFILE. The variable to be written can be a user type allowing complex records to be used. The record number must be greater than zero.

Example fragment:

```
TYPE HighScoreTopTen
DEF Score:INT
DEF Time:INT
DEF pname[20]:ISTRING
ENDTYPE
DEF TopTen:HighScoreTopTen
DEF myfile:BFILE
...
IF(OPENFILE(myfile,"C:\HIGHSCORES.DAT","W") = 0)
...
TopTen.Score = 100
TopTen.Time = 25
TopTen.pname = "The Winner"
PUT myfile,7,TopTen
...
```

This would write the 7th record of the file with the contents of TopTen

COPYFILE function

Copies a file from a source directory to a destination. The syntax of COPYFILE is:

error = COPYFILE(source, dest, fail)

Source and destination are strings containing the full paths of the file to copy. If fail = 1 then the file will not be overwritten if it already exists in the destination directory. If fail = 0 then the file will be overwritten. COPYFILE returns 0 on error.

DELETEFILE function

Deletes the specified file. The syntax of DELETEFILE is:

error = DELETEFILE(name)

Name is a string containing the full path to the file. DELETEFILE returns 0 if the file does not exist or is locked by another

process.

CREATEDIR function
Creates a new directory. The syntax of CREATEDIR is:

error = CREATEDIR(name)

Name is a string containing the full path to the directory. The name should not end with a '\' character. CREATEDIR returns 0 on error.

REMOVEDIR function
Removes the directory specified. The directory must be empty before it can be removed. The syntax of REMOVEDIR is:

error = REMOVEDIR(name)

Name is a string containing the full path to the directory. The name should not end with a '\' character. REMOVE returns 0 on error.

Reading directories

FINDOPEN function
To read all of the file names in a directory use the FINDOPEN function to get a handle to the directory first. Once a directory is opened the names of the files can be obtained with the FINDNEXT function. The syntax of FINDOPEN is:

handle = FINDOPEN(directory)

Directory is a string the contains the full path to the directory plus any wildcard symbols for file matching. Example: "c:\windows*.txt". handle is an integer variable. FINDOPEN returns 0 if the directory could not be opened for reading.

FINDNEXT function
After a directory is successfully opened with the FINDOPEN function use FINDNEXT to retrieve the filenames in a loop. FINDNEXT returns an empty string when all of the file names have been retrieved. The syntax of FINDNEXT is:

name = FINDNEXT(handle)

Handle is the integer value returned by FINDOPEN.

FINDCLOSE statement
After you are finished reading a directory you must close the handle with FINDCLOSE. If the handle is not closed memory loss will occur. The syntax of FINDCLOSE is:

FINDCLOSE handle

You must not close a handle more than once.

Example:

```
OPENCONSOLE
DEF dir:INT
DEF filename:STRING

dir = FINDOPEN("c:\*.*")
IF(dir)
DO
filename = FINDNEXT(dir)
PRINT filename
UNTIL filename = ""
FINDCLOSE dir
ENDIF
```

```
PRINT "Press Any Key"
DO:UNTIL INKEY$ <> ""

CLOSECONSOLE
END
```

FILEREQUEST function

Opens a standard file dialog and returns a string containing the fully qualified pathname to the file. If type equals 1 then an 'Open' dialog is used. If type equals 0 then a 'Save As' dialog is opened. The syntax of FILEREQUEST is:

Name$ = FILEREQUEST (prompt, parent, type {, filter} {,ext} {,flags} {,Initial Directory})

The optional filter variable limits the dialog to showing only certain file types. The string consists of ordered pairs separated by the '|' character and ending with two ||.

Example:

```
Filter$ = "Text files|*.txt|All Files|*.*||"
```

A default extension can be supplied and will be used if the user does not type in an extension. This is a string parameter and should not contain the '.'. To allow users to select multiple files supply @MULTISELECT for the flags parameter. The returned string will contain complete paths to all of the files selected separated with the '|' character. If only one file is selected it will be terminated with the '|' character. The following example shows how to extract the filenames from the returned string:

```
REM Define a buffer to hold the returned filenames.
DEF filenames[10000]:ISTRING
DEF filter,filetemp:STRING
DEF pos:int
filter = "All Files (*.*)|*.*|Text Files (*.txt)|*.txt||"

filenames = filerequest("Select Multiple Files",0,1,filter,"txt",@MULTISELECT, "C:\")
do
pos = instr(filenames,"|")
if(pos)
filetemp = left$(filenames,pos-1)
filenames = mid$(filenames,pos+1)
REM do something with the file in filetemp
endif
until pos = 0
```

Subroutines

Subroutines are sections of programs that need to be executed numerous times. While you could use GOTO to jump into and out of a section of code, it would be very hard to manage. Subroutines can be called from anywhere in the program and return execution from where they were called. You define the start of a subroutine with the SUB statement. When your want to execute your subroutine you use the GOSUB statement. When your subroutine is finished, it should execute the RETURN statement.

Example:

```
OPENCONSOLE
DEF name$:STRING
INPUT "Please enter your name ", name$
GOSUB printname
PRINT "Returned from the print name subroutine"
PRINT
GOSUB printname
PRINT "Press Any Key To Close"
DO:UNTIL INKEY$ <> ""
CLOSECONSOLE
END

SUB printname
PRINT "Hello ",name$
PRINT "Now in the printname subroutine"
PRINT
RETURN
```

While this is a very simple example, it shows the basic operation of GOSUB. While in the subroutine your program has access to any variables that were declared outside of the subroutine. You can define new variables for use in your subroutines called 'local' variables.

```
SUB mySubroutine
DEF myString:STRING
...
RETURN
```

The variable "myString" is created when the subroutine begins. The variable will only be accessible while your subroutine is running. As soon as the RETURN statement is executed the variable is destroyed and is no longer accessible to the program.

Shortcut Definition

Creative BASIC allows a shortcut to define a subroutine. Instead of using SUB you can just type the name of the subroutine followed by a colon ':' . In the above example the printname subroutine could have been written like this:

```
printname:
PRINT "Hello ",name$
PRINT "Now in the printname subroutine"
PRINT
RETURN
```

This shortcut allows easier conversions from other languages.

Declaring Functions

A function, by definition, accepts parameters as input and may or may not return a value on exit. *This is really a subroutine.* You can define your own functions in Creative BASIC by creating a special kind of subroutine. The first step is to declare the subroutine to Creative BASIC with the DECLARE statement. The syntax of declare is:

```
DECLARE {DLLname,}functionname{(param1, param2...)}{,DLLret}
```

DLLname is used in a special form of the DECLARE statement to call Windows library functions which will be covered in later

chapters. functionname is any name you chose as long as it is not any built in statement, function or command. The optional parameter list lets you pass information to your subroutine.

Example fragment:

```
DECLARE docircle(x:INT,y:INT,size:INT,colora:INT,colorb:INT)
...
docircle 100,100,25,color1,color2
...
SUB docircle(x,y,size,colora,colorb)
CIRCLE win,x,y,size,colora,colorb
RETURN
```

Unlike normal subroutines, you do not use the GOSUB command. To call a subroutine that has been declared you simply use it like any other statement. The declared subroutine becomes part of Creative BASIC's language.

Returning Values

Your subroutine can optionally return data with the RETURN statement. It can return any of the built-in variable types.

Example fragment:

```
DECLARE calculate(x:FLOAT,y:FLOAT)
OPENCONSOLE
PRINT calculate(25,100)
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END

SUB calculate(x,y)
DEF ret:FLOAT
ret = (x + y)/2
RETURN ret
```

Passing Variables

In the examples above variables are passed to a subroutine by value. This is to say a copy of the contents of the variable is provided to your subroutine. Numeric variables like INT, FLOAT, DOUBLE and WORD are always passed by value. Some variable types are passed by reference. A variable passed by reference allows the subroutine to modify the contents of the variable. The WINDOW variable type is always passed by reference.

Example:

```
DECLARE DrawText(w:WINDOW,s:STRING,x:INT,y:INT)
DEF win:WINDOW

window win,0,0,640,400,@SIZE|@MINBOX|@MAXBOX|@CAPTION,0,"Demo",mainwindow

DrawText(win,"This is a test",10,10)

run = 1
do
wait
until run = 0
closewindow win
end

mainwindow:
select @class
case @idclosewindow
run=0
```

```
endselect
return

REM window variable passed by reference
SUB DrawText(w:window,s:STRING,x:INT,y:INT)
move w,x,y
print w,s
RETURN
```

Notice that the window variable is accessed in the subroutine by reference. It was declared as 'w' in the declare statement. When the subroutine is called 'w' is actually a reference to 'win' in the main program.

Passing Arrays

Arrays can be passed to subroutines by declaring the parameter with brackets []. Example:

```
DECLARE SomeFunction(numbers[]:INT)
DEF data[10]:INT
...
SomeFunction(data)
...
SUB SomeFunction(numbers[]:INT)
FOR x = 0 TO 9
numbers[x] = x * 10
NEXT x
RETURN
```

Arrays are passed by reference and can be directly modified by the subroutine.

The following table list Creative BASIC's variable types and whether they are passed by value or reference

TYPE	PASSED BY (SUBROUTINE)	PASSED BY (DLL CALL)
INT/UINT	VALUE	VALUE*
FLOAT	VALUE	VALUE*
DOUBLE	VALUE	VALUE*
STRING	REFERENCE	REFERENCE
ISTRING	REFERENCE	REFERENCE
CHAR	VALUE	VALUE*
WINDOW	REFERENCE	VALUE of HWND
DIALOG	REFERENCE	VALUE of HWND
FILE	REFERENCE	N/A
BFILE	REFERENCE	N/A
WORD	VALUE	VALUE*
Arrays []	REFERENCE	REFERENCE
POINTER	REFERENCE	REFERENCE
USER	REFERENCE	REFERENCE

*Numeric types may be passed by reference to DLL's if declared as type POINTER in the DECLARE statement

String and Character functions

APPEND$ function

Concatenates all the strings in the parameter list and returns the total string. This is similar to using the '+' operator on strings. APPEND$ is faster and can be used if speed is critical. The syntax of APPEND$ is:

Result$ = APPEND$ (string1, string2 {,stringn …})

ASC function

The ASC function returns the ASCII value of the character parameter. A string may be specified as the parameter in which case the first character of the string is used. The syntax of the ASC function is:

Value = ASC(char | string)

CHR$ function

Returns the character represented by the ASCII parameter. This is the opposite of the ASC function. This function is useful for creating characters that can not be normally typed on a keyboard. The syntax of the CHR$ function is:

Character = CHR$(value)

Example print a <RETURN> to the console:

PRINT CHR$(13)

DATE$ function

Returns the current date as a string in the format DD-MM-YYYY. The syntax of the DATE$ function is:

Result$ = DATE$

DATE$(format) function

Returns the current date as a string formatted by a specifier string. The specifier string is comprised of:

d	Day of month as digits with no leading zero for single-digit days.
dd	Day of month as digits with leading zero for single-digit days.
ddd	Day of week as a three-letter abbreviation..
dddd	Day of week as its full name.
M	Month as digits with no leading zero for single-digit months.
MM	Month as digits with leading zero for single-digit months.
MMM	Month as a three-letter abbreviation.
MMMM	Month as its full name.
y	Year as last two digits, but with no leading zero for years less than 10.
yy	Year as last two digits, but with leading zero for years less than 10.
yyyy	Year represented by full four digits.

For example, to get the date string

"Wed, Aug 31 94"

use the following input string:

"ddd',' MMM dd yy"

Example:

PRINT DATE$("ddd',' MMM dd yy")

Note the single quotes used to insert text into the output string.

HEX$ function

Converts the numeric parameter to a string representing the hexadecimal notation of that number. The syntax of HEX$ is:

Result$ = HEX$(value)

Example:

PRINT HEX$(255)

Would print FF to the console window.

INSTR function

Returns the position of the sub string 'string2' in 'string1'. Or returns 0 if string2 is not in string1. Optional start variable specifies a starting point in string1 to begin searching. position is 1 based. The syntax of the INSTR function is:

Position = INSTR({start, } string1, string2)

LCASE$ function

Returns the string parameter converted to all lowercase letters. The syntax of the LCASE$ function is:

Result$ = LCASE$(string)

LEFT$ function

Extracts count number of characters from the string starting from the leftmost character. Returns the resulting string. The syntax of LEFT$ is:

Result$ = LEFT$ (string, count)

LEN function

The LEN function works with any variable type in Creative BASIC. For strings it returns the number of characters in the string. For files it returns the length of the open file. The syntax of LEN is:

Length = LEN (variable | file)

LTRIM$ function

Removes all leading white-space characters from a string. White-space characters include spaces and tabs. LTRIM$ returns the result as a string leaving the parameter unchanged. The syntax of LTRIM$ is:

Result$ = LTRIM$(string)

MID$ function

Extracts count number of characters staring at position from a string. If count is omitted all of the characters from position to the end of the string are returned. The syntax of MID$ is:

Result$ = MID$ (string, position {,count})

RIGHT$ function

Returns count characters starting at end of the string. The syntax of the RIGHT$ function is:

Result$ = RIGHT$ (string, count)

RTRIM$ function

Removes any trailing white-space characters from a string. White-space characters include spaces and tabs. RTRIM$ returns the

result as a string leaving the parameter unchanged. The syntax of RTRIM$ is:

Result$ = RTRIM$ (string)

SPACE$ function

Returns a string filled with n spaces. The syntax of the SPACE$ function is:

Result$ = SPACE$(n)

STR$ function

The STR$ function returns the string representation of a number. The syntax of STR$ is:

Result$ = STR$(number)

STRING$ function

The STRING$ function returns a string filled with count number of the character specified. The syntax of the STRING$ function is:

Result$ = STRING$ (count, character)

TIME$ function

Returns the current system time in the format HH:MM:SS as a string. The syntax of TIME$ is:

Result$ = TIME$

UCASE$ function

Returns the string parameter converted to all uppercase letters. The syntax of the UCASE$ function is:

Result$ = UCASE$(string)

VAL function

Returns the value of a string representation of a number. This function is the opposite of the STR$ function. The syntax of the VAL function is:

Result = VAL(string)

REPLACE$ statement

The REPLACE$ statement replaces characters in one string with one or more characters from another. The syntax of the REPLACE$ statement is:

REPLACE$ dest, start, count, source

Dest is the string being modifies and source is the string where the characters are extracted from. Start and count must be greater than 0.
Example:

```
DEF s:string
s = "All good DOGs go to heaven"
REPLACE$ s,10,3,"dog"
PRINT s
```

Would print "All good dogs go to heaven"

Formatting output

While its easy enough to use PRINT and SETPRECISION to control how numbers are displayed in a window, or console. Sometimes more advanced formatting is necessary. The USING function returns a string that is formatted based on a specifier string and one or more parameters. The syntax of the USING function is:

USING(formatstring, param1 {,param2...})

The format string is a string literal or variable that can contain special formatting symbols as well as regular text to be inserted into the final output string. Creative BASIC understands the following format specifiers:

Symbol	Meaning
#	Reserves a place for one digit
Point (.)	Determines decimal point locations
Minus (-)	Left justifies within field. Default is right.
0	Prints leading zeros instead of spaces. Ignored if used with left justification.
Comma (,)	Prints a comma before every third digit to the left of the decimal point and reserves a place for one digit or digit separator.
&	Copies the string parameter directly

An example format string would look like "#,###.##" which would reserve two places to the right of the decimal place and four to the left. The result string would also have comma's inserted between every third and forth digit. The output can be used as a parameter to any function that accepts a string, assigned to a string variable or used with the PRINT statement

Example:

```
PRINT USING("$#,###.##", 1145.551)
A$ = USING("$######.##", 22.8)
PRINT A$
```

Would produce the output of:

```
$1,145.56
$ 22.80
```

Note that the '$' is copied to the result string directly. The comma does not need to be placed correctly in the format string. It just needs to be somewhere in the definition.

```
PRINT USING("#,######", 1234567)
```

Produces the output of:

```
1,234,567
```

Truncation and rounding

Creative BASIC will not truncate the output if the number of digits exceeds the number of # symbols on the left of the decimal point. Make sure you have enough # symbols to accommodate the output width. The right side of the decimal point is always rounded and truncated to match the format definition.

Example:

```
PRINT USING("##.###", 5115.1234)
```

Produces the output of:

```
5115.123
```

Filling with spaces or 0

The left side of the decimal point determines how many spaces or 0's to use as a fill value if there are not enough digits to fill the field.

```
PRINT USING("0######", 23)
PRINT USING("######",23)
```

Produces the output of:

```
000023
23
```

Justification

To left justify the output in the field use a minus sign (-) as a leading character.

```
PRINT USING("-#### -####", 55, 88)
```

Produces the output of:

```
55 88
```

Including string variables

To copy the contents of a string literal or variable use the & symbol in your definition

```
PRINT USING("&$#,###.##&", "The total cost is ", 3000.55, " Dollars")
```

Produces the output of:

```
The total cost is $3,000.55 Dollars
```

Using the pointer type

A pointer is a special type of variable that can reference, or point to, another variable. Many DLL functions require the use of a pointer type so the function may modify the *contents* of a variable. You can use pointers in your programs in the same manner.

Defining a pointer

A pointer variable is defined like any other variable in Creative BASIC, with the DEF statement.

```
DEF ptr:POINTER
```

After the pointer is defined it can reference, or point to, any variable by using the '=' symbol:

```
DEF i:INT

ptr = i
```

After the statement is executed the variable 'ptr' will be a reference to the variable 'i'. If a pointer is not referencing any variable it is said to be a NULL pointer. Referencing a variable means the pointer has an indirect route to the contents, or value, of the variable.

De-referencing a pointer

To get to the contents of the variable that a pointer is referencing it is necessary to use the '#' operator. This is called de-referencing a pointer. Think of de-referencing a pointer as asking a friend for money. Your friend has the money and you must go through him to get to it. In this way you indirectly have the money. De-referencing is sometimes called 'indirection'.

Example:

```
OPENCONSOLE
DEF ptr:POINTER
DEF i:INT

i = 55
ptr = i

PRINT #ptr

DO:UNTIL INKEY$<> ""
CLOSECONSOLE
```

This short example prints the number 55 to the console window. The # symbols tells Creative BASIC to get the contents of the variable that 'ptr' is referencing, In this case the pointer was referencing 'i' whose value was 55.

De-referencing also works in reverse. If you wanted to change the value of a variable using a pointer to that variable then de-reference the pointer on the left side of an equation

```
#ptr = 2
```

Changes the value of the variable that 'ptr' is referencing to 2.

Using pointers with subroutines

One of the advantages to using a pointer is having a generic variable type. Suppose you wanted to create a subroutine that could either add two numbers or combine two strings together. With pointers it is possible to pass variables of any type.

Example:

```
OPENCONSOLE
DECLARE MySub(var1:POINTER,var2:POINTER)
DEF a,b:INT
DEF str1,str2:STRING

a = 5
```

```
b = 6
str1 = "This is a "
str2 = "String"

PRINT MySub(a,b)
PRINT MySub(str1,str2)

DO: UNTIL INKEY$ <> ""
CLOSECONSOLE
END

SUB MySub(var1:POINTER.var2:POINTER)
RETURN #var1 + #var2
```

The example uses a very simple subroutine. It returns the de-referenced contents of the two variables added together. By using pointer types it was not necessary to know what type the variables were. Creative BASIC automatically references the variables when they were passed to the subroutine. This is because the DECLARE statement defined the parameters as POINTER types/

The TYPEOF function

Sometimes it is necessary to determine the type of a variable a pointer is referencing. The TYPEOF function returns a numeric identifier depeninding on the type. If a pointer is not currently referencing a variable (NULL pointer) TYPEOF returns -1. Syntax:

```
id = TYPEOF(pointer)
```

The id returned will be one of the following:

```
@TYPESTRING
@TYPECHAR
@TYPEINT
@TYPEFLOAT
@TYPEMEMORY
@TYPEFILE
@TYPEWINDOW
@TYPEDIALOG
@TYPEBFILE
@TYPEWORD
@TYPEDOUBLE
@TYPEUSER
@TYPEPOINTER
@TYPEUINT
@TYPEUINT64
@TYPEINT64
```

The ISARRAY function

ISARRAY returns 1 if the variable referenced by a pointer is an array. ISARRAY can also be used with variables types other than POINTER. Since a string is actually an array of characters, ISARRAY always returns 1 for a string variable. Syntax:

```
ISARRAY(pointer | variable)
```

The ISARRAY function is useful if your subroutine can handle a pointer either a single variable or an array.

