Contents for Flute

Introduction

Menus

The Editor

How To...

Programming Topics

Functions

Copyright Notice

Flute © WorkingTitle, 1992, 1993 1994, 1995.

Flute, CleanSheet, Window Acts and CeSk are trademarks of WorkingTitle.

WorkingTitle@cix.compulink.co.uk

Voice (+44) (0) 1946 590 697

Fax   (+44) (0) 1946 590 565

The Background to Flute

In 1992 WorkingTitle started work on a spreadsheet called CleanSheet. This was to be a ground breaking product in which we were free to redesign any aspect of spreadsheets we saw fit. Naturally this spreadsheet required a scripting language. Just as other spreadsheets spawned their versions of BASIC CleanSheet spawned Flute.

However at the design stage of Flute it was clear that we could either write a large amount of code into CleanSheet and concentrate on automating CleanSheet in isolation, or we could write a system-wide scripting language capable of automating the whole of Windows, that would, as a side effect, automate CleanSheet.

We chose the latter. Flute is primarily a scripting language for Windows, but with a few additional commands to speed transfer of data between Flute and CleanSheet.

Flute is based on our CeSk programming language, developed for CleanSheet and extended to add all the control functions we needed.

Flute Compared to Windows Recorder

1. When you operate a Windows application you really only have two basic actions - you manipulate the mouse, or you manipulate the keyboard.
2. These actions are then handled by each window. For example, pressing the space key on a button, selects the button. It is the *button window* that translates key presses into actions.
3. The effect of selecting the button affects the application. For example selecting the 'Find Next' button on a search dialogue will search for some text.

Windows Recorder records events at the lowest level, level 1. Every tiny movement of the mouse, every mouse click and mouse release, key press and key release are recorded in intricate detail.

OLE Automation is Microsoft's present answer to automation. OLE Automation is a list of commands that control the application making it a level 3 system.

Flute is a scripting language at level 2. It has commands that mimick most of the actions performed by windows; for example, clicking, menu selection etc..

To reliably identify windows Flute uses the name of the window, and where this isn't possible it uses the class name or position.

Flute automates most programs, regardless of any special support they have. It is the nearest we can get to level 2 without being able to extend the Windows operating system.

Flute additionally supports a technique known as window Acts. Acts are special functions that a window class can add to improve Flute's control of that class.

Finally Flute supports OLE Automation and DDEExecute, to control applications at level 3 where an application supports this.

Menu: File

Open	Open a Flute program for editing or running. Only one program can be loaded into one copy of Flute at a time.
Save	Save a program without further prompting. If the program hasn't been named, you will be prompted to name it.
Save As	Save a program under a different name or to a different place onto the hard disk.
Print	Print the Program to the default printer. If a block of text is selected, then only this block will be printed (Even if the block is a rectangular block, selected with the right mouse button).
Printer Setup	Select the destination printer and set the options for each printer.
Exit	Exit Flute, if the program is running, this will also terminate it provided SafeExit is False.

Menu: Edit

Undo	Undo the last change to the edited program
Redo	Redo the last action undone
Cut	Cut out a section of text and place on the clipboard.
Copy	Copy the selected text to the clipboard, leaving the original text in place.
Paste	Paste the text on the clipboard into the text. If any text is selected, it is replaced.
Clear	Clear the selected text, the clipboard is left unchanged.
Find	Search for the specified text within the document. The CeSk programming language is not case sensitive, and hence neither are the search facilities.
Replace	Replace one text string with another.
Check Brackets	Checks brackets within the text to see if they are balanced. If a section of the text is selected, then Check Brackets will only check that section.
Paste Function	Pastes any of the functions or commands and their parameters into your program. This is an easier way to get the parameters of a function correct. See Paste Function
Paste Mouse Action	Paste a mouse command such as Click, Drag etc., into the program. See Paste Mouse Action.
Paste Menu Select	Paste a menu selection command into your program. See Paste Menu Select for details.
Paste Find Window	Paste a FindWindow command into your program. See Paste Find Window for more details.
Paste Window Act	Pastes an Act command into your program. See Paste Window Act.
Paste OLE Automation	
	Pastes an OLE Automation object or function into the text. See Paste OLE Automation.

Menu: General

Run Runs a program. Refer to <u>Running a Program</u> for more details.

Debug Window This shows or hides the Debug Window. The Debug Window allows you to examine variables in the program, to single step and to see how your program is evaluating. See <u>Debugging a Program</u> for more details.

BreakPoint Sets or Clears a Breakpoint on the line of the cursor. If a block of text is selected, then any breakpoints within that block are cleared. When a line containing a breakpoint is encountered during program execution - the program stops and the debugger window is displayed.

Record Actions Records menu selections, keyboard edits and mouse actions automatically as you do them. Refer to <u>Recording Sequences of Instructions</u> for more details.

Button Style Sets the style and type of the button that Flute minimises to. This button can be used to run the attached program. See <u>Flute Buttons</u> for more details.

Font Style Sets the style of the font used to display and edit the program. The font style is stored with each program. See <u>Font Style</u> for details.

Editor Options Sets the tab size and syntax colouring of the editor text. See <u>Editor Options</u>.

Button Bar Switches on or off the button bar. A check mark next to this menu item indicates if the button bar is displayed. See the <u>Flute Button Bar</u> for details.

Recording Sequences of Instructions

To automatically record a sequence of instructions:
- Select 'Record Actions' from the General menu.
- The Flute window will disappear.
- Any editing, manipulation of data or selection will be interpreted by Flute and the corresponding instructions entered into your program at the cursor position.
- To stop recording hold down the `Shift key` and press `Escape`.

Things that Won't Record

Not everything will record correctly, anything time dependent is likely to fail. Consider the example of selecting text in a window, moving the mouse outside the window to scroll, then releasing the mouse button after 1 second of scrolling. This is inherently unreliable in any recording program: how far the text scrolls in that 1 second depends very much on how busy the computer is.

Flute does not attempt to record time, it only records actions. When a Flute program is run the instructions have some built in intelligence to wait while the system is busy. When a program is replayed it is replayed at the maximum speed an application can respond.

When recording selections that require scrolling use the scroll bar. When a scroll bar moves Flute can detect exactly where its new position is and can later mimic the action precisely. This is completely independent of the time spent scrolling.

Generally, if a Flute program is not replaying correctly due to time faults add a <u>Wait</u> instruction to the program.

Running a Program

Once you have a program you want to run select 'Run' from the General menu.
- If the Flute window is open; as each line is executed that line will be highlighted.

Does the program work? If not try to decide why it fails.
- The commonest problem is that the start conditions are not the same.
- The next cause of problems is timing. If an application stops responding Flute will attempt to detect it and wait, but if the application continues to handle messages, and simply ignores them, then Flute will mistakenly believe the program is active. To work around this edit the recorded program and add a Wait instruction into the code.
- Was the edit cursor in the correct place to record? For a program to work the instructions must be inside the curly brackets in the main { } function. If the cursor was not between those brackets when recording then the program will not be correctly placed. Remember - it is the program inside the curly brackets that is executed.

Remember that you do not need to record the program all in one go. You can record fragments of the program, then edit those fragments into a complete program.

See Also

Debugging a Program

Flute Buttons

Introduction

The Flute window can be minimized to a single button on the screen. To do this select Button Style on the General menu and select a button type. Once this is done, whenever you minimize the Flute program window it will minimize to a button not an icon.

Manipulating Buttons

Run   Clicking on the button will run the program contained within the button.

Move   To move a button around the screen hold down the shift key and drag the button with the mouse. The button's position is stored with the program; provided the program is saved afterwards the position will also be saved.

Restore   To get back the program editor window hold down Shift and Control keys and click on the button. The button will disappear and the program window will return.

See Also

Button Style Dialogue

Button Style Dialogue

Introduction

The button style dialogue sets the style of the button that Flute programs minimize down to. Refer to Flute Buttons for more details.

Off   The Flute program window will minimize to an icon on the desktop, rather than a button.

Picture Buttons   Selecting this option gives you a button with the selected icon on its face.

Text Buttons   Enter a line of text. This text will be displayed on the button's face.

Always on Top   The button will always stay on top. Other windows will move behind this button.

Starting Flute Programs Automatically

If you want a program to run automatically when Windows starts you need to place a copy of the Flute icon in the 'Start Up' window of the Program Manager.

- Select the Flute Icon in the Program Manager
- Select Copy
- In the dialogue that appears select 'StartUp'. This will tell the Program Manager that you want a copy in the StartUp window.

The next step is to tell Flute how to start and which program to start with.

- Open the Start Up group window in the Program Manager and select the Flute icon you just created.
- Select 'Properties' from the Program Manager menu
- The command line will contain a command to start Flute, for example C:\Flute\ Flute.exe
- Add a separating space and the name of the program to start with. For example, to start with program 'brthday2.flu' add ' brthday2.flu' to the end of the command line.
- Add the command options chosen from the list below. Each command option should be separated by a space.

-h   Hide the Flute Window. The Flute window will not be seen and cannot be modified.

-r   Run the program. The default action for a program is to load it for editing. However, if this option is specified, then the program will be run immediately after it is loaded.

-b   Minimized the Flute Window to a button. (For this to work you must have selected a button on the Button Style dialogue). With this option selected the Flute Window never appears, only the button.

-e   This is similar to -r. The program is run immediately after it is loaded, but, in addition, when the Flute program terminates, Flute exits.

Example:

Command Line: C:\Flute\Flute.exe setprog.flu -r -h

This will load the program setprog.flu, run it and hide the Flute window. The program will run in the background. If it does not exit the only way to get rid of it is to exit Windows.

Embedding Flute Programs in Other Applications

A Flute program can be embedded in another application and run simply by double clicking on it. This works if the application supports embedded packages. An embedded package is a file usually represented by an icon. You may find this useful when you have a Flute program that is specific to a particular document.

- Start the File Manager
- Drag the file you want to embed from the File Manager to the destination document.
- If the program supports embedded packages the Flute program will appear as an icon inside the document.
- When you want to run the program double click on it. Flute will start, load the program, execute it, and exit immediately after the program finishes.

The Flute Button Bar

The button bar allows quick access to the most commonly used menu items.

New Document, See <u>Menu: File</u>

Open a Program, See <u>Menu: File</u>

Save Program, See <u>Menu: File</u>

Print the program or selected text, See <u>Menu: File</u>

Cut selected section to the clipboard, See <u>Menu: Edit</u>

Copy the selected text to the clipboard, See <u>Menu Edit</u>

Paste the clipboard into the program, See <u>Menu: Edit</u>

Record action, See <u>Recording Instructions</u>

Display the Debug Window, See <u>Debugging a Program</u>

Run the Program, See <u>Running a Program</u>

Sets or clears a breakpoint, See <u>Debugging a Program</u>.

Paste a menu command into the program, See <u>Menu: Edit</u>

Paste a FindWindow function into the program, See <u>Paste Find Window</u>

Paste a mouse command into the program, See <u>Paste Mouse Action</u>

Paste a function into the program, See <u>Paste Function</u>

Paste an Act command. See <u>Paste Window Act</u> for further details.

Paste an OLE Automation function. See <u>Paste OLE Automation</u>.

Provides help on the topic selected within your program. If no topic is selected this displays a list of available topics.

The button bar can be dragged into position at any edge or any corner of the Flute edit window. It can also be free standing.

The button bar can be switched on or off using the 'Button Bar' menu option on the 'General' Menu.

The Basics of CeSk

The programming language within Flute is called CeSk (pronounced C-eSk). It is very similar to the programming language 'C' in its syntax but with a few exceptions. Where these occur the difference is noted.

CeSk is the programming language embedded within all WorkingTitle products and is essentially identical to the programming language found inside the Computer Object within CleanSheet - our spreadsheet.

There are some general rules you should familiarize yourself with:
- CeSk is not case sensitive. Commands can be entered in upper or lower case, e.g., `Make` is the same as `make`.
- To separate commands a semicolon is used. e.g. a:=3; b:=4; c:=5;
- Sections of commands are specified using curly bracket to group the commands together, e.g., IF (a=2) { a:=3; b:=5; };
- Comments can be placed anywhere by enclosing them in **/*** and ***/** , e.g. `IF (a=2 /*  if the count has reached 2 */)  { a:=3; b:=5; };`
- // indicates that comments follow to the end of the line. E.g.

```
    main {        // comments follow here
          a:=10;

  };
```

The main function within a CeSk program is called **main{}**. When a program is run the function **main{}** is called regardless of whether there are any functions above it.

A simple program to close the Notepad window might look like this:

```
main {
    closewindow{"notepad"};
};
```

Important Topics

Vague Logic

CeSk contains fuzzy logic facilities - it has many *vague operators* which can compare objects that are dissimilar.

This is necessary as so many objects cannot be compared directly. We decided to implement a systematic strategy for handling object comparisons.

All vague operators are based on the Compare function, this is the root fuzzy logic operation.

[>=<,>==<,>===<,>====< Vague Equalities](#)

[<=>,<==>,<===>,<====> Vague Inequalities](#)

[>>, >>>, >>>>, >>>>> Vague Greater Than](#)

[<<,<<<,<<<<,<<<<< Vague Less Than](#)

[Compare](#)

Data Object Types

CeSk is an object based language. This means that it can handle more than the limited numbers and strings that, for example, the language BASIC handles.

There are many functions and commands whose action vary according to the objects they are used with. For example, if you add two numbers together the result is the sum of the numbers. If you add a date object and a number together, the result is the date with the equivalent number of days added.

Differing data types are specified in different ways. For example, a date is entered by preceding it with a backslash,   such as \2nd May 1992.

The full list of available objects is given below:

Type	Description
0	Null Object, i.e., Nothing
1	Integers, ranging from -2147483678 to +2147483647
2	Floats; A floating point number, having 6 significant decimal places and a range of -1E-38 to 1E38.
3	Reals; A double length floating point number, having 15 internally significant decimal places and a range of -1E-308 to 1E308.
4	A String of characters; the maximum length is 32767 characters.
5	Boolean Logic TRUE/FALSE.
6	An Error object - used to signal error conditions.
7	A Date object.
8	A Time object.
9	A Complex number.
10	An Equation.
11	An Array object.

The function 'Type' can be used to obtain the type number of an object. For example, `type(\22nd May 90)` returns the number 7, because date objects are type 7.

Entering Objects

The following list shows how these objects appear inside your program:

Null	A Null (Nothing) object
3@	Integers have an @ after them
3.42	Numbers without anything after them are double precision numbers

12:05	Numbers containing colons are Time objects
\14th Feb 1990	Dates have a backslash in front of them
"Strings"	Strings of text are enclosed in the double quote character
True	Boolean True
False	Boolean False
3+2i	Complex Numbers have an i, or j after the imaginary part
{1,2,3}	Arrays are enclosed in curly brackets

The Null Object

Null is the absence of other data object. When a pipe is empty, or a function returns nothing, it actually returns a NULL object.

When writing programs any variable that has not been initialised contains a NULL.

Think of NULL as the absence of other objects.

Integers Objects

Integers are whole numeric values (no decimal point) within the range -2147483678 to +2147483647.

To enter an integer into CeSk type the number followed immediately by an @ symbol.

For example 3@ is the integer 3. Similarly 3.4@ is truncated to the integer 3 and stored as an integer.

If you omit the @ then the number you type is taken as a double real number. This is true even if the number has no decimal places; 3 is a double, 3@ is an integer.

Integer objects may also be used to hold a sequence of (up to) four characters, i.e.,

'A' is an integer (value 65) representing 'A'

'AB' is an integer (value (65 * 256) + 66) representing 'AB'

Single Precision Numbers (C Type Floats)

Single Precision Numbers are floating-point values, to 6 significant decimal places, within a range of -1E-38 to +1E+38.   They cannot be directly entered into CeSk, but can be produced by 'Make' and some other functions.

Single Reals are a much more compact form of floating point number and hence their main use is for storage where the space consumed is more important than the precision.

Double Precision Numbers

Double Precision Numbers (called Doubles for short) are the normal type of number objects within CeSk. Values are held to 15 significant decimal places within the range -1E-308 to +1E308.

Examples

3.45

15.29e+3   The number 15290.

3   The double 3.0000000. To specify an integer you must use the @ symbol.

When you enter a number into CeSk if there is no qualifying symbol after it (such as @ for integers, colon for Time objects etc.), then it is assumed to be a Double Precision Floating point number.

Strings

Strings hold a sequence of characters up to a maximum length of 32767 characters.   A string is identified by enclosing it in *double* quotes e.g. "This is a sample string".

Note that characters surrounded by *single* quotes are converted to integers

Contrast: 'a'   The Number 96 representing character 'a'.

with "a"   The String, consisting of the letter "a".

See Also

Overdimensioning

Boolean Objects

Boolean items are logical TRUE or FALSE objects; these are entered into a program as the names 'FALSE' and 'TRUE'. They are also returned as the result of comparisons. For example 3>2 returns a Boolean TRUE.

Error Objects

Error objects are returned by CeSk when an expression cannot be evaluated. For example the expression

`\12th Jan 1992 + "This is a test string"`

will produce a result

`Q007 Error: Attempt to add mismatched objects type 7 and 4.`

You can use the function 'Type' which returns the type of an object to detect errors within your programs. Errors are type 6.

Some errors cause programs to stop. These Run Time errors represent faults within a program, while the less serious represent faults within a calculation.
An error will cause a program to stop if it begins with an 'R'.

You can create your own errors within programs using the Make{} function, but remember, if they begin with an 'R' then they will terminate your program.

See Also

Predefined Error Codes

Overdimensioning

Date Objects

Date objects are, quite literally, dates. A date object is declared by starting a constant with the '\' character (note that this is a backslash character rather than a forward slash - top left to bottom right). CeSk will then do its best to interpret the characters that follow as a valid date. For example

Correct:

```
\12th Jan 1966
```

```
\12 Jan 66
```

```
\12/1/66          In the UK English edition this is treated as 12th Jan 1966
```

```
\1/14/66          Because 14 can only be a day of the month, it is taken as 14th Jan 66
```

When parts of the date are missing the current system date is used to complete it. For example \3rd September would be interpreted as 3rd September 1994 if typed during the year 1994.

Some things are unacceptable within the date formats.
A dash (-) is not allowed since subtraction is a valid operation on date objects. A dot is not allowed as a separator because it is used internally by CeSk. Backslash is not allowed since this is used to specify a date object.

Incorrect:

```
\12\1\66          Incorrect due to the use of backslash.
```

```
\12-1-66          Incorrect due to the use of dash.
```

```
12/1/66           Missing backslash at start of date.
```

See Also

Overdimensioning

Time Objects

A time object is used to store a time value, in HH:MM:SS form.   It is entered by separating the three items with the ':' (colon) character.   If the seconds field is omitted a default value of 00 seconds is used. The entry may be followed with an 'am' or 'pm' indicator if a twelve hour format is required, otherwise a 24-hour clock is assumed.

Example

12:03

12:05:04

11:05am

11:06pm

Note that the am / pm immediately follows the numbers without any separating spaces.

See Also

Overdimensioning

Complex Numbers

Complex numbers are simply entered as the real+imaginary values as one composite item. To enter a complex number in CeSk the imaginary part must immediately be followed by an i or j. For example 3+2j is a complex number, but 3+2   j is incorrect, due to the space between the 2 and j.

They can also be reversed 2j+3 is the same as 3+2j, is the same as 3+2i . The **i** is usually used in mathematics, whereas the **j** is usually used in engineering notations.

Complex numbers can also be returned by expressions that evaluate to the complex domain. For example sqrt(-4) returns the complex number 0+2i.

See Also

Overdimensioning

Equation Objects

Equation objects are used by CeSk to define program code and equations, and are stored in a special tokenised form.

Equation objects cannot be directly entered into CeSk. Instead they are returned as results by some of its functions (mainly the Make{} function and FitLine{})

Note: an equation object is not the same as an equation entered in the editor. Naturally you can type equations directly into your program, you cannot however create an *equation object* by directly typing one in.

Arrays

An array is a group of other CeSk data objects, such as numbers, strings, dates etc.. These objects can be mixed freely. In other words you are not restricted to having only numbers in an array, or only dates.

All arrays in CeSk are zero based, i.e. the first entry is entry zero.

The array object can comprise of an array of single types of a base object, or any combination of base objects stored within particular elements,

Example

  MyInts[0] = 1@   ; here MyInts is an array of two integers

  MyInts[1] = 2@

while

  Composite[0] = "hello"   ; is a three element array comprising a string,

  Composite[1] = 123 + 24j   ; a complex number,

  Composite[2] = {1234,8279} ; and an array of 2 numbers.

Notice that the array 'Composite' is an array of 3 elements, the third of which is itself an array. Each element is independent of the others.

Arrays can have up to 10 dimensions, A[1][2][3][4][5][6][7][8][9][10].

Array Constants

Constant arrays are created by enclosing the list of elements within {} characters. For example,

  {1,2,3,4,5}   ; Is a one-dimensional array with 5 elements

  {{1,2},{3,4}}   ; Is the two-by-two array

Note that the first dimension of the array runs across, and the second dimension runs down.

You should be careful to distinguish between the use of {} when initialising arrays, and when accessing sub-sections of an object by overdimensioning (see overdimensioning later).

1	2
3	4
5	6

The above table is {{1,3,5},{2,4,6}} in array notation.

Indexing into Arrays

Indexing into an array can be specified in several forms. For, say, the 2 by 2 variable X, containing the array {{1,2},{3,4}}, access to individual items can be done as follows:

X[0]   ; returns the list {1,2}, while

X[1][1]   ; returns the value 4, and

X[1][0][1]   ; returns the value NULL (since nothing has been assigned here)

Note this last example also has the implicit action of promoting the array element x[1][0] into an array itself. After you have requested element X[1][0][1], X[1][0] becomes an array of 2 elements, the first of which is   the number 3. The action of referring to a variable or part of a variable as though it were an array *promotes it into being an array*.

Further Example:

Y:={{"A Sample","Array"},{"Of Strings"}};

Therefore:

Y[0] is the array {"A Sample","Array"}

and hence

Y[0][1] is the string "Array".

The index into the array can be given in several styles.   You can:
- Use separate parameters for each index, e.g. X[1][2]
- Use an array list within {}'s, e.g. X[{1,2}] is equivalent to X[1][2].
- Use complex numbers to index into arrays e.g. X[3+2i] is equivalent to X[3][2]
- Use time objects to index into arrays e.g. X[12:03:50] is equivalent to X[12][3][50]

These methods of access can be used in any combinations of styles you like, to gain access to a given element of an array, e.g.

array[1][2][3]   ; returns the same as

array[1][{2,3}]   ;

array[{1,2,3}]   ;

array[{1,2}][3]   ; etc. etc.

Implicit Promotion of Arrays

Simply by implying that a variable is an array you promote it to an array.

Simply indexing into a variable within an expression will cause that variable to be promoted.   For example, given the simple integer variable 'J', initialised with the instruction:

    J := 3@

The following all promote J to an array:

J[2] := 5@   ; promotes to the list {3,NULL,5}

J[0]   ; promotes to the list {3}.

J[1] := "Smith"   ; promotes to the list {3,"Smith"}

while

J[2][1] := 6   ; promotes to the array {3,NULL,{NULL,6}}

Note the second example carefully. 'J[0] '. This doesn't assign anything to the variable J, but simply by referring to J as though it is an array implicitly makes it into one.

Overdimensioning

For some data objects part of the data can be extracted separately using a technique known as overdimensioning. This is done by placing a value in curly brackets { } immediately after the object to select just a subset of the object in question. This is best explained by examples:

Given the constant string "Mr Smith grows day by day", then using overdimensioning as follows

"Mr Smith grows day by day"{3}

　　　; returns the 'S' of Smith, while

"Mr Smith grows day by day"{15}

　　　; returns the 'd' of   the first 'day'

Overdimensioned access can only be used on certain types of   objects - objects that have multiple parts.

Note that all the items extracted via overdimensioning are returned as a double floating point number   - this should be particularly noted when using overdimensioning on strings - the first example doesn't actually return letter 'S' it returns 83.0 the *code* for 'S'.

Complex - Overdimensioned complex numbers return either the real or imaginary part depending on the value used i.e.,

　Complex := 123 + 4j

　Complex{0}　　　　; returns the real part (123)

　Complex{1}　　　　; returns the imaginary part (4,   NOT 4j)

Dates - Overdimensioned dates return the day, month and year from items 0, 1 and 2 respectively, where day is 1 through 31, month 1 through 12 (Jan is 1), and year is the complete century and year (e.g. 1993).

　bd := \3rd Jan 67　　; initialise bd with a date

　bd{0}　　　　　; returns 3 - the day of month

　bd{1}　　　　　; returns 1 - month of year

　bd{2}　　　　　; returns 1967 - Year

Times - Overdimensioned times return hours, minutes and seconds as items 0, 1 and 2.

　Current :=09:11:18　;Assign a Time object to 'Current'

　Current{0}　　　　; returns 9 Hours

　Current{1}　　　　; returns 11 Minutes

```
Current{2}            ; returns 18 Seconds
```

Strings - Overdimensioned strings return a (single) character from the string, e.g.

```
MyString := "Four score years and ten"
```

```
MyString{6}                    ; returns the character 'c'
```

```
MyString{0}                    ; returns 'F'
```

Strings are based on the ANSI character set, thus the above would return 99 as the value. If you use a value as the overdimension argument that is greater than the current string length, then the string is extended (by adding spaces) to the required length   e.g.

```
MyString := "Hello"   ; if current length is 5 characters
```

```
MyString{10}         ; then access beyond string length...
```

```
sets MyString equal to "Hello<5 spaces>"
```

This is another example of implicit promotion of variables. By implying that the string is longer that it really is you extend the string.

Sections of a string may be replaced by assigning a value to an overdimensioned access to a string, e.g. given

```
MyString := "This is a T"
```

then

```
MyString{10} := 83              ; sets MyString to "This is a S", and
```

```
MyString{2} := "at"             ; sets MyString to "That is a S", while
```

```
MyInt := 87@                    ; Value for 'W'
```

```
MyString{0} := MyInt            ; sets MyString to "What is a S", and
```

```
MyString{11} := "ponge"         ; sets MyString to "What is a Sponge"
```

Equations   - Equations that have been returned by a function   can also have overdimensioning applied to them. However, this will return the byte token from the equation in CeSk internal format.

Function Arguments

Functions within CeSk always take just one parameter as an argument. This may at first seem like a serious limitation. However the single parameter can be any type of CeSk data object - including an Array.

Functions that require more than one piece of data accept an array as input. Using the array system means that any number of parameters can be passed or even *variable* numbers of parameters.

- When a function requires just one parameter the single data object is passed to it.
- Where functions require more than one parameter an array of parameters is passed.

For example, the function EXP (exponential) requires just 1 parameter; hence

`EXP(2.3)`, which returns 9.974

Note the rounded brackets, indicating that EXP is a function taking just one parameter. These are not obligatory, but make it easier for you to follow the declaration e.g. EXP(2.3) = EXP 2.3.

The function ATAN2, however, requires two parameters. These must be passed as a single array, e.g. given MyArray := {23,12}, then ATAN2({23,12}) = ATAN2{23,12} = ATAN2 MyArray

:= Assign to Variable

Assigns an object (a piece of data) to a variable name.

The variable is just a place holder. It has no type of its own; it takes the type of the variable it contains.