Advanced pointer de-referencing

When de-referencing a pointer to an array use the [] the same as you would for the array variable itself.

```
DEF i[10]:INT
DEF ptr:POINTER

ptr = i

#ptr[0] = 5
```

For user defined types use the dot operator in the same manner you would with the referenced variable

```
TYPE mytype
DEF i:INT
DEF g:FLOAT
ENDTYPE

DEF mt:mytype
DEF ptr:POINTER
ptr = mt

#ptr.g = 3.145
```

A user type may contain a pointer to another type. You can use multiple de-referencing operators to access the nested data. Example:

```
TYPE salary
DEF weekly:FLOAT
DEF hourly:FLOAT
ENDTYPE

TYPE employee
DEF wage:POINTER
DEF name:STRING
ENDTYPE

def emp:employee
def sal:salary
def pemp:POINTER

emp.wage = sal

pemp = emp
#pemp.#wage.hourly = 21.50
```

While not really useful in this example, multiple de-referencing is a powerful tool and will be discussed further in the section on dynamic variables.

Arrays of pointers

While Creative BASIC does not support using arrays of pointers directly, an array of pointers can be passed to a DLL function. If you need an array of unknown variables use a user type that contains one pointer type. Example:

```
TYPE vararg
DEF arg:POINTER
ENDTYPE

DEF args[100]:vararg
DEF pargs:POINTER
DEF x:INT
DEF y:FLOAT
DEF z:STRING

pargs = args

x = 1
y = 2.15
z = "Hello There!"

#pargs[0].arg = x
#pargs[1].arg = y
#pargs[2].arg = z
```

Creating dynamic variables

The NEW function

Creative BASIC supports two methods for creating variables. The first you are already familiar with is the DEF statement. The DEF statement creates the variable in a hard coded way that doesn't allow the flexibility needed for advanced programming structures like linked lists. The second method creates unnamed variables at run time and returns a pointer to that variable. To create the variable define a pointer and use the NEW function to create the variable in memory. The syntax of new is:

pointer = NEW(type,length)

Type can be any built in or user defined type. Length must be 1 or greater. If length is greater than one then an array is created. NEW supports creating single dimensioned arrays only.

The DELETE statement

When you are finished using the variable you must use the DELETE statement to free the memory used by the variable. Failure to use DELETE will result in memory leaks. The syntax of DELETE is:

DELETE pointer.

Example usage:

```
OPENCONSOLE
DEF p:POINTER
'Create an INT in memory
p = NEW(INT,1)
#p = 75
PRINT #p
'delete the INT
DELETE p
PRINT "Press any key to close"
DO: UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

The real power in dynamic variables is with user defined types. Using dynamic variables allows creation of lists of data only limited to the available memory in the computer, without needing to know the size of the list first. To demonstrate this concept lets define a couple of user data types. The entire program, linkedlist.cba, can be found in the samples directory.

```
'A node consists of two pointers
'one for the previous element
'and one for the next
TYPE node
def nxt:POINTER
def prev:POINTER
ENDTYPE

'our list has a node so we can
'create a linked list
TYPE list
def node:node
def data:string
def number:int
ENDTYPE
```

The major difference from hard coded user types is the use of a node type. The node type consists of two pointers. These pointers will reference the next and previous elements of out list. To create the elements we will use two subroutines. One to add to the beginning or 'head' of the list and one to add to the end or 'tail' of the list

```
declare addtail(plist:pointer,data:string,number:int)
declare addhead(plist:pointer,data:string,number:int)
```

```
sub addtail(plist:pointer,data:string,number:int)
if(plist = 0)
'if our list is empty just create
'a new one
plist = new(list,1)
else
'find the last element
WHILE #plist.node.nxt
plist = #plist.node.nxt
ENDWHILE
'create a new list variable
#plist.node.nxt = new(list,1)
'point the new lists prev variable
'to the current last element
'this syntax uses multiple indirection
'both plist and nxt are pointers
#plist.node.#nxt.node.prev = plist
plist = #plist.node.nxt
endif
#plist.data = data
#plist.number = number
return plist

sub addhead(plist:pointer,data:string,number:int)
if(plist = 0)
'if our list is empty just create a new one
plist = new(list,1)
else
'create a new list variable and put it first in line
#plist.node.prev = new(list,1)
'make our new head point to the old head element
'this syntax uses multiple indirection
'both plist and prev are pointers
#plist.node.#prev.node.nxt = plist
plist = #plist.node.prev
endif
#plist.data = data
#plist.number = number
return plist
```

Once our subroutine is declared we can begin creating list members by using either one. The subroutines are designed to create the first element automatically if the list is empty:

```
DEF p:POINTER

p = addtail(p,"entry0",12)
addtail(p,"entry1",77)
addtail(p,"entry2",55)
addtail(p,"entry3",108)
addtail(p,"tail",2)
p = addhead(p,"head",99)
```

Notice that when we used addhead that we set the pointer to the new head element of the list. When we are done with the list we must clean up by deleting each element. A subroutine takes care of this nicely:

```
DECLARE deleteall(plist:pointer)

deleteall(p)

sub deleteall(plist:pointer)
def temp:pointer
'iterate through the list
'until the end
```

```
'deleting elements as we go.
while #plist.node.nxt
temp = plist
plist = #plist.node.nxt
delete temp
endwhile
delete plist
return
```

The complete program, included with the samples, also includes a subroutine for finding an element by name and demonstrates accessing the elements by printing the contents to the console. Instead of printing, you could use the same technique to save all of the list elements to a file.

Using Memory

Allocating memory

Before memory can be read or written to if first needs to be allocated. Allocating tells the system that a certain amount of memory will be used by the program and should not be written to by any other programs. Allocate memory with the ALLOCMEM function. The syntax of ALLOCMEM is:

error = ALLOCMEM (variable, count, size)

Variable must be defined as type MEMORY with the DEF statement. Count is the number of items this memory block can hold and size is the size of the items. The LEN statement can be used to determine the size of any variable type. If variable was already in use then ALLOCMEM will re-allocate the memory to be the size and count specified. ALLOCMEM returns -1 if an error occurs.

Freeing Memory

After your program is done using the memory it needs to be freed. The FREEMEM statement returns a previously allocated block of memory to the system. Once memory is freed it cannot be read or written to again unless reallocated witht ALLOCMEM function. The syntac of the FREEMEM statement is:

FREEMEM variable

Variable must have been successfully defined as type MEMORY and allocated with the ALLOCMEM function.

Reading and Writing Memory

After allocating memory your program can store any built-in or user defined variable using the WRITEMEM statement. The syntax of WRITEMEM is:

WRITEMEM memory, record, variable {,offset}

Record must be greater than 0 and memory must have been successfully allocated with the ALLOCMEM function. This statement is similar to the PUT statement for files. Reading memory is done with the READMEM statement. The syntax of READMEM is:

READMEM memory, record, variable {,offset}

Record must be greater than 0 and memory must have been successfully allocated with the ALLOCMEM function. This statement is similar to the GET statement for files.

Optional offset value allows specify the position where reading and writing will occur. Amount is in bytes. This is useful for creating a 'header' at the beginning of the memory block.

Example:

```
REM example of memory functions
OPENCONSOLE
DEF num:INT
DEF buffer:MEMORY
DEF buffer2:MEMORY
TYPE mytype
DEF name[40]:ISTRING
DEF age:INT
DEF money:FLOAT
ENDTYPE
DEF record:mytype
ALLOCMEM buffer,100,LEN(num)
ALLOCMEM buffer2,10,LEN(record)

FOR x = 1 to 100
WRITEMEM buffer,x,x
```

```
NEXT x

FOR x = 1 to 100
READMEM buffer,x,num
PRINT num
NEXT x

record.name = "John Doe"
record.age = 32
record.money = 100.01

WRITEMEM buffer2,1,record
READMEM buffer2,1,record
PRINT record.name
PRINT record.age
PRINT record.money
FREEMEM buffer
FREEMEM buffer2

PRINT "Press any key to continue"
DO
UNTIL INKEY$ <> ""
CLOSECONSOLE
END
```

Calling external DLL functions.

Creative BASIC has the ability to call an external DLL to further expand the capabilities of your program. DLL stands for Dynamic Link Library and can contain hundreds of functions for your use. Many DLL's are included with the Windows™ operating system and more are available on the Internet.

To use a DLL you first need to define it to Creative BASIC using the DECLARE statement. DECLARE was discussed in an earlier chapter. The syntax of DECLARE when using a DLL is:

DECLARE "[!]DLL name", function({parameter list}) {, return type}
DECLARE "[!]DLL name", localname ALIAS function({parameter list}) {,return type}

If the DLL is in the windows system directory then a path is not required. For DLL's elsewhere you should use a fully qualified path name. Once the DLL is declared you can use it in your program by simply treating it as any other function or statement. The second form of the syntax allows using a different name for the function to avoid naming conflicts with other DLL's or to declare the same function with different parameters.

The optional ! symbol specifies that the DLL function uses the CDECL calling convention. For most of the Windows API this is unnecessary with the exception of a few functions. wsprintfA is one of the functions that requires the cdecl calling convention.

Example declaration:

DECLARE "User32",MessageBoxA (wnd:window,text:string,title:string,flags:int),int

Declares a function called MessageBoxA in the DLL "User32" which is located in the system directory. The function takes a window, two strings and an integer as parameters and returns an integer result.

Example usage:

Result = MessageBoxA(w,"This is a message","Press OK",0)

Documentation for Windows™ DLL's can be found in any good programmers reference. Commercially available DLL's include documentation that outlines all of the functions available. In the documentation, you may see some variable types that do not match

any of Creative BASIC's built-in types. The following table lists some common substitutes.

Requested type	Equivalent Creative BASIC type
HWND	WINDOW, DIALOG or INT
LPCSTR/LPSTR	STRING, ISTRING
BYTE / UBYTE / TCHAR	CHAR
LONG	INT
WORD	WORD
DWORD	INT
UINT	INT
BOOL	INT
WPARAM	INT
LPARAM	INT
LP*	POINTER

Passing user types

Many DLL functions will require passing a structure, or user type, to the function. Creative BASIC's TYPE statement can be used to create the needed structure to be passed or modified by a function. An example of this is the GetLocalTime function located in kernel32.dll. The function expects a variable of type SYSTEMTIME. The type can be specified as follows:

```
TYPE SYSTEMTIME
DEF wYear:WORD
DEF wMonth:WORD
DEF wDayOfWeek:WORD
DEF wDay:WORD
DEF wHour:WORD
DEF wMinute:WORD
DEF wSecond:WORD
DEF wMilliseconds:WORD
ENDTYPE
```

Once the type is defined you can create a variable that can be used with the function

Example:

```
DECLARE "kernel32",GetLocalTime(tm:SYSTEMTIME)
DEF tm:SYSTEMTIME

GetLocalTime(tm)

PRINT tm.wYear

...
```

The DLL function may also require a certain packing to be used when passing the user type. Packing represents how a structure is stored in memory and can be specified in the TYPE statement as an optional parameter.

Example:

```
TYPE MYTYPE,2
DEF letter:CHAR
DEF num:INT
DEF small:WORD
ENDTYPE
```

Would convert the type to a structure with 2 byte packing when passed to the DLL function. If a packing number is not supplied you can generally omit the number in which case Creative BASIC defaults to a pack value of 8. Note that the packing number has no effect on user types used within your program and only effects DLL calls.

The POINTER type

Certain DLL functions might require a parameter of a pointer. This allows the DLL function to modify the contents of a variable that was declared in Creative BASIC. Normally this is done with numeric variable types. An example of this would be the GetComputerName function located in kernel32.dll. The GetComputerName function expects a string variable and an integer variable which can be modified by the function. A pointer variable can 'point' to any of the built in variable types and then be passed to a function

Example:

```
DEF pSize:POINTER
DEF nSize:INT
DEF name:STRING
DECLARE "kernel32",GetComputerNameA(buffer:STRING,size:POINTER),int

nSize = 255
pSize = nSize
```

```
GetComputerNameA(name,pSize)
PRINT name
PRINT nSize
...
```

On return the function will have stored the computers name in 'name' and the number of characters in the name in 'nSize'. The function could have also been called with nSize directly and is internally converted to a pointer when the function is called. This happens since the function was declared as requiring type POINTER. We did not have to define a pointer for the string variable as all strings are passed by reference in Creative BASIC which allows the function to modify the contents of the string.

See the sample programs structuredemo.cba and pointerdemo.cba for more examples of using pointers and user types with DLL functions.

Passing Arrays

Arrays can be passed to DLL's by declaring the parameter with brackets []. Example:

```
DECLARE "mydll",SomeFunction(numbers[]:INT),INT
DEF data[10]:INT
...
SomeFunction(data)
```

Arrays are passed by reference and can be directly modified by the DLL.

Unloading a DLL

If your program is only going to use a DLL for a short time you can unload the DLL from memory with the FREELIB statement. Once a DLL is unloaded a call to a previously declared function will load the DLL again. The syntax of FREELIB is:

```
FREELIB name
```

It is not necessary to unload DLL's that are part of the system such as kernel32.dll or user32.dll since those libraries are always loaded when Windows start. A good example of when to use FREELIB would be a setup program that uses a DLL to create icons and then unloads it when finished. Another use would be a temporary DLL that you extract from resources. The DLL cannot be deleted until all references to it are freed.

All DLLs your program uses are unloaded when your program ends.

Miscellaneous functions

Functions and statements not covered elsewhere in the manual

COLORREQUEST function

Opens the standard palette dialog to request a color. Returns the color selected or –1 if the user cancels the dialog. Optional color parameter specifies a color to highlight when the dialog is displayed. The syntax of COLORREQUEST is:

Color = COLORREQUEST (window {,color})

FILEREQUEST function

Opens a standard file dialog and returns a string containing the fully qualified pathname to the file. If type equals 1 then an 'Open' dialog is used. If type equals 0 then a 'Save As' dialog is opened. The syntax of FILEREQUEST is:

Name$ = FILEREQUEST (prompt, parent, type {, filter} {,ext} {,flags} {,initial dir})

The optional filter variable limits the dialog to showing only certain file types. The string consists of ordered pairs separated by the '|' character and ending with two ||.

Example:
Filter$ = "Text files|*.txt|All Files|*.*||"

A default extension can be supplied and will be used if the user does not type in an extension. This is a string parameter and should not contain the '.'. To allow users to select multiple files supply @MULTISELECT for the flags parameter. The returned string will contain complete paths to all of the files selected separated with the '|' character. If only one file is selected it will be terminated with the '|' character. The following example shows how to extract the filenames from the returned string:

```
REM Define a buffer to hold the returned filenames.
DEF filenames[10000]:ISTRING
DEF filter,filetemp:STRING
DEF pos:int
filter = "All Files (*.*)|*.*|Text Files (*.txt)|*.txt||"

filenames = filerequest("Select Multiple Files",0,1,filter,"txt",@MULTISELECT, "C:\")
do
pos = instr(filenames,"|")
if(pos)
filetemp = left$(filenames,pos-1)
filenames = mid$(filenames,pos+1)
REM do something with the file in filetemp
endif
until pos = 0
```

GETKEYSTATE function

Returns the asynchronous state of the key specified. This will tell you if a key on the keyboard is being pressed when this function is executed without having to wait for a message in a window. The syntax of the GETKEYSTATE function is:

state = GETKEYSTATE (keycode)

The return value is greater than 0 if a key is pressed. The key code is the virtual keyboard code and does not always correspond to the ASCII value of the key. See Appendix C for a list of key codes.

NOTE: GETKEYSTATE will not work for a console only program under Windows 95/98 or ME.

GETSTARTPATH function

Returns the path to your program or executable. When your program is run from the Creative BASIC editor, this will return the path your source file is located in. When your program is in executable form, this will return the path to the executable. This provides a convenient way of specifying relative paths to data files. The path returned automatically has a "\" appended to it.

Example:
start$ = GETSTARTPATH

datafile$ = start$ + "data\mystuff.dat"

SETID statement

Provides a method for defining user constants that can be used in the same manner as system constants. The identifier must be unique and not conflict with any defined system constant. The syntax of the SETID statement is:

SETID identifier, value

Example usage fragment:
SETID "ISDRAWING",10
SETID "ISPRINTING",20
IF state = @ISDRAWING THEN GOSUB drawme
IF state = @ISPRINTING THEN GOSUB printme
...

SYSTEM statement

Runs another executable. Name must be a fully qualified path name or the executable must exist in the windows or system directory. Syntax of the SYSTEM statement is:

SYSTEM name {, Parameters}

Example:
SYSTEM "notepad.exe", "c:\windows\setuplog.txt"

TYPE statement

The TYPE statement begins defining a user variable type. The new type will contain all of the variables between TYPE and ENDTYPE. Accessing the variables in a user type is done with the dot operator `.'. Any variable type can be defined between TYPE and ENDTYPE. Syntax of TYPE:

TYPE name {,pack}
DEF ...
{DEF ...}
ENDTYPE

Once a user variable type is defined you can create variables of that type using the DEF statement.

Example fragment:

TYPE phonerecord
DEF Name:STRING
DEF Age:INT
DEF Phone[20]:ISTRING
ENDTYPE

DEF Rec:phonerecord

Rec.Name = "Joe Smith"
Rec.Age = 35
Rec.Phone = "555-555-1212"
...

Typed variables provide a convenient way of accessing groups of related information. The optional pack parameter specifies how this variable will be sent to DLL functions.

Printing

The PRINTWINDOW command

The PRINTWINDOW command takes a window as a parameter and sends the contents of the window to the printer. The output is properly scaled to fit the page in either portrait or landscape mode. PRINTWINDOW opens the standard printer dialog to allow the user to select printing options. The syntax of PRINTWINDOW is:

PRINTWINDOW window

PRINTWINDOW is good for obtaining snapshots of a window. You do not need to use OPENPRINTER before using PRINTWINDOW.

Opening a printer

In order to send data to a printer you must first open it. The OPENPRINTER function returns an integer handle to the printer specified or 0 if the printer could not be opened. The syntax of OPENPRINTER is:

handle = OPENPRINTER (name, title, mode)

Name is a string returned by GETDEFAULTPRINTER or the PRINTERDIALOG function. Title is your documents title which is used by the print spooler in windows and mode can be either "TEXT" or "RAW". TEXT mode will be the most commonly used option.

Sending data to the printer

After a printer is successfully opened you can send data to the printer with the WRITEPRINTER statement. WRITEPRINTER is very similar to the PRINT statement and can accept the same data types as parameters. The syntax of WRITEPRINTER is:

WRITEPRINTER handle, exp {,exp}...

Starting a new page

To eject the current page and start a new one use the ENDPAGE statement. Syntax:

ENDPAGE handle

Closing the printer

After you are finished using the printer use the CLOSEPRINTER statement. CLOSEPRINTER finishes the current page, closes the document and frees any memory used by the system. You cannot use the handle for the printer again unless you reopen the printer with OPENPRINTER. The syntax of CLOSEPRINTER is:

CLOSEPRINTER handle

Getting the default printer

To obtain the name of the current default printer, indicated by a checkmark in the printer control panel, use the GETDEFAULTPRINTER function. If there is no default printer the function returns an empty string. The syntax of GETDEFAULTPRINTER is:

name = GETDEFAULTPRINTER

Opening the printer dialog

Opening the printer dialog allows the user to select the printer of their choice, pages to print, number of copies and whether you need to collate the copies. The syntax of PRTDIALOG is:

name = PRTDIALOG (window, from, to, copies, collate)

The function returns the name of the selected printer and modifies the integer variables from, to, copies and collate to reflect the user's choices. Before using PRTDIALOG set the 'from' and 'to' variables to indicate the number of pages available to print. If you set 'from' and 'to' to the same number then only the 'all pages' radio button will be available. 'window' can be any open window or 0 if your program has no windows or only consists of a dialog.

Printing Example:

```
DEF hPrt:INT
DEF data:STRING
```

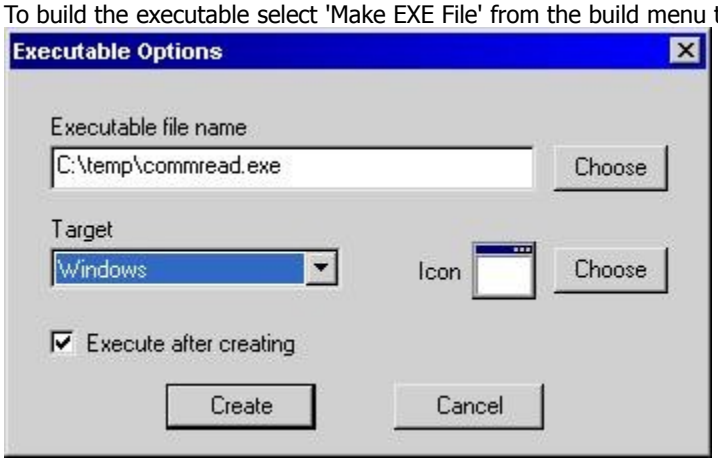
```
DEF name:STRING
DEF pagefrom,pageto,copies,collate:INT
pagefrom = 1
pageto = 1
copies = 1
name = PRTDIALOG(0,pagefrom,pageto,copies,collate)
hPrt = OPENPRINTER(name,"Test Document","TEXT")
IF (hPrt)
data = "This is a test of printing"
data = data + chr$(13)
data = data + "This is line 2"
WRITEPRINTER hPrt,data
CLOSEPRINTER hPrt
ENDIF

END
```

Note: Not all printers support printing text directly. If your printer is referred to as a 'GDI only' printer then you may have trouble using WRITEPRINTER. Consult you printer documentation for more information.

Compiling your program

Once your program is working to your specification you can build an executable that will run on any 32bit Windows system. The executables do not require any runtimes to be used and can be built with or without DirectX.

To build the executable select 'Make EXE File' from the build menu to open the Executable options dialog:



The Executable file name is the file Creative BASIC will create. Select an appropriate directory with the Choose button. The icon selected will be used for both the programs icon displayed in explorer as well as the icon used in any windows opened by your program.

The Target specifies what kind of code your program uses. If you using DirectX or Direct3D in your programs then you must select "Windows DirectX/3D" in order for your executable to correctly run. Select "Console Only" if you program only uses the console for input and output. For all other programs select "Windows".

Select the 'Execute after creating' option to run the executable immediately after compiling.

Once all of the options are selected press the 'Create' button and your executable will be built.

NOTES

The "Console Only" target links your code with a console startup routine and allows for creating command line utilities. Because the code uses a console startup instead of a normal Windows startup, your program cannot use any of the window, dialog or control creation and manipulation functions built into to Creative BASIC. Also unavailable to the console only program are the FILEREQUEST, COLORREQUEST and printer dialogs. Your program may still access the Windows API and external DLL's for additional functionality when compiled with "Console Only".

If you need to use the FILEREQUEST, COLORREQUEST and the printer functions with a console program then select the "Windows" target.

Debugging your program

When writing software in any language it is inevitable that a bug or error will happen. Finding the bug would be difficult if you could not stop the program at certain points and verify the contents of variables. The STOP command was designed for this purpose. Insert a STOP command anywhere in your program that you wish to examine. Once STOP is executed you will be returned to the editor and the debugging options of the Build menu will be available.

Showing the variable dialog

After your program is stopped select 'Show Variables' from the build menu. A dialog will open displaying all of the global and locally defined variables in the program along with the variables type, size and contents. See below for an example. For strings, array's and user types a '+' sign will appear next to the variable name. Clicking on the '+' sign will expand the variable to show the contents of each element.

Single stepping

When a program is stopped it is sometimes useful to single step through the program to watch the contents of variables change. Open the variable dialog as explained above, then select 'Single Step' from the build menu. The cursor will advance to the line that was just executed and the variable display dialog will be updated to show the changes to the contents of any variables. You can also use F5 to single step as well. Before using F5 make sure the window that contains the stopped program is selected.

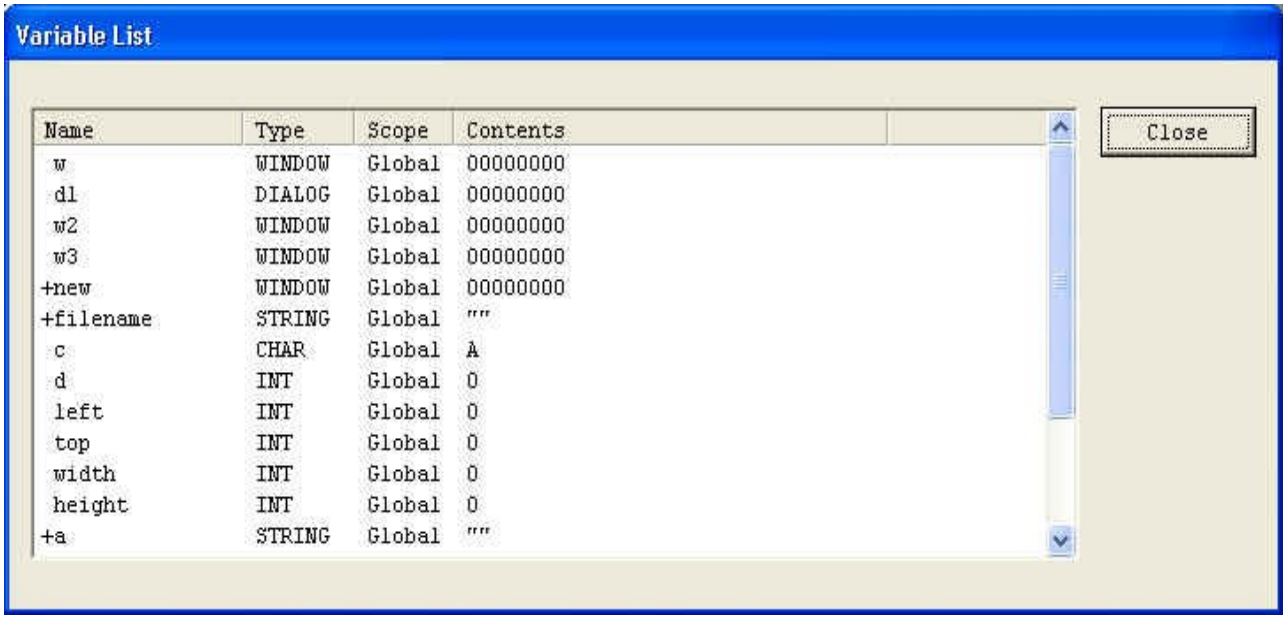
Continuing execution

After you have examined the contents of variables you can continue execution at the next line after the STOP statement, or the next line to be executed if you were single stepping, by selecting 'Continue' from the build menu. You can also continue execution by pressing F7. Continuing from within a subroutine will end the program at the first RETURN statement.

The recovery file

Every time a program is run from the editor, Creative BASIC saves a special recovery file in the main Creative BASIC directory. The recovery file is named 'recover.cba' and will protect your work in the event the program crashes Creative BASIC. After restarting Creative BASIC select Open from the main menu to load the file.

Figure 1 The variable display dialog



Components

Components are precompiled Creative BASIC subroutines than can be included into other projects, this allows reusing code many times without having to cut and paste. Components are written like any other Creative BASIC program with the exception that they cannot contain any global variables or definitions. A component is similar to statically linked libraries in other languages such as 'C'. The subroutines in a component can contain DECLARE statements to predefine API calls.

Creating a component

To create a component out of the current source file select 'Component' from the main menu and then select 'Create'. The Create Component dialog will open. Select a filename for the component and enter a description. The description should include usage instructions and any declarations needed to use the component. Components are saved with the extension ".cbo".

Including a component

To include a component into the current source file select 'Component' from the main menu and then select 'Include'. The include component dialog will open. Select a component to include and select 'OK'. The title of the current source file will be modified to show the number of components included. If you no longer wish to include a particular component, remove it form the list in the component dialog.

Once one or more components are included in a current file a new file is created in the same directory as the source file. This file has the same name as the source file with the extension ".cbl". If you rename your source file remember to rename the .cbl file as well.

Example component source:

```
REM This component declares the WinAPI functions needed to work with the registry
SUB INIT_REGFUNCTIONS
DECLARE "advapi32",RegConnectRegistryA(computer:STRING,hkey:INT,pkey:POINTER),INT
DECLARE "advapi32",RegOpenKeyExA(hkey:INT,subkey:STRING,options:INT,regsam:INT,pkey:POINTER),INT
DECLARE "advapi32",RegCloseKey(hkey:INT),int
DECLARE "advapi32",RegQueryValueExA(hkey:INT,name:STRING,reserved:INT,type:POINTER,data:POINTER,size:POINTER),INT
SETID "HKEY_CLASSES_ROOT", 0x80000000
SETID "HKEY_CURRENT_USER", 0x80000001
SETID "HKEY_LOCAL_MACHINE", 0x80000002
SETID "HKEY_USERS", 0x80000003
RETURN
```

Once the component is created you can include the component into as many other programs as needed without retyping the code. In the program that uses this particular component only one line is needed, a GOSUB INIT_REGFUNCTIONS before any of the declared functions are used.

Tips

If you find yourself using a subroutine over and over again it is a good candidate for a component. The subroutines in components can be as simple or as complex as needed, accept and return values or just set up API declares as above.

Save your component source file like any other Creative BASIC program. Once a component is created, it is compiled and cannot be turned back into the original source.

Test your components thoroughly. Once compiled, Creative BASIC will report any errors in the components but cannot know the line number of the error.

Windows

Creating a Window

Opening a window in Creative BASIC is easily done with the WINDOW command. When you use Creative BASIC to write Windows based software your program becomes *event driven*. Every action a user takes when running your program is sent to a special subroutine as a message. Your program must then decide whether or not to respond to this message. A message is really just a number. This number represents an action taken by the user.

The syntax of the WINDOW command is:

WINDOW variable, left, top, width, height, flags, parent, title, handler

Parameters:
variable - The name of the WINDOW variable used to store the window
left - The left edge of the window
top - The top edge of the window
width - The width of the window
height - The height of the window
flags - A numeric value specifying creation style flags
parent - a WINDOW variable if this is a child window or 0
title - The text shown in the caption of the window
handler - The name of a subroutine to handle messages for the window

The WINDOW command will return 1 if the window was created successfully or 0 if the window could not be created. You can also check the window variable to see if it equals 0. If so then the window could not be opened.

Example:

```
REM define a window variable
DEF w1:WINDOW
REM open the window
WINDOW w1,0,0,350,350,@MINBOX|@MAXBOX|@SIZE,0,"Simple Window",main
REM print a message
PRINT w1,"Hello World"
REM when w1 = 0 the window has been closed
WAITUNTIL w1 = 0
END
'---
REM every time there is a message for our window
REM the operating system will GOSUB here
SUB main
IF @CLASS = @IDCLOSEWINDOW
REM closes the window and sets w1 = 0
CLOSEWINDOW w1
ENDIF
RETURN
```

The example shows the steps necessary to open a window and wait for a message. If you examine the program, you will notice that there are no GOSUB statements. Windows will call your subroutine anytime there is a message for the window you created.

System variables and constants

The previous example also introduces the '@' symbol. Creative BASIC defines a number of constants for use in your program. Special variables contain information about the message sent to a window. In either case the '@' symbol is used to differentiate these names from variables you define in your program. System variables like @CLASS are set by Creative BASIC and cannot be used as normal variables. You can define your own constants with the SETID statement.

The appendix contains a list of all of the system variables and their meanings.

Creation style flags

In the above example we used @SIZE as the flags parameter. This tells Creative BASIC that the window should be resizable.

More than one flag can be used and combined with the '|' symbol (meaning OR). Creative BASIC contains many predefined flags:

NAME	PURPOSE/STYLE
@SIZE	Creates a window that is resizable
@MINBOX	The window has a minimize box
@MAXBOX	The window has a maximize box
@MINIMIZED	Creates a window that is initially minimized
@MAXIMIZED	Creates a window that is initially maximized
@CAPTION	Default. Creates a window with a caption
@NOCAPTION	Creates a window with no caption
@SYSMENU	Default. Creates a standard system menu
@BORDER	Creates a bordered window. Use with the @NOCAPTION flag
@HSCROLL	Window has a horizontal scroll bar
@VSCROLL	Window has a vertical scroll bar
@MDIFRAME	Creates a window that can contain other windows.
@USEDEFAULT	Child windows can use this for the 'left' parameter to let windows pick a default size for the window
@TOOLWINDOW	Creates a window with a half sized caption.
@NOAUTODRAW	Allows your program to handle @IDPAINT messages
@BROWSER	Creates a window with an embedded internet explorer window.

Waiting for messages

In the previous example, we introduced a few new ideas. The WAITUNTIL command tells Creative BASIC to process messages until a condition is true. The WAITUNTIL command *sleeps* until something is done with one of the windows your program is using. When a message is sent to your window, the handler subroutine is called that was defined when the window was opened. After your subroutine executes a RETURN statement, your program will continue waiting for messages. In our example, we check the value of the variable 'w1' to see of we should wait for more messages or end the program. The syntax of WAITUNTIL is:

WAITUNTIL condition

Creative BASIC also contains another statement for processing messages. The WAIT statement can be used when more control over message processing is desired. WAIT processes any messages that are available for your window, sleeps if none are available, and then returns. The syntax of WAIT is:

WAIT {NoSleep}

If the optional NoSleep parameter is set to 1 then the WAIT command will check for messages and return immediately. WAIT will be covered in more detail in the Messages and Message loops section

Handling Messages

When your windows handler subroutine is called, the system variable **@CLASS** will contain the message ID. This variable can then be compared against any of the messages you wish to handle using an IF or SELECT statement. Some messages contain additional information and set **@QUAL** and **@CODE** accordingly. A good example of this is information from the keyboard. When a key is pressed your window will receive, among others, the message @IDCHAR. In this case @CODE will contain the ASCII value of the key.

Example fragment:

```
SUB mywin
SELECT @CLASS
CASE @IDCHAR
Key = @CODE
CASE @IDCLOSEWINDOW
run = 0
ENDSELECT
RETURN
```

When developing your handler subroutine you can handle as many or as few messages as you need. Any messages that you do not respond to are simply 'thrown away' by the system. At a minimum you should check for @IDCLOSEWINDOW to handle the close button of the window.

Returning Values

If a particular message requires a return value you can specify it in the RETURN statement.

The next section will explore messages and the message loops in more detail.

See Also: System Variables and Constants in the appendix for a list of message IDs

Messages and message loops

Introduction

In the previous section we discussed how to create a window and introduced the concept of *messages.* Any action taken by a user that effects a window or dialog is reported to your program. This report is sent as a message that contains three main pieces of information. The message class or ID, the message code also known as the WPARAM, and the message qualifier also known as the LPARAM.

Windows sends the report to your program by calling, or GOSUBing, the subroutine you specified when the window or dialog was created. This subroutine is known as the "handler" subroutine. In other languages it may be referred to as the "windows procedure". Before your handler subroutine is called, Windows sets the system variables @CLASS, @CODE and @QUAL so your program can tell what has happened.

The message queue and loop

Since Windows is a multitasking system, and there may be literally hundreds of programs running on your computer at once, it would be very inefficient for Windows to wait for your program to finish handling a message. So all messages sent to your program are placed in a special buffer known as the message queue. The messages are placed in the queue in the order received to prevent your program from losing track of what has been happening to the window. The queue also prevents losing messages that may have been sent while your program was busy doing something else.

Because there may be many other programs running at the same time it is also not system friendly to "busy wait" for messages. When there are no messages left to process on the queue your program should sleep, waiting for the next message to be reported to the window, and placed in the message queue. In Creative BASIC the message queue, looping and sleeping are handled by using either the WAIT or WAITUNTIL statements.

The WAITUNTIL statement.

As discussed in the section on creating windows the WAITUNTIL statement will be used almost exclusively in your programs. To fully understand what is happening when you use the WAITUNTIL statement we will use some Creative BASIC statements to break it down. This code is just an example of what WAITUNTIL does and is not an actual program:

```
SUB WAITUNTIL(condition)
DO
GOSUB SLEEP: REM Wait until some messages are available
FOR x = 0 TO GetNumMessages() : REM get the number of messages in the queue
msg = GETMESSAGE(x)
@CLASS = msg.messageID
@CODE = msg.wparam
@QUAL = msg.lparam
IF @CLASS = @IDCONTROL
@CONTROLID = msg.control_id
ENDIF
IF @CLASS = @MENUPICK
@MENUNUM = msg.menu_id
ENDIF
GOSUB msg.window.handler : REM call the handler subroutine for this window or dialog
NEXT x
UNTIL condition
RETURN
```

As you can see by the pseudo code above there actually is a loop comprised of the DO and FOR statements.

The WAIT statement

The WAIT statement is very similar to WAITUNTIL with the exception that it only executes once. It is up to your program to determine how long messages should be processed and what condition(s) to check to exit the loop. WAIT should only be used in special circumstances where you need more control of when messages are processed. The WAIT statement has an optional parameter to specify not to sleep and to immediately check for messages and return. Using Creative BASIC statements again the code for WAIT would look something like this:

```
SUB WAIT(nosleep)
IF nosleep <> 1
```

```
GOSUB SLEEP: REM Wait until some messages are available
ENDIF
FOR x = 0 TO GetNumMessages() : REM get the number of messages in the queue
msg = GETMESSAGE(x)
@CLASS = msg.messageID
@CODE = msg.wparam
@QUAL = msg.lparam
IF @CLASS = @IDCONTROL
@CONTROLID = msg.control_id
ENDIF
IF @CLASS = @MENUPICK
@MENUNUM = msg.menu_id
ENDIF
GOSUB msg.window.handler : REM call the handler subroutine for this window or dialog
NEXT x
RETURN
```

The purpose of having a nosleep parameter is to allow your program to remain responsive to user input while it is very busy doing something else. For example:

```
FOR x = 1 TO 10000000
...some complex math and drawing here
WAIT 1
IF cancel = 1 THEN x = 10000001
NEXT x
```

This assumes that there is some menu option, or button that sets the variable cancel equal to 1. If you did not use WAIT 1 in this case the program would appear to be 'locked' and would not respond to menu's or buttons until the long loop was finished. You should limit your use of WAIT 1 since this creates a "busy wait" loop. Since your program is never allowed to sleep it consumes the majority of the processor time on the system. This will reduce overall system performance.

WAIT can also be used to create a custom WAITUNTIL loop:

```
SUB MYWAITUNTIL
DO
GOSUB processdata
WAIT
UNTIL (cancel = 0) | (run = 0)
RETURN
```

Using WAIT in this manner allows using the time between messages for custom processing.

Message ID's and the handler

Now that you have a general idea of what is happening with the message queue and loop we can explore some of the message ID's your handler will receive. To review the handler is just a subroutine that is called by the system, through either the WAITUNTIL or WAIT statements. Creative BASIC predefines many of the common message ID's that your program will use. Windows defines many hundreds more that can be used with your Creative BASIC programs. A good source of information on Windows messages is the Windows SDK available from Microsoft, the windows header files, or from the Microsoft developers website

Mouse messages

The mouse generates an input events whenever the user moves the mouse, or presses or releases a mouse button. Windows converts mouse input events into messages and posts them to the appropriate programs message queue. When mouse messages are posted faster than a program can process them, Windows discards all but the most recent mouse message.

A window receives a mouse message when a mouse event occurs while the cursor is within the borders of the window, or when the window has captured the mouse. Mouse messages are divided into two groups: client area messages and nonclient area messages. Typically, an application processes client area messages and ignores nonclient area messages.

When a mouse message is received and your handler is called the system variables @MOUSEX and @MOUSEY will contain the position of the pointers hot spot at the time the message was generated. The following mouse message ID's are predefined in Creative BASIC

Message ID	Meaning	Windows equivalent
@IDMOUSEMOVE	The mouse was moved in the client area	WM_MOUSEMOVE
@IDLBUTTONDN	Left mouse button was pressed while the pointer was in the client area	WM_LBUTTONDOWN
@IDLBUTTONUP	Left mouse button was released while the pointer was in the client area	WM_LBUTTONUP
@IDLBUTTONDBLCLK	Left mouse button was double clicked while the pointer was in the client area	WM_LBUTTONDBLCLK
@IDRBUTTONDN	Right mouse button was pressed while the pointer was in the client area	WM_RBUTTONDOWN
@IDRBUTTONUP	Right mouse button was released while the pointer was in the client area	WM_RBUTTONUP
@IDRBUTTONDBLCLK	Right mouse button was double clicked while the pointer was in the client area	WM_RBUTTONDBLCLK

Additional information:
@CODE contains the status of the other mouse buttons and the CTRL/SHIFT keys when the message was received.
0x0001 = left mouse button is down
0x0002 = right mouse button is down
0x0004 = SHIFT is being held down
0x0008 = CTRL is being held down
0x0010 = middle mouse button is down

The values should be test by using a bit wise AND (&). For example:
IF (@CODE & 0x0004) = 0x0004 : REM shift key is held down

Keyboard messages

Messages from the keyboard are generated whenever a key is pressed while your window has focus. Keyboard messages are sent in two different forms, keystroke messages and character messages. Keystroke messages are sent as raw keyboard scan codes, before the system translates them. Character messages are sent when the system translates the raw key code, taking into account the SHIFT, ALT and CTRL keys. The following keyboard messages ID's are predefined by Creative BASIC:

Message ID	Meaning	Windows equivalent
@IDKEYDOWN	A key was pressed while the window had focus	WM_KEYDOWN
@IDKEYUP	A key was released while the window had focus	WM_KEYUP
@IDCHAR	A keystroke or combination generated a valid ASCII character.	WM_CHAR

Additional information:
@CODE will contain the raw virtual keycode for @IDKEYDOWN and @IDKEYUP messages or the ASCII character for @IDCHAR messages. See for a list of virtual key codes.

Window and system messages

Messages for your window are generated whenever an action that would effect the window happens. These include sizing messages, creation messages, drawing messages, close events, system wide messages, etc.

Message ID	Meaning	Windows equivalent
@IDCREATE	Sent when the window is first created but before it is displayed	WM_CREATE
@IDDESTROY	Sent when the window is about to be destroyed.	WM_DESTROY
@IDINITDIALOG	Sent when the dialog is about to be shown. Initialize all controls here	WM_INITDIALOG
@IDCLOSEWINDOW	The close button was pressed in a window or modeless dialog	WM_CLOSE
@IDSIZE	Sent when the window or dialog is being sized.	WM_SIZE/WM_SIZING
@IDERASEBACKGROUND	Sent when the background of a window needs to be painted. The window background is shown when transparent objects, such as toolbars with the @TBFLAT style, are visible.	WM_ERASEBKGND
@IDPAINT	Sent when a window created with @NOAUTODRAW needs repainting.	WM_PAINT
@IDMENUINIT	The @IDMENUINIT message is sent when a menu is about to become active. It occurs when the user clicks an item on the menu bar or presses a menu key. This allows the application to modify the menu before it is displayed.	WM_INITMENU
@IDTIMER	Sent when a timer, set with the SETTIMER statement, expires. Multiple times may be used with a window. @CODE contains the timer ID.	WM_TIMER
@IDDXUPDATE	Sent when a DirectX or Direct3D window should be redrawn.	NONE
@IDMENUPICK	Sent when a used selects a menu item. Check @MENUNUM for the ID of the menu item selected	WM_MENUSELECT
@IDCONTROL	Sent by controls when activated. Check @CONTROLID for the controls ID and @NOTIFYCODE for the notification message.	WM_COMMAND/ WM_NOTIFY
@IDHSCROLL/ @IDVSCROLL	Sent when a horizontal or vertical scrollbar is used.	WM_HSCROLL/ WM_VSCROLL

Additional information.
Menus and are covered in detail in other sections of the user's guide.
System Variables and Constants

Window Manipulation functions

Once your window is successfully opened you can control its size, position and appearance with Creative BASIC's window manipulation functions.

CENTERWINDOW dialog|window
Will center the window or dialog in the middle of the screen

SETSIZE dialog|window, L, T, W, H {,ID}
Will set the size and position of the window. The window will start at the point specified by the L(eft) and T(op) variables and the size will be set by the W(idth) and H(eight) variables. If ID is specified then the control with that ID is sized

SHOWWINDOW window, flags {,ID}
Will show the window based on one of the flags: @SWMINIMIZED, @SWMAXIMIZED, @SWRESTORE or @SWHIDE. ID is an optional control ID. If ID is specified then the action is performed on the control.

CLOSEWINDOW window
Closes the window. Make sure you close all of your open windows before your program exits.

SETCAPTION window, string
Sets the caption text of the window.

SETFOCUS window | dialog {,ID}
Activates the window, dialog or control and sets the input focus.

SENDMESSAGE window | dialog, msg, wparam, lparam {,ID}
Sends a message to the window, dialog or control if ID is specified. msg and wparam are numeric values. lparam can be a numeric value, string, pointer or memory variable depending on the message being sent. Can return a value if used as a function

Example sends a message to a list box with an ID of 10.
SENDMESSAGE d1,0x18d,0,"c:\",10
or
result = SENDMESSAGE(d1,0x18d,0,"c:\",10)

SETWINDOWCOLOR window, color
Changes the window background color. Color is an integer value that can be created with the RGB function.

ENABLETABS window, enable
If enable = 1 then the window's message loop handles the same shortcuts as a dialog would allowing tabbing between controls and using a default button. To disable this set enable = 0. Note that when enabled a window will not receive @IDCHAR messages.

SETUSERDATA window, variable
Sets the windows user data to the variable. Variable may be of type INT, UINT or POINTER. Data may be retrieved by using the GETUSERDATA statement. This statement is similar to the Windows API SetWindowLong function using GWL_USERDATA

STARTTIMER statement
Starts a timer in the window that will send an @IDTIMER message every count milliseconds. An optional ID number can be supplied to have multiple timers. The default ID is 1. The syntax of STARTTIMER is:

STARTTIMER window | dialog, count {,ID}

STOPTIMER window | dialog {,ID}
Stops the previously started timer in the window. The syntax of STOPTIMER is:

Window Information functions

When a window or dialog is open, you can get size and position information by using one of the following functions:

GETSIZE window | dialog,varL,varT,varW,varH {, ID}
Will return the size of the window in the variables specified. Variables must be of type INT. For a parent window the left and top coordinates are relative to the upper left corner of the screen. For child windows the left and top coordinates are relative to the upper left corner of the window. If ID is specified then the size of the control is returned.

GETCLIENTSIZE window | dialog,varL,varT,varW,varH
Will return the size of the drawing area of the window in the variables specified. L and T will always be 0 with this function. Variables must be of type INT.

GETCAPTION (window)
Returns the windows caption text.

Example:
A$=GETCAPTION (mywin)

GETSCREENSIZE varW,varH
Returns the system screen size. Useful for determining a size to set your window. Variables must be of type INT.

GETTEXTSIZE window, string, varWidth, varHeight
Returns the size of a string in pixels when printed to the window. varWidth and varHeight must be of type INT. The size of the string is useful for determining line positions.

GETUSERDATA window, variable
Retrieves data previously set with SETUSERDATA. Variable may be of type INT, UINT or POINTER. This statement is similar to the Windows API function GetWindowLong using GWL_USERDATA.

GETPOSITION window, varX, varY
Retrieves the current drawing position in the window. varX and varY must be of type INT. The current drawing position is set with the MOVE statement, graphics operations, and text PRINTing.

GETCARETPOSITION window, varX, varY
Retrieves the current caret position in the window. varX and varY must be of type INT.

Window Text functions

A blank window is uninteresting so lets put some text into it. Earlier we discussed how to use the PRINT statement to output text into the console window. To output text to a window we use the PRINT statement and specify the window as the first parameter.

In the console we could use LOCATE to specify where text would be printed. For a window we use the MOVE statement. Since both text and graphics are sent to a window as bitmapped images, the MOVE statement takes its parameters in pixels and not characters. Pixels start from the upper left corner at location 0,0. The syntax of MOVE is:

MOVE window, x, y

The window keeps track of the current position specified by the MOVE statement. When text is printed, the position is adjusted to the end of the line.

Example:

```
DEF w:WINDOW
WINDOW w,0,0,640,200,@SIZE|@MINBOX|@MAXBOX,0,"Text",main
CENTERWINDOW w
MOVE w,4,20
FOR x=0 TO 50
PRINT w,"X"
NEXT x
MOVE w,4,40
PRINT w,"This is a test!"
run = 1

WAITUNTIL run = 0
CLOSEWINDOW w:END

SUB main
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
CASE @IDMOUSEMOVE
MOVE w,4,60
PRINT w,"Mouse Position: ",@MOUSEX,@MOUSEY," "
ENDSELECT
RETURN
```

Changing the font

The text printed to a window will use the default font specified in the display control panel unless changed with the SETFONT statement. The SETFONT statement has the syntax of:

SETFONT window, typeface, height, weight {, flags | charset} {,ID}

Height and weight can both be 0 in which case a default size and weight will be used. Weight ranges from 0 to 1000 with 700 being standard for bold fonts and 400 for normal fonts. Flags can be a combination of @SFITALIC, @SFUNDERLINE, or @SFSTRIKEOUT for italicized, underlined, and strikeout fonts. If an ID is specified then the font of a control in the window or dialog is changed.

Example fragment:

```
...
SETFONT mywin, "Ariel", 20, 700, @SFITALIC
PRINT mywin, "ARIEL bold italic"
...
```

Selecting character sets.

Certain fonts may have more than one character set. Normally this information is set automatically by the flag value returned by FONTREQUEST. You can set the character set manually by using the following values ORed in with the flags

ANSI_CHARSET = 0
DEFAULT_CHARSET = 0x00010000
SYMBOL_CHARSET = 0x00020000
SHIFTJIS_CHARSET = 0x00800000
HANGEUL_CHARSET 0x00810000
GB2312_CHARSET = 0x00860000
CHINESEBIG5_CHARSET = 0x00880000
OEM_CHARSET = 0x00FF0000
JOHAB_CHARSET = 0x00820000
HEBREW_CHARSET = 0x00B10000
ARABIC_CHARSET = 0x00B20000
GREEK_CHARSET = 0x00A10000
TURKISH_CHARSET = 0x00A20000
VIETNAMESE_CHARSET = 0x00A30000
THAI_CHARSET = 0x00DE0000
EASTEUROPE_CHARSET = 0x00EE0000
RUSSIAN_CHARSET = 0x00CC0000
MAC_CHARSET = 0x004D0000
BALTIC_CHARSET = 0x00BA0000

For example to set a terminal font which requires the OEM character set with an italic style:
SETFONT mywin, "Terminal", 20, 700, @SFITALIC | 0x00FF0000

If a character set doesn't exist in a particular font then the system will pick a font that closely matches the requested one.

Changing Colors

Text and graphics default to black on white. To change the current foreground drawing color of a window use the FRONTPEN statement. For the background color use the BACKPEN statement. The syntax of FRONTPEN and BACKPEN are:

FRONTPEN window, color

BACKPEN window, color

The color chosen by the FRONTPEN will be used by text, lines, outlines of rectangles and ellipses, and borders. The BACKPEN color is used as a fill for text if the drawing mode is not transparent.

The color variable can be set easily with the RGB function. RGB takes three numbers from 0 to 255 representing the intensity of red, green and blue components.

Example fragments:

```
FRONTPEN mywin, RGB(0,0,255):REM light blue
FRONTPEN mywin, RGB(100,0,0):REM medium red
BACKPEN mywin,RGB(200,200,200):REM light gray
```

FONTREQUEST function

The FONTREQUEST function opens the standard system font dialog. The functions returns the name of the font and sets four variables with the attributes of the requested font. The syntax of the FONTREQUEST function is:

name = FONTREQUEST(window, varSize, varWeight, varFlags, varColor {,dispname})

The variable parameters must be of type INT. FONTREQUEST returns an empty string if the user cancels the dialog.

Example fragment:

```
DEF size,weight,flags,col:INT
DEF fontname:STRING
...
fontname = FONTREQUEST(win,size,weight,flags,col)
IF fontname <> ""
SETFONT win,fontname,size,weight,flags
```

```
FRONTPEN win,col
ENDIF
```

The variables can be preset to show initial font settings when the dialog is displayed. Optional dispname string presets the font name pull down.

Graphics

Creative BASIC has graphic functions for lines, rectangles, ellipses, points, bitmaps, icons and cursors. Primitive graphic elements are drawn in the current foreground and background colors.

LINE statement

The line statement draws a solid line between the start and end points specified or between the last position and the end point specified. The line will be drawn in the current foreground color unless changed by the optional color parameter. The syntax of the line statement is:

LINE window {, startx, starty}, endx, endy {, color}

If startx and starty are not specified the line will start from the ending point of the last line or the last position set by the MOVE statement.

Example:

```
DEF w:WINDOW
WINDOW w,0,0,640,220,@SIZE|@MINBOX|@MAXBOX,0,"Lines",main
CENTERWINDOW w
LINE w,4,20,620,20,RGB(255,0,0)
LINE w,620,180,RGB(0,0,255)
LINE w,4,180,RGB(0,255,0)
LINE w,4,20,RGB(255,255,0)
run = 1

WAITUNTIL run = 0
CLOSEWINDOW w
END
SUB main
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
ENDSELECT
RETURN
```

RECT statement

The RECT statement is used to draw rectangles in the window. Rectangles can be filled with an optional fill color. The rectangle is drawn in the current foreground color unless a border color is specified. The syntax of the RECT statement is:

RECT window, left, top, width, height {, border {, fill}}

ELLIPSE statement

The ELLIPSE statement is used to draw ellipses in the window. The ellipse will be bound by the rectangle specified in the left,top,width and height parameters. The ellipse drawn in the current foreground color unless a border color is specified. The ellipse may be filled with an optional fill color. The syntax of the ellipse statement is:

ELLIPSE window, left, top, width, height {, border {, fill}}

CIRCLE statement

The CIRCLE statement is used to draw device independent circles. The circle will be drawn at the starting point specified with the radius specified. The circle will be drawn in the current foreground color unless a border color is specified. The circle may be filled with an optional fill color. The syntax of the CIRCLE statement is:

CIRCLE window, centerx, centery, radius {, border {, fill}}

Example:

```
DEF w:WINDOW
WINDOW w,0,0,640,220,@SIZE|@MINBOX|@MAXBOX,0,"Demo",main
CENTERWINDOW w
RECT w,20,20,50,100,RGB(0,0,255),RGB(0,255,0)
ELLIPSE w,90,20,100,50,RGB(255,0,0),RGB(255,255,0)
CIRCLE w,250,75,50,RGB(0,255,0),RGB(0,0,255)
run = 1

WAITUNTIL run = 0
CLOSEWINDOW w
END

SUB main
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
ENDSELECT
RETURN
```

PSET statement

The PSET statement changes one pixel of the window to the foreground color or the optional specified color. The syntax of PSET is:

```
PSET window, x, y {, color}
```

GETPIXEL function

GETPIXEL retrieves the color of a pixel in the window at the coordinates specified. The syntax of GETPIXEL is:

```
color = GETPIXEL( window, x ,y )
```

Drawing Modes

The DRAWMODE statement sets the background mode. The background mode defines whether the system removes existing background colors on the drawing surface before drawing text. The syntax of the DRAWMODE statement is

```
DRAWMODE window, flags
```

Flags can be either @TRANSPARENT or @OPAQUE. If the mode is set to @TRANSPARENT then the background color is not changed when printing text.

Raster operations

The RASTERMODE statement sets the current drawing mode. The drawing mode specifies how the colors of the pen and the interior of filled objects are combined with the color already on the display surface. The syntax of RASTERMODE is:

```
RASTEMODE window, flags
```

Flags can be any one of the rastermode flags listed in the appendix.

Example:

```
DEF w:WINDOW
WINDOW w,0,0,640,220,@SIZE|@MINBOX|@MAXBOX,0,"Demo",main
CENTERWINDOW w
RASTERMODE w, @RMXORPEN
RECT w,20,20,50,100,RGB(0,0,255),RGB(0,255,0)
ELLIPSE w,20,20,100,50,RGB(255,0,0),RGB(255,255,0)
CIRCLE w,50,50,25,RGB(0,255,0),RGB(0,0,255)
run = 1

WAITUNTIL run = 0
```

```
CLOSEWINDOW w
END

SUB main
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
ENDSELECT
RETURN
```

Images, Icons and Cursors

Loading an image

Creative BASIC can load an image, icon or cursor with the LOADIMAGE function. LOADIMAGE returns a handle to the loaded image as an integer value. This value can then be passed to any of the functions that accepts a handle as a parameter. The syntax of LOADIMAGE is:

handle = LOADIMAGE (filename | resource ID, type)

Type is a numeric value defining what kind of image to load.
The valid values for type are:
@IMGBITMAP - bitmap (*.bmp)
@IMGICON - Icon (*.ico)
@IMGCURSOR - Cursor (*.cur)
@IMGEMF - Enhanced meta file (*.emf)
@IMGSCALABLE - scalable bitmap, JPEG (*.jpg) or GIF (*.gif) files.

@IMGOEM - OR in with @IMGBITMAP, @IMGICON or @IMGCURSOR to load an OEM (system) image.
@IMGMAPCOLORS - OR in with @IMGBITMAP to have the system search the bitmap color table and map the following shades of grey:

Dk Gray, RGB(128,128,128)	3DSHADOW COLOR
Gray, RGB(192,192,192)	3DFACE COLOR
Lt Gray, RGB(223,223,223)	3DLIGHT COLOR

The mapping of colors with @IMGMAPCOLORS is used in loading toolbars and button images so the current system colors are used for 3D elements.

If a filename is specified the image is loaded from disk. Creative BASIC can also load bitmaps, icons and cursors directly from the executables resources. Enhanced meta files cannot be loaded from the resource table. Scalable images may be loaded from resources as custom type RTIMAGE.

Resource ID is either the string or integer identifier of a resource compiled with the project.

Freeing the image

After your program is finished with an image it should call the DELETEIMAGE statement to free memory used by the image. The syntax of DELETEIMAGE is:

DELETEIMAGE handle, type

Type *must* be the same value specified in the LOADIMAGE function.

Displaying the image

An image can be drawn in a window using the SHOWIMAGE statement. SHOWIMAGE can draw images, icons and enhanced metafiles. The syntax of SHOWIMAGE is:

SHOWIMAGE window, handle, type, x, y , {w, h{, flags}}

Type is the same value specified in the LOADIMAGE statement. x and y specify the upper left corner of the image. W and h specify the width and height of the image or EMF file. The flags variable is for advanced users and are passed directly to the Windows function BitBlt.

If the image type equals 4 then SHOWIMAGE will show a JPEG, GIF or bitmap and w,h will scale the image to the width and height specified. If w and h are omitted the exact size of the image will be used with no scaling.

Width and Height must be specified for bitmap files.

Changing cursors

The currently displayed cursor in a window can be changed with the SETCURSOR statement. The syntax of SETCURSOR is:

SETCURSOR window, style {, handle}

Valid values for style are @CSWAIT for a wait cursor, @CSARROW for the standard arrow cursor and @CSCUSTOM for a cursor loaded with the LOADIMAGE function.

Changing icons

The current icon for a window, displayed in the titlebar and taskbar, can be changed with the SETICON statement. SETICON accepts a handle from the LOADIMAGE function. The syntax of SETICON is:

SETICON window, handle

See the samples bitmap.cba and imgview.cba for examples of using the image functions.

Advanced graphics statements

For advanced users Creative BASIC allows using WIN32 (WINAPI) functions to draw in a window. In order to use any of the WIN32 graphics functions you must first obtain a handle to a device context. While you could use the GetDC/ReleaseDC functions in the USER32.DLL, It is more convenient and compatible to use Creative BASIC's built-in GetHDC and ReleaseHDC. The syntax of the GetHDC and ReleaseHDC functions are:

handle = GetHDC window
ReleaseHDC window, handle

The handle returned is an integer value and is a valid HDC.

The @NOAUTODRAW flag

To handle PAINT messages directly in your program specify the @NOAUTODRAW flag when creating the window. Your windows subroutine will begin receiving @IDPAINT messages whenever the window needs updating.

Example fragment:

```
...
WINDOW w,0,0,350,350@SIZE|@NOAUTODRAW,0,"Test",mainwindow
...
SUB mainwindow
select @CLASS
case @IDCLOSEWINDOW
run=0
case @IDPAINT
if(bitmap)
ShowImage w1,bitmap,0,x,y,w,h
endif
endselect
return
```

MDI windows

Multiple Document Interface, or MDI, windows consist of a parent frame window and one or more child windows. Most word processors use the MDI interface as well as Creative BASIC's editor. To create a MDI interface first open a window with @MDIFRAME as one of the creations flags:

WINDOW frame,0,0,640,400,@MDIFRAME|@SIZE,0,"Main",main

Any child windows would use that window as the parent parameter:

WINDOW w,0,0,200,100,@SIZE|@MINBOX|@MAXBOX,*frame*,"Child",main

The frame window will automatically contain a standard menu for minimizing, maximizing, restoring and arranging icons of the child windows. When ending your program it is not necessary to close all of the child windows individually. Closing just the frame window will also close all of the child windows.

For default sized child windows you can use a special variable @USEDEFAULT for the left parameter. Windows will then pick a standard size for your window:

WINDOW w,@USEDEFAULT,0,0,0…

You can use the same message subroutine for both the frame window and the child windows. In order to differentiate where the message is coming from, use the special @HITWINDOW variable. @HITWINDOW will contain the originating window when your message subroutine is called. The following example fragment demonstrates this:

```
SUB wndproc
SELECT @CLASS
CASE @IDCLOSEWINDOW
IF @HITWINDOW = frame
run = 0
ELSE
CLOSEWINDOW @HITWINDOW
ENDIF
```

The example 'mdidemo.cba' contains a complete example of a skeleton MDI program. It is a good place to start building MDI applications.

Menus

Defining

Creative BASIC has three menu creation statements MENU, INSERTMENU and CONTEXTMENU. The MENU statement will replace the entire window menu with the items specified. The INSERTMENU statement will add a menu to a window at a position specified. CONTEXTMENU displays a popup menu at the specified x and y positions in the window. Syntaxes:

MENU window | dialog, definition {,definition…}
INSERTMENU window | dialog, position, definition {,definition…}
CONTEXTMENU window | dialog, x, y, definition {,definition…}

Each definition is a string literal or variable containing the type of menu item to add, the text of the menu as will as an ID for your message subroutine to use. The format of the definition string is:

"[T|I|S], text, flags, ID"
T = Title
I = Item
S = Sub item (popup)

MENU and INSERTMENU must have a Title menu as the first definition. A context menu can consist of only items. The sub item menu will create a popup menu that appears when selected by the mouse.

Example definitions:

MENU w, "T, File, 0, 0" , "I, New, 0, 1" , "I, Quit, 0 ,2"
INSERTMENU w,1, "T,Edit,0,0" , "I,Cut,0,3" , "I,Copy,0,4"
CONTEXTMENU w, @MOUSEX, @MOUSEY, "I,Color,0,99","I,Clear,0,1"

The INSERTMENU statement takes the same definitions as parameters as well as a position variable. The menus specified will be inserted at the position and any existing menus will be moved to the right. You should always use INSERTMENU for a MDI frame window since the standard menu is already in place.

Context menus are normally displayed when the user right clicks in the window. To do this respond to the @IDRBUTTONUP message by displaying the menu.

Example:

```
SELECT @CLASS
...
CASE @IDRBUTTONUP
CONTEXTMENU w, @MOUSEX, @MOUSEY, "I,Color,0,99","I,Clear,0,1"
...
```

When creating a sub item menu you can return to the previous level by adding a ^ character to the next item

Example:

MENU w, "T,Properties,0,0", "S,Font,0,0", "I,Courier,0,1", "^I,Page,0,2"

Messages

Once you define menus for your window, your programs handler subroutine will receive an @IDMENUPICK message when an item is selected. The ID of the menu will be returned in @MENUNUM.

Adding items to an existing menu

To add items to an already existing menu use the ADDMENUITEM statement. ADDMENUITEM uses the zero-based position of the menu starting from the the left to determine where to add the item. The item is added to then end of the menu. Syntax:

ADDMENUITEM window | dialog, position, text, flags, ID

Removing menus or menu items

To remove menu items or an entire menu use the REMOVEMENUITEM statement.

REMOVEMENUITEM window | dialog, position, ID

Position is the zero-based index of the menu starting from the left. ID is the item to remove. If ID = 0 then the entire menu at position is removed.

Controlling menu appearance

When the menu is created items can optionally be disabled or have a check shown next to them. To create a disabled menu item us the @MENUDISABLE flag. For a checked menu item use the @MENUCHECK flag. While your program is running it will receive a @IDMENUINIT message every time the menu is about to be displayed. When your program receives the @IDMENUINIT message you can change the disabled and checked settings of an item with the CHECKMENUITEM, ENABLEMENUITEM and ENABLEMENU statements. The syntax of the statements is:

CHECKMENUITEM window | dialog, ID, state
ENABLEMENUITEM window | dialog, ID, state
Where state is either 0 for disabled/unchecked or 1 for enabled/checked.

ENABLEMENU window | dialog, position, state
Enables/disables a top level menu. Position is the zero based index of the menu title.

Example:

```
...
MENU mywin,"T,Options,0,0", "S,Line Color,0,0", "I,RED,@MENUCHECK,5", "I,BLUE,0,6"
...

handler:
SELECT @CLASS
CASE @IDMENUINIT
CHECKMENUITEM mywin, 5, (linecolor = red)
CHECKMENUITEM mywin, 6, (linecolor = blue)
...
```

The samples draw.cba and mapedit.cba provide good examples of menu definitions and message handling.

Creating embedded browser windows

Creative BASIC allows creating a self contained internet browser when opening a window. The embedded browser functions require Internet Explorer 4.0 or greater to be installed on the system. To open a browser window specify @BROWSER as a creation flag for the window. @NOAUTODRAW should also be used to prevent any overwriting of the displayed HTML document.

Example:

```
DEF wb:WINDOW
WINDOW wb,0,0,640,480,@SIZE|@BROWSER|@NOAUTODRAW|@MINBOX|@MAXBOX,0,"Test Browse",main

BROWSECMD wb,@NAVIGATE,"http://www.ionicwind.com"

run = 1
WAITUNTIL run=0
CLOSEWINDOW wb
END

main:
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
ENDSELECT
RETURN
```

Controlling the browser

As indicated in the above example, controlling the embedded browser is done with BROWSECMD. BROWSECMD serves as both a statement and a function depending on the command issued. The syntax of BROWSECMD is:

{return = }BROWSECMD (window, command {,parameters})

The available commands are:

Command	Details
@NAVIGATE	Navigates to the specified web page or local file. Uses 1 string parameter.
@GOHOME	Loads the default 'home' page
@GOBACK	Moves back one page in the history list. No parameters
@GOFORWARD	Moves ahead one page in the history list. No parameters
@BROWSESTOP	Stops loading of the current document. No parameters
@REFRESH	Refreshes the current document. No parameters
@GETTITLE	Returns the title of the current document. No parameters. Used as a function.
@BACKENABLED	Returns 1 if there is at least one page to @GOBACK to. 0 otherwise. No Parameters. Used as a function
@FORWARDENABLED	Returns 1 if there is at least one page to @GOFORWARD to. 0 otherwise. No Parameters. Used as a function.
@CANCELNAV	Cancels navigation to the current page. Use in response to @IDBEFORENAV to limit navigation. No parameters.
@GETPOSTDATA	Returns a string containing the data from a form. Filled in before @IDBEFORENAV is sent. No Parameters. Function.

@GETHEADERS	Returns a string containing the headers sent. Filled in before @IDBEFORENAV is sent. No Parameters. Function.
@GETNAVURL	Returns a string containing the URL. Filled in before @IDBEFORENAV or @IDNAVCOMPLETE is sent. No Parameters. Function.
@BROWSELOAD	Loads the browser with the contents of a string. 1 string parameter.
@BROWSEPRINT	Prints the currently displayed document. No Parameters.

Navigating

The @NAVIGATE command allows specifying either a network URL or path to a local file or directory. When used with a directory path the browser will display an Explorer like window allowing standard file operations. The embedded browser can display any of the file types that are viewable in Internet Explorer including pictures, plug-ins, and html documents.

Example navigation statements:

```
BROWSECMD window, @NAVIGATE, "c:\"
BROWSECMD window, @NAVIGATE, "e:\images\picture.jpg"
BROWSECMD window, @NAVIGATE, "http://www.ionicwind.com"
BROWSECMD window, @NAVIGATE, "ftp://ftp.ionicwind.com"
```

Messages

The embedded browser will send an @IDBEFORENAV message to the window just before it navigates to a page. If you wish to limit access to certain websites or files you can call BROWSCMD with a command of @CANCELNAV. This message can also be used to extract the data from an HTML form.

Once navigation is complete the browser will send an @IDNAVCOMPLETE message. Use this message to redirect to a different URL or to save data gathered during an @IDBEFORENAV message.

Example:

```
DEF wb:WINDOW
DEF data,url:STRING
WINDOW wb,0,0,640,480,@SIZE|@BROWSER|@NOAUTODRAW|@MINBOX|@MAXBOX,0,"Test Browse",main

BROWSECMD wb,@NAVIGATE,"c:\forms\userdata.html"

run = 1
WAITUNTIL run=0
CLOSEWINDOW wb
END

main:
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
CASE @IDBEFORENAV
url = BROWSECMD(wb,@GETNAVURL)
IF url <> "c:\forms\userdata.html"
data = BROWSECMD(wb,@GETPOSTDATA)
ENDIF
CASE @IDNAVCOMPLETE
url = BROWSECMD(wb,@GETNAVURL)
IF url <> "c:\forms\complete.html"
BROWSECMD wb,@NAVIGATE,"c:\forms\complete.html"
ENDIF
ENDSELECT
RETURN
```

In the above example we first navigate to an html page containing a form. In response to @IDBEFORENAV we check the URL to

see if it is not the first page we navigated to. This means the user has pressed the 'send' or 'post' button. Then we retrieve the posted data. In a real world example we would parse the data in the string and act upon it. If one of the fields was in error then the @CANCELNAV command could prevent the user from continuing on.

Posted data normally consists of name-value pairs separated by the '&' symbol. If our form had two fields 'first' and 'last' and the user entered John Smith the string returned would be:

first=John&last=Smith

Notes

Be careful when navigating to a page in response to @IDNAVCOMPLETE. If you do not test the current URL, the browser will end up in a loop since every page sends the message.

Don't use the @NAVIGATE command in response to an @IDBEFORENAV message. Doing so will send the browser into an endless loop and end your program without warning.

Defining a Control

Controls such as an edit box or check box can be created in either a window or a dialog. The control is defined after the window or dialog is opened and dynamically added. Controls are automatically removed when the window or dialog is closed. The syntax of the CONTROL statement is:

CONTROL window|dialog, definition {, definition…}

Each definition is a string literal or variable containing the type of control, title text, dimensions, flags and an ID for your message subroutine. The control definition has the format of:

"Type, title, L, T, W, H, flags, ID"

Type is a single letter representing the type of control to be created. The possible values for type are:

B creates a button
E creates an edit box
S creates a scrollbar
R creates a radio button
C creates a check box
L creates a list box
M creates a combo box
T creates a static text control
LV creates a List View control
RE creates a Rich Edit control
SW creates a status window

Example definitions:

CONTROL w,"B,Save,56,100,50,20,0,1"
CONTROL d,"M,,450,100,100,100,@TABSTOP,7"

Each control has its own creation flags and returns specific messages to your window or dialog subroutine. The @IDCONTROL message is sent when any operation is done to one of the controls. @CONTROLID will contain the ID value specified when the control was created. @CODE may contain additional information.

The easiest way to create and size a control is with the dialog editor.

Bitmap buttons and static controls

Buttons and static controls have the capability of displaying a bitmap instead of text. To define a bitmap button use the flag @CTLBTNBITMAP for a button or @CTLSTCBITMAP for static controls To specify the bitmap to display set the buttons text to the complete pathname to the bitmap file with the SETCONTROLTEXT statement. If your button is contained in a dialog use SETCONTROLTEXT in response to the @IDINITDIALOG message.

Example of bitmap static control:

```
DEF d1:Dialog

DIALOG d1,0,0,295,168,0x80C80080,0,"Caption",Handler
CONTROL d1,"T,,13,14,102,102,@CTLSTCBITMAP,1"