Example Assignments

```
MyString := 1@;          Assigns the Integer 1 to the variable MyString

MyString := "Hi There"   Assigns the String "Hi There" to the variable MyString

MyString := {1,2}        Assigns an Array to MyString. Note that this example implicitly
                         defines an array object.
```

Promotion during Assignments

You can assign to part of a variable using the assignment operator. This effectively promotes it to an array if it is not already one.

MyVar := {"hello",34.5}	Creates an array of two elements in which MyVar[0] is equal to "hello", and MyVar[1] is equal to 34.5.
MyVar := "hello"	Assign the string object "hello" to 'MyVar'. This replaces the previous contents of MyVar.
MyVar[1] := 34.5	Implicitly converts MyVar to an array, keeping the original value as element [0]. By treating an object as an array you implicitly promote it to an array.
MyVar[1][3] := 10	Adds a further dimension to MyVar, converting element [1] into an array in its own right i.e. MyVar[1] becomes the list {34.5,null,null,10}

The whole of MyVar now contains {"hello",{34.5,null,null,10}}

Replacing Sections of Variables using :=

An assignment may also to used to replace a value or section of a string in a similar manner to the way Overdimensioning extracts parts of objects.

Example:

MyDate := \12th Jan 1966	Assigns the date \12th Jan 1966 to the variable MyDate
MyDate{0} := 10	Then assigns 10 to replace part 0 - the Day of Month part. After this is done MyDate now contains 10th Jan 1966 as a date.

Example:

MyString := "hello how"	Assigns the string "hello how" to the variable MyString.
MyString{6} := "this is an example"	

Assigns the string "this is an example" to character 6 onwards in the original string.

MyString now contains "hello this is an example"

+=, -=, *=....Complex Assign to Variable

Complex assignments are contractions of an operator and an assignment. a/=2 expands to a:= a / 2 conversely a := a + 2 can be written as a += 2.

If MyArrayA := {{2,3},{4,5}} and MyArrayB := {2,4},

then MyArrayA := MyArray A * MyArrayB

 can be written as

MyArrayA *= MyArrayB

The full list of complex assignments is:

#~=	Bitwise XOR
^=	Raise to the Power
#&=	Bitwise AND
#|=	Bitwise OR
#>=	Right Shift
#<=	Left Shift
+=	Add
-=	Subtract
*=	Multiply
/=	Divide
?=	Modulus

User Functions

Functions may be used to encapsulate commonly used sequences of code, allowing a simpler structure and easier flow of logic for the whole program.

They are defined by simply following a (non-reserved) name with curly brackets { }, which enclose the code for the function, i.e.

```
main { };          - is just an empty function called 'main'

add3{return(ip+3);};
```

- is a function called 'add3', which takes a single passed value of whatever object type, adds three, and returns the result. Hence a numeric will return its current value + 3, a date object returns the date 3 days beyond the passed date, and so on.

The order the functions are placed within a file is unimportant.
Functions may call themselves recursively, or other functions as required; the nesting limit for function calls is dependent only on the available stack space within the computer.
Functions can accept only 1 argument and return 1 result. This may seem like a serious limitation, but the argument can be an array, and similarly the result can be an array.
Thus an argument list could contain any of the CeSk object types, either as single values, lists or arrays of the same object types, or lists or arrays of disparate objects,
e.g. MyFunc {123,"This is a string",{2+3j,4+6i},{{1,1},{2,2}},\26th Jan 62}
calls the function MyFunc with a whole range of different parameters, passed as a single array.
All of the passed parameters are assigned to a local function variable called 'ip' (short for input) - if the function takes no parameters then ip will contain a null object. Using the above example, ip[0] refers to the value 123, ip[2][1] refers to the complex number 4+6i, and so on. Hence, using the appropriate intrinsic functions, a user function can customise itself to deal with any form or number of passed arguments.

Most variables are only valid within the function they are used in, this keeps each function separate. Refer to <u>Variables and Their Scope</u> for more details.

Operators

The following list covers the major operators within the CeSk programming language.

+ Addition Operator

- Subtraction Operator

* Multiplication Operator

/ Division Operators

% Percentage Operator

? Modulus Operator

^ Raise to the Power

#& Bitwise AND

#| Bitwise OR

#~ Bitwise XOR

#> Bitwise Right Shift

#< Bitwise Left Shift

! Factorial

+ Addition Operator

The addition operator's exact action depends on the objects it operates on. The following gives an exact list, but you will generally find that addition operates sensibly;   it behaves as you would expect addition to behave.

Null + Any	Adding NULL to an object returns the object, e.g., 1 + NULL returns 1, "Smith" + NULL returns "Smith"
Integer + Integer	Adding a integer value to an integer returns an integer, e.g. 1@ + 2@ returns 3@
Integer + Float	Adding an integer to a single real value, or a single real value to a single real value returns a single real value, e.g. 3@ + 0.142 returns 3.142, 6.234 + 1.766 returns 8.0
Integer + Double	Adding an integer, single real, or double real value to a double real value returns a double real value, e.g., if Pi := 3.141592654, then Pi + 3@ returns 6.141592654, Pi + 3.142 returns 6.283592654, while Pi + Pi returns 6.283185307.
Integer + String	Adding an integer to a string will add the character represented by the value (mod 256) to the start or end of the string, i.e. 65 + "Hello" returns "AHello", while "Hello" + 65 returns the string "HelloA".
Time + Numeric	Adding an integer, single real, or double real value to a time object returns the time object plus or minus the value in seconds, e.g. 10:23:59 + 200@ returns 10:26:19.
Complex + Numeric	Adding an integer, single real, double real, time or complex number to a complex number returns a complex number, e.g. 10+2j + 10@ returns 20+2j. If a time object is used, then the time object is converted to an absolute number of seconds since midnight and then added as a number, i.e. MyComplex + 10:24:04pm returns 80654+2j (80644 seconds since midnight)
Boolean + Numeric	Adding a Boolean object to any object (except NULL) is equivalent to adding the value 0 (for FALSE) or 1 (for TRUE).
String + String	Adding a string object to a string object returns the concatenated strings, e.g. "Hi " + "There" returns "Hi There".
Date + Numeric	Adding an integer, single real, or double real value to a date object returns the date object plus N days, e.g. \12th Jan 74 + 12 returns \24th Jan 1974
Error + Any Object	Adding an error object to any other object returns the error object.

Array Objects

Adding an object to an array object is equivalent to adding the object to each of the array elements in turn.   When adding two array objects each dimension is treated separately; for arrays of the same size and dimensions corresponding values are simply added for each element.   For example

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} + 1@ = \begin{pmatrix} 2 & 4 \\ 3 & 5 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} + \{2,3\} = \begin{pmatrix} 3 & 3 \\ 5 & 4 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} + \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix}$$

Note that arrays may not just contain numbers; they could hold string elements or even whole lists or arrays.

- Subtraction

The subtraction operator's exact action depends on the objects it operates on. The following is an exact list. This list is similar to addition but note that subtraction is not commutative; a - b is not the same as b - a, and similarly, Date - Number is not the same as Number - Date.

Any - Null	Subtraction NULL from an object returns the object unchanged., e.g., 1 - NULL returns 1, "Smith" - NULL returns "Smith"
Integer - Integer	Subtracting an integer value from an integer returns an integer, e.g. 1@ - 2@ returns -1@
Integer - Float	Subtracting an integer from a single real value, or a single real value from a single real value returns a single real value, e.g. 3@ - 0.142 returns 2.858, 6.234 - 1.766 returns 4.468
Integer - Double	Subtracting an integer, single real, or double real value from a double real value returns a double real value, e.g., 3.141- 3@ returns 0.141.
Time - Numeric	Subtracting a number from a Time object, treats the number as seconds, and adjusts the time accordingly. e.g. 12:15:20 - 80@ return 12:14:00.
Complex - Numeric	Subtracting an integer, single real, double real, time or complex number from a complex number returns a complex number, e.g. 10+2j - 10@ returns 0+2j.
Boolean - Numeric	A Boolean object is treated as a number of value 0 (for FALSE) or 1 (for TRUE).
Date - Numeric	Subtracting an integer, single real, or double real value from a date object returns the date object plus the corresponding number of days, e.g. \12th Jan 74 - 3 returns \9th Jan 1974
Date - Date	Subtracting one date from another returns the number of days between them.
Time - Time	Subtracting one Time from another returns the number of seconds between them.

Other conversion return errors. For example, it is not acceptable to subtract a date from a number since there is no sensible value such an operator could return.

Array Objects

Subtracting an object from an array object is equivalent to subtracting the object from each of the array elements in turn.   When subtracting two array objects, each dimension is treated separately. For arrays of the same size and dimensions corresponding values are simply subtracted, element - by - element. For example

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} - 1@ = \begin{pmatrix} 0 & 2 \\ 1 & 3 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} - \{2,3\} = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} - \begin{pmatrix} 1 & 3 \\ 2 & 4 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

Note that arrays may not just contain numbers, they could hold string elements, or even whole lists or arrays within each element being looked at.

* Multiplication Operator

The exact operation of the multiply operator depends on the objects being multiplied. The list below gives the sensible actions that this operation performs.

Numeric * Numeric	Multiplying Numeric types returns the product (the multiplication result) as the higher type of object. For example 2@ * 3.2 returns 6.4 as a double precision number. (Integer * double returns a double because it has more precision and hence is a higher type of object).
Null * Any Object	Multiplying any object, except an error object, by a NULL object returns a NULL object. An error object will be returned unchanged.
Error * Any Object	Multiplying an Error by anything returns the Error unchanged.
String * Numeric	Multiplying a string object by a value returns the string replicated N times, i.e. "Hello" * 3 returns "HelloHelloHello".
Any * Boolean	Multiplying any object by a Boolean object returns the object unchanged if the Boolean value is TRUE, otherwise a NULL object is returned. For example: "abc" * TRUE returns "abc"
Complex * Complex	Multiplying two complex numbers returns the complex product, e.g. (2+3j) * (3+3j) returns -3 + 15j. Note that the preceding expression needs to be bracketed since multiplication has a higher precedence than addition. Without the brackets 2+3j * 3+3j would be evaluated as 2+(3j*3)+3j.
Array * Array	Multiplying arrays; multiplies the corresponding objects in each array. For example {100,200,300} * {2,3,4} produces an array {200,600,1200}. Similarly 100*{1,2,3} produces an array {100,200,300}.

Example

If A is a two dimensional array with the first column containing the price of products and the second column containing the quantity sold {A[0]*A[1]} produces a column array containing the value of the stock sold.

A[0] is the 1 dimensional array {4.5,6.7,5.5,2,6}

A[1] is the 1 dimensional array {1,3,6,3,2}

{A[0] * A[1]} is the 2 dimensional array {{4.5,20.1,33,6,12}}

/ Division Operator

The exact operation of the division operator depends on the objects being divided. The list below gives the sensible actions that this operation performs.

Unlike Multiplication division is not commutative (a/b is no the same as b/a), hence the order of the operands is important.

Numeric / Numeric   Dividing Numeric types returns the quotient (the result) as the higher type of object. For example 2@ / 3.2 returns 0.625 as a double precision number. (Integer / double returns a double because it has more precision and hence is a higher type of object).

Error / Any Object   Dividing an Error by anything return the Error unchanged.

Complex / Complex   Division of two complex numbers returns the complex quotient, e.g. (2+3j) / (3+3j) returns 0.8333333+0.16666666j. Note that the preceding expression needs to be bracketed since division has a higher precedence than addition. Without the brackets 2+3j / 3+3j would be evaluated as 2+(3j/3)+3j.

Array / Numeric   Dividing an Array by a Numeric divides all elements of that array separately. e.g.

$$\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} / 2 = \begin{pmatrix} 1 & 1.5 \\ 2 & 2.5 \end{pmatrix}$$

Array / Array   Divides corresponding elements of the arrays e.g. $\begin{pmatrix} 2 & 3 \\ 4 & 5 \end{pmatrix} / \begin{pmatrix} 2 & 2 \\ 4 & 10 \end{pmatrix} = \begin{pmatrix} 1 & 1.5 \\ 1 & 0.5 \end{pmatrix}$.

% Percentage Operator

This operator is exactly equivalent to dividing by the number 100. It is mainly used to enter percentages into a worksheet.

Example

50% is 50/100 is 0.5

For an full list of actions that this operator can perform see the <u>Division</u> Operator.

When used to input a number % will divide by 100. When used in the display format to display a number % will multiply by 100.

? Modulus Operator

The modulus operator returns the remainder of a division. For example, 4 ? 3 is 1, four divided by three is 1 remainder 1.

The modulus operator is similar to division in that it operates differently on different objects. The following list gives the useful combinations.

Numeric ? Numeric   Returns the modulus (the remainder) of the equivalent division. e.g. 4?3 = 1

Complex ? Complex   The result of this calculation is produced by dividing the complex number by the divisor. The real and imaginary parts are then processed to remove any integer part, leaving only the fraction. Finally the result is multiplied by the divisor. Notice that this is not the Modulus part of {Mod, Arg} form for a complex number. e.g. (32+4i)?3i = (2+1i). Modulus has a higher precedence than addition so the brackets are needed, otherwise this expression would appear as 32+(4i ? 3i).

Array ? Numeric   Returns the array, with each element separately evaluated using the normal modulus rules. e.g. $\begin{pmatrix} 2 & 4 \\ 5 & 7 \end{pmatrix} ? 2 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$

Array ? Array   The modulus is applied to corresponding elements of the array e.g.
$\begin{pmatrix} 2 & 4 \\ 5 & 7 \end{pmatrix} ? \begin{pmatrix} 3 & 3 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$

^ Raise to the Power Operator

The ^ operator raises the first value to the power of the second, e.g. 2^2 returns 4,   3.2 ^ 6 returns 1073.472.

The returned type is always double (a double precision floating point number).

Complex numbers can be raised to a power, i.e. (2+3j)^2 returns -5+12j and can be returned as the result of a calculation e.g. (-3)^0.5 returns 0 + 1.7320508j

#& Bitwise AND Operator

Converts any numeric object to an integer value, and then performs a Boolean logic bitwise AND on the numbers, returning the integer result e.g. 5@ #& 4@ returns 4@, 20.7 #& 7.0001 returns 4.

⭐ Applying a bitwise AND to an array performs the AND on each element in turn if a single value is used, while an AND involving two arrays performs the AND on corresponding elements.

$$\begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix} \text{\#\&} 3 = \begin{pmatrix} 2 & 2 \\ 0 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 7 \\ 12 \end{pmatrix} \text{\#\&} \begin{pmatrix} 6 \\ 11 \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \end{pmatrix}$$

#| Bitwise OR Operator

This function converts any numeric object to an integer value, and then performs a Boolean logic bitwise OR on the numbers, returning the integer result e.g. 5@ #| 4@ returns 5@, 20.7 #| 7.0001 returns 23.

⭐ Applying a bitwise OR to an array performs the OR on each element in turn if a single value is used, while an OR involving two arrays performs the OR on corresponding elements, e.g.

$$\begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix} \#|3 = \begin{pmatrix} 3 & 7 \\ 7 & 11 \end{pmatrix}$$

$$\begin{pmatrix} 7 \\ 12 \end{pmatrix} \#| \begin{pmatrix} 6 \\ 11 \end{pmatrix} = \begin{pmatrix} 7 \\ 15 \end{pmatrix}$$

#~ Bitwise XOR Operator

This function converts any numeric object to an integer value, and then performs a Boolean logic bitwise XOR on the numbers, returning the integer result e.g. 5@ #~ 4@ returns 1@, 20.7 #~ 7.0001 returns 19.

Applying a bitwise XOR to an array performs the XOR on each element in turn if a single value is used, while an XOR involving two arrays performs the XOR on corresponding elements, e.g.

$$\begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix} \#\sim 3 = \begin{pmatrix} 1 & 5 \\ 7 & 11 \end{pmatrix}$$

$$\begin{pmatrix} 7 \\ 12 \end{pmatrix} \#\sim \begin{pmatrix} 6 \\ 11 \end{pmatrix} = \begin{pmatrix} 1 \\ 7 \end{pmatrix}$$

#>Bitwise Right Shift Operator

This function converts any numeric object to an integer value, and then performs a logical shift right by the specified number of bits, returning the integer result e.g. 23@ #> 2@ returns 5@, 200.7 #~ 3.0001 returns 25.

Note that zeros are shifted into the upper bits.

Applying a right shift to an array performs the shift on each element in turn if a single value is used, while a right shift   involving two arrays performs the shift on corresponding elements, e.g.

$$\begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix} \# > 2 = \begin{pmatrix} 0 & 1 \\ 1 & 2 \end{pmatrix}$$

$$\begin{pmatrix} 7 \\ 12 \end{pmatrix} \# > \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

#< Bitwise Left Shift Operator

This converts any numeric object to an integer value, and then performs a logical shift left by the specified number of bits (using a shift count greater than 32 acts the same as 32), returning the integer result e.g. 23@ #< 2@ returns 92@, 200.7 #~ 3.0001 returns 1600.

Applying a left shift to an array performs the shift on each element in turn if a single value is used, while a left shift   involving two arrays performs the shift on corresponding elements, e.g.

$$\begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix} \#< 2 = \begin{pmatrix} 8 & 24 \\ 16 & 32 \end{pmatrix}$$

$$\begin{pmatrix} 7 \\ 12 \end{pmatrix} \#< \begin{pmatrix} 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 28 \\ 96 \end{pmatrix}$$

! Factorial Operator

The Factorial Operator, returns the factorial of the number to the left of it. It must be to the right of the calculation.

When the exclamation mark is to the left of a calculation this operator behaves as a Logical NOT, rather than Factorial.

The Factorial function is defined as   n! = n (n-1) (n-2)....1 and as such it only operates on integers. When used on a floating point number the number is rounded. The returned result is always of type Double since this operation can produce very large numbers.

30! = 2.6525e32

Window Functions

In Flute there is a range of functions to control windows. What is a window? Most things on the screen are a window, most buttons for example are windows, as are edit fields, fixed text, and a lot of scroll bars.

This is a very important concept. Consider the function SetWindowRect which sets the size of the rectangle that a window covers. This function can be used to set the size of a main window, but it can also be used to set the size of a button in a dialogue.

The type of a window, is defined by a piece of text called its class name. Buttons are usually class "BUTTON". Not surprisingly, edit fields are usually class "EDIT" and so on. Each program can define its own classes for any special types of window they need.

Each window has two parts, a client area, and a non-client area.

The client area is the inside of the window, the non-client area is everything else and usually includes the border, menus, title bars and so on.

Window Functions

FindWindow	Obtain a Window Handle
GetWindowText	Get text of a window
GetWindowRect	Get outer rectangle of a window
GetClientRect	Get Inner rectangle of a window
GetTopWindow	Get handle of top window
GuessClient	Guess which window is the client window
BringWindowToTop	Bring a window to the top
BringWindowToTopmost	Bring a window to topmost
BringBackTopmost	Bring topmost window back to normal
CloseWindow	Close specified window
IsZoomed	Is a window maximized (zoomed)
IsIconic	Is a window iconised
IsEnabled	Is a window enabled

MaxWindow	Maximized (zoom) a window
MinWindow	Minimized (iconize) a window
RestoreWindow	Restore a window to normal size
EnableWindow	Enable a window
DisableWindow	Disable a window
GetParentWindow	Get windows parent
GetChildWindows	Get list of child windows
GetClassName	Get Class Name of a window
SetWindowText	Set a windows text
GetActiveWindow	Get main window being edited
GetFocus	Get window being editied
SetWindowRect	Set outer rectangle for a window
IsVisible	Test if a window is visible
HideWindow	Make a window invisible
ShowWindow	Unhide a window
ActivateWind	Make a window the active window

FindWindow

Every window on the screen has an associated number used to identify it. These numbers are called handles.

FindWindow finds and returns the handle to a window given the window's name, class name or position. Many of Flute's control functions require a handle to a window.

If the window is not found, a Boolean FALSE is returned.

Windows are organised into parent windows (the main windows for an application) and child windows (children of a parent window) in a hierarchical manner.

```
FindWindow("TestNameWindow")
```

Here it searches for a window called "TestNameWindow" and if that window is found its handle is returned. If no matching window can be found then a Boolean FALSE is returned. FindWindow first searches all the parent windows and then searches all the children of those parent windows (and children of children where this is appropriate).

You can also call FindWindow with an array of strings giving more detail as to where to search for the window.

```
FindWindow{"Program Manager","main"}
```

Here FindWindow searches for the window called "main" which is a child of the window "Program Manager".

You can specify as many windows as you like, for example:

```
FindWindow{"CleanSheet","Document 1","PipeData"}
```

This refers to the window "PipeData" which is a child of window "Document 1" which is a child of window "CleanSheet".

If you know the handle of any of the windows, you can substitute that instead. For example `FindWindow{1045, "Document 1"}` means find the handle of the window called "Document 1" which is a child of a window whose handle is 1045.

You can also specify the XY mouse coordinates of the window relative to its parent window. This is done by using a sub array of {x,y} coordinates. Where {0,0} is the top left corner of the parent window, or, if this is a top level window, the {x,y} screen coordinates.

`FindWindow{1045,{10,50}}` will return the handle of the window that is at x=10, y=50 inside the parent window whose handle is 1045.

Hence to find the window whose coordinates are {10,20}, use `FindWindow{{10,20}}`.

Note: When searching for a matching name. Flute treats upper and lower case letters as being equal and will accept partial matches.

Often in multi-document applications, the name of the document is appended to the end of the main window name. For example: "CleanSheet - [Untitled 1]" is window CleanSheet, document Untitled 1. To refer to the main program window (for example, to select a menu item) you can refer to the window as "CleanSheet". This is enough to select the main window. If, however, you wanted to refer to the document window an array is required : {"CleanSheet","Untitled 1"}.

Another way of specifying a window is by its class name. To do this add the character '>' in front of the name. For example, {"Flute", ">edit"} is the first window that has the class 'edit' inside the 'Flute' window. To specify the second window that has the "edit" class name, use ">>edit", and obviously the third window with that class name would be ">>>edit".

Paste Find Window

Generating <u>FindWindow</u> commands by hand is not easy. There is a much simpler way of doing it. Select 'Paste Find Window...' from the 'Edit' Menu. This dialogue lets you choose the window by pointing at it or selecting it from a list and generates the correct `FindWindow` command required to specify it.

Click on any part of this window that you need help with.
There are two ways to select a window using this dialogue:
▪   To select using the mouse, drag the arrow button off the dialogue and over the window you want to refer to.
▪   To select the window from a list; click on the 'More>>>'   button to bring down the list. This list mentions every window that is on screen. Windows that are hidden are shown in grey. As windows are arranged hierarchically, so is the list. Each entry shows the class name of the window, followed by the name of the window in quotes.

When you select a window information showing the text, class name, handle and attributes of the window are displayed.

Once you have selected your window select OK to paste the appropriate `FindWindow` command into your program.

To choose a window with the mouse, drag this mouse button out from the dialogue and over the window you want. The window will be highlighted with a rectangle.

The resultant FindWindow command will be shown in this edit window.

Information about the window you have chosen is displayed in this section of the dialogue.

Selecting this 'More>>>' button displays the lower half of the dialogue.

This is a hierarchical list of all the windows on the screen. Windows which are hidden or disabled are shown in grey. A typical entry in this list might consist of `Button, "OK"` which indicates a window of class 'Button' with the text 'OK' on its face.

GetWindowText

This function obtains the text from a window into a string object. For most windows the text is the title of the window, but this is not always the case:

Static text items on dialogues are really windows. This function can obtain the text from them.

EditFields - the text from the edit field of a dialogue can be obtained using GetWindowText.

Buttons - most buttons are windows. The text on the face of a button can be obtained with GetWindowText.

This function takes the form:

```
GetWindowText(wind_hand)
```

wind_hand is the handle of the window whose text is to be obtained. You can use any of the methods used in <u>FindWindow</u> to locate the window. For example `GetWindowText("CleanSheet")` can return the full title of the CleanSheet window - "CleanSheet - [Untitled 1]".

The maximum length of text returned by this function is 32767 characters.

If you are obtaining text from a Multi-Line edit control use <u>GetMLText</u> in preference to this function.

GetWindowRect

This function obtains the rectangle containing the window in screen coordinates. Every window has two rectangles - the inner work area and the outer rectangle enclosing the whole window, its title bar, scroll bars, icons etc. This function returns the **outer** enclosing rectangle.

The function takes the form:

`GetWindowRect(wind_hand)`

wind_hand is the handle of the window whose rectangle is to be obtained.

The returned array looks like this:

{{x1,y1},{x2,y2}}

x1,y1 - the coordinates of the top left corner of the window (including its title, borders etc.)

x2,y2 - the coordinates of the bottom right corner of the window (including title, borders etc.).

x1,y1 and x2,y2 coordinates are specified relative to the screen.

You can substitute any method listed under <u>FindWindow</u> of identifying the window for the window handle. For example `GetWindowRect("Program Manager")` will obtain the non-client rectangle for the Program Manager's main window.

GetClientRect

Gets the rectangle containing the client or work area of a window in screen coordinates. Every window has two rectangles - the work area and the outer rectangle enclosing the whole window, its title bar, scroll bars, icons etc.

This function returns the coordinates of the work area, relative to the screen.

The function takes the form:

```
GetClientRect(wind_hand)
```

wind_hand is the handle of the window whose rectangle is to be obtained. (You can use any of the methods listed under FindWindow to specify the window).

The returned array looks like this:

{{x1,y1},{x2,y2}}

x1,y1 - the coordinates of the top left corner of the client area of the window.

x2,y2 - the coordinates of the bottom right corner of the client area of the window.

x1,y1 and x2,y2 coordinates are specified relative to the screen.

The Client area of a Window.

The Non-Client Area of a Window.

Menu Functions

An important part of controlling an application is manipulating its menus. Flute provides a range of commands to directly handle and examine menus. Most of these commands refer to a menu by its ID number. Each menu item has its own unique ID number and these numbers are fixed within the programs.

The following list covers the menu commands within Flute

FindMenu   Find the ID of a menu item

SelectMenu   Select the specified menu item

GetMenuText   Get the text of a specified menu item

GetMenuState   Get   disabled/checked etc. information

See Also

Paste Select Menu   Constructions SelectMenu commands.

Microsoft - Breaking Automation Programs

There are many automation programs on the market that manipulate the user interface. One of the most stable things these programs can do is to select menus by their menu ids.

Menu ids are normally fixed, they are the agreed means by which the operating system tells an application which menu has been chosen. To vary the menu ids requires wasteful mapping functions to generate a unique id and to convert from that id back to a command.

Microsoft has broken most controlling programs with Windows 95. Consider the following menu (from "My Computer"):

The menu id of 'Open' is 30734. Pull the menu down again and the menu id of 'Open' changes to 30739. Open the menu a third time and it changes again.

Microsoft is constantly reassigning menu IDs to its menus - Why?!

When you use Flute menu selection functions they perform a consistency check. If the menu id changes, Flute will use a different method of referencing the menu item.

FindMenu

The find menu command searches for a menu item and returns the ID number assigned to it. ID numbers are the numbers used by programs to identify the menus. Once an ID is known the menu can be selected.

If FindMenu doesn't find a matching menu item it returns a Boolean FALSE.

`FindMenu{wind_hand,menu}`

This is the general form of the FindMenu command. The wind_hand is the handle of the window containing the menu to be searched. You can obtain window handles using the FindWindow function, or you can refer to windows by name.

The menu to search for can be specified in several ways:

`FindMenu{4044, "Minimize on Use"}` searches for a menu item called "Minimize on Use" in the menu held by window 4044.