DOMODAL d1
END

SUB handler
SELECT @class
CASE @IDINITDIALOG
SETCONTROLTEXT d1,1,GETSTARTPATH + "bug.bmp"
CENTERWINDOW d1
ENDSELECT
```

RETURN

Changing the font of a control
The text printed in a control will use the default font specified in the display control panel unless changed with the SETFONT statement. The SETFONT statement has the syntax of:

SETFONT window | dialog, typeface, height, weight {, flags} {,Control_ID}

Height and weight can both be 0 in which case a default size and weight will be used. Weight ranges from 0 to 1000 with 700 being standard for bold fonts and 400 for normal fonts. Flags can be a combination of @SFITALIC, @SFUNDERLINE or @SFSTRIKEOUT for italicized and underlined fonts. If an ID is specified then the font of a control is changed.

The height parameter is specified in points. 1 point is equal to 1/72nd of an inch. If you want a font that is 1/2 an inch high you would specify a point size of 36.

Control Functions

SETCONTROLCOLOR window|dialog, ID, fg, bg
Changes the foreground and background colors of a control in a window or dialog. When used with controls in a dialog use SETCONTROLCOLOR in response to the @IDINITDIALOG message. The foreground color is used by text displayed in the controls. These colors can be specified using the RGB function.

SETCONTROLTEXT window | dialog, ID, text
SETCONTROLTEXT is used to change the text of a control. If the control is a bitmap button or bitmap static control then the text is the path to a bitmap file.

text$ = GETCONTROLTEXT (window | dialog, ID)
GETCONTROLTEXT is used to retrieve the text of a control. GETCONTROLTEXT and SETCONTROLTEXT will work with single and multi-line edit controls to get/set the typed text. For any other control the caption is get/set. GETCONTROLTEXT returns the result as a string.

return = CONTROLEXISTS (window | dialog, ID)
Returns 1 if the specified control exists in the dialog or window. Returns 0 if the control could not be found. Useful for controls that are dynamically created based on user options.

SETSCROLLRANGE window | dialog, ID, min, max
SETSCROLLRANGE Sets the minimum and maximum range of a scrollbar control to min and max. All values returned by the scrollbar will be between min and max. If ID = -1 then sets the range of the windows horizontal scrollbar. If ID = -2 then sets the range of the windows vertical scrollbar. ID must be a scrollbar control.

GETSCROLLRANGE window | dialog, ID, varMin, varMax
Stores the scrollbars range into the variables specified by varMin and varMax. The variables must be of type INT. If ID = -1 then stores the range of the windows horizontal scrollbar. If ID = -2 then stores the range of the windows vertical scrollbar. ID must be a scrollbar control. :

SETSCROLLPOS window | dialog, ID, position
Sets the slider position of a scrollbar. If ID = -1 then sets the scroll position of the windows horizontal scrollbar. If ID = -2 then sets the scroll position of the windows vertical scrollbar. Any other ID value is a user-defined scrollbar. Position must be between the minimum and maximum values set by the SETSCROLLRANGE statement.

position = GETSCROLLPOS (window | dialog, ID)
Returns the slider position of a scrollbar. If ID = -1 then returns the scroll position of the windows horizontal scrollbar. If ID = -2 then returns the scroll position of the windows vertical scrollbar. Any other ID value is a user-defined scrollbar.

SETSELECTED window | dialog, ID, position
Sets the currently selected item in a list or combo box control. The position value is zero-based and the list or combo box must contain a string at that position.

index = GETSELECTED (window | dialog, ID)
Returns the zero-based index of the currently selected item in the list box of a combo or single selection list box. Returns -1 if no item is selected. ID must be a list box or combo box.

selected = ISSELECTED (window | dialog, ID, position)
Returns TRUE if the string at position is selected in a multi-selection list or combo box.

count = GETSTRINGCOUNT (window | dialog, ID)
Returns the number of strings in a list box or combo box control.

text$ = GETSTRING (window | dialog, ID, position)
Returns the string at position in a list box or combobox. Position is a zero based index.

ADDSTRING window | dialog, ID, string
Adds a string to a list box or combo box control. String is added to the end of the list unless sorting is specified in the style of the control.

INSERTSTRING window | dialog, ID, position, text
Inserts a string into a combo or listbox control at position. All other strings are moved down by one position.

DELETESTRING window | dialog, ID, position
Removes a string from a list box or combo box control. Remaining strings are moved up to fill the empty position.:

SETLBCOLWIDTH window | dialog, ID, width
Sets the width of columns in a multi-column list box. The list box must have been created with the style @CTLISTMULTI either in the CONTROL statement or by selecting the multicolumn checkbox in the dialog editor.

SETHORIZEXTENT window | dialog, ID, width
Sets the horizontal scroll width of a list box or list box portion of a combo box. Control must have been created with the @HSCROLL style. The width is specified in pixels.

SETSTATE window | dialog, ID, state
Sets or resets a checkbox or radio button control. State can either be 0 to uncheck the control or 1 to check the control.

state = GETSTATE (window | dialog, ID)
Returns the state of a checkbox or radio button control. Returns 1 if control is checked and 0 if control is unchecked. Radio buttons in a group are mutually exclusive. When a radio button is selected your program will receive an @IDCONTROL message with @CONTROLID containing the newly selected radio button. If the radio button is created outside of a group, you will need to use GETSTATE to determine whether the button is selected.

ENABLECONTROL window | dialog, ID, 0 | 1
The ENABLECONTROL function enables or disables a control in a window or dialog. Controls that are disables are grayed out and cannot be selected. Use 0 to disable the control or 1 to enable. Example: ENABLECONTROL mydialog, 2, 0

SETFOCUS window | dialog {,ID}
Activates the window, dialog or control and sets the input focus.

{return = } SENDMESSAGE (window | dialog, msg, wparam, lparam {,ID})
Sends a message to the window, dialog or control if ID is specified. msg and wparam are numeric values. lparam can be a numeric value, string, pointer or memory variable depending on the message being sent.

See the sample file dirselector.cba for a complete example of using SENDMESSAGE

{return =} CONTROLCMD(window | dialog, ID ,command, {,param1...})
Used to communicate with edit, rich edit, list view, status window and toolbar controls. See the sections on using those controls for more details.

Using edit controls

About edit controls

An *edit control* is a rectangular control window typically used in a dialog box to permit the user to enter and edit text from the keyboard. Edit controls are single font, single text color controls. If you need more advanced word processing features see the documentation on *rich edit* controls.

Creating the control

Edit controls are created either through the dialog editor or manually with the CONTROL statement.

Edit control styles

The following edit style flags can be specified in the CONTROL statement or by ticking the corresponding check box in the control properties of the dialog editor:

@CTEDITLEFT

Text is left justified in the edit control

@CTEDITRIGHT

Text is right justified in the edit control

@CTEDITMULTI

Designates a multiline edit control. The default is single-line edit control. When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the @CTEDITRETURN style. When the multiline edit control is not in a dialog box and the @CTEDITAUTOV style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify @CTEDITAUTOV, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed. If you specify the @CTEDITAUTOH style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify @CTEDITAUTOH, the control automatically wraps words to the beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The window size determines the position of the word wrap. If the window size changes, the word wrapping position changes and the text is redisplayed. Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Note that edit controls without scroll bars scroll as described in the previous paragraphs and process any scroll messages sent by the parent window.

@CTEDITPASS

Displays an asterisk (*) for each character typed into the edit control.

@CTEDITCENTER

Text is centered within the edit control

@CTEDITRO

The edit control is read only and text can be displayed but not entered

@CTEDITAUTOH

Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.

@CTEDITAUTOV

Automatically scrolls text up one page when the user presses the ENTER key on the last line.

@CTEDITRETURN

Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiline edit control in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.

@CTEDITNUMBER

Restricts text entered in a single line edit control to numerals only (0 - 9)

In addition to the edit styles the following window styles can also be specified:

@TABSTOP
Moves the input focus between controls if controls are in a dialog. This style has no effect for controls created in a window.

@GROUP
Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the @GROUP style after the first control belong to the same group. The next control with the @GROUP style starts the next group (that is, one group ends where the next begins)

@HSCROLL
The control has a horizontal scroll bar

@VSCROLL
The control has a vertical scrollbar

Edit control functions and statements
The following Creative BASIC functions and statements are used to communicate with the edit control.

SETCONTROLTEXT window | dialog, ID, text$
Changes the text in the edit control. Any text previously in the control will be replaced.

text$ = GETCONTROLTEXT (window | dialog, ID)
Retrieves the text in the edit control.

SETCONTROLCOLOR window|dialog, ID, fg, bg
Sets the text color and background color of the edit control. The edit control can only have a single text color.

return = CONTROLEXISTS (window | dialog, ID)
Returns 1 if the control with ID exists in the window or dialog.

ENABLECONTROL window | dialog, ID, 0 | 1
Disables or enables the edit control. If the edit control is disabled no text can be entered and the control will not receive input focus when clicked on. A disabled control will also not accept any new text with the SETCONTROLTEXT statement.

SETFOCUS window | dialog, ID
Gives the control the input focus. The edit control shows the caret.

SENDMESSAGE window | dialog, msg, wparam, lparam ,ID
Sends a message to the control for advanced functionality.

SHOWWINDOW window, flags, ID
Changes the visibility state of the control. Use @SWHIDE to hide the control and @SWRESTORE to show the control

SETSIZE dialog|window, L, T, W, H ,ID
Changes the size of the control. The edit control is redrawn and the text will be formatted to match the new size of the control. The dimensions include the borders of the control.

Clipboard operations

CONTROLCMD window | dialog, ID, **@EDCUT**
Use this command to delete (cut) the current selection (if any) in the edit control and copy the deleted text to the Clipboard.

CONTROLCMD window | dialog, ID, **@EDCOPY**
Use this command to copy the current selection (if any) in the edit control to the Clipboard.

CONTROLCMD window | dialog, ID, **@EDPASTE**
Use this command to insert the data from the Clipboard into the edit control at the insertion point, the location of the caret. Data is inserted only if the Clipboard contains data in text format.

Line operations

count = CONTROLCMD (window | dialog, ID, **@EDGETLINECOUNT**)
Use this function to retrieve the number of lines in the edit control

line$ = CONTROLCMD (window | dialog, ID, **@EDGETLINE**, linenum)
Use this function to retrieve a line of text from the edit control.
Linenum is the 0 based index of the line to retrieve

line = CONTROLCMD (window | dialog, ID, **@EDGETFIRSTLINE**)
Use this function to retrieve the zero-based index of the uppermost visible line.

line = CONTROLCMD (window | dialog, ID, **@EDLINEFROMCHAR**, index)
Use this function to retrieve the line number of the line that contains the specified character index.
Index is the number of characters from the beginning of the edit control.

length = CONTROLCMD (window | dialog, ID, **@EDGETLINELENGTH**, index)
Use this function to retrieve the length of a line in a edit control. When @RTGETLINELENGTH is called for a multiple-line edit control, the return value is the length (in bytes) of the line specified by index. When @RTGETLINELENGTH is called for a single-line edit control, the return value is the length (in bytes) of the text in the edit control.
Index specifies the character index of a character in the line whose length is to be retrieved. If this parameter is –1, the length of the current line (the line that contains the caret) is returned

Selection operations

CONTROLCMD window | dialog, ID, **@EDGETSELECTION**, varStart, varEnd
Use this command to retrieve the current selection of the edit control.
varStart and varEnd must be of type INT. The zero-based index of the first and last characters selected are copied into the two variables. The selection includes everything if varStart =0 and varEnd = -1.

CONTROLCMD window | dialog, ID, **@EDDELETESEL**
Use this statement to delete the current selection. The deletion performed by @EDDELETESEL can be undone by using @EDUNDO

CONTROLCMD window | dialog, ID, **@EDSETSELECTION**, start, end
Use this statement to set the current selection in the edit control
start and end are the zero-based character indexes of the selection. If start = 0 and end = -1 then all of the text is selected.

CONTROLCMD window | dialog, ID, **@EDREPLACESEL**, text$
Use this statement to replace the current selection text

Editing operations

CONTROLCMD window | dialog, ID, **@EDUNDO**
Use this statement to undo the last editing operation. An undo operation can also be undone. For example, you can restore deleted text with the first call to Undo. As long as there is no intervening edit operation, you can remove the text again with a second call to Undo.

return = CONTROLCMD (window | dialog, ID, **@EDCANUNDO**)
Use this function to determine if the last editing operation can be undone. Returns 0 if the last operation cannot be undone.

CONTROLCMD window | dialog, ID, **@EDEMPTYUNDO**
Use this statement to reset (clear) the undo flag of this edit control. The control will now be unable to undo the last editing operation. The undo flag is set whenever an operation within the rich edit control can be undone.

General operations

return = CONTROLCMD (window | dialog, ID, **@EDGETMODIFIED**)
Use this function to determine if the contents of the edit control have changed. Returns 1 if the contents have been modified, 0 otherwise.

CONTROLCMD window | dialog, ID, **@EDSETMODIFIED**, mod
Sets the modified flag of the edit control.
Mod can be 0 to reset the flag or 1 to set it.

length = CONTROLCMD (window | dialog, ID, **@EDGETLIMITTEXT**)
Use this function to get the text limit for this edit control. The text limit is the maximum amount of text, in bytes, the edit control can accept either through pasting or typing.

CONTROLCMD window | dialog, ID, **@EDSETLIMITTEXT**, length
Use this function to set the text limit for this edit control. The text limit is the maximum amount of text, in bytes, the edit control can accept either through pasting or typing. The default limit is 32767 bytes for multi line controls.

CONTROLCMD window | dialog, ID, **@EDSETMARGINS**, left, right
Use this statement to set the visible left and right margins of the edit control.
Left and right are specified in pixels.

Notification messages
An edit control sends notification messages to the parent window or dialog in the @NOTIFYCODE variable. The ID of the control is found in @CONTROLID. The following notification messages are supported:

@ENKILLFOCUS
Control has lost input focus

@ENSETFOCUS
Control has been given the input focus

@ENERRSPACE
Control could not complete an operation because there was not enough memory available

@ENMAXTEXT
While inserting text, the user has exceeded the specified number of characters for the edit control. Insertion has been truncated. This message is also sent either when an edit control does not have the @CTEDITAUTOH style and the number of characters to be inserted exceeds the width of the edit control or when an edit control does not have the @CTEDITAUTOV style and the total number of lines to be inserted exceeds the height of the edit control.

@ENUPDATE
The contents of the control are about to change

@ENCHANGE
The contents of the control have changed.

@ENHSCROLL
The user has clicked the edit control’s horizontal scroll bar. Windows sends this message before updating the control.

@ENVSCROLL
The user has clicked the edit control’s vertical scroll bar. Windows sends this message before updating the control.

Using list box controls

About list box

A *list box* is a control window that contains a list of items from which the user can choose.

List box items are represented by text strings. If the list box is not large enough to display all the list box items at once, the list box can provide a scroll bar. The user maneuvers through the list box items, scrolling the list when necessary, and selects or removes the selection from items. Selecting a list box item changes its visual appearance, usually by changing the text and background colors to the colors specified by the operating system metrics for selected items. When the user selects an item or removes the selection from an item, Windows sends a notification message to the parent window of the list box.

Creating the control

List box controls are created in the same manner as the standard control types, either through the dialog editor or manually with the CONTROL statement.

List box control styles

The following list box style flags can be specified in the CONTROL statement or by ticking the corresponding check box in the control properties of the dialog editor:

@CTLISTEXTENDED

Allows multiple items to be selected by using the SHIFT key and the mouse or special key combinations.

@CTLISTMULTI

Turns string selection on or off each time the user clicks or double-clicks a string in the list box. The user can select any number of strings.

@CTLISTSORT

Sorts strings in the list box alphabetically.

@CTLISTSTANDARD

Sorts strings in the list box alphabetically. The parent window receives an input message whenever the user clicks or double-clicks a string. The list box has borders on all sides.

@CTLISTNOTIFY

Notifies the parent window with an input message whenever the user clicks or double-clicks a string in the list box.

@CTLISTTABS

Enables a list box to recognize and expand tab characters when drawing its strings.

@CTLISTCOLUMNS

Specifies a multi-column list box that is scrolled horizontally. The SETLBCOLWIDTH statement sets the width of the columns.

In addition to the list box styles, the following window styles can also be specified:

@TABSTOP

Moves the input focus between controls if controls are in a dialog. This style has no effect for controls created in a window.

@GROUP

Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the @GROUP style after the first control belong to the same group. The next control with the @GROUP style starts the next group (that is, one group ends where the next begins)

@HSCROLL

The control has a horizontal scroll bar

@VSCROLL

The control has a vertical scrollbar

List box control functions and statements

The following Creative BASIC functions and statements are used to communicate with the list box control.

SETSELECTED window | dialog, ID, position

Sets the currently selected item in a list or combo box control. The position value is zero-based and the list or combo box must contain a string at that position.

GETSELECTED (window | dialog, ID)

Returns the zero-based index of the currently selected item in the list box of a combo or single selection list box. Returns -1 if no item is selected. ID must be a list box or combo box.

ISSELECTED (window | dialog, ID, position)

Returns TRUE if the string at position is selected in a multi-selection list or combo box.

GETSTRINGCOUNT (window | dialog, ID)

Returns the number of strings in a list box or combo box control.

GETSTRING (window | dialog, ID, position)

Returns the string at position in a list box or combo box. Position is a zero based index.

ADDSTRING window | dialog, ID, string

Adds a string to a list box or combo box control. String is added to the end of the list unless sorting is specified in the style of the control.

INSERTSTRING window | dialog, ID, position, text

Inserts a string into a combo or list box control at position. All other strings are moved down by one position.

DELETESTRING window | dialog, ID, position

Removes a string from a list box or combo box control. Remaining strings are moved up to fill the empty position.

SETLBCOLWIDTH window | dialog, ID, width

Sets the width of columns in a multi-column list box. The list box must have been created with the style @CTLISTMULTI either in the CONTROL statement or by selecting the multicolumn checkbox in the dialog editor.

SETHORIZEXTENT window | dialog, ID, width

Sets the horizontal scroll width of a list box or list box portion of a combo box. Control must have been created with the @HSCROLL style. The width is specified in pixels.

SETCONTROLCOLOR window|dialog, ID, fg, bg

Sets the text color and background color of the list box control.

return = CONTROLEXISTS (window | dialog, ID)

Returns 1 if the control with ID exists in the window or dialog.

ENABLECONTROL window | dialog, ID, 0 | 1

Disables or enables the list box control.

SETFOCUS window | dialog, ID

Gives the control the input focus.

SENDMESSAGE window | dialog, msg, wparam, lparam ,ID

Sends a message to the control for advanced functionality.

SHOWWINDOW window, flags, ID

Changes the visibility state of the control. Use @SWHIDE to hide the control and @SWRESTORE to show the control

SETSIZE dialog | window, L, T, W, H ,ID
Changes the size of the control.

Notification messages
An list box control sends notification messages to the parent window or dialog in the @NOTIFYCODE variable. The ID of the control is found in @CONTROLID. To receive @LBNDBLCLK or click messages it is necessary to create the control with the @CTLISTNOTIFY style. The following notification messages are supported:

@LBNDBLCLK
The user double-clicks an item in the list box.

@LBNERRSPACE
The list box cannot allocate enough memory to fulfill a request.

@LBNKILLFOCUS
The list box loses the keyboard focus.

@LBNSETFOCUS
The list box receives the keyboard focus.

@LBNSELCHANGE
The selection in a list box is about to change.

@LBNSELCANCEL
The user cancels the selection of an item in the list box.

Using combo box controls

About combo box
A *combo box* is a unique type of control that combines much of the functionality of a list box and an edit control.

A combo box consists of a list and a selection field. The list presents the options a user can select and the selection field displays the current selection. Except in drop-down list boxes, the selection field is an edit control and can be used to enter text not in the list.

Creating the control
Combo box controls are created either through the dialog editor or manually with the CONTROL statement.

Combo box control styles
The following combo box style flags can be specified in the CONTROL statement or by ticking the corresponding check box in the control properties of the dialog editor:

@CTCOMBODROPDOWN
Similar to @CTCOMBOSIMPLE, except that the list box is not displayed unless the user selects an icon next to the edit control.

@CTCOMBODROPLIST
Similar to @CTCOMBODROPDOWN, except that the edit control is replaced by a static text item that displays the current selection in the list box.

@CTCOMBOSIMPLE
Displays the list box at all times. The current selection in the list box is displayed in the edit control.

@CTCOMBOSORT
Automatically sorts strings added to the list box.

@CTCOMBOAUTOHSCROLL
Automatically scrolls the text in an edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed.

In addition to the combo box styles, the following window styles can also be specified:

@TABSTOP
Moves the input focus between controls if controls are in a dialog. This style has no effect for controls created in a window.

@GROUP
Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the @GROUP style after the first control belong to the same group. The next control with the @GROUP style starts the next group (that is, one group ends where the next begins)

@HSCROLL
The control has a horizontal scroll bar

@VSCROLL
The control has a vertical scrollbar

Combo box control functions and statements
The following Creative BASIC functions and statements are used to communicate with the combo box control.

SETSELECTED window | dialog, ID, position
Sets the currently selected item in a list or combo box control. The position value is zero-based and the list or combo box must contain a string at that position.

GETSELECTED (window | dialog, ID)
Returns the zero-based index of the currently selected item in the list box of a combo or single selection list box. Returns -1 if no item is selected. ID must be a list box or combo box.

ISSELECTED (window | dialog, ID, position)
Returns TRUE if the string at position is selected in a multi-selection list or combo box.

GETSTRINGCOUNT (window | dialog, ID)
Returns the number of strings in a list box or combo box control.

GETSTRING (window | dialog, ID, position)
Returns the string at position in a list box or combo box. Position is a zero based index.

ADDSTRING window | dialog, ID, string
Adds a string to a list box or combo box control. String is added to the end of the list unless sorting is specified in the style of the control.

INSERTSTRING window | dialog, ID, position, text
Inserts a string into a combo or list box control at position. All other strings are moved down by one position.

DELETESTRING window | dialog, ID, position
Removes a string from a list box or combo box control. Remaining strings are moved up to fill the empty position.

SETHORIZEXTENT window | dialog, ID, width
Sets the horizontal scroll width of a list box or list box portion of a combo box. Control must have been created with the @HSCROLL style. The width is specified in pixels.

SETCONTROLCOLOR window | dialog, ID, fg, bg
Sets the text color and background color of the list box control.

return = CONTROLEXISTS (window | dialog, ID)
Returns 1 if the control with ID exists in the window or dialog.

ENABLECONTROL window | dialog, ID, 0 | 1
Disables or enables the list box control.

SETFOCUS window | dialog, ID
Gives the control the input focus.

SENDMESSAGE window | dialog, msg, wparam, lparam ,ID
Sends a message to the control for advanced functionality.

SHOWWINDOW window, flags, ID
Changes the visibility state of the control. Use @SWHIDE to hide the control and @SWRESTORE to show the control

SETSIZE dialog|window, L, T, W, H ,ID
Changes the size of the control.

Notification messages
An combo box control sends notification messages to the parent window or dialog in the @NOTIFYCODE variable. The ID of the control is found in @CONTROLID.

@CBNDBLCLICK
Indicates the user has double-clicked a list item in a simple combo box.

@CBNERRSPACE
Indicates the combo box cannot allocate enough memory to carry out a request, such as adding a list item.

@CBNKILLFOCUS
Indicates the combo box is about to lose the input focus.

@CBNSETFOCUS
Indicates the combo box has received the input focus.

@CBNDROPDOWN
Indicates the list in a drop-down combo box or drop-down list box is about to open.

@CBNCLOSEUP
Indicates the list in a drop-down combo box or drop-down list box is about to close.

@CBNEDITCHANGE
Indicates the user has changed the text in the edit control of a simple or drop-down combo box. This notification message is sent after the altered text is displayed.

@CBNEDITUPATE
Indicates the user has changed the text in the edit control of a simple or drop-down combo box. This notification message is sent before the altered text is displayed.

@CBNSELCHANGE
Indicates the current selection has changed.

@CBNSELENDOK
Indicates that the selection made drop down list, while it was dropped down, should be accepted.

@CBNSELENDCANCEL
Indicates that the selection made in the drop down list, while it was dropped down, should be ignored.

Using scroll bar controls

About scroll bar

A scroll bar consists of a shaded shaft with an arrow button at each end and a *scroll box* (sometimes called a thumb) between the arrow buttons. A scroll bar represents the overall length or width of a data object in a window's client area; the scroll box represents the portion of the object that is visible in the client area. The position of the scroll box changes whenever the user scrolls a data object to display a different portion of it. Windows also adjusts the size of a scroll bar's scroll box so that it indicates what portion of the entire data object is currently visible in the window. If most of the object is visible, the scroll box occupies most of the scroll bar's shaft. Similarly, if only a small portion of the object is visible, the scroll box occupies a small part of the shaft.

A *scroll bar control* is a control that is separate from the window frame and can be placed anywhere in a dialog or window. A scroll bar control appears and functions like a standard scroll bar, but it is a separate window. As a separate window, a scroll bar control receives direct input focus, indicated by a flashing caret displayed in the scroll box. Unlike a standard scroll bar, a scroll bar control also has a built-in keyboard interface that enables the user to direct scrolling. You can use as many scroll bar controls as needed in a single window. When you create a scroll bar control, you must specify the scroll bar's size and position. However, if a scroll bar control's window can be resized, adjustments to the scroll bar's size must be made whenever the size of the window changes.

Creating the control

Scroll bar controls are created through the dialog editor, manually with the CONTROL statement or by specifying @HSCROLL and @VSCROLL when creating a window.

Scroll bar control styles

The following scroll bat style flags can be specified in the CONTROL statement or by ticking the corresponding check box in the control properties of the dialog editor:

@CTSCROLLHORIZ