`FindMenu{4044,{"Options","Minimize on Use"}}` here an array is used to specify the menu. This array tells FindWindow to search for a menu titled "Options" and then search on that sub menu for "Minimize on Use". These constructs are useful when there are several menu items with the same name.

Note: Not all menu items that look like text are text. Some menu items are bitmaps, others are drawn by the applications themselves rather than by Windows. If the menu item is not real text then it cannot be searched for by name.

You can also retrieve a menu by its position.

`FindMenu{"program manager" , {"Options", 1}}` here the number 1 is used. This tells FindMenu to return the ID of the item at position 1 on the "Options" menu.

Positions start at zero, and separating lines count as one position.

SelectMenu

This command selects a menu item, either by its name (the text shown on the menu), or more usually by the special ID number assigned to it.

```
SelectMenu{wind_handle, Menu_id}
```

wind_handle is the handle for the window containing the menu.

This is the most reliable way of selecting menus. Each menu item has a number assigned to it by the program's designer. Windows uses this number to tell the program that its menu has been selected. The ID numbers can be obtained by using Flute's Paste Menu command.

Example:
```
SelectMenu{4450, 342}
```

Selects menu item ID 342 in the window whose handle is 4450.

```
SelectMenu{wind_hand, menu_text}
```

In this form the text of the menu is used to select it. For example if the menu item was called "Open" enter `SelectMenu{4450,"Open"};` Flute will search the menu in window 4450 for a menu item called "Open" and select it.

This is not a reliable way of selecting menus. Many programs vary the text of the menu. For example "Hide Ruler" may become "Show Ruler". Also the menu item may only appear to be text when it is actually a menu item drawn by the program itself.

➡ Whenever possible, use the SelectMenu option specifying the **ID** of the menu item.
Note: You can also specify the window name rather than the handle of the window. For example `SelectMenu{"Program Manager","New"}` is equivalent to selecting the "New" option within the Program Manager's menus. Refer to <u>FindWindow</u> for more details.

GetMenuText

This function returns the text of the menu object.

```
GetMenuText{wind_hand,menu_id}
```

wind_hand is either the handle of the window, or a search string that can be used to find the window (see FindWindow for more details).

menu_id is the ID assigned to that menu item. You may also specify a search string or search array to locate the menu_id, (see FindMenu for more details).

The text of the menu item is returned as a string object. If no such item exists, an error object is returned. If the item doesn't have any text (either it is drawn by the application, or it is a bitmap) then a Null is returned.

Notice that the string may not be what you expect. The menu item "New" on the File menu of the program manager actually looks like "&New..." The ampersand (&) tells Windows to draw an underscore under the N. Also note that it has 3 dots after the name to indicate that selecting that menu option displays a dialogue.

A menu item such as "Open" has a keyboard shortcut next to it. These keyboard shortcuts are separated from the text by a tab character. The actual text of "Open" is "&Open <TAB> Enter"

Keyboard shortcuts and underscore characters are ignored when you use Find Menu but are included in the string returned by GetMenuText.

GetMenuState

This function returns information about the specified menu item.

`GetMenuState{wind_hand, menu_id}`

wind_hand is a handle to the window containing the menu

menu_id is ID number used to specify which menu item.

You can use any of the methods listed under FindWindow to specify the window handle, similarly you can use any of the methods listed under FindMenu to specify the menu ID.

The information it returns consists of an array of Boolean TRUE/FALSE Objects as follows:

`{Checked, Disabled, Grayed,Bitmap}`

`Checked`	TRUE if the menu item has a check mark (a tick) next to it.
`Disabled`	TRUE if the menu item is not selectable, (but not grayed).
`Grayed`	TRUE if the menu item is not selectable and has been Grayed.
`Bitmap`	TRUE if the menu item is a bitmap.

Example:

`GetMenuState{"CleanSheet","Cut"}[2]` produces a Boolean TRUE when the Cut menu is grayed.

The GetMenuState returns the array {FALSE,FALSE,TRUE,FALSE} and the square brackets access element 2 from this array to return the Boolean TRUE (remember arrays start at element 0).

Paste Select Menu

Generating menu IDs by hand can be very difficult -you cannot be expected to know which menu item has which menu ID. To automate this process, Flute provides a command on its Edit menu - 'Paste Menu Select...' .

Click on any part of the dialogue you need help with.

Selecting a menu item is a two state process:

- Select the window whose menu is to be manipulated. You can do this either with the mouse or by selecting from a list.
- ▶ To select using the mouse; drag the arrow icon off the dialogue and over the window to be used.
- ▶ To select from the list; click on one of the windows listed. Only windows that have menus are listed, hidden or programs reduced to icons are shown in grey.
- When a window is selected, its menu is copied onto the Paste Menu dialogue
- Select the menu item on this dialogue (not on the application). You will notice as you select the menu items, the command needed to obtain the correct menu ID is shown at the bottom of the dialogue.
- When you are happy with your selection click on OK.

To select a window with the mouse, drag this button off the dialogue and over the window.

The text (the title) of the chosen window is displayed here.

A list of windows that contain menus is shown in this section, windows which are hidden or disabled are shown in grey.

The SelectMenu command constructed by this dialogue is shown here.

Miscellaneous Control Functions

This group of functions covers running programs, obtaining information for the user, and various flags and information functions.

[Execute](#)	Run a program
[GetCaretPosition](#)	Returns position of vertical bar cursor
[Ask](#)	Ask the user for information
[MessageBox](#)	Display a multiple choice message box
[MessageWindow](#)	Displays a modeless message window
[MessageWindowState](#)	Returns the state of the message window buttons
[MessageWindowClose](#)	Closes a messagewindow
[Changed](#)	Flag to indicate program has changed
[Wait](#)	Wait for specified number of milliseconds
[SafeExit](#)	Flag is it safe to exit?
[GetListBoxState](#)	What text does a list box contain?
[GetButtonState](#)	Is a button checked?
[SelectListBoxItem](#)	Select (or deselect) an item in a list box.
[GetComboText](#)	Returns the text from a Combo Box.
[SelectComboItem](#)	Select an item from a Combo Box.
[Act](#)	Call a window act.
[ActList](#)	Return a list of acts supported by a window.

Execute

Starts a program. The program runs independently from Flute. The Execute command returns immediately, it does not wait until the executed program has terminated.

```
Execute{prog_name, command_line, working_dir};
```

prog_name is a string object containing the name of the program to execute (which can include a full pathname).

command_line is the optional command string.

working_dir is the default directory for this program.

You can omit the command_line and working directory, or substitute NULL for them. If the working directory is not specified then the program name is examined and if it includes a pathname then that path is used as the working directory.

Example

```
Execute{"c:\csheet\csheet.exe", "example1.cln"};
```

Opens CleanSheet (CleanSheets program is called Csheet.exe), and passes it the name example1.cln. When CleanSheet receives a command in this way it opens the file.

The working directory in this case is c:\csheet, because no working directory was specified.

The return is TRUE if the program was executed, or an appropriate error message.

GetCaretPosition

The caret is a blinking vertical bar that indicates where the insertion point is when editing text. Not all applications make use of the Windows Caret, some implement their own and so GetCaretPosition cannot be used on those applications.

This function takes no parameters. For example

```
a:=GetCaretPosition;
```

note it is not necessary to use brackets ( ) after the function.

The return is a two element array containing the {x,y} screen coordinates of the centre of the caret.

Note: to set the caret position, use LClick.

Ask

This function constructs and displays dialogues on the screen asking for information to be entered or giving information back to the user of Flute. This is an easy way of obtaining information from the user.

A single request for information looks like this:

ask{"Enter Number X",5@}

Notice that the request has 2 parts to it; the text of the request and the initial value. The initial value determines what type of data is being called for. In this example the initial value is the integer 5, hence the request will return an integer value.

A typical return from this could be the single integer 4@.

If the user selects CANCEL an error object is returned.

The type of control that appears on the dialogue depends on the type of object used as the initial value:

```
ask{"Enter Integer X",5@}
```
        Results in an input box with up/down spin dials.

```
ask{"Enter Date D",\20 Mar 88}
```
        Results in a Date selector control

```
ask{"Enter String",""}
```
        Results in a long edit field

```
ask{"Enter Complex Number",1+2i}
```
        Results in a long edit field

```
ask{"Select Button",TRUE}
```
        Displays a single checkbox, the checkbox is initially selected

```
ask{"Enter Number",5}
```
        Displays a short edit box

```
ask{"Enter CeSk Data Object",NULL}
```
        This item is displayed as a large edit field. The return from this can be any type of object, simply use the CeSk's object notation when entering the data. For example 2@ will result in the integer 2, "Text" will result in a string object because double quotes are placed around the text.

```
ask{"Type of Particle",{"Odd","Even","Both"}}
```
        Here a selection of 3 items is given. These appear as 3 radio buttons. The return from this type of request is the number 0,1, or 2 indicating the first, second or third choice was made. (Remember that CeSk references all arrays starting at zero).
        If there are more than 4 selections then a drop down list is used in place of the radio buttons. This control is more suited to displaying large numbers of choices.

```
ask{"This is a test string"}
```
        Here there is no initial value, the text is simply displayed without any request for information. The return from this item is always NULL.

You can ask for several pieces of information at the same time simply by grouping the items into arrays:
```
request_1:={"Enter X",0};
request_2:={"Enter Y",0};
request_3:={"Enter Z",0};
```

```
ask{request_1, request_2,request_3};
```

Asks for the X, Y and Z to be entered, all three being numbers.

The return from this ASK takes the same form as the array. For example if you entered 1,2,3, then the result would be the array {1,2,3}
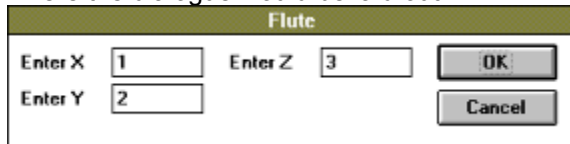
Notice that these three controls are laid out vertically in a single column. You can have multiple columns in your dialogue, simply by using a two dimensional array of requests:
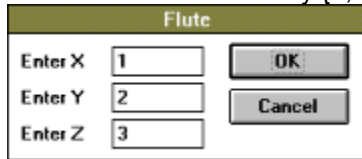
```
request_1:={"Enter X",0};
```

```
request_2:={"Enter Y",0};
```

```
request_3:={"Enter Z",0};
```

```
ask{{request_1, request_2},request_3};
```

Here the dialogue would be laid out:

The return from this request would take the same form. For example if the user entered X=1 and Y=2 and Z=3 into the dialogue, the return would be {{1,2},3}

MessageBox

Creates a multi-choice message box or displays a simple message to the user.

```
MessageBox ("This is a message");
```

In its simplest form   it displays a message to the user and waits for them to select 'OK'. To display multiple lines of text use a line feed (character 10) to separate the lines, for example:

```
MessageBox ("This is Line 1"+10+"This is Line 2");
```


If the message text includes a question mark or an exclamation mark, the information icon is replaced with an exclamation, or question icon.

You can give the user multiple choices to select from by providing a list of possible selections in an array:

```
a := MessageBox{"An Error has occurred"+10+"Should we continue anyway?",
"Yes","No","Cancel","&Help"};
```
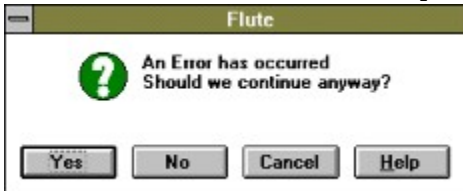

The first element of the array is the text of the message. The remaining entries are the text of the buttons.
▪   The return from this function is the text of the chosen button. If the user selected 'Cancel' a string "Cancel" would be returned.
▪   Notice that putting the ampersand '&' character inside the text of a button defines a keyboard shortcut for that button. Notice that the ampersand in the "&Help" button, defines Alt-'H' as the shortcut for selecting Help.

Note:
        A modeless version of this function is available MessageWindow

MessageWindow

Creates a multi-choice message window or displays a simple message to the user. This is very similar to MessageBox, except that this call returns immediately leaving the message displayed on screen. The message window can subsequently be examined or manipulated by other message window functions.

```
a:=MessageWindow ("This is a message");
```

The return from this function is a window handle to the MessageWindow. This handle is required by other MessageWindow functions.

In its simplest form   it displays a text message to the user. To display multiple lines of text use a line feed (character 10) to separate the lines, for example:

```
a:=MessageWindow ("This is Line 1"+10+"This is Line 2");
```
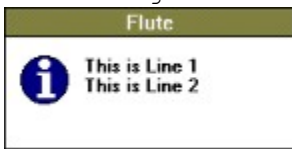

If the message text includes a question mark or an exclamation mark, the information icon is replaced with an exclamation, or question icon.

You can give the user multiple choices to select from by providing a list of possible selections in an array:

```
a:=MessageWindow{"An Error has occurred"+10+"Should we continue anyway?",
"Yes","No","Cancel","&Help"};
```
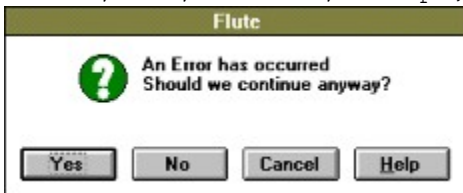

The first element of the array is the text of the message. The remaining entries are the text of the buttons.
▪	Notice that putting the ampersand '&' character inside the text of a button defines a keyboard shortcut for that button. Notice that the ampersand in the "&Help" button, defines Alt-'H' as the shortcut for selecting Help.
▪	Multiple message boxes can be displayed at the same time.

Manipulating the Message Window

Once the message window is displayed your program can continues to run. The message window continues to be displayed and can be manipulated and checked with two message window functions:

MessageWindowClose Closes the message window.

<u>MessageWindowState</u> Returns the state of the message box buttons.

MessageWindowClose

Closes a message window previously created with the <u>MessageWindow</u> function.

```
MessageWindowClose(WindowHandle);
```

WindowHandle   the handle of the window as returned by the MessageWindow function.

Returns TRUE if successful or an error if failed.

MessageWindowState

Returns an array indicating how many times each button on a <u>MessageWindow</u> has been pressed.

```
a:=MessageWindowState(WindowHandle);
```

WindowHandle   the handle of the window as returned by the MessageWindow functions.

Return: An array of integers, each integer corresponds to a button on the MessageWindow and indicates how many times it has been pressed since the Message Window was first created.

For example:

Consider the message window below. If the 'Cancel' button had been clicked on twice since the message window was first created then the return from a MessageWindowState function would be {0, 0, 2, 0}
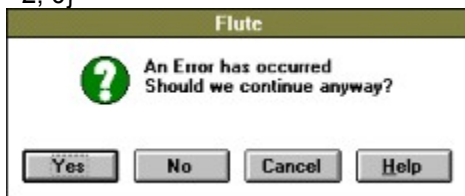
Changed

This function sets or returns the state of the Changed flag. When you exit Flute, if the changed flag is TRUE Flute will save the current program, together with the variables beginning l_ ,ls_, lo_, los_, g_, gs_, w_ and wp_. These variables are special in Flute because they can be seen by any function. Other variables are only valid within the Function that created them.

You can use these variable to hold important settings that need to be saved with the program.

Changed(TRUE)   Sets the Changed Flag to TRUE.

Changed(FALSE)   Clears the Changed Flag.

flag:=Changed;   Puts the current value of the changed flag into the variable 'flag'.

Example:
```
main {
    if (l_birthdate =NULL) {
        l_birthdate:=ask{"What is Your Birthday?",\1st Jan 1950};
        Changed(TRUE);
    };
};
```

In this example, if the variable l_birthdate has not been initialised then this program asks you when your birthday is. This setting will then be saved with the program. When this program is run again l_birthdate will still contain the date.

The rules for saving when Changed is true are simple:
- If the Flute program window is visible Flute will prompt you if you want to save.
- If the Flute program window is hidden Flute will save without further prompting unless the file doesn't have a name, in which case it will ask.

Wait

This command waits for a specified number of milliseconds before program execution continues.

```
wait(count);
```

Example:
```
tn:=make{Now,1}; /* current time in seconds */
st:=make{03:00 ,1};    /* time we are waiting for */
di:=st - tn;
if (di<0) di+=86400;   /*di is num of seconds to wait */
wait(di*1000);   /* wait for di*1000 millisecs */
/* execution continues at 03:00 */
```

This freezes the program but does not freeze Windows. When in a Wait command Flute takes the absolute minimum amount of processor time possible, giving the maximum time to the other Windows programs.

Consider the example of a Flute program designed to start a Fax program at 3am. It is important that this program should not waste processor time executing a loop, so it is better to use 'Wait'.

SafeExit

This function sets or clears the SafeExit flag.

```
SafeExit(TRUE);    /*The program can terminate safely. */

SafeExit(FALSE);   /*The program must not be closed at this point. */

val:=SafeExit;     /* Returns the value of the SafeExit flag */
```

When Windows is closed, or Flute is exitted, the program's execution will stop. If it is currently saving a file, or in the middle of a critical piece of code, then it is not safe to exit. You can set SafeExit to False at this point in the program, and restore it to True after the critical piece of code.

When SafeExit is False, Windows cannot be shut down, and Flute cannot be closed.

GetButtonState

This function is used to test the state of a button window. A button window can have one of three states: checked, unchecked, or indeterminate. The latter state is used only by three state buttons.

☒ Checked
☐ Unchecked
▩ Indeterminate

◉ Checked
○ Unchecked

```
ButtonState(wind_hand)
```

wind_hand is the handle of the button window. Any of the methods listed under FindWindow can be used to specify this window.

The return from this function is the number 0 - Unchecked, 1- Checked, or 2 - Indeterminate.

Example:

result:=ButtonState{"Program Item Properties","Run Minimized"};

Will test whether the "Run Minimized" button is checked on the "Program Item Properties" dialogue.

To set the state of a button, use LClick.

Note: many of the buttons within applications produced by Microsoft are not the standard Windows buttons. These buttons do not follow the commands set out in the Windows SDK manuals and their state cannot be obtained with the GetButtonState function.

GetListBoxState

This function obtains data from a list box. It gets the text of each list item and whether the item is selected.

Two examples of list boxes are shown below:
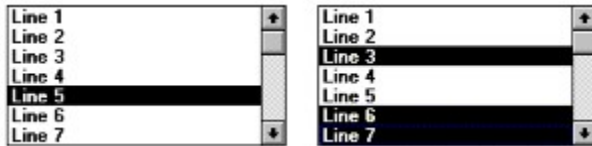
Notice that list boxes can have several items selected.

```
GetListBoxState(wind_hand)
```

wind_hand is the handle of the list box window, you can use any of the methods listed under FindWindow to specify this window.

The return from this function is an array of data entries. Each entry takes the form: {Item_Text, Item_State}.

Item_Text is a string object containing any text in the list item. Some list boxes are drawn by the application and contain no text: the Paste Find Window dialogue within Flute is a good example of this.

Item_State is a Boolean TRUE or FALSE: TRUE means its selected.

Example:

The return from the left list box would look like this:

{{"Line 1",FALSE}, {"Line 2",FALSE}, {"Line 3",FALSE}, {"Line 4",FALSE}, {"Line 5",TRUE}, {"Line 6",FALSE}, {"Line 7",FALSE},{"Line 8",FALSE}}

SelectListBoxItem

This command selects or deselects an item in a list box. It takes two forms, dependent on whether the list box is a multiple selection list box or a single selection list box. In a single-selection list box you cannot *deselect* an item, you can only select a *different* item.

```
SelectListBoxItem{wind_hand, item_index, flag}
```

wind_hand   is the handle of the list box window, any of the methods listed under FindWindow can be used to specify the window.

item_index specifies the entry to be selected, the first entry is entry zero.

flag is TRUE to select an item, FALSE to deselect it.

If the list box allows only single selections then the flag can be omitted, e.g., SelectListBoxItem{wind_hand,item_index}

The return from this function is TRUE if successful, or an error object if the window could not be found.

GetComboText

This function obtains the text from a Combo Box.
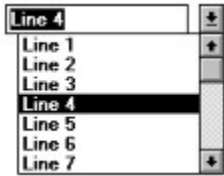
A combo box is a drop down list attached to an edit field.

```
GetComboText(wind_hand)
```

wind_hand is the handle of the combo window, you can use any of the methods listed under FindWindow to specify this window.

The return from this function is a 1 dimensional array containing the text within the combo box.

Note: Some combo boxes are drawn by the applications; they do not contain any text.

Example:

The return from the above Combo box might be

{"Line 1","Line 2","Line 3","Line 4","Line 5", "Line 6","Line 7","Line 8"}

SelectComboItem

This function selects an item from a drop down combo box.

```
SelectComboItem( wind_hand, item_index)
```

wind_hand is the handle of the combo box item. Any of the methods listed under FindWindow can be used to specify this window.

item_index is a number identifying the item to select. Item numbers being at zero.

The return from this function is a Boolean TRUE if successful, or an error object if the window could not be found.

Act

Each window behaves the way it does because of its class. It makes sense for each class of window to support a range of automation commands that represent the actions it supports. This is the purpose of the Act command.

Act calls the specified window function for the specified window.

```
result:=Act{wind_hand, funct_name, parameters}
```

wind_hand; is the handle of the window. Any of the methods listed under <u>FindWindow</u> can be used to specify this window.

funct_name; is a string object which specifies the function to call.

parameters; are optional parameters to pass to the function. If, for example, the function required two numbers then parameters would be an array such as {1,2}.

Act requires special code within each application. CleanSheet version 1.12 adds support for Acts: most of its automation features are window acts.

Applications writers can add support for Act very easily, contact WorkingTitle UK for details.

ActList

This is a complimentery function to Act. It obtains a list of the Act functions that the specified window supports.

result:=ActList(wind_hand)

wind_hand is the handle of the window to obtain information on. Any of the methods listed under FindWindow can be used to specify this window.

If the window does not support Act the return from this function is NULL. Otherwise a two dimensional array of strings is returned.

Consider an example returned array for a (ficticious) slider window.

{{"GetSliderPos","Returns the position of the slider 0-1"},{"SliderTop","Moves the slider to its topmost position"}}

Each function pair gives the name of the function and its purpose as two string objects. For this example the window supports two functions, "GetSliderPos" and "SliderTop".

Scroll Bar Functions

Scroll bars are the sliders that control the view a window displays. They are a special case within Windows as they can either be a window themselves or, more usually, they are simply a *part* of another window, just like the caption or menu is a part of a window.

In Flute all scroll bars are treated equally. Flute attempts to separate scroll bar *windows* from scroll bar *sections* of a window and handle the difference automatically.

Most of the functions are directly equivalent to clicking on the scroll bar with the mouse:

See Also

<u>VScrollPos</u>   Returns the position of a vertical scroll bar

<u>HScrollPos</u>   Returns the positition of a horizontal scroll bar

VScrollThumb

This function scrolls a window as though you had operated the **thumb** of a vertical scroll bar.

The thumb is the rectangular box that defines the scroll bar's position. The new position is specified as a number in the range 0 to 1, 0% is the top of the scroll bar, 100% is the bottom of the scroll bar (remember % in Flute divides by 100, so 50 % = 50/100 = 0.5).

```
VScrollThumb{wind_hand,amount}
```

wind_hand - the handle of the scroll bar window, or the handle of a window containing a scroll bar - any of the methods listed under FindWindow can be used to identify the window.

amount - a number in the range 0 to 1 indicating the position of the scroll bar.

If the specified window is itself a scroll bar the thumb is moved to its new location exactly as if you had dragged it there with the mouse. If the window is a normal document window VScrollThumb looks for a vertical scroll bar at the right edge of the window and scrolls that.

Example:

```
a:=findwindow{"Program Manager","Main"};
```

```
VScrollThumb{a,100%}
```

Will scroll the "main" group window in the Program Manager to its bottom position.

The return value from this function is either TRUE if a scroll bar was found and the window scrolled, or an appropriate error.

Note: There is no perfect way of identifying scroll bars. Flute does its best, but may fail with some applications. If this happens use lclick to scroll the window using mouse clicks.

VScrollLineUp
VScrollLineDown
VScrollPageUp
VScrollPageDown

These functions scrolls a window as though you had operated the up or down arrows on the scroll bar or the page up or page down sections.

If the specified window is itself a scroll bar, this bar is manipulated directly. If the window is a normal document window, These functions look for a vertical scroll bar at the right edge of the window and manipulate that window instead.

```
VScrollLineUp(wind_hand)
```

wind_hand - the handle of the scroll bar window, or the handle of a window containing a scroll bar.

Example:

```
a:=findwindow{"Program Manager","Main"};
```
```
VScrollLineDown(a)
```

Will scroll down in the "main" group of the Program Manager.

The return value from this function is either TRUE if a scroll bar was found and the window scrolled, or an appropriate error.

VScrollPos

This function returns the position of the vertical scroll bar thumb for the specified window.

```
VscrollPos(wind_hand)
```

wind_hand is the handle of the window whose vertical scroll bar is to be examined, or the handle of the scroll bar itself. Any of the methods listed under FindWindow can be used to specify the window.

The return from this function is a value 0 to 1, where 0 is the top and 1 is the bottom thumb position. An error is returned if the window specified by wind_hand was invalid, or no suitable scroll bar could be found.

HScrollThumb

This scrolls a window as though you had operated the thumb of a horizontal scroll bar.

The thumb is the rectangular box that defines the scroll bar's position. The new position is specified as a number in the range 0 to 1, 0% is the left of the scroll bar, 100% is the right of the scroll bar (remember % in Flute divides by 100, so 50 % = 50/100 = 0.5).

```
HScrollThumb{wind_hand,amount}
```

wind_hand - the handle of the scroll bar window, or the handle of a window containing a scroll bar.

amount - a number in the range 0 to 1 indicating the position of the scroll bar.

If the specified window is itself a scroll bar,   the thumb is moved to its new location exactly as if you had dragged it there with the mouse. If the window is a normal document window, HScrollThumb looks for a horizontal scroll bar at the bottom edge of the window and scrolls that.

Example:

```
a:=findwindow{"Program Manager","Main"};
```

```
HScrollThumb{a,100%}
```

Will scroll the "main" group window in the Program Manager to its rightmost position.

The return value from this function is either TRUE if a scroll bar was found and the window scrolled, or an appropriate error.

HScrollLineLeft
HScrollLineRight
HScrollPageLeft
HScrollPageRight

This scrolls a window as though you had operated the left or right arrows on the scroll bar or the page left or page right sections.

```
HScrollLineLeft(wind_hand)
```

wind_hand - the handle of the scroll bar window, or the handle of a window containing a scroll bar.

If the specified window is itself a scroll bar, this is manipulated directly, if the window is a normal document window, these functions look for a horizontal scroll bar along the bottom edge of that window.

Example:

```
a:=findwindow{"Program Manager","Main"};
```

```
HScrollLineRight(a)
```

Will scroll right the "main" group of the Program Manager.

The return value from this function is either TRUE if a scroll bar was found and the window scrolled, or an appropriate error.

HScrollPos

This function returns the position of the horizontal scroll bar thumb for the specified window.

```
HscrollPos(wind_hand)
```

wind_hand is the handle of the window whose horizontal scroll bar is to be examined. Any of the methods listed under FindWindow can be used to specify the window.

The return from this function is a value 0 to 1, where 0 is the left and 1 is the rightmost thumb position. An error is returned if the window specified by wind_hand was invalid, or no suitable scroll bar could be found.

Mouse Operations

This list of the commands deal with ways of mimicking mouse operations. Flute does not record each mouse down, mouse move and mouse up events, rather it has a set of commands that performs the standard combinations of these events.

LClick	Left Button Single Click
LDoubleClick	Left Button Double Click
MClick	Middle Button Single Click
MDoubleClick	Middle Button Double Click
RClick	Right Button SingleClick
RDoubleClick	Right Button Double Click
LDrag	Drag with Left Button
MDrag	Drag with Middle Button
RDrag	Drag with Right Button
MousePos	Return position of mouse

See Also

Paste Mouse Action	Simplifies the pasting of the above mouse commands into your program.

LClick
lDoubleClick
MClick
MDoubleClick
RClick
RDoubleClick

These commands perform mouse clicks. A click is a single press and release of the mouse button. A double click is two clicks at the same place within a short space of time.

The commands RClick and RDoubleClick are for the right mouse button, the commands LClick and LDoubleClick are for the left mouse button, and MClick and MDoubleClick are for the middle mouse button.

These commands all take the form:

```
LClick{wind_handle,{x,y},key_flags}
```

Where wind_handle is the handle of the window to receive the mouse click. Refer to FindWindow for details on obtaining window handles.

key_flags is an optional value that specifies whether the shift key or control key was pressed during the clicking action.

There are predefined values : `shift` and `control`, which can be added together. For example a click with the left button while holding down the shift and control keys would be:

```
LClick{4222, {10,20}, Shift + Control};
```

There is an additional flag: `activate` - this means that the window should be activated before the mouse click. Some applications discard the mouse clicks that make them active, it is better to make these applications active *before* any mouse clicks.

Note: You can use any of the ways of referring to window handles listed under the FindWindow command. For example `LClick{"Program Manager",{40,50}}` is equivalent to clicking on the Program Manager window 40 pixels in from the left and 50 down from the top of the window.

`LClick{{"Program Manager","Main"},{40,50}}` is equivalent to clicking in the Main sub window of the Program Manager application window at {40,50}.

If the window handle is NULL, then the mouse clicks are positioned relative to the screen rather than any particular window.

`LClick{null,{40,50}}` clicks at position x=40, y=50 on the screen.

If you replace the coordinates with a NULL then the centre of the window is used. This is useful if the window is a button. Exactly where in the button you click makes no difference, as long as it is within the rectangle covered by the button.

`LClick{4055, null}` clicks in the centre of the window whose handle is 4055.

LDrag
RDrag
MDrag

Drags the mouse pointer within a window. A Drag action involves clicking the mouse button at a point {x1,y1} in the window, holding the button down and moving it to a new position {x2,y2}. The coordinates are specified relative to the window client area. {0,0} is the top left corner.

`LDrag` is a drag using the left mouse button.

`MDrag` is a drag with the middle mouse button.

`RDrag` is a drag with the right mouse button.

`LDrag{wind_handle , {x1,y1}, {x2,y2}, key_state}`

wind_handle is the handle of the window to receive the mouse drag action. Refer to FindWindow for details on obtaining window handles.

key_state is an optional value that specifies if shift, control or the other mouse buttons were pressed at the same time as the click.

There are predefined values : `shift` and `control`, which can be added together. For example a click-and-drag with the left button while holding down the shift and control keys would be `LDrag{4222, {10,20},{20,20}, Shift + Control};`

Note: You can use any of the ways of referring to window handles listed in the <u>FindWindow</u> command. For example `LDrag{"Program Manager",{40,50},{50,50}}` is equivalent to clicking on the Program Manager window 40 pixels in from the left and 50 down from the top of the window then dragging to {50,50}.

If the window handle is NULL then the mouse actions are positioned relative to the screen rather than any particular window.

MousePos

This function sets or reads the position of the mouse cursor.

```
MousePos{x,y}
```

In this form it sets the position of the mouse cursor to the coordinates x,y relative to the screen where (0,0) is the top left corner of the screen.

Example:
```
MousePos{10,30};
```

Moves the mouse to x=10 and y=30

Another form of this function is:
```
Pos:=MousePos;
```

Here MousePos returns the current position of the mouse, as an {x,y} array.

You can combine the two forms:
```
Pos:=MousePos{10,30};
```

After this call, Pos will contain {10,30} - the current position of the mouse cursor.

Paste Mouse Action

The 'Paste Mouse Action' command on the Edit Window simplifies the process of generating mouse commands.

Click on any part of this dialogue you need further help with.

To use Paste Mouse Action:
- Select Paste Mouse Action, the Flute window will be hidden and the cursor will change.
- Perform the clicking or dragging on the destination window. If the window is not top use Alt-Tab to bring it to the top of the pile of windows. Do not attempt to click with the mouse to do this.
- The Paste Mouse Action dialogue will appear. This displays the mouse action you have just performed.

'Window' shows the window that the mouse clicks applied to.

The Start Pos and End Pos display the start and end positions of the mouse action. For a single or double click, the start and end positions are the same, for a drag operation they are different. The coordinates of the start and end positions are always relative to the chosen window.

Mouse Button - shows which mouse button was used.

Modifier Keys - shows whether Shift or Control was held down during the action.

You can change the start and end positions by dragging the arrow button out from the dialogue to the new reference point. If you drag the start position this also sets the window.

Once you are happy with the settings click on OK and the relevant command will be pasted into your program.

The window that will be manipulated is shown here.

To set the first {x,y} position for a mouse action, drag this button off the dialogue and over the window. This sets the window and the {x,y} starting position.

The start coordinates are shown here.

The end coordinates can be set by dragging this button off the dialogue and over the window.

This section displays the end mouse coordinates, for example if the mouse is dragged, this is the coordinates where it will end up.

This section shows which mouse button is being manipulated. Note that most programs do not handle the middle mouse button because original Microsoft Mice did not have a middle mouse button.

These are the 3 basic actions you can paste:

Single Click - is a click and release of the mouse.

Double Click is a rapid click, release, click, release of the mouse.

Drag is a click, mouse move, release of the mouse.

If shift or control was held down during the mouse action - this section shows it.

Text Transfer Functions

Entering Text into an Application

There are many techniques for placing text into a program. The exact method you choose depends on the recipient of the text.

- For single line edit fields use <u>SetWindowText</u>
- For multiple line edit fields (such as Windows NotePad) use <u>SetMLText</u>
- For the main editing window of a Wordprocessor, Spreadsheet or Database, either transfer text into the clipboard with <u>SetClipboardText</u> and paste the text in with an appropriate menu command, or type the text in using <u>TypeText</u>.

The following section lists the various commands available for placing text into an application.

Getting Text from Other Programs

Obtaining text from applications is only slightly more difficult than placing it. The same basic strategies apply. If the window you want text from is a single line edit window use <u>GetWindowText</u>. A multi-line edit window such as NotePad or Flute's Edit window requires a <u>GetMLText</u> command. Obtaining text from the main window of a Wordprocessor or Spreadsheet is a multi-stage process:

- Use the mouse commands to select the text (or where there is a menu item called 'Select All' this is a better choice - see <u>SelectMenu</u>).
- Select the 'Copy' command to copy the text onto the clipboard.
- Use <u>GetClipboardText</u> to read the text off the clipboard.

An alternative way of extracting text from a program is to save the document or sheet as Text Only and use Flute's file commands to read the file.

See Also

<u>ShiftKey, ControlKey, AltKey</u>

TypeText

TypeText enters text into a window just as though it had been typed.

```
TypeText{wind_hand, text}
```

Where wind_hand is the handle of the window

Text is a string object to be typed into the window, or a number representing a key code.

If you are entering text into an edit field on a dialogue box it is often much faster to use <u>SetWindowText</u> or <u>SetMLText</u>. These two commands enter the text in one go, while this command enters the text character by character. This makes TypeText more suited to entering text into the main window of wordprocessors and editors.

Example:
```
TypeText{3304,"This is text"};
```

TypeText can also enter keys, such as F10, cursor up etc., which have no character equivalent. This is done by entering a number instead of a string - each special key has a corresponding number. Notice that not all keys listed in the following table are available on all keyboards.

Number	Key Assignment
8	Backspace key
9	Tab Key
12	Clear Key
13	Return/Enter Key
19	Pause Key
27	Escape Key
33	Page Up
34	Page Down
35	End Key
36	Home Key
37	Left Arrow Key
38	Up Arrow Key
39	Right Arrow Key
40	Down Arrow Key
41	Select Key
43	Execute
44	Print Screen Key
45	Insert Key
46	Delete Key
47	Help Key
96-105	Numeric Keypad 0 to 9
106	Multiply (Numeric Keypad)
107	Add (Numeric Keypad)
108	Separator Key
109	Subtract (Numeric Keypad)
110	Decimal Point
111	Divide (Numeric Keypad)
112	F1
113	F2
114	F3
115	F4
116	F5

117	F6
118	F7
119	F8
120	F9
121	F10
122	F11
123	F12
124	F13
125	F14
126	F15
127	F16
128	F17
129	F18
130	F19
131	F20
132	F21
133	F22
134	F23
135	F24
144	Numeric Lock
145	Scroll Lock

Example:

```
TypeText{"Microsoft Word",120}
```

Is equivalent to pressing F9 in Microsoft Word.

Multiple keys or strings can be entered with one command:

```
TypeText{"Notepad","This is line 1",13,"This is line 2",13};
```

Here two lines of text are typed into Notepad.

The Shift, Alt and Control keys can changed by TypeText.

Key	Down	Up
Shift	16	-16
Control	17	-17
Alt	18	-18

For example:

```
TypeText{"Notepad",18,"F",-18};
```

This is equivalent to pressing Alt-F inside Notepad. Notice that the Alt key is released by adding a -18 after the "F". Without this the Alt key would remain pressed.

ShiftKey, ControlKey and AltKey

These functions set the shift, control or Alt keys or return the key state as a function return.

`Shiftkey(TRUE)`	This is equivalent to pressing and holding the shift key down.
`ShiftKey(FALSE)`	This is equivalent to releasing the shift key.
`state:=ShiftKey;`	Here it is used as a function to return the state of the shift key - TRUE means the shift key is pressed, FALSE means the key is released.

SetMLText

Sets the text in a multi-line edit control. For most windows the command <u>SetWindowText</u> will set the text in that window or edit field. The exception is multi-line edit controls which store their text in a different way. If your attempt to set the text in an edit field fails, then try this command instead.

```
SetMLText{wind_hand,text}
```

Where wind_hand is the handle of the edit window whose text is to be set. You can use any of the ways of referring to windows listed under FindWindow for this parameter.

Text is a string object containing the text.

Example:

```
a:=FindWindow{"My Dialogue",{500,10}};
SetMLText{a,"This is a test piece of text"};
```

This first finds the handle to a multi-line edit window which is at location {500,10} inside the dialogue window called "My Dialogue". It then changes the text of that window.

SetClipboardText

This command places text onto the clipboard. This text can then be pasted into an application by selecting 'Paste' on its menu (refer to the Menu Commands for details on selecting menus).

SetClipboardText("Sample piece of text");

This command takes one parameter - the text object to place on the clipboard.

This command does not return a value.

GetMLText

Gets the text from a multi-line edit control. Although GetWindowText can normally get the text from a multi-line edit control you should use this function in preference. The reason is that there is no reliable way of detecting multi-line edit controls and although the GetWindowText function may detect it sometimes, other times it may not.

When getting text from normal windows, or single line edit windows, use GetWindowText.

```
GetMLText(wind_hand);
```

wind_hand is the handle of the edit window whose text you want. You can use any of the methods listed under FindWindow to specify the window you want the text of.

GetClipboardText

This function obtains any text that is on the clipboard and return it as a string object. Strings are limited to 37 thousand characters, if the text is any longer than this it will be cropped.

If there is no text on the clipboard this function returns a Null object.

```
a:=GetClipboardText;
```

File Functions

The following list shows functions that manipulate files. Flute provides the ability to manipulate some file formats, but it is worth remembering that Flute is a control program - it can save or load in **any** format, simply by opening other applications and using their file facilities.

- To read from a file; use OpenFile to open the file, use ReadFile, or ReadFileCS to read the data and finish off with CloseFile.
- To write to a file; Create it with CreateFile, use WriteFile or WriteFileCS to write the data and close the file with CloseFile.
- To modify an existing file; Open it with OpenFile, use FilePtr to set the position within the file to write data, use WriteFile or WriteFileCS to write the data and finish up with CloseFile.

Function List

OpenFile	Open an existing file for reading or writing
CreateFile	Create a new file
CloseFile	Close a file
ReadFile	Read data from a file
WriteFile	Write data to a file
ReadFileCS	Read CeSk data from a file
WriteFileCS	Write CeSk data to a file
CurrentDir	Set or return the current directory
WindowsDir	Returns the directory for Windows
StartDir	Returns the directory Flute was started from
FilePtr	Sets the read/write position within a file

OpenFile

This function opens the specified file for reading and writing. This takes the name of the file, or the full pathname of the file. The return is a handle that identifies the file to other file commands.

```
handle:=OpenFile{"Filename",access};
```

Where access=0 read only, 1=write only, 2=read or write

If you omit the access then read/write is assumed.

```
handle:=OpenFile{"Filename"};
```

CreateFile

This function creates a file. If the file already exists this truncates it to zero length. The return from this function is the handle identifying the file, or an error code.

```
handle:=CreateFile{Pathname,Access};
```

Pathname is a string object representing the filename and (optionally) the path.

Access specifies the access allowed when the file is opened on future occasions:

0=read only, 1=write only, 2=read or write.

If you create a file that is read only you can still write to it. Only once the file has been closed and re-opened with OpenFile does the read-only restriction apply.

If you omit the access value, read/write access is assumed.

Example:

```
CreateFile "c:\file.txt"
```

Creates a file with read/write access, called file.txt, on the root directory of drive c.

CloseFile

This function closes a previously opened file.

```
CloseFile(filehandle)
```

filehandle is the handle identifying the file that was returned from a previous call to <u>OpenFile</u> or <u>CreateFile</u>.

This function returns TRUE if successful, or an error if the handle is invalid.

ReadFile

Reads bytes (characters) from a file and returns them as a string object.

```
ReadFileString{handle,length};
```

The handle is the file handle previously returned by the <u>OpenFile</u> function.

If you omit the length a line of text is read in. The line is terminated with a Carriage return and/or linefeed character. These characters are not included in the string. The read position for the file is moved on to the start of the next line (i.e., *after* the carriage return or line feed).

String objects cannot be more than 32767 characters in length. If you attempt to read more than this number from the file, Flute returns an Error: String too long.

When ReadFile hits the end of file it returns a NULL object. This is distinct from a null length string which could simply indicate a blank line has been read in. In Flute a null string is equal to NULL (NULL="" is TRUE), so, to test for the end of a file test if the `Type` of the return is a string as in the following example:

```
do {
    /*read a line of text */
    a:=ReadFile{file_handle};
    /* reached end of file so exit loop*/
    if (type(a)!=4) break;
    /* manipulate the line of text */
} while (TRUE);
```

NOT

```
do {
    /*read a line of text */
    a:=ReadFile{file_handle};
    /* (wrong) reached end of file so exit loop*/
    if (a=NULL) break;
    /* manipulate the line of text */
} while (TRUE);
```

The second example will fail when an empty line of text is read because ""=NULL.

WriteFile

This function writes the bytes from a string object out to a file. If the data is not a string it is converted to a string representation.

```
WriteFileString{handle,data};
```

handle is the file handle returned by either <u>OpenFile</u> or <u>CreateFile</u>.

data is a string object that is to be written to the file.

Example:
```
myhandle=CreateFile{"c:\1.txt"};
WriteFileString{myhandle,"This is line 1"+13+10};
WriteFileString{myhandle,"This is line 2"+13+10);
Closefile(myhandle);
```

This example writes two lines of text out to the file 1.txt. Each line is terminated by a carriage return and line feed.

Adding a number to a string in this way adds the corresponding character. Character 13 is a carriage return, character 10 is a line feed.

The return from this function is TRUE if successful, or an error if there was a fault.

ReadFileCS

Read a block of CeSk data from a file. The data must have been written with WriteFileCS. ReadFileCS/WriteFileCS allows you to quickly read and write CeSk data into a file.

```
ReadFileCS(filehandle)
```

filehandle is the handle of the file to be read from. After the call the file pointer points to the next piece of CeSk data.

If the data does not look like CeSk data, an error is returned. The same error is returned if you attempt to read past the end of the file.

See the example after WriteFileCS.

WriteFileCS

Writes a block of CeSk data into a file. The file format is internal and should only be read by a <u>ReadFileCS</u> function.

WriteFileCS{filehandle,data};

filehandle is the handle of the file to write to, as returned from a previous OpenFile or CreateFile function.

data is any CeSk data structure.

Example:
```
filehandle:=CreateFile{"c:\1.dat"};
WriteFileCS{filehandle, {"Date of Birth", \ 3rd June 1970}};
WriteFileCS{filehandle, "Steven Blakey"};
CloseFile(filehandle);
filehandle:=OpenFile{"c:\1.dat"};
a:=ReadFileCS(filehandle);
b:=ReadFileCS(filehandle);
CloseFile(filehandle);
```

After this fragment of code, variable **a** contains the array {"Date of Birth", \ 3rd June 1970} and variable **b** contains the string "Steven Blakey".

CurrentDir

This function sets or returns the current directory and drive.

```
CurrentDir("c:\dos")
```

Sets the directory to \dos and the drive to C

```
pathstore:=CurrentDir;
```

Returns the Current Directory and assigns it to the variable pathstore

WindowsDir

Returns the full pathname for the main windows directory. This directory contains initialisation files, utilities and other files. It is a useful place to store options and settings.

The pathname is returned as a string object.

```
a:=WindowsDir;
```

StartDir

Returns the directory that Flute was run from as a string object. This function requires no parameters.

```
result:=StartDir;
```

FilePtr

This function gets or sets the position of the file pointer. When data is read or written to a file it is read or written at the position known as the file pointer.

```
a:=Fileptr(filehandle);
```

This retrieves the file pointer for the file whose handle is filehandle and assign it to variable a. The filehandle is returned by a previous call to OpenFile or CreateFile.

```
Fileptr{filehandle, newpos}
```

Passing an array of 2 values to FilePtr sets the position to newpos bytes from the start of the file.

```
FilePtr{filehandle,newpos,mode}
```

Passing an array of 3 values to FilePtr sets the position to newpos bytes from either the start, end or current position.

If mode=0 the offset is referenced from the start of the file. For example, if newpos is 100 and mode is 0 the file pointer is set to 100 bytes in from the start of the file.

If mode=1 the offset is from the current file position. Positive values move the pointer nearer the end of the file, negative values move it closer to the start of the file.

If mode=2 the offset is taken from the end of the file. Only negative values of newpos are valid. For example, -2 will move you 2 bytes nearer the start of the file.

The return from this function is the position of the file pointer from the start of the file, for example

```
filesize:=Fileptr{myfile,0,2};
/* move to end of file */
Fileptr{myfile,0,0};
/* move back to start of file */
```

This fragment of code obtains the size of a file, then moves the fileptr back to the start.

Adding functions to Flute

You can extend Flute by implementing your own functions in Dynamic Link Libraries (DLLs). Writing DLLs is a job for professional programmers. The information required to write your own DLLs is well beyond the scope of this documentation and the tools required are not provided with Flute.

Flute cannot simply call any function in any DLL. The function must be specially written to handle Flute's powerful fluid array structure.

The normal sequence of events for a DLL call is as follows:

DLLOpen   Open the library, load the code into memory

DLLCall   One or more calls to functions in the library

DLLClose   Close the library after use

However, to simplify this, you can use DLLCall to open the library for you and close it after the call. See DLLCall for details.

Example:

There is an example DLL, called SampDll.Dll provided with Flute. In it is the single function 'sumofpos' which adds together all the positive numbers in an array and returns the total as a double precision object.

The call for this function is:
```
data:={1,3,-2,6,7};
result:=DLLCall{"sampdll.dll","sumofpos",data};
ask{"Result was: "+make{result,4}};
```

The result 17 is displayed.

See Also

How Data is Passed to a DLL Function

DLLOpen

Open a Dynamic Link Library prior to calling functions in it.

```
Library:=DLLOpen(DLL_Filename);
```

This function takes the filename of the DLL as its only parameter and returns a number which identifies the DLL. This number is used when calling functions in a DLL and closing the DLL after use.

If the library could not be opened this function returns an error object.

Example:

```
Library_ID1:=DLLOpen("DLLCode1.dll");
Library_ID2:=DLLOpen(c:\"DLLCode2.dll");
```

If the pathname is not specified, then DLLOpen will search for the DLL as follows:
- The current directory
- The Windows Directory
- The Windows System Directory
- The directory Flute was started from
- The directories listed in the Path command in the Autoexec.bat file
- The mapped in Network Directories.

DLLClose(Library_ID)

Closes a DLL after use, freeing the resources it uses.

```
Ret:=DLLClose(Library_ID);
```

This takes 1 parameter - the ID of the opened DLL as returned by DLLOpen.

The return is either a Boolean TRUE object if the function was successful, or an error object if the function failed.

DLLCall

Calls a function inside a Dynamic Link Library. You can only call specially written functions from within CeSk, the reason being that most DLLs are written to be called by programs written in 'C', and do not support the fluid arrays that CeSk supports.

```
result:=DLLCall{Library_ID, "Function_Name", parameters};
```

This function takes 3 parameters:

Library_ID is the number that identifies the opened library. This is returned by DllOpen.

Function_Name is a string object that identifies the name of the function within the DLL. Calling function this way is inefficient - if you know the ordinal number of a function within a DLL, you can use this number instead. Ordinal numbers are defined by the designer of the DLL.

parameters - optional parameters that are passed to the function in the DLL.

The return from this function is the data returned from the DLLCall, or an error object if the data is not recognised.

If you only plan to make one call in a particular library, then, rather than use DLLOpen, DLLCall and DLLClose, you can replace the Library_ID parameter with the filename of the DLL.

For example:
```
result:=DLLCall{"testdll.dll","sumfunction",{1,2,3}};
```

Here "sumfunction" is called inside the library "testdll.dll" with the array as its parameter {1,2,3}.

The library is opened, the call made and the library closed immediately - if you make many calls to the same library this way time is wasted opening and closing the library needlessly.

See Also

How Data is Passed to a DLL Function

How Data is Passed to a DLL Function

The parameter passed to the DLL is a 16 bit global handle (an HGLOBAL).

This HGLOBAL is freed by CeSk when the function returns, The DLL function should not free it, but can modify its contents.

The return from the function should also be a HGLOBAL to a memory object, in the same form as the data was passed.

If the function returns 0, then a NULL object is substituted by CeSk.

The data inside the global memory object depends on the type of data being passed. The following lists the 12 data types currently supported by CeSk.

```
/* Storage for an NULL object  */
struct t_int {
    short   vtype;      /*NULL is type 0 */
    long    value;      /*for a NULL,this is always 0*/
};
/* Storage for an integer 32 bit */
struct t_int {
    short   vtype;      /* INTEGER is type 1 */
    long    value;      /* The value of the integer */
    };

/* Storage for a floating point number */
struct t_float {
    short   vtype;      /* FLOAT types are type 2 */
    float   value;      /* the value of the FLOAT */
    };

/* Storage for a double floating point number */
struct t_double {
    short   vtype;      /* Doubles are Type 3*/
    double  value;      /* the value of the double */
    };

/* Storage for a string object */
struct t_string {
    short   vtype;      /* Strings are Type 4*/
    unsigned short   length;
                    /* the number of characters in the string (can be zero)
                    */
    char    value[];
                    /* the array of characters (NOT zero terminated )*/
```

```
    } ;
```

The length of the string structure is always rounded up to make it an even number of bytes long, so if the length field contains 1 the total length of the string object is 6.

String objects can contain zeros. Zero is NOT used to terminate the string.

```
/* storage for Boolean object */
struct  t_boolean {
    short   vtype;      /*Boolean TRUE/FALSE are type 5 */
    long   value; /*0=FALSE, anything else TRUE */
    } ;


/* Storage for a error object */
struct t_error {
    short   vtype;      /* Errors are type 6 */
    short   length;
                /* the number of characters in the error string */
    char    value[];
                /* array of characters giving the error (NOT zero
                terminated) */
    } ;
```

Errors are stored exactly like strings. The text of the error explains the problem. Just as with string objects, error objects are rounded up to make them an even number of bytes long.

```
/* storage for a date object */
struct  t_date {
    short   vtype;      /* Date objects are type 7*/
    short   day; /* Day of month 1-31 */
    short   month;      /* Month 1-12 */
    short   year; /* Year (e.g. 1994) */
    } ;
```

It is possible to produce date objects that have values outside the correct ranges using overdimensioning. A DLL should not assume that a date is a valid one.

```
/* storage for a time object type */
struct t_hms {
    short   vtype;      /* time objects are type 8 */
    short   hours;      /* number of hours (0-23) */
    short   minutes;    /* minutes 0-59 */
    short   seconds;    /* seconds 0-59 */
    };
```

It is possible to produce time objects that have values out of range using overdimensioning. A DLL should not assume that a time is a valid one.

```
/* storage for a complex number type */
struct t_complex {
   short   vtype;       /* Complex numbers are type 9 */
   double   realpart;
                   /* The real part of the complex number */
   double   imagepart; /* the imaginary part */
   };


/* storage for an equation */
struct t_equation {
   short   vtype;       /* equations are type 10 */
   short   length;
                   /* the length of the equation in bytes */
   char    value[]; /* length bytes of information */
   } ;
```

The length of equation objects are rounded up to an even number of bytes. The format for equation objects is undocumented and subject to variations - make no assumptions about it.

```
/* storage for an array */
struct t_array {
   short   vtype;       /* arrays are type 11 */
   short   elements;/*number of elements in array */
   char    value[];    /* the data in the array */
   };
```

Arrays simply group the other objects, the data for the array immediately follows the array header.

Example:

The CeSk array {1.2, "ABC", {1+2i}} would be passes to the DLL as:

Type	Value	Comments
short	11	Type is array
short	3	3 elements in Array
short	3	Type is Double
double	1.2	Value of double is 1.2
short	4	Type is string
char[3]	ABC	3 Characters of string information
char[1]	0	Padding character to make string even number of bytes long
short	11	Type is array
short	1	1 element in array
short	9	Type is Complex Number
double	1	Real part is 1
double	2	Imaginary part is 2

This is almost exactly the same format as data is written to disk in the WriteFileCS command. The

difference is that a long value precedes the data - this is the number of bytes of data that follow.

In the above example, if we had written the data to file using WriteFileCS the file would contain:

Type	Value	Comment
long	42	Length of data that follows
char[42]	.....	42 bytes of data
long	n	Length of next block of data
char[n]	....	Next data block

Communicating via DDE

Many applications implement Dynamic Data Exchange (DDE) - a system of inter program communications. Flute can talk to these application using its DDE functions.

The provider of DDE services is called a server, Flute acts as a client of the server.

The normal sequence of events when communication is:

DDEConnect   Connect to a Server

DDEPoke/DDEPeek/DDEExecute
                    These three commands tell the server to perform an action, or transfer data.

DDEDisconnect   To close the conversation.

However, you may find it simpler to use the facility within DDEPoke, DDEPeek and DDEExecute to make the connection, perform the command and then disconnect. See the documentation on these functions for details.

DDEConnect

Make a DDE Connection to a DDE server.

```
conv:=DdeConnect{"Service","Topic"};
```

This takes two parameters, the service name and the topic name, and connects to the server that provides those services on that topic. If there are several servers available that provide matching services and topics then only **one** connection is made to **one** of the servers.