Creates a horizontal scroll bar. Automatically set when using the dialog editor.

@CTSCROLLVERT

Creates a vertical scroll bar. Automatically set when using the dialog editor.

@TABSTOP

Moves the input focus between controls if controls are in a dialog. This style has no effect for controls created in a window.

@GROUP

Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the @GROUP style after the first control belong to the same group. The next control with the @GROUP style starts the next group (that is, one group ends where the next begins)

Scroll bar control functions and statements

The following Creative BASIC functions and statements are used to communicate with the scroll bar control.

SETSCROLLRANGE window | dialog, ID, min, max

SETSCROLLRANGE Sets the minimum and maximum range of a scrollbar control to min and max. All values returned by the scrollbar will be between min and max. If ID = -1 then sets the range of the windows horizontal scrollbar. If ID = -2 then sets the range of the windows vertical scrollbar. ID must be a scrollbar control.

GETSCROLLRANGE window | dialog, ID, varMin, varMax

Stores the scrollbars range into the variables specified by varMin and varMax. The variables must be of type INT. If ID = -1 then stores the range of the windows horizontal scrollbar. If ID = -2 then stores the range of the windows vertical scrollbar. ID must be a scrollbar control. :

SETSCROLLPOS window | dialog, ID, position

Sets the slider position of a scrollbar. If ID = -1 then sets the scroll position of the windows horizontal scrollbar. If ID = -2 then sets the scroll position of the windows vertical scrollbar. Any other ID value is a user-defined scrollbar. Position must be between the minimum and maximum values set by the SETSCROLLRANGE statement.

position = GETSCROLLPOS (window | dialog, ID)
Returns the slider position of a scrollbar. If ID = -1 then returns the scroll position of the windows horizontal scrollbar. If ID = -2 then returns the scroll position of the windows vertical scrollbar. Any other ID value is a user-defined scrollbar.

SETCONTROLCOLOR window|dialog, ID, fg, bg
Sets the text color and background color of the edit control. The edit control can only have a single text color.

return = CONTROLEXISTS (window | dialog, ID)
Returns 1 if the control with ID exists in the window or dialog.

ENABLECONTROL window | dialog, ID, 0 | 1
Disables or enables the edit control. If the edit control is disabled no text can be entered and the control will not receive input focus when clicked on. A disabled control will also not accept any new text with the SETCONTROLTEXT statement.

SETFOCUS window | dialog, ID
Gives the control the input focus. The edit control shows the caret.

SENDMESSAGE window | dialog, msg, wparam, lparam ,ID
Sends a message to the control for advanced functionality.

SHOWWINDOW window, flags, ID
Changes the visibility state of the control. Use @SWHIDE to hide the control and @SWRESTORE to show the control

SETSIZE dialog|window, L, T, W, H ,ID
Changes the size of the control. The edit control is redrawn and the text will be formatted to match the new size of the control. The dimensions include the borders of the control.

Messages
Windows sends two messages to indicate the user has performed and action with the scrollbar.

@IDHSCROLL
This message is sent whenever an action has been performed with either the windows horizontal scroll bar or a horizontal scroll bar created with the CONTROL statement. @CONTROLID will contain the ID of the scroll bar control or 0 for a windows horizontal scroll bar.

@IDVSCROLL
This message is sent whenever an action has been performed with either the windows vertical scroll bar or a vertical scroll bar created with the CONTROL statement. @CONTROLID will contain the ID of the scroll bar control or 0 for a windows vertical scroll bar.

Notification messages
Unlike other controls, a scroll bar returns notification messages in @CODE instead of @NOTIFYCODE. Notification messages for scroll bars are sent to inform the program that the user *wants* to perform an action. It is up to the program to scroll the data and position the scroll box (thumb track) of the scroll bar to the correct position,

@SBLEFT
Scroll to the far left. Sent when the user drags the scroll box to the far left

@SBENDSCROLL
End scroll.

@SBLINELEFT
The user clicked the left scroll arrow.

@SBLINERIGHT
The user clicked the left scroll arrow.

@SBPAGELEFT
Scroll one page left

@SBPAGERIGHT
Scroll one page right

@SBRIGHT
Scroll to the far right. Sent when the user drags the scroll box to the far right

@SBTHUMBPOS
Scroll to absolute position. Check @QUAL for the position

@SBTHUMBTRACK
Drag scroll box to a position. Check @QUAL for position

@SBBOTTOM
Scroll to the bottom. Sent when the user drags the scroll bar to the bottom.

@SBLINEDOWN
The user clicks the bottom scroll arrow.

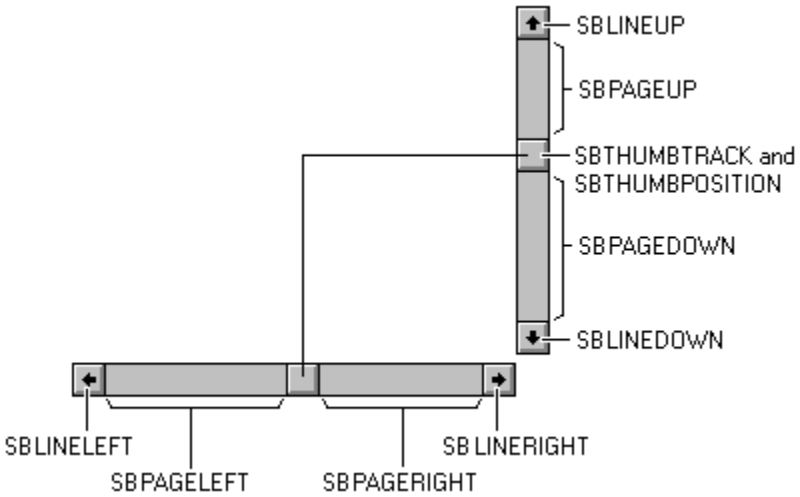
@SBLINEUP
The user clicked the top scroll arrow.

@SBPAGEDOWN
Scroll one page down

@SBPAGEUP
Scroll one page up

@SBTOP
Scroll to the top. Sent when the user drags the scroll box to the top.

The following illustration shows the relationships between the notification codes and the parts of the scroll bars



Using rich edit controls

About rich edit

A "rich edit control" is a window in which the user can enter and edit text. The text can be assigned character and line formatting, and can include embedded OLE objects. Rich edit controls provide a programming interface for formatting text. However, an application must implement any user interface components necessary to make formatting operations available to the user.

Rich edit controls support almost all of the commands and notification messages used with multiline edit controls. Thus, applications that already use edit controls can be easily changed to use rich edit controls. Additional commands and notifications enable applications to access the functionality unique to rich edit controls.

Rich edit controls can save and load text in either straight ASCII format or in rich text format (*.rtf)

Creating the control

Rich edit controls are created in the same manner as the standard control types, either through the dialog editor or manually with the CONTROL statement. Rich edit controls support the same creation flags as standard multiline edit controls except for the @CTLEDITPASS style. A rich edit control can also be created as a single line edit control.

Rich edit control styles

The following rich edit style flags can be specified in the CONTROL statement or by ticking the corresponding check box in the control properties of the dialog editor:

@CTEDITLEFT

Text is left justified in the edit control

@CTEDITRIGHT

Text is right justified in the edit control

@CTEDITMULTI

Designates a multiline edit control. The default is single-line edit control. When the multiline edit control is in a dialog box, the default response to pressing the ENTER key is to activate the default button. To use the ENTER key as a carriage return, use the @CTEDITRETURN style. When the multiline edit control is not in a dialog box and the @CTEDITAUTOV style is specified, the edit control shows as many lines as possible and scrolls vertically when the user presses the ENTER key. If you do not specify @CTEDITAUTOV, the edit control shows as many lines as possible and beeps if the user presses the ENTER key when no more lines can be displayed. If you specify the @CTEDITAUTOH style, the multiline edit control automatically scrolls horizontally when the caret goes past the right edge of the control. To start a new line, the user must press the ENTER key. If you do not specify @CTEDITAUTOH, the control automatically wraps words to the beginning of the next line when necessary. A new line is also started if the user presses the ENTER key. The window size determines the position of the word wrap. If the window size changes, the word wrapping position changes and the text is redisplayed. Multiline edit controls can have scroll bars. An edit control with scroll bars processes its own scroll bar messages. Note that edit controls without scroll bars scroll as described in the previous paragraphs and process any scroll messages sent by the parent window.

@CTEDITCENTER

Text is centered within the edit control

@CTEDITRO

The edit control is read only and text can be displayed but not entered

@CTEDITAUTOH

Automatically scrolls text to the right by 10 characters when the user types a character at the end of the line. When the user presses the ENTER key, the control scrolls all text back to position zero.

@CTEDITAUTOV

Automatically scrolls text up one page when the user presses the ENTER key on the last line.

@CTEDITRETURN

Specifies that a carriage return be inserted when the user presses the ENTER key while entering text into a multiline edit control

in a dialog box. If you do not specify this style, pressing the ENTER key has the same effect as pressing the dialog box's default push button. This style has no effect on a single-line edit control.

@CTEDITNUMBER
Restricts text entered in a single line edit control to numerals only (0 - 9)

In addition to the edit styles the following window styles can also be specified:

@TABSTOP
Moves the input focus between controls if controls are in a dialog. This style has no effect for controls created in a window.

@GROUP
Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the @GROUP style after the first control belong to the same group. The next control with the @GROUP style starts the next group (that is, one group ends where the next begins)

@HSCROLL
The control has a horizontal scroll bar

@VSCROLL
The control has a vertical scrollbar

Controlling the rich edit control.
The rich edit control is accessed through the CONTROLCMD statement/function. CONTROLCMD can be used as a statement or function depending on the command issued.

Clipboard operations

CONTROLCMD window | dialog, ID, **@RTCUT**
Use this command to delete (cut) the current selection (if any) in the rich edit control and copy the deleted text to the Clipboard.

CONTROLCMD window | dialog, ID, **@RTCOPY**
Use this command to copy the current selection (if any) in the rich edit control to the Clipboard.

CONTROLCMD window | dialog, ID, **@RTPASTE**
Use this command to insert the data from the Clipboard into the rich edit control at the insertion point, the location of the caret. Data is inserted only if the Clipboard contains data in a recognized format.

Line operations

count = CONTROLCMD (window | dialog, ID, **@RTGETLINECOUNT**)
Use this function to retrieve the number of lines in the rich edit control

line$ = CONTROLCMD (window | dialog, ID, **@RTGETLINE**, linenum)
Use this function to retrieve a line of text from the rich edit control.
Linenum is the 0 based index of the line to retrieve

line = CONTROLCMD (window | dialog, ID, **@RTGETFIRSTLINE**)
Use this function to retrieve the zero-based index of the uppermost visible line.

line = CONTROLCMD (window | dialog, ID, **@RTLINEFROMCHAR**, index)
Use this function to retrieve the line number of the line that contains the specified character index.
Index is the number of characters from the beginning of the rich edit control. For character counting, an OLE item is counted as a single character

length = CONTROLCMD (window | dialog, ID, **@RTGETLINELENGTH**, index)
Use this function to retrieve the length of a line in a rich edit control. When @RTGETLINELENGTH is called for a multiple-line edit control, the return value is the length (in bytes) of the line specified by index. When @RTGETLINELENGTH is called for a single-line edit control, the return value is the length (in bytes) of the text in the edit control.
Index specifies the character index of a character in the line whose length is to be retrieved. If this parameter is –1, the length of the current line (the line that contains the caret) is returned

CONTROLCMD window | dialog, ID, **@RTSCROLL**, lines, chars
Use this command to scroll the text of a multiple-line edit control. The rich edit control does not scroll vertically past the last line of text in the edit control. If the current line plus the number of lines specified by lines exceeds the total number of lines in the edit control, the value is adjusted so that the last line of the edit control is scrolled to the top of the edit-control window.

Selection operations

CONTROLCMD window | dialog, ID, **@RTGETSELECTION**, varStart, varEnd
Use this command to retrieve the current selection of the rich edit control.
varStart and varEnd must be of type INT. The zero-based index of the first and last characters selected are copied into the two variables. The selection includes everything if varStart =0 and varEnd = -1.

text$ = CONTROLCMD (window | dialog, ID, **@RTGETSELTEXT**)
Use this function to retrieve the text of the selection.

CONTROLCMD window | dialog, ID, **@RTDELETESEL**
Use this statement to delete the current selection. The deletion performed by @RTDELETESEL can be undone by using @RTUNDO

CONTROLCMD window | dialog, ID, **@RTSETSELECTION**, start, end
Use this statement to set the current selection in the rich edit control
start and end are the zero-based character indexes of the selection. If start = 0 and end = -1 then all of the text is selected.

CONTROLCMD window | dialog, ID, **@RTREPLACESEL**, text$
Use this statement to replace the current selection text

CONTROLCMD window | dialog, ID, **@RTHIDESEL**, hide
Use this statement to change the visibility of the current selection. If hide = 1 then the selection is hidden. If hide = 0 then the selection is shown.

Formatting operations

CONTROLCMD window | dialog, ID, **@RTSETDEFAULTFONT**, name$, height, bold, effects
Changes the default font in the rich edit control. The default font is used when no other character formatting has been specified. Height is specified in points. If a bold font is required set bold = 1. Effects can be a combination of @SFITALIC, @SFUNDERLINE, or @SFSTRIKEOUT. Combine effects with the '|' operator.

CONTROLCMD window | dialog, ID, **@RTSETSELFONT**, name$, height, bold, effects
Changes the font of the currently selected text in the rich edit control. If there is no selection the font is changed for all characters entered after the insertion point. Height is specified in points. If a bold font is required set bold = 1. Effects can be a combination of @SFITALIC, @SFUNDERLINE, or @SFSTRIKEOUT. Combine effects with the '|' operator.

CONTROLCMD window | dialog, ID, **@RTSETDEFAULTCOLOR**, color
Changes the default text color of the rich edit control. The default text color is used when no other character formatting has been specified. Color is a color value created with the RGB function.

CONTROLCMD window | dialog, ID, **@RTSETSELCOLOR**, color
Changes the selection text color of the rich edit control. If there is no selection the color is changed for all characters entered after the insertion point.
Color is a color value created with the RGB function.

CONTROLCMD window | dialog, ID, **@RTSETALIGNMENT**, alignment
Changes the alignment of the current selection. If there is no selection the alignment is set for the insertion point.
Valid alignment values are @RTALIGNLEFT, @RTALIGNCENTER and @RTALIGNRIGHT

CONTROLCMD window | dialog, ID, **@RTSETCHAROFFSET**, offset
Character offset, in twips, from the baseline. If offset is positive, the character is a superscript; if it is negative, the character is a subscript. There are 1440 twips in an inch, 20 twips in a point.

Editing operations

CONTROLCMD window | dialog, ID, **@RTUNDO**
Use this statement to undo the last editing operation.An undo operation can also be undone. For example, you can restore deleted text with the first call to Undo. As long as there is no intervening edit operation, you can remove the text again with a second call to Undo.

return = CONTROLCMD (window | dialog, ID, **@RTCANUNDO**)
Use this function to determine if the last editing operation can be undone. Returns 0 if the last operation cannot be undone.

CONTROLCMD window | dialog, ID, **@RTEMPTYUNDO**
Use this statement to reset (clear) the undo flag of this rich edit control. The control will now be unable to undo the last editing operation. The undo flag is set whenever an operation within the rich edit control can be undone.

return = CONTROLCMD (window | dialog, ID, **@RTSAVE**, FILE | STRING, type)
Use this function to save the contents of the rich edit control to an open file or into a string variable.
Type is either 0 to store the contents in plain ASCII format or 1 to store the contents in rich text format (*.rtf). Returns 0 if the operation successfully completed. FILE is a variable of type FILE and must have been successfully opened for writing with the OPENFILE function.

return = CONTROLCMD (window | dialog, ID, **@RTLOAD**, FILE | STRING, type)
Use this function to load the contents of the rich edit control from an open file or a string variable.
Type is either 0 to load the contents in plain ASCII format or 1 to load the contents in rich text format (*.rtf). Returns 0 if the operation successfully completed. FILE is a variable of type FILE and must have been successfully opened for reading with the OPENFILE function.

General operations

return = CONTROLCMD (window | dialog, ID, **@RTGETMODIFIED**)
Use this function to determine if the contents of the rich edit control have changed. Returns 1 if the contents have been modified, 0 otherwise.

CONTROLCMD window | dialog, ID, **@RTSETMODIFIED**, mod
Sets the modified flag of the rich edit control.
Mod can be 0 to reset the flag or 1 to set it.

pos = CONTROLCMD (window | dialog, ID, **@RTFINDTEXT**, text$, start_pos, ignore_case)
Use this function to find text within the rich edit control. Returns the zero-based index of the first character in the control that matches the string in text$. Returns -1 if there are no matches. start_pos specifies a starting character index to begin the search. If ignore_case = 0 then the function performs a case sensitive search, if ignore_case = 1 then the case of the search string is ignored.

length = CONTROLCMD (window | dialog, ID, **@RTGETLIMITTEXT**)
Use this function to get the text limit for this rich edit control. The text limit is the maximum amount of text, in bytes, the rich edit control can accept either through pasting, typing or with the @RTLOAD function.

CONTROLCMD window | dialog, ID, **@RTSETLIMITTEXT**, length
Use this function to set the text limit for this rich edit control. The text limit is the maximum amount of text, in bytes, the rich edit control can accept either through pasting, typing or with the @RTLOAD function. The default limit is 32767 bytes.

CONTROLCMD window | dialog, ID, **@RTSETLINEWIDTH**, width
Use this statement to set the line width for word wrapping in the rich edit control.
Width is specified in twips. There are 1440 twips in 1 inch.

CONTROLCMD window | dialog, ID, **@RTSETMARGINS**, left, right
Use this statement to set the visible left and right margins of the rich edit control.
Left and right are specified in pixels.

length = CONTROLCMD (window | dialog, ID, **@RTGETTEXTLENGTH**)
Returns the text length, in bytes, of the rich edit control.

CONTROLCMD window | dialog, ID, **@RTPRINT** {, margin}
Opens the standard print dialog and prints the contents of the rich edit control. Optional margin value specifies the printer margin in twips. 1440 twips = 1 inch. Default margin is 1/2 inch.

Notification event control

mask = CONTROLCMD (window | dialog, ID, **@RTGETEVENTMASK**)
Returns the event mask of the rich edit control.

CONTROLCMD window | dialog, ID, **@RTSETEVENTMASK**, mask
Sets the notification event mask for the rich edit control. The event mask controls which notification messages are sent to the parent window or dialog in the @NOTIFYCODE variable. Multiple events can be specified by using the | operator The following mask values are available:

@ENMNONE
Only the standard events @ENKILLFOCUS, @ENSETFOCUS, @ENMAXTEXT and @ENERRSPACE are sent.

@ENMCHANGE
The control sends @ENCHANGE events

@ENMUPDATE
The control sends @ENUPDATE events

@ENMSCROLL
The controls sends @ENHSCROLL and @ENVSCROLL events

@ENMREQUESTRESIZE
The control sends @ENREQUESTRESIZE events

@ENMSELCHANGE
The control sends @ENSELCHANGE events

Notification messages
A rich edit controls sends notification messages to the parent window or dialog in the @NOTIFYCODE variable. The ID of the control is found in @CONTROLID the same as standard controls. The following notification messages are supported:

@ENKILLFOCUS
Control has lost input focus

@ENSETFOCUS
Control has been given the input focus

@ENERRSPACE
Control could not complete an operation because there was not enough memory available

@ENMAXTEXT
The limit set by @RTSETLIMITTEXT has been reached,

@ENUPDATE
The contents of the control are about to change
Must be enabled with @RTSETEVENTMASK

@ENSELCHANGE
The current selection has changed.
Must be enabled with @RTSETEVENTMASK

@ENCHANGE
The contents of the control have changed.
Must be enabled with @RTSETEVENTMASK

@ENHSCROLL
The user has clicked the edit control's horizontal scroll bar. Windows sends this message before updating the control.
Must be enabled with @RTSETEVENTMASK

@ENVSCROLL
The user has clicked the edit control's vertical scroll bar. Windows sends this message before updating the control.
Must be enabled with @RTSETEVENTMASK

@ENREQUESTRESIZE
The control's contents are either smaller or larger than the control's window size.
Must be enabled with @RTSETEVENTMASK

Miscellaneous
The rich edit control also supports the following Creative BASIC statements and functions:

SETCONTROLCOLOR window|dialog, ID, fg, bg
Sets the background color of the rich edit control. The foreground color is ignored as the rich edit controls supports individual character formatting.

return = CONTROLEXISTS (window | dialog, ID)
Returns 1 if the control with ID exists in the window or dialog.

ENABLECONTROL window | dialog, ID, 0 | 1
Disables or enables the rich edit control. If the rich edit control is disabled no text can be entered and the control will not receive input focus when clicked on.

SETFOCUS window | dialog, ID
Gives the control the input focus. The rich edit control shows the caret.

SENDMESSAGE window | dialog, msg, wparam, lparam ,ID
Sends a message to the control for advanced functionality.

SHOWWINDOW window, flags, ID

Changes the visibility state of the control. Use @SWHIDE to hide the control and @SWRESTORE to show the control

SETSIZE dialog|window, L, T, W, H ,ID

Changes the size of the control. The rich edit control is redrawn and the text will be formatted to match the new size of the control. The dimensions include the borders, if any, of the control.

Using list view controls

About list view

A list view control is a window that displays a collection of items, each item consisting of a label. List view controls provide several ways of arranging items and displaying individual items. For example, additional information about each item can be displayed in columns to the right of the label. List view controls can be shown in report mode in which case a selectable header is used. The Windows explorer in 'detail' view is one example of a list view control.

Image lists and icon views are not currently supported directly in Creative BASIC. This functionality will be added in a future version. You can create image lists with the Windows API and the DLL comctl32.dll.

Creating the control

List view controls are created in the same manner as the standard control types, either through the dialog editor or manually with the CONTROL statement.

List view control styles

The following list view style flags can be specified in the CONTROL statement or by ticking the corresponding check box in the control properties of the dialog editor:

@LVSALIGNLEFT

Specifies that items are left-aligned in icon and small icon view.

@LVSALIGNTOP

Specifies that items are aligned with the top of the control in icon and small icon view.

@LVSAUTOARRANGE

Specifies that icons are automatically kept arranged in icon view and small icon view.

@LVSEDITLABELS

Allows item text to be edited in place. The parent window must process the LVN_ENDLABELEDIT notification message.

@LVSICON

Specifies icon view.

@LVSLIST

Specifies list view.

@LVSNOCOLUMNHEADER

Specifies that a column header is not displayed in report view. By default, columns have headers in report view.

@LVSNOLABELWRAP

Displays item text on a single line in icon view. By default, item text can wrap in icon view.

@LVSNOSCROLL

Disables scrolling. All items must be within the client area.

@LVSNOSORTHEADER

Specifies that column headers do not work like buttons. This style is useful if clicking a column header in report view does not carry out an action, such as sorting.

@LVSREPORT

Specifies report view.

@LVSSHOWSELALWAYS

Always show the selection, if any, even if the control does not have the focus.

@LVSSINGLESEL
Allows only one item at a time to be selected. By default, multiple items can be selected.

@LVSSMALLICON
Specifies small icon view.

@LVSSORTASCENDING
Sorts items based on item text in ascending order.

@LVSSORTDESCENDING
Sorts items based on item text in descending order.

In addition to the list view styles, the following window styles can also be specified:

@TABSTOP
Moves the input focus between controls if controls are in a dialog. This style has no effect for controls created in a window.

@GROUP
Specifies the first control of a group of controls in which the user can move from one control to the next with the arrow keys. All controls defined without the @GROUP style after the first control belong to the same group. The next control with the @GROUP style starts the next group (that is, one group ends where the next begins)

@HSCROLL
The control has a horizontal scroll bar

@VSCROLL
The control has a vertical scrollbar

Controlling the list view control.
The list view control is accessed through the CONTROLCMD statement/function. CONTROLCMD can be used as a statement or function depending on the command issued.

Statements and Functions

CONTROLCMD window | dialog, ID, **@LVDELETEALL**
Use this statement to delete all items in the list view control

CONTROLCMD window | dialog, ID, **@LVDELETECOLUMN**, col
Use this statement to delete a column in the list view control.
col is the zero-based index of the column to delete.

CONTROLCMD window | dialog, ID, **@LVDELETEITEM**, position
Use this statement to delete a item in the list view.
position is the zero-based index of the item to delete.

CONTROLCMD window | dialog, ID, **@LVINSERTITEM**, position, item$
Use this statement to insert an item into the list view control.
Position is the zero-based index of the item to insert.
Item$ is the text of the item

CONTROLCMD window | dialog, ID, **@LVINSERTCOLUMN**, column, text$
Use this statement to insert a column into list control. The control must have been created in report view.
Column is the zero-based index of the new column
Text$ is the text of the column.

CONTROLCMD window | dialog, ID, **@LVSETTEXT**, item, subitem, text$
Use this statement to set the text of an item or subitem
item is the zero-based index of the item
subitem is the ones-based index of the subitem or 0 to change the item text
text$ is the new text.

text$ = CONTROLCMD(window | dialog, ID, **@LVGETTEXT**, item, subitem)
Use this function to retrieve the text of an item or subitem.
item is the zero-based index of the item
subitem is the ones-based index of the subitem or 0 to retreive the item text

count = CONTROLCMD(window | dialog, ID, **@LVGETSELCOUNT**)
Use this function to retrieve the number of selected items

count = CONTROLCMD(window | dialog, ID, **@LVGETCOUNT**)
Use this function to retrieve the total number of items in the list view control

CONTROLCMD window | dialog, ID, **@LVSETCOLUMNTEXT**, column, text$
Use this statement to change the text of a column in report view.
column is the zero-based index of the column to change.
text$ is the new column text

text$ = CONTROLCMD(window | dialog, ID, **@LVGETCOLUMNTEXT**, column)
Use this function to retrieve the text of a column.
column is the zero-based index of the column to change.

CONTROLCMD window | dialog, ID, **@LVSETCOLWIDTH**, column, width
Use this function to set a columns width
column is the zero-based index of the column to change.
width is the new width of the column in pixels

width = CONTROLCMD(window | dialog, ID, **@LVGETCOLWIDTH**, column)
Use this function to retreive a columns width
column is the zero-based index of the column to change.

CONTROLCMD window | dialog, ID, **@LVSETSELECTED**, item

selected = CONTROLCMD(window | dialog, ID, **@LVGETSELECTED**, item)
Use this function to determine the selected state of the item.
item is the zero-based index of the item.

position = CONTROLCMD(window | dialog, ID, **@LVFINDITEM**, text$)
Searches for an item. Returns the zero-based index of the item or -1 if the item could not be found.
text$ is the case sensitive string to search for.

position = CONTROLCMD(window | dialog, ID, **@LVGETTOPINDEX**)
Use this function to retrieve the zero-based index of the first item visible in the control

Notification messages

A list view control sends notification messages to the parent window or dialog in the @NOTIFYCODE variable. The ID of the control is found in @CONTROLID the same as standard controls. The following notification messages are supported:

@NMCLICK

User has left clicked in the control

@NMDBLCLK

User has double clicked in the control

@NMKILLFOCUS
The control has lost the input focus

@NMSETFOCUS
The control has received the input focus

@NMRCLICK
User has right clicked in the control

@LVNCOLUMNCLICK
Indicates that the user clicked a column header in report view.
@QUAL contains a memory handle to a **NMLISTVIEW** data type.

@LVNKEYDOWN
Signals a keyboard event
@QUAL contains a memory handle to a **LVKEYDOWN** data type

@LVNBEGINLABELEDIT
Signals the start of in-place label editing

@LVNENDLABELEDIT
Signals the end of label editing

@LVNITEMCHANGED
Indicates that an item has changed.
@QUAL contains a memory handle to a **NMLISTVIEW** data type.

@LVNITEMCHANGING
Indicates that an item is in the process of changing
@QUAL contains a memory handle to a **NMLISTVIEW** data type.

@LVNINSERTITEM
Signals the insertion of a new list view item.
@QUAL contains a memory handle to a **NMLISTVIEW** data type.

@LVNDELETEITEM
Signals the deletion of a specific item
@QUAL contains a memory handle to a **NMLISTVIEW** data type.

Data types
Some notification messages set @QUAL to a memory handle of a data type. The following data types are used with list view notifications.