Service and Topic names are defined by the servers. If you want to make a connection to a particular program refer to the documentation for that program to obtain the Service and Topic names it supports.

The service name and/or topic name can be replaced with a NULL to indicate that any service or topic name will do.

```
conv:=DdeConnect{"ProgMan",NULL};
```

Here the connection is made to the service "Progman" about any topic. "Progman" is a service provided by the Program Manager.

```
conv:=DdeConnect{Null,"System"};
```

Here a connection is made to any service provider that supports the topic "System". If there are several only one conversation to one of the servers is started.

The return from this function is a number that identifies the connection. You can have several conversations running at any particular time. This number is used to identify which connection the messages refer to.

If no suitable server is found this function returns an appropriate error.

DDEDisconnect

This ends a conversation with a DDE server.

```
result:=DdeDisconnect(conv);
```

The conv parameter is the number that identifies the conversation, this number is returned by DDEConnect.

The return from this function is TRUE if the disconnection was successful, or an appropriate error code if there was a problem.

DDEList

Lists available DDE Services and Topics.

```
result:=DdeList{"Service","Topic"};
```

This takes two parameters which specify the service or topic names which DdeList is to consider.

```
result:=DdeList{NULL,NULL};
```

If both parameters are NULL then all Services/Topic combinations are listed. For example the result might be:

{{"windserve","system"},{"testserve","system"},{"testserve","document1"}}

```
result:=DdeList{"testserve",NULL};
```

If a service name is provided, but the topic name is NULL, then all topics available from servers that provide the specified service are listed. In the above example an array of service and topic names are provided. All the service names will be "testserve". The result from this call might be:

{{"testserve","system"},{"testserve","document1"}}

```
result:=DdeList{NULL,"System"};
```

If the topic name is provided, but the service name is NULL, then a list of all services that relate to the specified topic are returned, together with the topic name. The result might look like this:

{{"windserve","system"},{"testserve","system"}}

Example:

The following example obtains a list of all the services provided about the topic "Program" then connects to the first server.

```
result:=DdeList{NULL,"Program"};
if (size(result)>0) {
    conv:=DdeConnect(result[0]);
} else {
    /* no server suitable */
};
```

result[0] is a two dimensional array containing the service and topic names. DDEConnect(result[0]) therefore connects to the correct service and topic. Notice the round brackets ( ) around the parameters. Result[0] is already an array, putting curly brackets around it would make it into a two dimensional array.

Note; This may not give you a complete list of available services or topics. Some servers do not process all the DDE message required to produce the list and do not appear within the list of servers.

DDEPoke

DDEPoke sends data to a DDE Server. This is typically used to enter text into a wordprocessor via DDE or enter data into a database or spreadsheet.

```
result:= DDEPoke{Conv, Item, Data}
```

Conv; The number that describes the connection to the server. This is the number returned by DDEConnect. (See note below).

Item; A string object describing the item of data being sent. This is defined by the DDE Server you are poking data into. For a Spreadsheet it might define the row, and column to place the data at. For a database the name of the data field to place the data at. If the DDE Server does not need this field use NULL or "".

Data; A string object representing the data being poked. Some server applications require data to be sent in formats other than text. You cannot poke data into these applications from CeSk. If the application does not support text transfer, use the other commands in Flute to manipulate the application directly.

The return from this function is TRUE if successful, or an error object if the DDEPoke failed. If the Server is not responding, DDEPoke will wait for 10 seconds to see if it comes back to life.

Example:
```
conv:=DdeConnect{"Bigword","document1"};
if (conv) {
    result:=DdePoke{conv,NULL,make{today,4}};
    DDEDisconnect(conv);
};
```

This example starts a conversation with the wordprocessor 'Bigword' about 'document1' and enters the current date into the document. Note that make{today,4} takes the current date and converts it to a string of text.

If you are only making *one* DDEPoke into a server then, rather than using DDEConnect, DDEPoke and DDEDisconnect, you can specify an array containing the server name and topic name in place of the conv parameter.

For example,
```
DDEPoke{{"Bigword","document1"}, NULL,make{today,4}}
```

is exactly the same as the example given earlier. It makes a connection, does the DDEPoke, then terminates the connection.

DDEExecute

DDEExecute tells a DDE Server to execute a series of instructions. The exact format for the instructions depends on the server but, typically, they are macro commands.

```
result:= DDEExecute{Conv, Instructions}
```

Conv; The number that describes the connection to the server. This is the number returned by DDEConnect. (See note below).

Instructions;   A string object representing the instructions to be executed by the server.

The return from this function is TRUE if successful, or an error object if the DDEExecute failed. If the Server is not responding DDEExecute will keep trying for 10 seconds before returning.

Example:
```
conv:=DdeConnect{"progman","progman"};
if (conv) {
    result:=DdeExecute{conv,"[CreateGroup(Test)][ShowGroup(Test,1)]"};
    DDEDisconnect(conv);
};
```

This example tells the server that provides the "progman" service (the Program Manager) to execute the command "[CreateGroup(Test)][ShowGroup(Test,1)]" - this creates and shows a group window called "Test".

If you are only instructing a server to perform one DDEExecute, then rather than using DDEConnect, DDEExecute and DDEDisconnect, you can specify an array containing the server name and topic name in place of the conv parameter.

```
DDEExecute{{"progman","progman"},"[CreateGroup(Test)][ShowGroup(Test,1)]"};
```

This instruction does exactly the same as the previous example. It makes a connection, does the DDEExecute, then terminates the connection.

DDEPeek

DDEPeek requests data from a DDE Server. This is typically used to obtain text from a wordprocessor that supports DDE, or obtain data from a database that supports DDE.

```
result:= DDEPeek{Conv, Item}
```

Conv; The number that describes the connection to the server. This is the number returned by DDEConnect. (See note below).

Item;   A string object describing the item of data to be sent. This is defined by the DDE Server you are asking for data. For a Spreadsheet it might define the row and column to get the data from. For a database, the name of the data field to provide the data for. If the DDE Server does not need this field use NULL or "".

The return from this function is the data requested if successful, or an error return if the function failed. If the server is not responding DDEPeek will try for 10 seconds before returning an error. The data is always requested from the server as text and returned as a string object.

Example:
```
conv:=DdeConnect{"SmallSheet","document1"};
if (conv) {
    result:=DdePeek{conv,"A1:B4"};
    DdeDisconnect(conv);
    if (type(result)!=4) {
        /* an error occurred , data not received*/
    };
};
```

This example starts a conversation with the spreadsheet 'SmallSheet' about 'document1' and requests the contents of cells A1 to B4. If the type of data is not type 4 (string objects) then an error has occurred.

If only one piece of data is required then the conv parameter can be replaced by an array containing the Service name and Topic name.

For example:
```
result:=DdePeek{{"SmallSheet","document1"},"A1:B4"}
```

Will make the connection, ask for the data and disconnect afterwards.

Control Structures Within CeSk

CeSk implements a range of control structures, loops, IF's and so on, in a similar way to all modern languages.

IF(expression) {action} ELSE {action}

WHILE (expression) action;

DO action WHILE (expression);

FOR (initaction;condition;postaction;) mainaction;

SWITCH (expression)

BREAK

BREAK2, BREAK3, BREAK4, BREAK5

CONTINUE

BREAKCONTINUE

BREAK2CONTINUE

RETURN

Variables and Their Scope

IF (expression) ...ELSE...

The IF..ELSE statements switches the flow of execution of a program depending on a TRUE/FALSE test.

```
if (expression)
    trueaction;
else
    falseaction;
```

The (expression) must be bracketed with ( ).

The value of (expression) is evaluated and if found to be TRUE (any non-zero result) then the statements represented by 'trueaction' are executed, otherwise the (optional) statements represented by 'falseaction' are executed.

Note that (expression) can consist of any test that returns a TRUE or FALSE, zero or non-zero, e.g. a Boolean, a number, relative comparison of strings, etc.   Where the test involves the use of floating point values, which cannot always be represented exactly, then CeSk will regard any value between -1e-15 and +1e-15 as being zero (or FALSE).

If there is no action to be taken for a FALSE result, then the ELSE clause may be skipped.

```
if (expression)
    trueaction;
```

Should you wish a group of statements to be run for 'trueaction' or 'falseaction', then the statements should be grouped using { } brackets.

For example:

```
if (abc) am := 1; else pm := 1;
```

If the variable 'abc' is non-zero (or TRUE), then the variable 'am' is set to the value 1, otherwise the variable 'pm' is set to 1.

```
if (abc > 2) {am := 1; hm -= 12;} else {pm := 1; hm += 12;};
```

If the variable 'abc' is greater than 2, then the variable 'am' is set to 1, and the variable 'hm' has 12 subtracted from its current value. If 'abc' is less than or equal to 2, then the variable 'pm' is set to 1 and the variable 'hm' has 12 added to its current value.

Note that the first set of curly brackets do not end in a semicolon, but the second set does, to terminate the IF command.   If there was no ELSE group then the semicolon would appear after the first closing curly bracket.

IF statements can be nested, but when this is done the statements to be run as trueaction or falseaction **must** be enclosed within { }'s, e.g.

```
if (abc > 2) <--------- Start of 1st level IF
   {              <------- Start of trueaction for 1st level IF
     if (pm) {       <--------- Start of 2nd level IF
        kom := 1;   <--------- Trueaction for 2nd IF/end of 2nd level IF
   }               <--------- End of trueaction for 1st level IF
else
```

```
{                <--------- Start of falseaction for 1st level IF
   if (xyz) {        <--------- Start of 2nd level IF
      tim:= "Hello";              <--------- Trueaction for 2nd level IF
   };              <--------- end of 2nd level IF
};   <--------- End of falseaction for 1st level IF
```

The indentation is simply for illustration, Flute ignores how a program is indented.

This is different from the language 'C', which doesn't require nested IF's to be enclosed in { }.

WHILE (expression) action;

WHILE evaluates the expression and, if TRUE, executes the action. It then repeats the process, evaluating the expression and if TRUE, executing the action, finally stopping when (expression) evaluates to FALSE.

(expression) must be given within the ( ) brackets, and if several actions are required as part of the loop they should be enclosed within { } brackets.

For example:

```
while (abc > 2) abc -= 1;
```

While the value held in variable 'abc' is greater than 2, subtract 1 from 'abc', loop and then retest.

```
while (abc > 2) {abc -= 1; kom[abc] := 3;};
```

While the value of 'abc' is greater than 2, subtract 1 from 'abc', and then set kom[abc] to 3. Return to the start of the loop and retest. Note carefully the semicolon after the closing bracket, used to terminate the WHILE statement.

Contrast the `while` loop with the <u>do</u> loop. The `while` loop tests the condition *before* the action is executed, the `do` loop tests the condition *after* the action is executed. Hence the do loop always executes the action at least once, but the while loop may never execute the action at all.

DO action WHILE (expression)

The DO ... WHILE is a variant of the WHILE loop. It differs in that the actions are always executed at least once, since the test does not occur until the end of the loop.

Contrast this with the normal WHILE loop, which tests at the top of the loop and hence the action may never be performed.

For example:

```
do abc -= 1; while (abc > 2);
```

Subtract one from 'abc'. If 'abc' is greater than 2, repeat.

```
do {abc -= 1; kom[abc] := 3;} while (abc > 2);
```

Subtract 1 from 'abc'. Use the current value of 'abc' to set list element 'kom[abc]' to 3. If 'abc' is greater than 2, repeat.

The expression **must** be enclosed in brackets ( ).

The multiple actions must always be enclosed in curly brackets { }.

FOR (initaction; conditional; postaction;) mainaction

The FOR loop is another form of looping construct, generally used where the flow of a `while` or `do...while` loop would obscure the logic of the program.

It first evaluates 'initaction', and then performs the 'conditional' test. If the 'conditional' test returns a FALSE result, then the loop is skipped; if the test returns a TRUE result, then the main body of the loop is run. After the main body of the loop has run 'postaction' is then evaluated.

This is equivalent to the construct:
```
initaction;
while (conditional) {
    mainaction;
    postaction;
};
```

Whether you opt for this style of loop, or for a For loop depends on the situation.

As an example:
```
for (x := 0; x < 10; x += 1;) abd[2*x] := 3;
```

Set x to 0. If x is less than 10, set abd[2*x] to 3. Add one to x, loop and retest x.

which as a WHILE loop would be
```
x := 0; while (x < 10) {abd[2*x] := 3; x += 1;};
```

This is different from the C programming language in that a semicolon **must** be placed after the post action:

for (x := 0; x < 10; x += 1;)   /* semicolon correct */

for (x := 0; x < 10; x += 1)   /* semicolon missing */

SWITCH (expression)

The SWITCH statement allows a multi-way choice to be made without requiring several levels of complex IF ... ELSE ... statements. It takes the form

```
switch (expression)
   {
      case (value1):   action1;  break;
      case (value2):   action2;  break;
               .
               .
      default:  defaultaction;
   };
```

The value of (expression) is calculated, and then compared in turn to each of the values given in the case selections. If a match is found then the actions specified for that case are executed. A default selection can be used as a catch-all if none of the case tests are explicitly matched.

Notice a case construct close up `case (10) :`

The case ends in a colon, which tells CeSk where the next statement begins.

Note that each of the values attached to a case must be enclosed within ( )   brackets - this is different from the C convention, which does not require the ( ).

If you wish to use several statements for a particular matching case, then the statements do **not** need to be enclosed within { } brackets.

The case tests can be in any order.

Unless you use a BREAK statement to exit the switch execution will continue through any CASE statements below the matching one.

For example, assuming the variable abc currently has the value of 10, then

```
switch (abc - 2) {
   case (3) :
   default:      abd := 3;
   case (10):   mik := 5;
};
```

Then because the expression (abc - 2) does not have an explicitly matching case value of 8, the default selection is taken, setting abd to the value 3. Since there is no BREAK terminating this group the code then falls through to the statements associated with the case(10) test, setting the variable 'mik' to 5. Note that 'abd' will also be set if abc has a value of 5, since the case(3) match will fall through to the default code; only if abc has the value 12 will abd not have a value assigned.

The switch statement is exactly identical to groups of IF statements.

```
Switch (abc) {
   case (1): a:=1; break;
   case (2): b:=1; break;
   case (3): c:=1; break;
```

```
        default: d:=1; break;
};
```

Is equivalent to:

```
If (abc=1) a:=1;
else {
    if (abc=2) b:=1;
    else {
        if (abc=3) c:=1;
        else {
            d:=1;
        };
    };
};
```

Notice that the equivalent IF statements are nested in curly brackets. CeSk insists you use { } brackets when IF's are nested.

BREAK

BREAK is used to step up from the deepest <u>WHILE</u>, <u>DO</u>, <u>FOR</u> or <u>SWITCH</u> construct by one level. It will not break out of a series of IF statements. The loop or switch command is terminated, and the code is continued from just after the closing bracket for the construct. For example, using a switch statement:

```
abc:=8;
switch (abc - 2)  {
    case (3):
    default:     abd := 3;
                 break;
    case (10):   mink := 5;
};
```

Here the break statement breaks out of the Switch construct, after executing abd:=3;

```
a:=0; c:=0;
while (a<10) {
    c+=a;
    if (c>5) break;
};
d:=1;
```

Here the break statement breaks out of the while loop when c is greater than 5 and execution continues at the line d:=1;

BREAK2, BREAK3, BREAK4, BREAK5

The above statements all work in a similar manner to BREAK, except that they will break out of the 2, 3, 4 or 5 deepest levels of <u>WHILE</u>, <u>DO</u>, <u>FOR</u> or <u>SWITCH</u> constructs, i.e.

```
while (abc > 3) {
  mik := 5;
    while (mik != 0) {
        if (mik = 3 & abc = 1) break2;
        mik -= 1;
    };
    abc -= 1;
};
/* when mik is 3 and abc is 1, break2 causes execution to continue here */
```

When mik equals 3 and abc equals 1, then the Break2 will terminate *both* WHILE loops.

CONTINUE

The CONTINUE statement continues execution at the end of a <u>WHILE</u>, <u>DO</u> or <u>FOR</u> loop, skipping intermediate statements.

```
Cnt := 0
while (abc > 10) {
    abc -= 1;
    if (abc > 6) continue;
    Cnt += 1;
};
```

When abc is greater than 6, the statement Cnt+=1; is skipped and the loop continues with its next iteration.

BREAKCONTINUE

Breaks out of the innermost loop and continues with the next iteration of the next outer loop. As its name implies, it performs the equivalent of a <u>BREAK</u> followed by an immediate <u>CONTINUE</u>.

You may wonder why this is different from simply putting Break; Continue; within a program. If you were to put Break; then follow it with Continue; the continue would never be executed. Break breaks out of the loop.

BREAK2CONTINUE

This statement breaks out of the two deepest loops and continues on the next outer loop.

In effect it executes two <u>Break</u> commands and follows it with a <u>Continue</u> command.

RETURN

Return is used to transfer control back from a function to the function's caller.

It can just return to its caller, or return a result for use of the caller e.g.

`return;`   just returns

`return {abc,"a"};`   returns an array of 2 values, the contents of variable 'abc', and the string "a".

If this command is used within the main{} function then the program stops. The return value from main{} is ignored by Flute.

Variables and their Scope.

Variables do not have to be explicitly declared - simply assigning a value to a variable name (this is a difference from the programming language C) both creates the variable and assigns the initial value. Within a function, variables that are created are automatic by default. i.e. When the function exits the variable will be destroyed. For example, consider the following functions

```
MyFunc {
    abc += 1;
    return(abc);
};
Main{
    V1 := MyFunc;
    V2 := MyFunc;
};
```

After the calls in Main{ }, both V1 and V2 would equal 1, since the value of the variable 'abc' in function MyFunc{ } is *lost* when MyFunc{ } terminates, and thus starts as a Null object The call then becomes null+1 -> 1, which is the value returned to Main{ }.

Variables are also local within the function. You can use the same variable name in any number of functions and they will all act independently.

```
MyFunc{ abc := 20}
Main{ abc := 100; MyFunc;};
```

With the function Main, abc has the value 100, while during the call to MyFunc{ }, the local variable abc contains the value 20. After the call to MyFunct, abc is once again 100; The value of 20 only applied *inside* of MyFunc.

Normal variables are limited to the function in which they are used.

However, by prefixing a variable with one of the following prefixes, these variables can be seen by all functions inside the program.

l_   (L underscore)

ls_   (LS underscore)

los_   (LOS underscore)

lo_   (LO underscore)

g_   (G underscore)

gs_   (GS underscore)

w_   (W underscore)

wp_   (WP underscore)

Any variable with any one of these prefixes can be seen by all functions, e.g.

```
MyFunc{ l_abc := 20}
Main { l_abc := 100; MyFunc;};
```

Now when main has finished running, l_abc contains the value 20 because the function MyFunc has set it to be 20.

In CleanSheet these various prefixes mean different things. They are retained in Flute for consistency between the two products, but in Flute they are all the same.

There is another very important feature of variables beginning with these prefixes. When the program is saved **these variables are saved with it**. This allows you to store settings in the variables and retrieve them the next time the program is loaded. If you change a variable and want it saved it is a good idea to flag the program as having been edited, using the <u>Changed</u> function.

Controlling CleanSheet from Flute

CleanSheet is the spreadsheet from WorkingTitle. Code has been implemented in versions 1.12 of CleanSheet and onwards to support these functions. If you have a version prior to 1.12, request an upgrade.

There are two commands for sending and receiving data from CleanSheet, CSPoke (send) and CSPeek (receive). Obviously Flute shares the system of fluid arrays with CleanSheet and has exactly the same data types - so it would be a waste not to pass this data directly between the two applications.

There are also a few commands to simplify mode settings within CleanSheet. These commands could be implemented using mouse clicks and menu selections, but it makes the program easier to read if the appropriate command is used.

The example 'automate.flu' is an example Flute program to plot equations.

Functions and Commands

CSPoke   Poke data into CleanSheet

CSPeek   Peek data from CleanSheet

CSRecalc   Recalculate a sheet

CSMode   Change CleanSheet's mode

CSAutoRecalc   Switch on/off automatic recalculation

CSSelect   Select an Object in Cesk

Other Topics

Preparing CleanSheet

Opening a CleanSheet Worksheet

Preparing CleanSheet

Before a worksheet is manipulated CleanSheet must be running and the worksheet loaded. The following fragment of code illustrates how to start CleanSheet.

```
/* is CleanSheet already running? */
if (!findwindow{"CleanSheet"}) {
   if (l_where=null) {
     l_where:="d:\csheet";
   };
   while (!execute{l_where+"\csheet.exe","",l_where}) {
     /* lets ask where Cleansheet is */
     new_where:=ask{"Which directory is CleanSheet in?",l_where};
     if (type (new_where)=6) {
       /* cancel button selected */
       ask{"Could not start CleanSheet"};
       return;
     };
     l_where:=new_where;
     changed(true);
   };
};
/* CleanSheet is running */
```

The first line, `if (!findwindow("CleanSheet"))` tests if CleanSheet is already running. If this test fails then it must be run with the `execute` function. CleanSheet does not register the directory it is stored in with the operating system, so this Flute program first uses the default directory "c:\csheet", and if that fails, it asks you to enter the directory, using the function `ask{"Which directory is CleanSheet in?",l_where}`. The result is stored in the variable l_where, which is saved when Flute exits. Once the directory has been entered, it will not have to be asked for again.

Opening a CleanSheet WorkSheet

After CleanSheet is running, the next step is to load the worksheet to be manipulated. Consider the following fragment of code. It first checks if the window is already open, and if it isn't it selects the 'Open' menu commands.

The setmltext command is then used to enter the path and name of the worksheet (in this example, our worksheet is called "automate.cln").

Finally a click with the left mouse button on the 'OK' button on this dialogue completes the operation.

```
if (!findwindow{"CleanSheet","automate.cln"}) {
   /* automate isn't already open, so open it */
   selectmenu{{"CleanSheet"},5013};    /* Open */
   wait(1000);
   setmltext{ findwindow{"Open",">edit"}, startdir+
           "\automate.cln"};
   lclick{{"Open","OK"},{24,8},activate};
   /* file is now loaded */
};
```

CSPoke

CSPoke pokes a piece of data into input objects on a CleanSheet worksheet. This command

It takes the form:

```
CSPoke{sheet_name, object_name, data}
```

sheet_name is the name of the worksheet to receive the data. Names are not case sensitive ("Untitled 1" = "untitled 1"). The name should be spelled out in full including the extension ("wave.cln" **not** "wave"). If there are several windows open to the same sheet CleanSheet numbers them with :1, :2, :3 etc. - these extra numbers must **not** be included within the name ("wave.cln", **not** "wave.cln:2").

object_name is the name of the CleanSheet object   (the name of a CleanSheet object is set using the Object Attributes dialogue in Edit Mode within CleanSheet).

data is the data (for example if you were sending an array, the data might look like {2,3,4}).

The return from this function is TRUE if successful, or an error object explaining the problem.

The objects you can poke data into are as follows:

Input Table - The data is overlaid onto the input table. For example if you poke {1,2,3} into the table, numbers 1,2,3 will appear across the top row of the table. (If the top row are headings and the input table has been set to remove those headings then the *second* row will contain {1,2,3} instead).

If you Poke a 2 dimensional array into an Input Table, then 2 dimensional data is pasted. {{1,2,3},{4,5,6}} will paste numbers 1 to 6 over an area 2 columns by 3 rows of the table.

Object Input Box - Poking data into an object input box is just the same as typing it in. For example, if the data is {1,2,3} then the object input box will contain an array {1,2,3}.

Text Input Box - A text representation of the object you Poked in is entered into the text box.

Button Group - If you Poke a single number into a Button Group, it is the same as clicking on the equivalent button with the mouse. The buttons in that case are numbers from 0, reading left to right, top to bottom.

If you Poke an array with 2 elements in, these are taken as x and y coordinates of the button. For example the data {2,3} is taken as Column 2, row 3.

Slider - This object accepts a number in the range 0 to 1 where 0 is the left, or top value, and 1 is the right or bottom value.

Tumbler - You can Poke an Angle (in degrees) into the tumbler object. This corresponds to the angle of the tumbler mark.

Polar Input Graph - This accepts an array of 2 elements {radius, angle}. The angle is either radians or degrees according to the settings on the Polar Graph.

XY Scatter Input Graph - This takes an array of 2 numbers {x,y} the x and y coordinates to input.

CSPeek

This function obtains the data from the first pipe carrying data into an object within CleanSheet. Normally this function is used on Output Objects to obtain the results of calculations.

The function takes the form:

`CSPeek{sheet_name, object_name}`

sheet_name is the name of the worksheet to get data from.

object_name is the name of the output table object to obtain the data from.

The return is either the data, or an error object indicating the problem.

NOTE: when worksheets are saved the data in the pipes is NOT saved. In order for there to be data in the pipes the sheet must have been recalculated since it was last opened.

CSRecalc

This function causes CleanSheet to recalculate the value of a sheet. It is exactly equivalent to clicking on the manual recalculation button with that worksheet window on top.

```
CSRecalc(sheet_name)
```

sheet_name is a string object containing the name of the worksheet to recalculate. Names are not case sensitive.

The return from this function is TRUE if successful, or an appropriate Error if communications failed.

Automatic recalculation is switched off after this command, just as it would be if you had clicked on the manual recalculation button.

Example:
```
CSRecalc("Wave.cln");
```

CSMode

This function sets the Use/Edit mode for a specified sheet.

```
CSMode{sheet_name, mode};
```

sheet_name is the name of the worksheet whose mode is to be set.

mode is a Boolean TRUE for Use Mode, or FALSE for Edit Mode. You can also use any non-zero value to represent TRUE and zero to represent FALSE.

The return from this function is TRUE if successful, or an appropriate error object.

Example:
```
CSMode{"Untitled 1",TRUE);
```

CSAutoRecalc

This function switches on or off automatic recalculation. This is equivalent to clicking on the automatic recalculation button within CleanSheet. If CleanSheet is in Use Mode when this is done, the worksheet will be recalculated.

```
CSAutoRecalc{sheet_name,autoflag}
```

sheet_name is the name of the worksheet to set automatic recalculation mode for. Note that CleanSheet does not store automatic recalculation settings with the work sheets, however it is anticipated that a future version will store this setting with the sheet.

autoflag - TRUE if automatic recalculation should be switched on, or FALSE if automatic recalculation should be switched off. Any non zero value can also be used in place of TRUE and zero in place of FALSE.

The return from this function is TRUE, if successful, or an appropriate error object if this function failed.

Example:
```
CSAutoRecalc{"Untitled 1",FALSE};
```

CSSelect

This function selects an object on a particular sheet. It is equivalent to clicking on the frame of the object. In Edit Mode you can also select objects by manipulating the 'Select Special' dialogue.

`CSSelect{sheet_name, object_name}`

sheet_name is a string object containing the name of the worksheet to select an object from.

object_name is a string object containing the name of the object, as set on the Object Attributes dialogue within CleanSheet. The names are case insensitive ("OBJ1" = "obj1").

The return from this function is TRUE if successful, or an appropriate error return if the function failed.

Example:

CSSelect{"Untitled 1","Frequency Shift"};

Debugging a Program

Debugging means fixing program errors, Flute provides a debugger to track and examine the flow and actions of your program. If the program window is visible, then Flute will highlight each line of code being executed *as* it is being executed.

BreakPoints

To stop execution of a program at a particular point, set a breakpoint.
- To set a breakpoint on a line, place the cursor anywhere on the line and select 'BreakPoint' on the 'General' menu, or select the breakpoint button on the button bar.

- To clear a breakpoint, place the cursor onto the line and select 'BreakPoint' on the 'General' menu. Or select the breakpoint button on the button bar.
- To clear breakpoints for a whole range of text, select the text and select the 'Breakpoint' menu. When text is selected this menu option clears any breakpoints within the selected area.

When the program is run, and execution reaches a breakpoint line, the program stops and the debug window appears. If the debug window is already up, the program is simply paused.

Debug Window

This dialogue allows you to monitor the progress of a program, examine variables, pause and stop programs.

Click on any part of the dialogue you need help with.

Run   Begins execution of the program; the program is immediately paused allowing you to single step through it.

Step   Steps through one *phase* of execution - this is **not** one line of code. Flute breaks complex expressions down into simpler ones, the debugger allows you to see what is being calculated (and when the fault you are searching for occurs). By repeatedly clicking on Step you can examine in detail the execution of the program.

Stop   Stops execution of the program. The program must be run from scratch after this command.

Continue   When the program is paused the Run button is renamed Continue. Clicking on Continue executes the program continuously until Pause is selected.

Pause   Pauses the program and switches back into single step mode.

The section labelled 'Instruction Being Executed' displays the command being evaluated. If you consider the following line of code you can see how this code breaks down into individual commands.

```
a:=make{make{"sin(degtorad(30))",10},3};
```

The commands look like this:

```
make{"sin(degtorad(30))",10}
```
        this converts the text to an equation

```
make{sin(degtorad(30)),3}
```
        this converts the equation to a number

```
degtorad 30
```
        converts 30 degrees to radians

```
sin 0.52359877
```
        the sin function is evaluated

```
:= 0.5
```
assign the value 0.5

You can see from this step by step execution that this combination of two `make` commands evaluates an equation and returns the result.

If your program is long and you do not want to step through all of it run the program continuously until the piece of code to be examined, then select 'Pause' and single step through just that section.

If you want Flute to automatically stop at the position to be examined and display the debug window simply add the following command into your program:

```
SelectMenu{"Flute",5010};
```

This command selects Debug Window from the General Menu of Flute.

The variables section lists what variables are in memory. To examine one of these variables select the group (these correspond to the prefixes explained under <u>Variables and Their Scope</u>), select the name of the variable, and its content will be displayed in the text window.

This section shows the instruction being executed. Instructions consist of 1 function, or 1 calculation - if you are tracing through a complex expression, this section will show each step of the evaluation.

Variables are stored in groups, for example all variables beginning I_ are in group I_. These groups have special meanings, refer to <u>Variables and their Scope</u> for details.

To show a particular variable, select the group, then the name of the variable. The contents of that variable will be shown in CeSk array notation.

This section shows the variables that have been defined in the selected group.

The contents of the selected variable are shown in this section of the dialogue.

Run - Starts execution of the program, the program immediately switches into Pause mode to enable you to single step through it.

Continue - When in pause mode, selecting Continue will run the program continuously.

Step - Steps through one instruction when the program is paused.

Pause - Pauses execution of the program.

Stop - stops execution of the program, after this the program must be restarted using 'Run'.

When the program is running continuously, you can adjust its speed using this slider.

Comparator Functions

This group of functions is used for comparisons between objects. Their main use is in the <u>IF</u> structures and within the <u>WHILE</u>, <u>DO</u> and <u>FOR</u> loops.

<u>= Equality and !=Inequality</u>

<u>< and > Greater Than and Less Than</u>

<u>! Logical NOT</u>

<u>& Grouping AND</u>

<u>| Grouping OR</u>

<u>>=<,>==<,>===<,>====< Vague Equalities</u>

<u><=>,<==>,<===>,<====> Vague Inequalities</u>

<u>>>, >>>, >>>>, >>>>> Vague Greater Than</u>

<u><<,<<<,<<<<,<<<<< Vague Less Than</u>

= Equality and != Inequalities

= Equality Operator

This operators tests for equality between two compared objects, returning a Boolean TRUE if the objects are the same and a Boolean FALSE if they aren't.

!= Inequality Operator

The inequality operator is exactly the reverse of the equality operator, returning TRUE if the objects are not identical.

Rules

Numeric = Numeric	Numeric objects (of whatever form) are always compared by value. If the values are the same to within 1e-15th of their magnitudes then the numerics are equal. This small value is to allow for slight rounding errors within floating point numbers.
Complex = Complex	An equality test on two complex number objects will only return TRUE if both the real and imaginary parts have the same values in each object - all other values will return a FALSE result. Again, a slight error of 1e-15th of their value is allowed.
Array = Array	An equality test on two lists or arrays will only return TRUE if all corresponding element are equal, e.g. $\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$ and

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} != \begin{pmatrix} 1 & 2 \\ 3 & 3 \end{pmatrix}$$

Null = Numeric A value of zero (0) is considered equal to a NULL object
Null = String An Null is considered equal to a string if the string has no characters and hence no length.

String = String	Two strings are considered equal if they match character by character, ignoring the case or attributes (accents, cedillas etc.) of any letters. Any punctuation character is equivalent to any other punctuation character found at the same offset in each string, i.e. "clean-sheet" = "clean?sheet" but "clean-sheet"!= "cleanasheet".
Error = Error	Error objects can be compared. The same rules apply as for strings.
Date = Date	Date objects are compared like for like. Two dates are consider equal if they represent the same day, e.g. \12/06/93 = \12th June 1993 = \Saturday, 12 June 93
Time = Time	Time objects are also compared like for like, e.g., 11:12pm = 23:12:00 = 11:12:00pm

< and > Greater Than and Less Than

The < and > operators compare two objects to determine their relative values - is one less than, or greater than, the other.   A Boolean TRUE or FALSE object is returned.

Rules

The list of rules mostly represent the Greater Than comparison. To obtain the Less Than comparison, simply swap the left and right sides of the > sign.

Numeric > Numeric	Numeric objects are compared by value, returning TRUE if the left is greater than the right.
String > String	Strings are compared according to their relative position in the alphabet, e.g. "Nigel" > "Fred" will return TRUE, while "B" < "A" returns FALSE. Accented characters within the string have the accents ignored during comparison, and upper and lower case are treated equally.
Date > Date	Date objects are compared chronologically e.g. \12/06/93 > \11th June 1993
Time > Time	Time objects are compared on a 24-hour clock basis, with midnight being 00:00:00.

❌ Complex numbers cannot be compared for magnitude, it is mathematically nonsensical so do so.

<= and >= Less Than or Equals and Greater Than Or Equals

These comparisons are composites of the Greater Than and Equals, and the Less Than and Equals comparison. Their behaviour is therefore very predictable.

Rules

Numeric >=Numeric	Numeric objects are compared by value, returning TRUE if the left is greater than or equal to the right.
String >= String	Strings are compared according to their relative position in the alphabet, e.g. "Nigel" >= "Fred" will return TRUE, while "B" <= "A" returns FALSE. Accented characters within the string have the accents ignored during comparison, and upper and lower case are treated equally, hence "John" >= "JOHN"
Date >= Date	Date objects are compared chronologically e.g. \12/06/93 >= \11th June 1993
Time >= Time	Time objects are compared on a 24-hour clock basis, with midnight being 00:00:00.

Complex numbers cannot be compared for magnitude, but they can be compared for equality hence this operator returns TRUE if they are equal, and FALSE if they are not.

! Logical NOT Operator

The logical Not reverses the sense of any comparisons. To do this it must be to the left of the comparison, or object.

When placed to the right of a calculation this operator behaves as a Factorial function.

Examples

!TRUE = FALSE   Not TRUE equals FALSE

!(1=2) = TRUE   (1=2) is FALSE, the NOT operator reverse this to make it TRUE

See Also

! Factorial Operator

& Grouping AND

The & operator groups several related comparisons into one composite.

If both sides are TRUE (or non-zero for numeric values), then a Boolean TRUE is returned. In other words they group comparisons together.

Example:

(a=2 & b=2) is TRUE only if a=2 AND b=2

| Grouping OR

The | operator groups several related comparisons into one composite.

If either side is TRUE (or non-zero for numeric values), then a Boolean TRUE is returned. In other words this operator groups comparisons together.

Example:

(a=2 | b=2) is TRUE only if either a=2 OR b=2

>=<,>==<,>===<,>====< Vague Equalities

The above operators all access the <u>Compare</u> function, to test that two objects are equal within a specified % of error.   The number of '=' symbols give the % error to use, having the ranges +/- 5%, +/- 10%, +/- 25% and +/- 50% respectively.   They all return a Boolean TRUE or FALSE result.

For example, the test "Nigel Johnstone" >==< "Nigel Johnson" returns TRUE, but the test "Nigel Johnstone" >=< "Nigel Johnson" returns FALSE.

Note that the >< signs point inwards to indicate that the result is within a specified range of vagueness. Also notice that the narrower the symbol the narrower the tolerance to differences.

It helps to think of these operators by associating names with them. For example   A >=< B means A is very close to B. A>====<B mean A is fairly close to B and so on.

<=>,<==>,<===>,<====> Vague Inequalities

The above operators all access the <u>Compare</u> function to test that two objects are not equal within a specified % of error.   The number of '=' symbols give the % error to use, having the ranges +/- 5%, +/- 10%, +/- 25% and +/- 50% respectively.   They all return a Boolean TRUE or FALSE result.

These operators are an example of the Vague Logic operations within CeSk.

Note that the <> signs point *outwards* to indicate that the result of the compare is *outside* the specified range of vagueness.

It helps to think of these by names. For example A<====>B means that A is very different from B. It may also help to think of these in terms of negative equalities. For example A<=>B is not (A is close to B).

Example, the test "Nigel Johnstone" <==> "Nigel Johns" returns TRUE, but the test "Nigel Johnstone" <==> "Nigel Johnson" returns FALSE, i.e., they are NOT unequal hence they are equal. The second example returns FALSE because the names are too similar to be considered unequal.

>>, >>>, >>>>, >>>>> Vague Greater Than

The above comparison operators all perform relational tests between two objects, returning a Boolean TRUE or FALSE value. They compare objects using the <u>Compare</u> function within CeSk's Vague Logic comparison engine.

a >> b Indicates that a is greater than b and outside a the 5% error band over which the objects could be considered equal.

10 >> 9 returns TRUE (difference is > +/- 5%), while 10 >> 9.6 returns FALSE.
It helps to think of these operators by names, for example >>> is very much greater than. Similarly >> means slightly greater than and so on.

<<,<<<,<<<<,<<<<< Vague Less Than

The above comparison operators all perform relational tests between two objects, returning a Boolean TRUE or FALSE value. They compare objects using the Compare function within CeSk's Vague Logic comparison engine.

a << b Indicates that a is Less Than b and outside a the 5% error band over which the objects could be considered equal.

9 <<<< 10 returns FALSE, because 9 is too close to 10 to be considered as Less Than.
It helps to think of these operators by names, for example <<<< is very much less than. Similarly << is slightly less than and so on.

General Functions - Listed

Asec, Acosec, Acot

Asin, Acos, Atan

acosh, asinh, atanh

Atan2

Clean

Compare

DegtoRad

Exp

False

Int

Ln

Log

Log10

Lower

Mag

Make

Now

Null

Pi

Proper

RadToDeg

Rand

Round

Sec, Cosec, Cot

Sign

Sin, Cos, Tan

Sinh,Cosh, Tanh

Size

Asec, Acosec & Acot

The ASec, ACoSec & ACot functions return the inverse secant, inverse cosecant, and inverse cotangent of the passed numeric value.

The returned angle is in Radians, to convert this to degrees see RadToDeg.

Asin, Acos & Atan

These functions return the inverse sin, inverse cosine and inverse tangent of the passed numeric value.

The returned angle is in Radians, to convert to degrees, see RadtoDeg.

Asinh, Acosh & Atanh

These functions return the inverse hyperbolic sin, cosine, or tangent of the passed numeric value.

Atan2

Atan2 takes two numeric parameters, i.e. atan2{a,b}. returning the inverse tangent of b/a. The difference between Atan2 and Atan, is that Atan considers the sign of the parameters a and b to return a result in the correct quadrant. Atan only returns results in the range 0 to Pi/2.

Atan2 can also take a complex number as its single parameter, which uses the real value as the first parameter, and the imaginary value as the second, i.e. atan2(a+bi).

In this form Atan2 returns the Arg value of the Modulus/Arg form for the complex number. Note that when using a complex number, since only one argument is being supplied, rounded brackets are used rather than curly brackets.

Clean

Clean removes all characters with codes (according to the ANSI tables) less than 32 or greater than 127 from the passed string, returning the cleaned string.

Compare

The general basis function for all vague operations is the COMPARE function, which takes two objects (A & B) as arguments, compares them, and returns a value indicating how close the objects are.   This value is approximately scaled (where possible) to be a percentage in the range +/- 100.

A value of 0 (zero) means that the objects are the same.   A positive value means that   A > B, while a negative value means A < B.

➤ If two objects of differing types are compared, e.g. a number and a string, then the relative rankings of the object types are used, and the value returned is always one of +/- 100.   For example, comparing a number (N) with a string (S) via COMPARE{S,N} will return +100, since strings rank 'higher' than numbers (a string can contain representations of numbers, but numbers cannot contain strings).

▪   Numbers are compared and the difference between the values is scaled against the first of the two values; this may result in results outside the +/- 100 range, i.e.

COMPARE{1000,2000} returns -100, since A < B by 100%, but

COMPARE{100,2000} returns -1900, since (A-B) = -1900, which is then * (100/100), while

COMPARE{10,2000} returns -19900, since (A-B) = -1990 which is then * (100/10), and

COMPARE{2000,10} returns 99.5, since (A-B) = 1990 which is then * (100/2000)

▪   Strings can be compared by using the COMPARE function. Case and accents on characters are ignored.   The value is calculated by comparing the letters within the strings, and to a lesser extent the position of the letters e.g.

COMPARE{"Nigel Johnstone","Nigel Johnston"}

returns 9.615, indicating that the strings are a pretty close match.   The returned value is positive, since "Nigel Johnstone" ranks greater than "Nigel Johnson" alphabetically.

▪   Error objects can be compared with other error objects. An array of error objects could be scanned to find a corresponding error message in a string array.

▪   Equation objects can be compared with other equation objects.

▪   Date objects can be compared with other date objects.   The result is scaled so that a one year difference (365 days) corresponds to the value +/- 100 exactly.   As per numbers, this may result in values greater than +/- 100, i.e.

COMPARE{\22/01/63,\22nd Jan 1963} returns 0 (zero), while

COMPARE{\22/01/63,\22nd Jan 1962} returns 100.

▪   Time objects can be compared with other time objects, returning a value scaled by the first argument expressed as seconds since midnight, e.g.

compare{00:00:01,00:00:02} returns -100, since the difference is 1, and the scale 100/1, while

compare{23:59:58,23:59:59} returns -0.001, since the scale is now 100/86398.

▪   Arrays are compared element by element, and the average of all the comparisons returned as the result, e.g.

compare{{2,3},{2,4}} returns -16.66

since compare{2,2} returns 0, and compare{3,4} returns -33.3, which is then averaged.

- Complex numbers are compared using the sum of the differences of their real and imaginary parts, scaled to the magnitude of the first complex number; this can return results greater than +/- 100.

Degtorad

Takes a single numeric argument, expressing a degree value, and returns the value in radians, e.g. degtorad(100) returns 1.74532925.

Exp

This function calculates the value of e^x where e is 2.71828...

The Exp function takes either a single numeric argument N and returns the value of e^N, or a complex number object, which returns a complex result,.

False

The FALSE function returns a Boolean FALSE object.

Int

The int function returns the integer portion of any single passed numeric object, e.g. int (2.3) returns 2, as does int(2.7).

Ln (Naperian Log)

The Ln function takes a single numeric argument, returning the Naperian log (log to the base e, where e is 2.71828...).

Log

Log returns the log of argument a to the base b, i.e. log{10,2} returns 3.322.

log10

The log10 function returns the log to the base 10 of a single numeric argument.

Lower

The lower   function converts all alphabetic characters in the passed string to lower case.

Lower("This is A TEST") returns "this is a test"

Mag

The mag function returns the magnitude of an object.
For a numeric object, this returns the absolute value e.g. mag(-2) returns 2.

- For a string object; mag returns the length of the string, i.e. mag("Hi there") will return the value   8.
- For a Boolean object; TRUE will return a value of 1, while FALSE will return a value of 0.
- When used on a date object; mag returns the number of days from 1st Jan 1900 (which is considered day 0).   Note that for dates prior to this a negative value will be returned, e.g. mag(\31st Dec 1899) returns the value -1.
- When used on a time object; the number of seconds since midnight is returned.
- For a complex number, mag returns the modulus part of the complex number, e.g. for a complex number a+bi mag returns sqrt(a*a+b*b).
- When the argument is a list or array the argument is treated as a matrix.   A one dimension matrix (a list) is treated as a vector, and its length returned, e.g.

  mag{10,20} equals sqrt(10*10+20*20) returns 22.36, while
  mag{10,20,30} equals sqrt(10*10+20*20+30*30) returns 37.42

- When applied to a 2   dimensional array mag returns the determinant of the array. This will work for any N x N array up to 16 * 16.

Make{Object, Optional_New_Object_Type}

Make is one of the most powerful functions within CeSk, allowing conversion of the supplied object into the new object type. Within this function is a whole wealth of powerful features.

Essentially this function converts one object type (See Data Types) to another, but since CeSk types include equations, complex numbers and so on, the conversion process provides a range of very useful conversions.

No Conversions

If the requested conversion type is the same as the original object then no conversion is done; the original object is simply returned 'as is'.

e.g. Make{"A String",4} returns "A String" since type 4 is the string type.

Make on String Objects.

When converting a string object to another type, in general it is assumed that the string object contains an expression to be evaluated.   The string object is converted into an equation object, which is then evaluated.   The output of the expression is then converted to the requested object type, or to a Null object if conversion is not possible, e.g. Make{"\28th Feb 92 + 3"} will return the date object \2nd Mar 92. Note that since no type conversion is given, the string is evaluated and returned exactly as if the equation \28th Feb 92 + 3 was typed in.
Omitting the type causes Make to return the evaluated type.
A slightly different example is:
Make{"2@ + 3@",3}. Here the string is evaluated to give the integer 5@. Type 3 is a double precision type and so the finished integer is then converted to a double precision number.
 If you specify a particular type to convert to, then whatever the string evaluates to is then converted to that type.

Make with Numeric Objects

If a numeric value is supplied as the source object, and if no output object type is given, a default conversion to string type is assumed, i.e.. Make{32.34,4} is the same as Make{32.34} and returns the string of characters "32.34".

Type 4 is a string object.

If a numeric value is converted to a string, then normally the default format currently set is used to determine the format of the string. However, you can override this by explicitly giving the format to use as the second parameter to Make, i.e. make{1032.34,"N.Nx"} results in a string representation of the value as "1,032.34".   There is a large collection of predefined formats, e.g. Make{3.2,CurrencyForm} would return £3.20 for the UK version and $3.20 for the USA version. For a complete description of these see Standard Number Formats. For a list of the formatting characters you can use with Make see Number Formatting Characters.

Make with Date Objects

If a date value is supplied, and no output format given, then it is assumed that a default conversion to string format should be done, e.g. Make{\12 Jan 92,4} is the same as Make{\12 Jan 92} and results in a string object "12th Jan 92".

As per numbers, you can also give a specific conversion format that the date should be styled to, i.e.

make{\1st Jan 86,"%S-%d-%n-%y} returns a string "WED-1-1-1986".   Again, there are several formats already defined as defaults see Standard Date Formats.

For the list of available date formatting characters, refer to <u>Date Formatting Characters</u>.

Other Conversions

The conversions allowed for each type are listed below.

> 0 - Make{Object,0}   Convert to Null Object.
> No matter what the original object type, the conversion to type zero always results in a Null object type being returned.

> 1 - Make{Object,1}   Convert to Integer type.
> ▪ Any type of numeric value will be converted to an integer by truncation of any fractional part.

▪ Date objects are converted to the number of days since 1st Jan 1900.

▪ Time objects are converted to the number of seconds since midnight.

▪ Boolean objects are converted to 0 (False) and 1 (True) values.

▪ Complex number objects have the imaginary part discarded and the real part treated as a numeric value.

▪ String objects are assumed to contain expression (e.g. "2 + 2"), which are evaluated and any result of the calculation returned, such return being treated as a numeric value.

▪ Arrays and Error objects cannot be converted by this method (though individual array elements can).

> 2 - Make{Object,2}   Convert to Floating Point Type.
> The same rules apply for type 2 conversions as appear in type 1 conversions; the only difference is that the result is a floating point value.

> 3 - Make{Object,3}   Convert to Double Type.
> Type 3 conversions apply the same rules as type 1 and 2 conversions; the output is a double precision floating point value.

> 4 - Make{Object,4}   Convert to String type.
> Any object converted to a type 4 value gets converted to its string representation.
> Null objects are converted to null strings i.e. "".
> Numbers (both simple and complex) are converted to string representations, according to the default format styling for that numeric type. Dates are converted to strings, according to the default date format.
> Arrays are converted into array notation, i.e. "{{12,43},{\12th Jan 92}}"
> Boolean objects are converted into the strings "FALSE" and "TRUE", respectively.

Equation objects are converted from their tokenised form into a string, e.g. "1 + cos(X)".

5 - Make{Object,5}	**Convert to Boolean Type.** Any numeric value may be converted to Boolean by use of this call. If the supplied numeric value (of whatever form) is zero, the FALSE is returned, otherwise a TRUE value is returned.
6 - Make{String,6}	**Convert to Error Type.** Type 6 conversions are one of the two conversions that will only work on string objects - It will convert the string object into an error object. Any CeSk programs can use this to flag errors.
7 - Make{Object,7}	**Convert to Date Type.** Type 7 conversions return a date object when passed a numeric value by treating the value as the number of days since 1st Jan 1900. For equation and string objects the equation or string is evaluated as a calculation, and any numeric result converted as above. If evaluation of the equation or string does not result in a simple numeric value then a Null object is returned.
8 - Make{Object,8}	**Convert to Time Type.** Type 8 conversions return a time object when passed a numeric value, by treating the value modulus 86400, as the number of seconds from midnight (00:00:00). For equation and string objects the equation or string is evaluated as a calculation, and any numeric result converted as above. If evaluation of the equation or string does not result in a simple numeric value then a Null object is returned.
9 - Make{Object,9}	**Convert to Complex Number Type.** A type 9 conversion takes a numeric value (of any form) and converts it to a complex number having an imaginary part of 0i. A complex number object is simply returned unchanged. Equation and string objects are evaluated for a numeric result, treated as above.
10 -Make{String,10}	**Convert to Equation Type.** Type 10 conversions are the second type of conversion that only take strings as the input object. They return a tokenised equation form of the string e.g. Make{"sin(X) * cos(Y)",10} will return the internal equation object for the expression Sin(X) * Cos(Y). Flute performs this tokenisation on its programs.

Null Function

The Null function returns a Null object.

Now

Now returns a time object for the current system time.

Pi

The Pi function returns the value of Pi (approximated to 3.14159265358979)

This function takes no argument.

Proper

When used on a string object the proper function returns the string with each initial letter of a word capitalised, all other letters being converted to lower case, e.g.

proper("tHOse vAlueS") returns "Those Values"

When passed a date or time object proper converts the date or time to the correct representation, e.g.
proper(\30th Feb 92) returns \1st Mar 92,   proper(12:61:66) returns 13:02:06

Radtodeg

Converts a single numeric value - an angle in radians, and returns the degree equivalent, e.g. radtodeg(1.745532925) returns 99.999...

Rand (Random Number)

Rand returns a decimal random number in the range 0 (zero) to 1 (one).

Round

This function rounds a number to the supplied number of decimal places.   Both parameters must be of numeric type, e.g.

round{1.2359,2} returns 1.24, while

round{1234.234,-2} returns 1200

Sec , Cosec & Cot

The Sec, CoSec, & Cot functions return the secant, cosecant, or cotangent of the passed numeric value. These functions require the angle to be specified in Radians, to convert degrees to radians use DegToRad.

Sign

Sign returns the sign of the passed numeric argument, as the value +1, 0 or -1 for a positive, zero, or negative value, respectively.

Sin, Cos & Tan

These functions return the sin, cosine or tangent of the passed numeric value. The angle is in radian, to convert from degrees to radians use DegToRad.

Sinh, Cosh & Tanh

These functions return hyperbolic sin, cosine, or tangent of the passed numeric value respectively.

Size

The size function returns a value representing the size of an object in accordance with the table below

Null	Size is always 0
Integer	Size is always 1
Float	Size is always 1
Double	Size is always 1
String	Size is the length of the string in characters
Boolean	Size is always 1
Error	Size is the length of the error string in characters
Time	Size is always 1
Date	Size is always 1
Equation	Size is the length of the equation token list in bytes
Arrays	Size is the number of elements

➡ Notice that the Size of an array returns the number of elements. This is the number of elements in the first dimension. For example, size {{1,2,3},{4,5,6}} returns 2 because the array is a 2 element array, {1,2,3} and {4,5,6}.

Sqrt

Returns the square root of the passed numeric or complex number argument.   If a negative numeric value is used, a complex number will be returned i.e. sqrt(-1) returns the complex number 0+1i.

Standard Complex Number Formats

The following is a list of standard conversion formats for converting Complex numbers using the Make{} function.

ComplexForm -   Returns the format string "n.nx*sn.nxi". For example,
                           make{ 12.45+3.2i,ComplexForm} returns the string "12.45+3.2i".

ComplexIntForm -   Returns the format string "n*sni", For example,
                           make{12.45+3.2i,ComplexIntForm} returns "12+3i" - an integer only
                           form.

ComplexFixForm -   Returns the format string "n.00x*sn.00xi", e.g.
                           make{12.45+3.2i,ComplexFixForm} returns "12.45+3.20i".

For a list of formatting characters refer to <u>Number Formatting Characters</u>.

Standard Date Formats

There are a number of standard date formats defined for use with the Make{} function.

FormalDateForm -   Returns a string of the form "%w %e %m %y" from a date object, e.g. Make{\2nd July 92,FormalDateForm} returns "Thursday, 2nd July 1992".

FullDateForm -   Returns a date object in the format "%s %e %a %Y", giving the complete date with abbreviated names e.g. Make{\2nd Jul 92,FullDateForm} returns "Thurs, 2nd Jul 92".

ShortDateForm -   Produces a string of the form "%e %a %Y" from a supplied date object, e.g. Make{\2/6/92,ShortDateForm} returns "2nd Jul 92"

DateForm -   Returns a string in the form "%d/%n/%Y", i.e., the normal numeric form dd/mm/yy, of a date object.

For a list of available character codes see Date Formatting Characters.

Standard Number Formats

There are a number of formats predefined for conversion of objects to strings. These are for use with the make{} function.