```
TYPE NMLISTVIEW
def hwndFrom:INT
def idFrom:INT
def code:INT
def iItem:INT
def iSubItem:INT
def uNewState:INT
def uOldState:INT
def uChanged:INT
def ptActionx:INT
def ptActiony:INT
def lParam:INT
ENDTYPE

TYPE LVKEYDOWN
def hwndFrom:INT
```

```
def idFrom:INT
def code:INT
def vkey:WORD
def flags: INT
ENDTYPE
```

Reading data types

To copy the memory handle to a data type first declare a user data type and a MEMORY variable, assign @QUAL to the memory variable and use READMEM to load the variable. You should not use FREEMEM with memory variables assigned in this manner.

Example:

```
'standard NM_LISTVIEW
TYPE NMLISTVIEW
def hwndFrom:INT
def idFrom:INT
def code:INT
def iItem:INT
def iSubItem:INT
def uNewState:INT
def uOldState:INT
def uChanged:INT
def ptActionx:INT
def ptActiony:INT
def lParam:INT
ENDTYPE

DEF mem:MEMORY
DEF lv:NMLISTVIEW
...

Handler:
SELECT @CLASS
CASE @IDCONTROL
IF(@NOTIFYCODE = @LVNCOLUMNCLICK)
mem = @QUAL
READMEM mem,1,lv
CONTROLCMD d1,1,@LVSETCOLUMNTEXT,lv.iSubItem,"Clicked!"
ENDIF
ENDSELECT
RETURN
```

Using status window controls

About status windows
A *status window* is a horizontal window at the bottom of a parent window in which an application can display various kinds of status information. The status window can be divided into panes to display more than one type of information..

Creating the control
Status window controls are created with the CONTROL statement. The status window is created with 1 pane by default. The dimensions in the CONTROL statement are ignored as the status window is auto sized to the parents client area.
Example creation statement:
CONTROL win,"SW,,0,0,0,0,0,2"

Controlling the status window control.
The status window control is accessed through the CONTROLCMD statement/function. CONTROLCMD can be used as a statement or function depending on the command issued.

Statements and Functions

CONTROLCMD window | dialog, ID, **@SWRESIZE**
Use this statement to inform the status window that the parent has been resized. A status window automatically sizes itself to the parent windows client area at the bottom of the window. The typical place to use this is in response to the @IDSIZE message in the parent window.

CONTROLCMD window | dialog, ID, **@SWSETPANES**, count, sizes[]
Sets the number of panes and the right edges of the panes.
count is the number of panes to create. sizes[] is an integer array, each element containing the pixel position of the right edge of the pane. If an element of the array is -1 then the corresponding pane extends to right edge of the client area.

CONTROLCMD window | dialog, ID, **@SWSETPANETEXT**, pane, text$
Use this statement to change the pane text. If there is only one pane then SETCONTROLTEXT can also be used.
pane is the zero-based index of the pane to set.

Using Toolbar controls

About Toolbar controls
A *toolbar* is a control window that contains one or more buttons. Each button sends a command message to the parent window when the user chooses it. Typically, the buttons in a toolbar correspond to items in the application's menu, providing an additional and more direct way for the user to access an application's commands.

Creating the control
A toolbar control is created and added to a window with the LOADTOOLBAR function. A bitmap is used as the images for the toolbar buttons and can be loaded directly from a handle returned by LOADIMAGE or from a bitmap resource compiled in the executable. A toolbar has a control ID that allows communicating to the control with the CONTROLCMD statement. The syntax of LOADTOOLBAR:

success = LOADTOOLBAR(window | dialog, resourceID | handle, toolbarID, buttonIDs[] , style)

When a button on a toolbar is pressed Windows will send an @IDCONTROL message to your handler subroutine with the ID of the button contained in @CONTROLID. The variable buttonIDs is an array of type INT that is dimensioned to the number of buttons in the toolbar. Each value in the array corresponds to the ID of a button or 0 for a separator.

The function returns zero on failure or non-zero for success.

The following is an example snippet of loading a toolbar with 7 button images. A separator is used between buttons 3 and 4.

... Assume a open window w1

```
DEF hbitmap:UINT
DEF tbArray[8]:INT
tbArray = 2,3,4,0,5,6,7,8
hbitmap = LOADIMAGE(GETSTARTPATH + "toolbar.bmp",@IMGBITMAP | @IMGMAPCOLORS)

REM this toolbar is loaded from a bitmap handle
IF LOADTOOLBAR(w1,hbitmap,998,tbArray,@TBTOP | @TBFROMHANDLE)
CONTROLCMD w1,998,@TBENABLEBUTTON,3,0
ENDIF
```

In the snippet above we assign an ID of 998 to the toolbar control. After the toolbar is loaded CONTROLCMD is used to disable a the button with an ID of 3.

Toolbar control styles

The following toolbar style flags can be specified in the LOADTOOLBAR function.

@TBTOP

Creates a toolbar that is positioned at the top of the window. This is the default

@TBBOTTOM

Creates a toolbar this is positioned at the bottom of the window

@TBRIGHT

Creates a toolbar that is positioned at the right of the window. Must be combined with @TBNORESIZE. Vertical toolbars cannot be resized automatically with @TBRESIZE. The width and height of a vertical toolbar must be set with the SETSIZE statement.

@TBLEFT

Creates a toolbar the is positioned at the left edge of the window. Must be combined with @TBNORESIZE. Vertical toolbars cannot be resized automatically with @TBRESIZE. The width and height of a vertical toolbar must be set with the SETSIZE statement.

@TBNOALIGN

Prevents the control from automatically moving to the top or bottom of the parent window. Instead, the control keeps its position within the parent window despite changes to the size of the parent window. If the @TBTOP or @TBBOTTOM style is also used, the height is adjusted to the default, but the position and width remain unchanged.

@TBWRAPABLE

Creates a toolbar control that can have multiple lines of buttons. Toolbar buttons can "wrap" to the next line when the toolbar becomes too narrow to include all buttons on the same line.

@TBFLAT

Creates a toolbar with flat buttons. The buttons and toolbar are transparent with this style. Available on systems with Internet Explorer 3.0 and above installed.

@TBLIST

Creates a toolbar with buttons that can have text to the right of the buttons. Available on systems with Internet Explorer 3.0 and above installed.

@TBNORESIZE

Prevents the control from using the default width and height when setting its initial size or a new size. Instead, the control uses the width and height specified in the request for creation or sizing. Control may then be positioned with the SETSIZE statement

@TBFROMHANDLE

Specifies the bitmap is to loaded from a handle returned by LOADIMAGE instead of from resources.

@TBTRANSPARENT

Allows non flat toolbars to have transparent backgrounds. Useable on systems with Internet Explore 4.0 or higher installed.

@TBTOOLTIPS
Allows toolbars to display a tool tip when the cursor hovers over a button. The text for the tool tips is set with the @TBSETTIP command.

Controlling the toolbar control.

The toolbar control is accessed through the CONTROLCMD statement/function. CONTROLCMD can be used as a statement or function depending on the command issued.

Statements and Functions

CONTROLCMD window | dialog, toolbarID, **@TBRESIZE**
Use in response to the @IDSIZE message to automatically change the size of the toolbar to fit the window. This command is ignored if the toolbar is created with the @TBNORESIZE flag.

CONTROLCMD window | dialog, toolbarID, **@TBSETBITMAPSIZE**, width, height
Sets the size of the bitmap images for the buttons. The default size is 16 x 15

CONTROLCMD window | dialog, toolbarID, **@TBSETBUTTONSIZE**, width, height
Sets the size of the buttons. The default size is 24 x 23. The size of the buttons must be at least 7 pixels greater in width and height than the bitmap size otherwise the bitmap images will be clipped.

CONTROLCMD window | dialog, toolbarID, **@TBENABLEBUTTON**, buttonID, enable
Enables or disables the button specified by buttonID. use 1 to enable and 0 to disable the button. A button that is disabled can not be pressed and has a 'grayed out' appearance.

CONTROLCMD window | dialog, toolbarID, **@TBSETLABELS**, strLabels
Sets the button labels used with the @TBLIST and @TBFLAT styles. Strings consist of text labels separated by a bar '|' symbol and ending with two bar '||' symbols. Example:

CONTROLCMD d1, 999, @TBSETLABELS, "New|Open|Save|Cut|Copy|Paste|Print|Help||"

CONTROLCMD window | dialog, toolbarID, **@TBSETBUTTONSTYLE**, buttonID, style
Changes the behavior of a toolbar button. This command is only available on systems that have Internet Explorer 4.0 or greater installed. Style can be a combination of:
@TBBUTTONCHECK - Button stays pressed down until released by clicking on again
@TBBUTTONGROUP - When combined with @TBBUTTONCHECK creates a button that stays pressed until another button in the group is pressed.

state = CONTROLCMD (window | dialog, toolbarID, **@TBGETBUTTONSTATE**, buttonID)
Returns 1 if button is currently pressed. Only applied to buttons with the @TBBUTTONCHECK style.

width = CONTROLCMD (window | dialog, toolbarID, **@TBGETBUTTONWIDTH)**
Returns the width of the toolbar buttons

height = CONTROLCMD (window | dialog, toolbarID, **@TBGETBUTTONHEIGHT)**
Returns the height of the toolbar buttons

CONTROLCMD window | dialog, toolbarID, **@TBSETTIP,** buttonID, tiptext
Sets the tool tip text for a button. Toolbar must have been created with @TBTOOLTIPS.

Design Considerations
Horizontal toolbars are more natural to view then vertical ones. The eyes naturally follow the left to right ordering of a standard toolbar placed at the top of a window. Vertical toolbars tend to be more awkward to use.

MDI windows will automatically resize the client window to fit toolbars and status windows. A regular window places the toolbar in its client area so you need to take the height (width) of the toolbar into account when drawing to the window. Even though a vertical toolbar won't respond to the @TBRESIZE command you should still use CONTROLCMD window | dialog, ID, @TBRESIZE on vertical toolbars in an MDI window. This tells the MDI frame to reserve space for the toolbar after sizing. Use the command after resizing the toolbar with SETSIZE

Toolbars on the bottom and right will flicker slightly when resized. This is due to the toolbar needing to refresh when moved. Consider using the top and right positions.

If you need more than one toolbar at the top you will need to specify @NORESIZE and handle the resizing and positioning manually with the SETSIZE statement. @TBRESIZE only handles single toolbars on an edge.

See Also

The sample programs loadtoolbar.cba and verticaltoolbar.cba

Defining a dialog

Using dialogs in Creative BASIC is similar to defining and opening a window. Many of the same window statements and functions apply to dialogs as well. Dialogs are first defined using the DIALOG statement and then activated elsewhere in your program with the DOMODAL command. The syntax of the DIALOG statement is:

DIALOG variable, L, T, W, H, flags, parent, title, procedure

The variable must have been previously defined as type DIALOG with the DEF statement. 'procedure' refers to the name of the subroutine that will handle messages from this dialog. Procedure is the name of the subroutine that will handle messages from this dialog. Flags should contain @CAPTION for a title and @SYSMENU for the standard system menu and close button. If @CAPTION is omitted the dialog will not have a title bar and will be fixed in place.

It is important to note the difference between a dialog and a window. A window is created and shown with one statement, WINDOW. A dialog is *defined* by the DIALOG statement but not created or shown until you use DOMODAL or SHOWDIALOG. This means a dialog is reusable and can be created and shown many times by just calling DOMODAL or SHOWDIALOG.

Define your dialogs at the beginning of your programs. A common programming error is to try and define the same dialog more than once in a subroutine.

Showing the dialog

Add controls to the dialog after it is defined and show the dialog with the DOMODAL or SHOWDIALOG functions. The DOMODAL function has the syntax of:

DOMODAL variable

The variable must be of type DIALOG and defined with the DIALOG statement. DOMODAL will return the value given to the CLOSEDIALOG statement or @IDCANCEL if the user presses the <Esc> key to dismiss the dialog. All input will be captured by the dialog while it is displayed. Note that any controls in the dialog cannot be changed until the dialog is displayed with the DOMODAL statement. All control definitions should be done in the dialog handler subroutine.

DOMODAL creates the dialog as a modal dialog. This means that all other windows in your program will be blocked until the dialog is closed. To show a non modal dialog use the SHOWDIALOG statement. The syntax of SHOWDIALOG is:

SHOWDIALOG variable

Once the dialog is displayed, your program continues to execute normally.

Closing the dialog:

CLOSEDIALOG variable, return

The return value can be any integer value, it is ignored for non modal dialogs shown with the SHOWDIALOG statement. You can use the predefined values of @IDOK and @IDCANCEL if applicable.

When your dialog is displayed, your program will receive the message @IDINITDIALOG. This is a good place to perform any initializations such as centering the dialog with the CENTERWINDOW statement and presetting any controls.

You should copy any control data before the dialog is closed. When DOMODAL returns all of the controls are invalid and accessing them will generate an error message.

Example:

```
DEF d1:DIALOG
DEF w:WINDOW
DEF result:INT
DEF answer:STRING
'Open our window and define a dialog
WINDOW w,0,0,640,200,@SIZE,0,"Dialog Test",wndproc
DIALOG d1,0,0,100,100,@CAPTION|@SYSMENU,w,"My Dialog",dialoghandler
CONTROL d1,"B,OK,25,75,50,20,@TABSTOP,1"
```

```
CONTROL d1,"E,,15,45,70,14,@TABSTOP,2"
'Show the dialog
result = DOMODAL d1
'Print the result
MOVE w,4,20
IF result = @IDOK
PRINT w, answer
ELSE
PRINT w, "DIALOG canceled"
ENDIF
'Just wait for the window to be closed
run=1
WAITUNTIL run = 0

CLOSEWINDOW w
END

'Our window subroutine
SUB wndproc
SELECT @CLASS
CASE @IDCLOSEWINDOW
run = 0
ENDSELECT
RETURN

'Our dialog subroutine
SUB dialoghandler
SELECT @CLASS
CASE @IDCONTROL
SELECT @CONTROLID
CASE 1
answer = GETCONTROLTEXT(d1, 2)
CLOSEDIALOG d1,@IDOK
ENDSELECT
'All controls should be initialized while processing the @IDINITDIALOG message
CASE @IDINITDIALOG
CENTERWINDOW d1
SETCONTROLTEXT d1,2,"Yipee!"
ENDSELECT
RETURN
```

The dialog can be created without a parent window in which case your program would be a dialog application.