GeneralForm -	Returns the string "n.nx", the general form for displaying numeric values. For example, make{323123123212123,GeneralForm} returns the string "3.2312312321213e14".
CurrencyForm -	Returns the string "£n.00". Note that this form will change to match whatever is relevant for the country that you are working in, e.g. "$n.00" for the US, etc.
CommaCurrencyForm -	Returns the string "£N.00". When the value requires it commas are inserted separating each group e.g. Make{1234223,CommaCurrencyForm} will return the string "£12,342.23", rather than the "£12342.23" of CurrencyForm.
CommaIntCurrencyForm -	Returns the string "£N", i.e. just the integer portion of any value. This is used when you are not interested in any fractional part of the amount.
ScientificForm -	Returns the string "n.ne", a value always expressed with an exponent e.g. Make{100.456,ScientificForm} returns "1.00456e2"
PercentageForm -	Returns the string "n.n%", expressing the value as a percentage e.g. Make{0.456,PercentageForm} returns "45.6%"
PercentageIntForm -	Returns the string "n%", expressing the value as an integer percentage, i.e. Make{0.456,PercentageIntForm} returns "46%".
LedgerForm -	Returns the string "b£n.00", which gives positive values as per CurrencyForm, but will show negative values within bracket. For example, Make{200.345,LedgerForm} returns the string "£200.35", while Make{-200.345,LedgerForm} returns the string "(£200.35)".
LedgerIntForm -	Returns the string format as LedgerForm, using the integer value of the supplied number.
FractionForm -	FractionForm returns the string "n.nx/*n", which, when used on a numeric, will attempt to express the value as a fraction, e.g. Make{0.333...,FractionForm} returns the string "1/3", while Make{0.0000324342,FractionForm} returns "3.23342e-5", since this cannot be expressed as a fraction.
SignedForm -	Returns the string "sn.nx", always returning the numeric value with a '+' or '-' sign prefix, for positive and negative values respectively.
IntegerForm -	Returns the string "N", i.e. the integral number closest to the supplied value. For example, Make{12.34,IntegerForm} returns "12".
SignedIntForm -	Returns the string "sN", a signed version of IntegerForm, starting the string with a '+' or '-' sign as required.

For a list of the character formatting codes see <u>Number Formatting Characters</u>.

Standard Time Formats

There are a number of standard time formats defined for use with the Make{} function.

Time24Form -	Returns the string "n:*00:*00" for a time object, ensuring a 24-hour display format, e.g. Make{9:2:57,Time24Form} returns "09:02:57", while Make{1:13pm,Time24Form} will return "13:13:00".
Time12Form -	Returns the string "an:*00:*00", showing the time object in 12 hour format with a trailing am/pm.
ShortTime24Form -	Returns the string "n:*00" i.e. the same as Time24Form without any seconds being displayed.
ShortTime12Form -	Returns the short version of Time12Form, i.e. the time without seconds, using the format "an:*00".

The time formats are defined using the same character codes as the number formats. For a list of these characters see <u>Number Formatting Characters</u>.

Today

Today returns a date object for the current system date.

Transpose

Transpose takes an array argument and transposes the first 2 dimensions. For example,

transpose{{1,2,3},{4,5,6},{7,8,9}} returns {{1,4,7},{2,5,8},{3,6,9}}, i.e.

$$\text{transpose} \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \text{ is returned as } \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

Note that a similar transpose using array notation is transpose{{"hello","array"},{"follow","on"}} which returns {{"hello","follow"},{"array","on"}}

Trim

Trim converts all whitespace (comprising spaces, tabs, cr/lf etc.) within a string into a single space. (In other words it trims needless white space) e.g.

trim("This   is   a   test") returns "This is a test"

True

The True function returns a Boolean TRUE object.

Type

The Type function returns a numeric value representing the type of the object as shown below

Type	Description
0	Null Object (i.e. nothing) . Note that this is not the same as zero, or a zero-length string.
1	Integers, ranging from -2147483678 to +2147483647
2	Reals; A floating point value, having 6 significant decimal places and a range of -1E-38 to 1E38.
3	Reals; A double length floating point value, having 15 significant decimal places and a range of -1E-308 to 1E308.
4	A String of characters; maximum length is 32767 characters.
5	Boolean True or False.
6	An error object.
7	A date object.
8	A time object.
9	A Complex number.
10	An equation.
11	An array object.

Upper

Upper converts all alphabetic characters in the passed string to upper case.

Upper("This is A TEST") returns "THIS IS A TEST"

Date Formatting Characters

When converting date objects to string objects using the Make function you can define exactly how the conversion should be done by construction a format string using the following character codes.

%d	The day of the month; 1,2....20,21...
%e	The day of the month with st, nd etc; 1st, 2nd...
%E	The day of the month with ST, ND etc; 1ST, 2ND...
%m	The month in full; January February, March....
%M	The month in capitals; JANUARY, FEBRUARY, MARCH...
%a	The month in short form; Jan, Feb, Mar...
%A	The month in short form capitals; JAN, FEB, MAR...
%n	The month as a number; 1,2,3...11,12
%N	The month as a number; same as %n
%y	The year in full; 1992, 1993, 1994..
%Y	The year in short; 92, 93, 94, 95
%w	Day of week; Monday, Tuesday, Wednesday...
%W	Day of week; MONDAY, TUESDAY..
%s	Day of week short form; Mon, Tues, Wed...
%S	Day of week short form Capitals; MON, TUE, WED...

And other characters are taken literally.

Example.

%d/%n/%Y   is 14/2/69

Predefined Error Return Codes

The following list covers errors returned by expressions, CeSk functions and programs written by the user. Errors beginning 'R' are Run Time errors in Cesk programs and will cause the program to abort. Other code letters are warnings from Flute to the User and are non-fatal.

Where <num> is shown in the error a number will be displayed which represents the data object type. A list of these numbers is covered under the Type function.

Q001 Error: attempt to add mismatch objects types <Num> and <Num>

An attempt has been made to add two data items whose types are incompatible.

Q002 Error: Bit operation on mismatched objects types <Num> and <Num>

Bit operations such as AND, OR, XOR can only be performed on numeric objects. An attempt to perform a bit operation on another data type results in this error.

Q003 Error: Cannot take inverse

It was not possible to take the inverse of an object because the object has no inverse, or the type of object is incorrect. For example Matrices with a determinant of 0 have no inverse.

Q004 Error: Wrong object type

The wrong type of object was encountered in a function requiring specific object types.

Q005 Error: Cannot convert object <Num> to <Num>

The Make function and similar functions perform conversions from one object type to another. This error will be returned if a request for an illegal conversion is made.

Q006 Error: Numerical overflow

A numerical overflow of the Floating Point routines.

Q007 Error: action performed on mismatched objects types <Num> and <Num>

This is returned by operators and functions when the objects are ill matched. The commonest use of this is in multiplication; for example "String1"*"String2" is illegal as you cannot multiple two strings.

Q008 Error: String too long

String objects can be no larger than 32767 characters long.

Q009 Error: Division by zero

Attempt to divide by zero.

Q010 Error: Cannot evaluate complex power

Attempts to raise an object to a complex power e.g. 2^(3+2i) are not allowed.

Q011 Error: attempt to sub mismatched objects types <Num> and <Num>

Attempt to subtract objects that cannot be subtracted.

Q012 Error: Incorrect function parameters

This is the commonest error within CeSk. When data is passed to a function, if the function expects particular types of parameter and receives different it returns this error as a function return.

Q013 Error: Two many parameters

More parameters than allowed have been passed to a function. The exact number of parameters required depends on the function.

Q014 Error: Two few parameters

Too few parameters were passed to a function that was expecting more.

Q015 Error: Number too large to display

When you use Make to convert numbers to strings and the format you specify does not permit a large number to be converted this error is returned.

Q016 Error: Maths Domain error in <Name>

The calculation went outside the allowable Domain. This is a floating point error, <Name> gives the name of the function that failed.

Q017 Error: Maths Singularity   Error in <Name>

The floating point package reported a singularity in the function <Name>.

Q018 Error: Maths Overflow Error in <Name>

The floating point package reported an overflow condition in the function <Name>.

Q019 Error: Maths Partial Loss of Significance in <Name>

The floating point package reported a partial loss of accuracy in the function <Name>.

Q020 Error: Maths Total Loss of Significance in <Name>

The floating point package reported a loss of accuracy.

Q021 Error: Maths Underflow in <Name>

The floating point package has lost its accuracy because the numbers have become too small.

Q022 Error: Maths Invalid Operation

An invalidate operation was performed by the floating point package. The exact meaning is not known.

Q023 Error: Maths Overflow

A floating point package overflow occurred in the normal multiplication, division and addition operators.

Q024 Error: Maths Underflow

A floating point underflow (number too small to store) occurred in the normal multiplication, division and addition operators.

Q025 Error: Maths Error

A general error, other than the predefined types, occurred in the floating point package during the normal multiplication, addition and division operators.

R001 Error: Incorrect assignment

An attempt has been made to assign data to something that is not a variable. For example today:=3. Although this may seem like assigning 3 to the variable 'today', today is a reserved function name, not a variable.

R002 Error: unbalanced brackets [<Num>] {<Num>} (<Num>)

When a program is tokenised (prepared for execution), or when the user types in an equation and the brackets mismatch, this error is displayed in a dialogue box. The <NUM> in this case represents a count of each type of brackets, +ve numbers mean too many opening brackets, -ve numbers mean too many closing brackets.

R003 Error: Syntax

An unexpected command or item was hit when executing a CeSk program.

R004 Error: Faulty Action Clause in IF

The Action clause of an IF command within a CeSk program is incorrect or missing.
IF (condition-clause) action-clause;

R005 Error: Faulty ELSE Clause in IF

The Else clause within a CeSk program is missing or faulty. This is only relevant when ELSE is specified.
IF (condition-clause) action-clause ELSE else-clause;

R006 Error: Malformed WHILE

Badly formed WHILE loop in a program. A common error is to omit the condition.
WHILE (condition) { Action };

R007 Error: WHILE not terminated correctly

The WHILE loop was not terminated correctly by a semicolon after the closing bracket or statement.

R008 Error: Malformed DO

A DO loop within a CeSk program is incorrectly set. The correct form for a DO loop is:
DO { action} while (condition);

R009 Error: DO not terminated correctly

The final semicolon after a DO-loop within a CeSk program is missing.

R010 Error: Malformed Condition in FOR

The condition part of a FOR loop is incorrect. The correct form is:
FOR (Init-action ; *condition*; post-action;) {main -action};

R011 Error: Malformed PostOp in FOR

The Post action in a FOR loop is badly formed. Unlike the programming language C, a semicolon is required after this action.
FOR (Init-action; condition; *post-action*;) {main-action};

R012 Error: Malformed Action in FOR

The main action clause within a FOR loop is madly formed.
FOR (init-action; condition; post-action;) {*main-action*};

R013 Error: FOR incorrectly terminated

A FOR loop is not terminated by a semicolon;

R014 Error: Missing (in FOR

A FOR loops main group is contained within a set of braces ( ). This error occurs if these braces are omitted.

R015 Error: Illegal command in FOR

It is not allowed to place FOR, DO, WHILE loops etc., within the initialisation, condition, or Post-Op phase of a FOR loop.

R016 Error: Expression does not evaluate

When an expression for a loop condition was evaluated it evaluated to nothing (not even a NULL object)

and hence this expression does not evaluate.

R017 Error: CASE/DEFAULT does not evaluate

The specifier in a CASE construct does not evaluate to any value and so could not be compared.

R018 Error: No such function

When a call is made to a function and no function can be found with a name that matches this error is returned.

R019 Error: Attempt to break/Continue out of a function

Break and Continues are used within program loops. If the program is not in a loop when these commands are used then this error will occur.

R020 Error: Cannot find object

Reserved for internal use.

R021 Error: Program Terminated by User.

When running a CeSk Program with the Debug window open, if you click on the Stop button the program is terminated and this error is returned.

R022 Error: Arrays are defined using *Curly* brackets

Functions that take multiple parameters are passed 'Arrays' of parameters. Arrays are defined using curly brackets, e.g., {1,2,3} is an array of 3 numbers, whereas (1,2,3) is an error.

U001 Error: Bad Window Handle

The window specified for a function is not valid. It may be that the window specification is wrong, or that the window is not open. Flute will keep trying to find the window for 5 seconds when it hits a problem. If the window is still not open by then this error is returned.

U002 Error: Menu Not Found

The specified menu is not correct, or the item specified is not found. For example, if you specify a window that does not contain a menu in FindMenu this error is returned.

U003 Error: Menu is Disabled

At attempt has been made to select a menu that is disabled and hence cannot be selected.

U004 Error: Cannot Find Named Object

When using CSPoke and CSPeek you specify an object to Poke and Peek into. If the name of the object you specified does not match the name of any objects on the specified sheet this error is returned.

U005 Error: CleanSheet not responding

CleanSheet is busy or not available and cannot respond to a CSPeek or CSPoke.

U006 Error: The Object has no Input Pipes

You cannot CSPeek the contents of the input pipe of a CleanSheet object if it has no input pipes.

U007 Error: The Pipe is Empty - Recalculate to fill it

When a sheet is first opened in CleanSheet the pipes are always empty. If you try to CSPeek from an object at this time you will get error U007. To fill the pipes recalculate the sheet inside CleanSheet.

U008 Error: Window has no Children

You have used GuessClient to guess the client window for a window which has no children and hence cannot possibly have a client window.

U009 Error: No Suitable Scroll Bar

When using scroll bar commands Flute attempts to find a window that appears to behave like a scroll bar. If no suitable candidate is found this error is returned.

U010 Error: Cancel Selected

During an Ask function if the user clicks on Cancel on the resulting dialogue Flute returns this error.

U011 Error: Failed to Open File

The requested file could not be opened.

U012 Error: Failed to Create File

The requested file could not be created.

U013 Error: Bad File Handle

The handle for the specified file is not a file handle.

U014 Error: Error when reading

There was an error when reading from the file.

U015 Error: Disk Full or Damaged

There was an error when attempting to write to the disk, the disk is either full, or there is a fault.

U016 Error: Error when writing data

An error occurred when writing data to a file.

U017 Error: Not a CeSk data structure

Attempt to read in (from a file) data which does not appear to be CeSk data. You may be reading from a corrupt, or incorrect file.

U018 Error: Invalid File Ptr Position

The file pointer specified is invalid for this file.

S000 Error: System out of Memory/Executable Corrupt/Relocations invalid

Attempt to execute a Dynamic Link Library (a DLL) or program that is faulty.

S001 Error: File not found

The specified DLL was not found.

S003 Error: Path not found

The pathname part of the DLL was not found.

S005 Error: Sharing/Network Protection fault, Dynamic link fault

The DLL cannot be opened due to a sharing fault on the network, or because the DLL does not allow multiple copies of itself to be used and another copy is already in use.

S006 Error: Library Required separate data segments for each task

This is a similar fault to S005, but indicates that the data segments could not shared.

S008 Error: Insufficient memory

There was insufficient memory available to load the library.

S010 Error: Incorrect Windows Version

The library was written for an incompatible version of Windows.

S011 Error: Invalid Executable File

The DLL file is not an executable, or DLL file.

S012 Error: Application intended for different OS

The DLL file is not for Windows.

S013 Error: Application was designed for MSDOS 4

The DLL requires an older version of DOS.

S014 Error: Type of executable file unknown

Executable file has an unknown type. This may occur when using DLLs written under a newer operating system on an older version.

S015 Error: Attempt to run Real Mode application

Application requires Real Mode Windows. Flute cannot be run in Real Mode.

S016 Error: Attempt to start multiple copies of application that does not allow it

The application specified permits only one copy of itself to run at a time.

S019 Error: Attempt to load compressed executable - decompress first

Programs are often distributed in compressed form (they require less disk space). These programs must be installed correctly before executing.

S020 Error: DLL was invalid

The DLL has an unspecified fault.

S021 Error: Applications requires Windows 32bit extensions

The DLL, or application requires Win32.DLL on the system.

S022 Error: Action Unknown

There is an unknown fault when loading a DLL.

S023 Error: DLL Function not known

The function to be called in the DLL could not be found.

S024 Error: DLL Function returned unknown data

The data returned by a DLLCall did not look like CeSk data. Only specifically written functions can be called by DLLCall.

D001 Error: DDE Library could not be initialized

The Windows Dynamic Data Exchange (DDE) handler could not be initialised.

D002 Error: DDE Synchronous Advice request timed-out

A request from a server timed out (the time out in Flute is set to 10 seconds). If the server does not respond in this time there is a definite fault.

D003 Error: DDE Server is Busy

The DDE server is too busy to process the transaction.

D004 Error: DDE Data transaction timed-out

A transaction involving the transfer of data took too long.

D007 Error: DDE Execute command timed out

A request for a DDE server to execute the requested commands took too long to execute.

D008 Error: DDE Invalid Parameter

An invalid parameter has been passed to the DDE manager.

D009 Error: DDE Insufficient memory

There is not enough memory to complete the requested DDE operation.

D010 Error: DDE Out of memory

There is not enough memory to complete any DDE operation.

D011 Error: DDE Transaction failed

The attempt to perform a transaction failed for an unspecified reason.

D012 Error: DDE No suitable server found for conversation

The attempt to connect to a DDE server failed because no server matched the requested system &/or topic

D013 Error: DDE Poke request has timed out

Request to Poke data into a server timed out.

D014 Error: DDE Post Message function failed

An attempt by the DDE server to post a message failed.

D016 Error: DDE Conversation has been terminated

The other partner in the DDE conversation has terminated the conversation.

D017 Error: DDE System fault

There is a system fault in the DDE manager.

D018 Error: DDE Time out

A general time-out between the DDE manager and client has occurred.

D019 Error: DDE Transaction identifier is invalid

The Item transaction or conversation identifier is invalid.

O001 Error: OLE Libraries could not be initialised.

The OLE2 Libraries could not be initialised, are missing, corrupt, the wrong version or the wrong registration information is present. Try re-installing Flute and if this fails, try re-installing any program you are attempting to control.

O002 Error: ProgID does not exist.

The description of the OLE object is invalid. The object may not support IDispatch interfaces or the text describing the object may be incorrect.

Number formatting Characters

When the function <u>Make</u> is used to convert numbers to string it does so in a defined format. If the standard defined formats are unsuitable then CeSk allows you to specify your own.

The formatting characters available are:

b	Bracket a number if negative, otherwise do nothing
B	Bracket a number if negative, else substitute spaces
N	Display a number grouped into thousands. This displays the section before or after a decimal point and will include a leading minus sign if negative.
n	Display a regular number, with a leading minus sign if b, B, S or s is not specified
S	Show a minus sign if negative, otherwise substitute a space
s	Display a minus sign if negative, otherwise nothing
9	Show a single digit, or a space character if a leading zero
0	Show a digit or nothing if a leading zero
.	Place a decimal point here. The decimal point is only displayed if there is anything to the right
E	Display the exponent here with an E
e	Display the exponent here with an e
X	Display the exponent if there is one with an E character
x	Display the exponent if there is one with an e character
*	Separates multi-part numbers, such as complex numbers
/	Display as a fraction if practical
%	Multiply by 100 before displaying - percentage
A	Must be before any numbers. Sets AM/PM display on time objects
a	Must be before any numbers. Sets am/pm display on time objects
!	Surrounds literal text e.g. !Pounds Sterling!

GetTopWindow

This function returns the handle of the top child window in a group of child windows. This takes one parameter - the handle of the parent window. You can substitute NULL if you want to obtain the top window from the desktop.

Just as you can with other functions, you can use any of the ways of specifying windows listed in the FindWindow function.

Example: `GetTopWindow("CleanSheet")` gets the top child window of the CleanSheet application window. This may not be what you expect - the documents in a program like CleanSheet are not children of the main window - they are children of children of the main window. For example a simplified version of the structure of CleanSheets windows is as follows:

```
Main Frame Window

    ┌── Button Bar
    │        └────── Edit Field
    ├── Object Pallette
    │        └────── Scroll Bar
    └── Client Window

             ┌────── Worksheet 1
             ├────── Worksheet 2
             ─────── Worksheet....
```
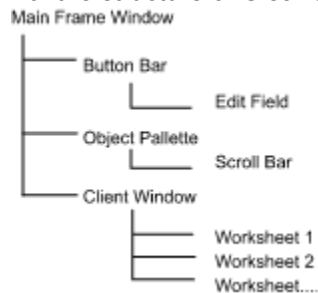
You can see that the worksheets are children of something called the Client Window. This is typical of most multi-document applications, If you use `GetTopWindow{"CleanSheet"}` you would find out which of the three windows, Button Bar, Object Pallette or Client Window is top, not which document window is top.

There is no reliable way of finding the client window. They are usually unnamed and there is no consistent way of identifying the type (or class) of the window. However, Flute provides a function GuessClient which examines the first level windows of a frame window and attempts to decide which is the ClientWindow. Therefore, to find the top document window use
`GetTopWindow(GuessClient("CleanSheet"))`

See Also

GuessClient

GuessClient

Multi-document applications have their document windows grouped under a window known as the client window. If you want to find which document window is top then you need to find the client window. (See GetTopWindow for more details).

`GuessClient(wind_hand)` - it takes one parameter which is the handle of the main application window. As with other window functions in Flute, you can use any of the methods specified under FindWindow to specify the window.

GuessClient examines the first level children of a window and takes a guess as to which is the Client window. It starts by searching for windows of type "MDIClient", "OpusDesk", "XLDesk" etc. - the common types of client window. Failing that it compares the area covered by each window and chooses the largest - companies want maximum workspace, so client windows tend to be much larger than button bars, pallettes or other windows.

These methods will work most of the time, but not all. Whenever possible you should use Paste Find Window to examine the applications windows and locate the client window.

GuessClient returns the handle of the found window or an error if no child window exists.

BringWindowToTop

This command bring a window to the front (top) of its sibling windows. This takes one parameter - the handle of the window. You can use any of the ways of identifying the window specified in FindWindow.

Example:

```
BringWindowToTop{"CleanSheet","Untitled 1"};
```

This brings "Untitled 1" to the top of document windows within CleanSheet.

```
BringWindowToTop{"CleanSheet"};
```

This command will bring CleanSheet to the top of application windows.

This command does not return a value.

BringWindowToTopmost

This command brings a window to the front of the screen. Windows that are always on top are called Topmost windows. The Clock application and Help applications have this option. This command allows you to make any window a top most window.

BringWindowToTopmost(windhand)

It takes one parameter; a handle to the window to become a topmost window. Just as with all window commands, you can specify this window using any of the methods listed under FindWindow.

Example:

`BringWindowToTopmost("Calculator");` will make the calculator stay on top of the screen no matter what you do with the other windows.

This command does not return a value.

BringBackTopmost

The command brings a window back from being the topmost window back to being a normal window.

`BringBackTopMost(wind_hand)`

This command takes one parameter, the handle of the window that is to be brought back. You can use any of the methods listed under FindWindow to specify the window.

Example:

`BringBackTopmost("Calculator")`

If you have made the calculator a topmost window this command will bring it back to a normal window.

This command does not return a value.

CloseWindow

This function closes the specified window.

```
CloseWindow(wind_hand);
```

This takes the handle of the window as a parameter.

Note: It doesn't actually close the window, rather it tells the application to close the window. If the window is a main application window this usually causes the application to exit.

Example:

`CloseWindow{"Microsoft Word"};` will ask Microsoft Word to close down.

This function returns TRUE if successful, or FALSE if the window would not close down.

IsZoomed

This function tests if the specified window is maximized (zoomed) to full size on the screen.

```
result:=IsZoomed(wind_hand);
```

It takes one parameter; the handle of the window to test. You can specify the window using any of the methods listed under FindWindow.

The return from this function is a Boolean TRUE if the window is zoomed, and FALSE if it isn't. If the window could not be found an error is returned.

IsIconic

This function tests if the specified window has been reduced to an icon.

```
IsIconic(wind_hand);
```

This takes the handle of the window as the only parameter. Any method listed under <u>FindWindow</u> can be used to specify the window.

This should be used on parent windows that can be iconised, and document windows that can be iconised. A child window of an iconised window is not iconised itself just because its parent is.

The return from this function is a Boolean TRUE if the window is iconised, or FALSE if it isn't. If the window could not be found an error is returned.

IsEnabled

Tests if the window is enabled. If a parent window is disabled then all its children behave as though they are disabled.

```
result:=IsEnabled(wind_hand)
```

The return from this function is TRUE if the window is enabled, FALSE if it is disabled, or an error object if the window could not be found.

MaxWindow

This function maximises (zooms up to full screen size) the specified window.

```
MaxWindow(wind_hand)
```

wind_hand is the handle of the window to maximize. Any of the methods listed under FindWindow can be used to specify the window.

Not all windows are intended to be maximized. You may get unusual results if you maximize a window that was never intended to be maximized. Some windows will prevent the maximize command and will not respond.

The return from this function is TRUE if the window became maximized, or FALSE if the command failed. If the window could not be found, an error is returned.

MinWindow

This function minimizes (reduces to an icon) the specified window.

MinWindow(wind_hand)

wind_hand is the handle of the window to minimize. Any of the methods listed under FindWindow can be used to specify the window.

Not all windows are intended to be minimized. This command requests the window to minimize itself, some windows will not respond to MinWindow.

The return from this function is TRUE if the window became minimized, or FALSE if the command failed. If the window could not be found, an error is returned.

RestoreWindow

This function restores a window from its maximized state to its normal state, or from an iconised state to the normal state. It is the equivalent to selecting 'Restore' on the system menu for a window.

RestoreWindow(wind_hand)

wind_hand is the handle of the window to be restored. You can use any of the methods listed under FindWindow to specify the window.

The return from this function is TRUE if successful, FALSE if the window did not respond, or an error return if the window could not be found.

EnableWindow

This functions enables a specified window to receive mouse and keyboard events.

```
EnableWindow(wind_hand)
```

wind_hand is the handle of the window to be enabled, or any of the ways of defining windows listed under FindWindow.

The return from this function is a Boolean TRUE if the window is enabled.

DisableWindow

This functions disables a window, preventing it from receiving mouse and keyboard events.

```
DisableWindow(wind_hand)
```

wind_hand is the handle of the window to be disabled, or any of the ways of defining windows listed under FindWindow.

The return from this function is a Boolean TRUE if the window is enabled.

GetParentWindow

This function gets the parent of the specified window, or returns NULL if no parent exists.

GetParentWindow(wind_hand)

wind_hand is the handle of the window whose parent is to be obtained. Any of the methods listed under FindWindow can be used to specify the window.

The return from this function is the handle of the parent window, NULL if this is a top level window, or an error if the specified child window does not exist.

GetChildWindows

This function returns a list of the child windows for a given parent window.

```
GetChildWindows(wind_hand)
```

This function takes one parameter - wind_hand the handle of the parent window. This can be specified using any of the methods listed under FindWindow.

The return is an array of window handles with one element for each child window. You can use the size function to find how many elements are in the array and hence how many children a window has.

For example, a return array {4034,4543} indicates that there are 2 child windows whose handles are 4034 and 4543.

For an example, consider the Program Manager, and imagine that 3 program group windows and 6 program group icons are displayed.

You might think that GetChildWindows("Program manager") would return 9 items (3+6=9). However, the Program Manager has a window, known as the client window, as its only child. You will get only one handle - the handle of the client window.

Suppose then that we got the child windows of the client window with GetChildWindows(GuessClient("Program manager"));

Will thisl return 9 window handles; 6 for the icons and 3 for the windows? No, in fact you will get 15 window handles; 3 for the windows, 6 for the icons and a further 6 for the names of the icons - each icon is actually 2 windows.

GetClassName

This function retrieves the class name for the given window. Class names are names that identify the type of window. For example, edit fields are usually class "EDIT", scroll bars are class "SCROLLBAR" and so on. Apart from the standard classes, each program has many of its own.

GetClassName(wind_hand)

wind_hand is the handle of the window whose class name is to be obtained or a description of the window (see FindWindow).

The return from this function is a string object containing the class name, or an error if the window does not exist.

Example:

`GetClassName("CleanSheet")` returns "CSheetFrameWClass"

Do not assume that all edit fields are class "EDIT" and all scroll bars class "SCROLLBAR". Applications can create variations on these windows and these variations can have their own unique names.

SetWindowText

Set the text of the specified window.

```
SetWindowText{wind_hand,text}
```

wind_hand is the handle of the window. You can use any method listed under <u>FindWindow</u> to specify the window.

text is a string object to set the window text to.

Remember that most items on a dialogue are windows. You could for example, use this function to change the text of a button, or of a text item, or a single line edit field.

Example:

SetWindowText{"program manager","Program Mangler"};

This example renames the Program Manager window.

This method may not work on some edit windows. Particularly multi-line edit windows, which use other methods to store their text. If you are attempting to set the text in an edit window and it is not responding use <u>SetMLText</u> instead.

GetActiveWindow

This function returns the handle of the active window. The active window is the main window that has focus. The difference between this function and <u>GetFocus</u> can be explained by a simple example: consider a dialogue box with an edit field; when you are entering text in the edit field the Active window is the dialogue box window and the Focus window is the edit field.

This function requires no parameters, for example `a:=GetActiveWindow;`

GetFocus

This function returns the handle of the current focus window. The focus window is the window into which text is typed. See GetActiveWindow for more details.

This function requires no parameters, for example `a:=GetFocus;`

SetWindowRect

This function sets the rectangle containing a window. This has the effect of moving and sizing the window. Note: some windows were never designed to be resized, changing their size may have unwanted effects.

SetWindowRect{wind_hand, {{x1,y1},{x2,y2}}}

Where wind_hand is the handle of the window,   {x1,y1} is top left corner of the window in screen coordinates, {x2,y2} is the bottom right corner of the window in screen coordinates.

With the confusing brackets it may help you to think in these terms:

top_left:={x1,y1};

bottom_right:={x2,y2};

window_rect:={top_left, Bottom_right};

SetWindowRect{wind_hand,window_rect};

wind_hand can be specified using any of the methods listed under FindWindow.

SetWindowRect is the complimentary function to GetWindowRect.

IsVisible

This function tests if the specified window is visible.

IsVisible(wind_hand)

wind_hand is the handle of the window to be tested for visibility.

This function returns a Boolean TRUE if the window is visible, or FALSE if it is invisible. If the window specified by wind_hand is invalid, an error is returned.

A window can be obscured by another window and even though it cannot be seem it is not invisible. An Invisible window is one specifically hidden by the operating system.

Example:

a:=IsVisible("Flute");

Tests if the flute window is visible.

HideWindow

This function hide the specified window, making it invisible on the screen.

```
HideWindow(wind_hand)
```

wind_hand is the handle of the window to be made invisible. Any of the methods listed under FindWindow can be used to specify the window.

All windows can be made invisible.

The return from this function is TRUE if successful, or an error code if wind_hand was invalid.

Example:

```
HideWindow("Program Manager");
```

This will hide the Program Manager window, preventing the user from running programs, or exiting windows without using Ctrl-Alt-Delete.

ShowWindow

This function unhides a hidden window. A hidden window is one that has been marked as invisible by the operating system (or with HideWindow). Windows that are simply obscured by other windows, or have been moved off the edge of the screen, are not hidden.

```
ShowWindow(wind_hand)
```

wind_hand is the handle of the window to be re-shown.

The return from this function is a Boolean TRUE if successful, or an error code if wind_hand was invalid.

ActivateWind

This function activates a window ready to receive input. This is the same activation that happens to a window when it is clicked on.

```
ActivateWind(wind_hand)
```

Where wind_hand is a handle to the window. You can use any of the methods of referring to windows listed under FindWindow.

The return from this function is a Boolean TRUE if successful, or an appropriate error code if the window was invalid.

Font

Introduction

The Font dialogue selects the typeface and style used to display and edit the program.
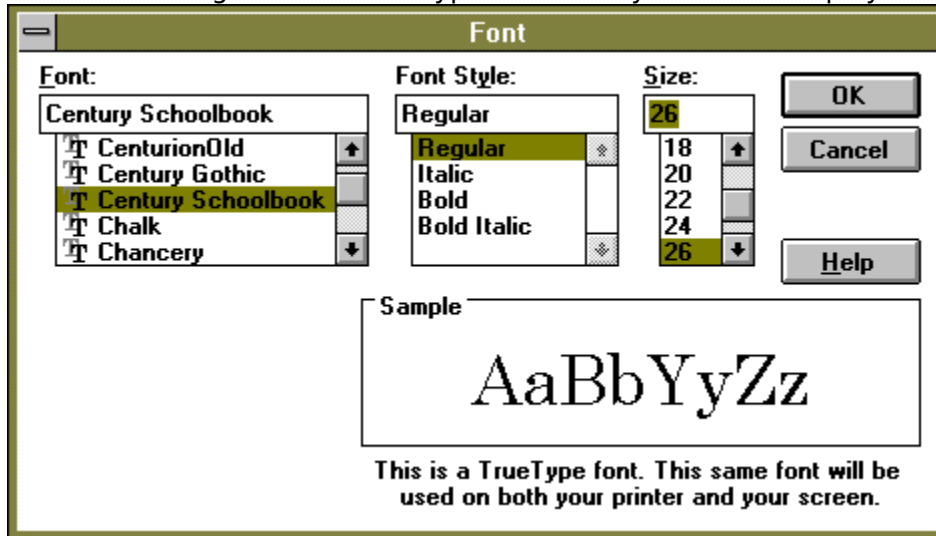
Click on any part you need help with.

The base typeface is selected from this list, this defines the basic style of the text. Typefaces with the ⊤ symbol before them are True Type typefaces that can be scaled smoothly to any size. Whenever possible you should use these typefaces.

The styles that the base typeface is available in are listed in this section. The exact wording depends on the supplier of the typefaces, but generally Italic typefaces are called Cursive, Italic or Oblique and the weights (how heavy or black a typeface is) generally follows a trend:

Light ► Roman
► Regular
► Medium
► Demi Bold
► Bold
► Black

With typefaces that aren't TrueType, the system may simulate the typeface by (for example) skewing a regular typeface to obtain an italic.

Size - The size of the typeface in Points (72nds of an Inch). 10pt is the regular text size for editing. If the typeface isn't True Type, then the equivalent bitmap typeface may have to be stretched to make it larger producing a jagged look.

A sample of the typeface and some explanatory text are provided in this section of the dialogue.

Open

Introduction

The Open dialogue is for selecting the file to be loaded from disk.

The Drive

Each floppy disk and each hard disk is called a Drive. Floppy disks are always labeled A (and B if you have 2 floppy disks), hard disks are labeled C, D, E.. and so on. If the file is only floppy disk A, then you would select disk A in the Drives section.

►   Select the Drive the file is on.

Directories

On each drive is a hierarchy of folders. These split up the disk into a logical manner, so that related files are grouped together in one folder. The directory specifies where in this system of folders you are, for example "c:\c700\source" reading left to right means Drive C:, inside the folder called "c700", inside the folder "source".

Notice the hierarchy of folder, the folder "source" is inside the folder "c700".

►   Select the folder the file is in.

Filenames

Each file is referred to by a name, with an extension (a full stop followed by 3 characters) after it. The extension tells the system what kind of file it is. Flute used the extension .FLU.

When this dialogue is first opened, the File Name is given as *.FLU, the star (*) means all filenames, and hence *.FLU means all filenames with the extension FLU. This text string tells the system to display all the files that end in .FLU that are in this folder.

►   Select the file from the list of displayed files.
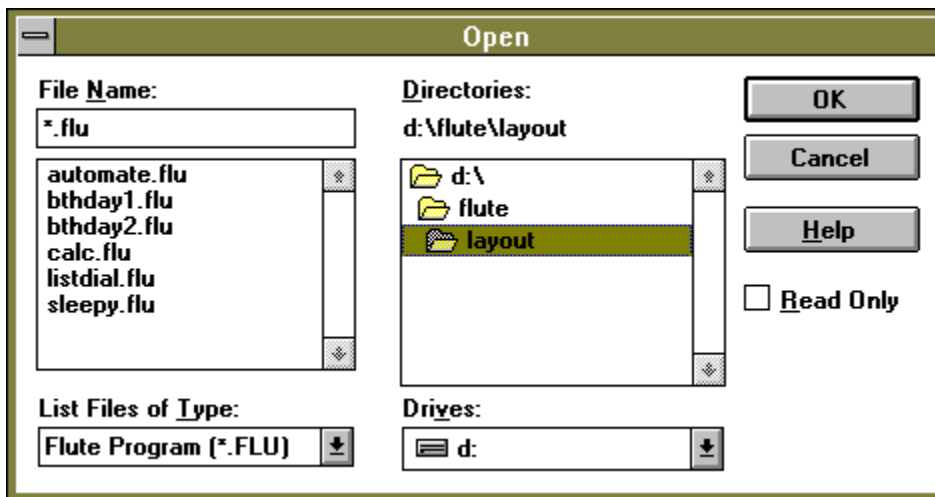
Click on any part of the dialogue you need further help with.

See Also the Help Available with the Windows File Manager.

File Name

Enter the name of the file in this section (you can omit the extension). You can also enter a file mask, for example *.FLU means all files ending in extension FLU, while *.* means all files with all extensions.

A list of the files in this folder is displayed here. Only files that conform to the file mask are displayed. For example if the File Name is *.FLU, then only files with the extension FLU are displayed.

Types of Files

Select which file type to display here. Each file type has a corresponding file extension, hence selecting the Flute Program file type, puts *.FLU in the filename field so that only Flute Programs (which always end in .FLU) are displayed.

Directories

This section displays the current directory, and provides a graphical list showing the directory tree.

In this list, open folders are shown as 📂, closed folders are shown as 📁.

▶   To open a folder and see what's inside, double click on its icon.

Notice the indent indicating the levels of folders.

📂 c:\
  📂 windows
    📁 msapps

This for example means that folder "MsApps" is inside the "Windows" folder which is inside Drive C.

Drives - this section lists the floppy disk drives and hard disks drives available to store files on.

Read Only

If this button is selected, then the file is opened as without a filename. You can still edit the program, but when you attempt to save it, you will be prompted for a **new** filename.

Save As

Introduction
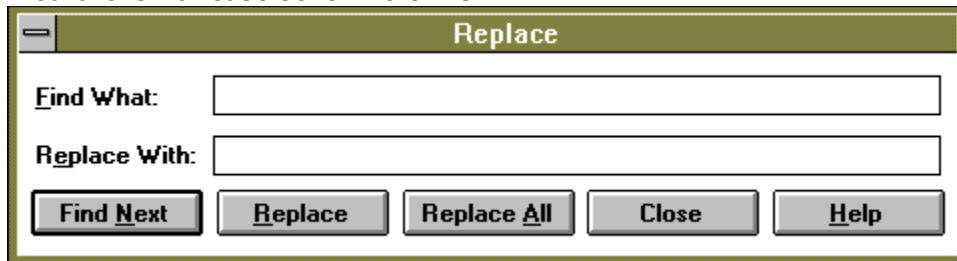
This dialogue sets the filename under which a program is saved. It is almost identical to the dialogue for Opening a file and for this reason, you should read the section on the Open Dialog first.

Step - by Step

- Select the Drive to save to.
- Select the Directory to save to.
- Enter the Filename the program is to be saved as. Choose a filename that will help you identify the file when you wish to open it again. You are allowed only 8 characters for a filename, you do not have to add the .FLU extensions to identify it as a Flute program, Flute will do this automatically.
- Select OK to save the file.

Replace

Replaces one piece of text with another. CeSk is not case sensitive and hence the replace feature is not case sensitive either.

Click on any part of the dialogue you need help with.

Find Next	Find the next occurrence.
Replace	Replace one occurrence and find the next.
Replace All	Replace all occurrences from the insert point downwards.
Close	Close the dialogue

Enter the text to be searched for in this section of the dialogue.

Enter the text to replace with in this section.

Find the next occurrence of the text specified in 'Find What'.

Replace the selected text with the text specified in the 'Replace With' section.

Replace all occurrences of the search text with the replace text. This replaces all occurrences from the insert point downwards.

Close the dialogue.

Find

Find the next occurrence of the specified text.

Click on any part of the dialogue you need help with.

Note that CeSk is not case sensitive (ATAN = atan) therefore neither are the search facilities.

Enter the text to search for in this section.

Searches for the next occurrence of the search text.

Close the dialogue.

Printer Setup

Introduction

This dialogue selects the printer which is the destination for all printing in Flute, it also sets options specific to each individual printer.

The list of available printers is shown on the dialogue, to change printers, simply select one from the list and click on OK.

To set options specific to a particular printer, select the printer from the list and select the 'Set Up' button.

 Selecting a Printer on this dialogue, sets that printer as the default for all the applications you are running.

If you want to add additional printers to the available list, Select Printers in the Windows Control Panel.

Paste Functions

Pastes the name of a function, together with its parameters - this is a more reliable way of getting the parameters of a function correct.
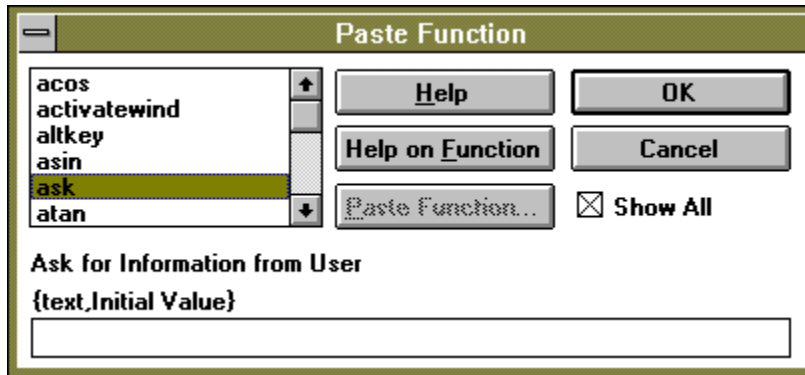
Click on any part of the dialogue you need help with.

Help on Functions	Provides help on the selected function.
Show All	Shows all the functions, the Flute specific functions **and** the more general functions.
Paste Function	When one of your parameters is itself derived from a function, you can click on this button and another paste function dialogue will appear.

This is a list of the functions available.

Provides help on the selected function.

If **Show All** is selected, then all the functions, not just the scripting functions, are listed.

The name of the function, a brief description and the parameters required, are listed in this section of the dialogue.

When you one of the parameters to a function is derived from another function, click on this button to get another copy of the Paste Function dialogue to appear.

Paste Window Act

The behaviour of a window is defined by its *class*. Some windows support special functions that relate to their normal behaviour. These functions are called Acts.

It may be easier to explain with an example: consider a spin dial window ⬍. This is a typical spin dial within CleanSheet. It supports only 1 Act : it can be spun up or down by a specified number of clicks.
The Act is called "Spin", it adjusts the edit control it is attached to by the amount you specify. It then returns the new contents of the edit control as a number.
To call the Spin function for a spin dial whose handle is wind_hand, you would use the command:
result:=Act{wind_hand,"Spin",-3};
This moves the spin dial down by 3 and returns the new result.

Click on any part of the dialogue you need help with.

The Paste Window Act makes it easier to find out which Acts a window supports and what parameters it requires.
- Select the window to examine.
▶ Either drag the arrow button off the dialogue and over the window to select with the mouse
▶ Or select the window from the list of windows.
- A list of Acts the window supports is shown at the bottom of the dialogue. Select one of these Acts.
- An explanation of the selected Act is shown at bottom right of the dialogue.
- Click on OK, the appropriate ACT command will be pasted into your Flute program

Drag this arrow off the dialogue and over a window to select that window.

The finished Act command is shown here.

Information about the chosen window is shown here.

This is a list of all the windows on screen.
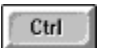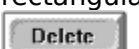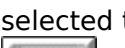
When a window is selected, this section displays a list of the window Acts it supports.

This section displays information about the chosen window act.

This button brings up help on the chosen Window Act if help exists.

Editor Keyboard Shortcuts

The following list covers the keyboard controls that are specific to the Flute editor. There are additional keyboard shortcuts for menu items. Refer to the menus for details.

Key **Action**

Home Moves to the start *of the text* on a line, or when pressed again moves to the start of the line.

End Moves the cursor to the end of the text on the line.

↑ Moves up one line, if there is selected text, moves one line up from the *start* of the text.

↓ Down one line. If there is selected text moves one line down from the *end* of the selected area.

← Left one character. If there is a selected area, moves to the start of the selected area.

→ Right one character. If there is a selected area, moves to the end of the selected area.

Ctrl

← Left 1 word.

Ctrl

→ Right 1 word.

Page Up Up by one page height

Page Down Down one page height.

Ctrl

Home Moves to the start of text.

Ctrl

End Moves to the end of the text.

⇥ If text is selected, this indents the selected text (adds a tab to each line selected). This can be used with either a normal selection or a rectangular selection.

Shift

⇥ If text is selected, this outdents the text (removes one set of tabs or spaces at the left edge).

Backspace Deletes a character to the left. If there is a selected piece of text (whether rectangular or normal), deletes the text.

Delete Deletes a character to the right of the text. If text is selected this deletes that selected text.

Ctrl

Y Deletes a line of text.

F1 Calls up the on-line help for the selected keyword or for the keyword that the cursor is in.

Ctrl

Z Undo the last change

Ctrl

A Redo the last undone action

Selecting Text

There are two types of selected region within Flute. A **block selection** covers all the text between a start and end point. A **rectangular selection**, covers a rectangle of text.

Selecting a Block

- Drag the mouse cursor over the text while holding down the **left** mouse button. If you hold down the shift key, then any previous selection is extended.
- Move the cursor using the keyboard while holding down the shift key (See <u>Editor Keyboard Shortcuts</u>).
- Click at the start of the required selection and shift-click at the end of the selection.
- Double click to select a word
- Triple click to select a line
- Quadruple click to select a block of code between balanced curly brackets { }'s.
- Quadruple click while holding down the Control key, selects a block of code between balanced round brackets   ( )'s.

Selecting a Rectangle of Text
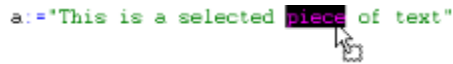
- Drag the mouse over the text while holding down the **right** mouse button.

Special Editor Features

Drag Drop

If a block of text is selected the text can be dragged to a new location. For example:

a:='This is a selected piece of text"

Click and drag the selected text using the left mouse button. The cursor will change to an arrow dragging a box.

a:='This is a selected piece of text"

As the text is dragged, a dotted bar shows where the text will be dropped.

a:='This ispiece a selected of text'

Rectangular blocks can also be dragged and dropped.

Window Splits

At the bottom right corner and top left corner of the edit window are small black rectangles, known as window splitters. If you want to view two sections of the program simultaneously, you can split the window in two using the splitters and scroll each section independently.
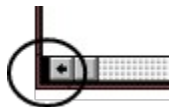
This splitter, can split the edit window into two vertical panes.
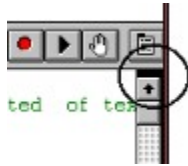
This splitter splits the window into two horizontal window panes.

- To split a window, drag the splitter bar using the mouse and drop it where you want the split.
- To remove a split, drag the splitter bar to the extreme top or bottom of the window.

The panes created using window splits are views onto **the same** program. They are **not** different programs. Changes made in one view are changes in **all** views.

Syntax Colouring

Text is coloured according to the same rules that Flute uses to interpret the program. This helps identify any typing errors or syntax problems.

You can set the colours used on the Editor Options dialogue.

Some fragments of text are coloured RED. These indicate possible errors and keywords from other languages that are not supported or have a different syntax. If you see red text in your program examine it closely - you may have made an error.

Editor Options

The edit options dialogue sets the colours used in syntax colouring and the size of the tab stops.

TabStops   The width of the tabs in characters. For a proportional font, the average character width is used.

Syntax Colours

User Functions   Used to colour text not contained in { }. Text outside of curly brackets is only used to name user defined functions and the 'main' function.

Functions   The colour of system functions (e.g., FindMenu, Sin, Cos,... etc.)

Comments   The colour of comments within the program

Constants   The colour of numeric and string constants (e.g. "Text" and 1.234 are both constants).

Operators   The colour of operators such as + - +=

Normal Text   The colour of text not in any of the above catagories.

Highlight Line   When the program is running, the line being executed is drawn with a background in this highlight colour.

BreakPoint Line   Lines that contain breakpoints are drawn with this background colour.

Window Acts, A Programmers Guide

Introduction

Flute manipulates applications by manipulating their windows. It provides a wide range of functions to control standard windows such as List boxes and Edit controls, but can only manipulate user defined windows in terms of key presses and mouse clicks.

You can improve control of your custom window classes by implementing Window Acts for those classes.

A Window Act is any function that the window supports.

For example, a colour selection box might provide two functions:

SetColour   a function to select a specified colour

GetColour   a function to return the currently selected colour

Advantages of Window Acts

- Window Acts are simple to implement, only the special window classes in your program require implementation - the standard window classes already have support functions. If you only use standard window classes such as List Boxes and Edit controls, then you do not need to implement anything.
- Flexible. Variable numbers of parameters can be passed and returned easily.
- Simple to Install: no type libraries are necessary. There is also no special OLE libraries to install. The executable file contains all the information for the Window Acts it supports so you no longer need to keep a Type library and Application in sync.
- Simple to Maintain: the system registry is left untouched and does not need repairing everytime you move a program on your disk.
- Intuitive: When you operate a program, you manipulate its windows. Window Acts manipulates those same windows.

Step through the following topics to see how WindowActs work. Then read either the C++ / MFC or C / SDK example on adding Window Acts to an application.

Samp_Act.Cpp   Adding Window Acts to a C++/MFC application

Samp_Act.C   Adding Window Acts to a C/SDK application

How These Functions are Called

In the following text the C / SDK code for implementing Window Acts is shown in blue and the C++ / MFC method is shown in green.

Window Acts are really registered Window messages. In the above example the colour selection control would register two messages with the RegisterWindowMessage function within Windows. When one of these messages is sent to the window, it is equivalent to calling the corresponding function for that window. The best place to do the message registration is in the WM_CREATE section of the window handler or in the OnCreate method in a class.

```
... // inside the window procedure
static     UINT Acts[2];          // UINT used to hold the ids of the Window Acts
switch (message) {
     case WM_CREATE:
          Acts[0] = RegisterWindowMessage("SetColour");
          Acts[1] = RegisterWindowMessage("GetColour");
          ... further initialisation here
};
```

```
//////////////////////////////////////////////////////////////////////////////
// inside the header file window declaration
class CMyNewClass : public CWnd
{
        ... various declarations follow
private:
        static UINT     m_Acts[2];              // UINT used to hold the window acts
                                                // notice that the Acts are static - they are shared
                                                // by all windows of the same class
}
```

```
//////////////////////////////////////////////////////////////////////////////
// inside the source file
static UINT CMyNewClass::m_Acts[2];

 int CMyNewClass::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
        m_Acts[0] = RegisterWindowMessage("SetColour");
        m_Acts[1] = RegisterWindowMessage("GetColour");
        ... further initialisation here
}
```

Rather than have lists of RegisterWindowMessage calls, a function RegisterActList is provided which allows you to register a complete list directly from your resource string table.

Handling the Message - Data Passing

Notice that you have not told the calling application what parameters to pass, or what parameters are returned. You have simply registered the **name** of the function. In Flute, all parameters to a function are passed in a standard format and all parameters returned are expected to be in the same standard format.

This format is the Fluid array structure within Flute (and CleanSheet our spreadsheet product). The data is formatted into a shared memory object allocated using the GlobalAlloc Windows function. The blocks are allocated using the GMEM_DDESHARE flag to allow them to be shared between applications.

The handle to this shared object is passed in the wParam of the message. The return handle is returned as the Window function return.

The code to handle the functions is shown below. Note that this code uses functions and declarations which are contained in the file WindActs.h and WindActs.c. These functions are explained in more detail later. They simplify implementation of Window Acts even further.

```
switch (message) {
    ... other message handlers
    default:
        if (message == Acts[0]) {
            // function to set the colour of a colour selection window
            DataObj  *pin;      // declare a pointed to the data
            long      lcolour;
            BOOL      btranslated;

            pin = DataGlobalLock( (HGLOBAL) wParam);      // lock the input data
            lcolour = GetIntValue(pin, &btranslated);    // get a long
            GlobalUnlock( (HGLOBAL) wParam);
            if ( ! btranslated) {
                return (LRESULT)GeneralError(IDS_BADPARAMS);
                // bad data, return an error object
            };
            SetChosenColour(lcolour);              // set the colour
            return (LRESULT)NULL;  // no return from this function
        } else if (message ==Acts[1]) {
            // function to return the current colour
            // uses the MakeIntObject function to
            // construct a Flute Integer object from a long
            return (LRESULT)MakeIntObject(lChosenColour);
        } else return DefWindowProc(hWnd, message,wParam, lParam);
};



/////////////////////////////////////////////////////////////////////////////////////////
// in the message map section, add two ON_REGISTERED_MESSAGE declarations
    //{{AFX_MSG_MAP(CMyNewClass)
    ON_WM_CREATE()
    ... further message maps added by Class Wizard
```

```
        //}}AFX_MSG_MAP
        // add your message macros after the normal Class Wizard ones
        ON_REGISTERED_MESSAGE(CMyNewClass::m_Acts[0], OnSetChosenColour)
        ON_REGISTERED_MESSAGE(CMyNewClass::m_Acts[1], OnGetChosenColour)


//////////////////////////////////////////////////////////////////////////////////
// in the header file declare the two handler functions after the Class Wizard functions
        //{{AFX_MSG(CMyNewClass)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        .... further declarations follow
        //}}AFX_MSG
        // declare your Window Acts
        afx_msg LRESULT OnSetChosenColour(WPARAM wParam, LPARAM lParam);
        afx_msg LRESULT OnGetChosenColour(WPARAM wParam, LPARAM lParam);


//////////////////////////////////////////////////////////////////////////////////
//   finally add the code to handle your two window acts

// Set the selected colour:
//      Input:   wParam - the handle to a numeric object specifying the colour
//               lParam - unused
//      Output: Either NULL, or an HGLOBAL to an error object
LRESULT OnSetChosenColour(WPARAM wParam, LPARAM )
{
        DataObj         *pin;     // declare a pointed to the data
        long            lcolour;
        BOOL            btranslated;

        pin = DataGlobalLock( (HGLOBAL) wParam);    // lock the input data
        lcolour = GetIntValue(pin, &btranslated); // get a long
        GlobalUnlock( (HGLOBAL) wParam);
        if ( ! btranslated) {
                return (LRESULT)GeneralError(IDS_BADPARAMS);
                // bad data, return an error object
        };
        SetChosenColour(lcolour);               // set the colour
        return (LRESULT)NULL;                   // no return from this function
};

// Get the selected colour
//      Input:   wParam - unused
//               lParam - unused
//      Output: An HGLOBAL to a long integer object containing the chosen colour
LRESULT OnGetChosenColour(WPARAM, LPARAM)
{
        // uses the MakeIntObject function to construct a Flute Integer object from a long
        return (LRESULT)MakeIntObject(lChosenColour);
};
```

The Format for Data

The format for the Global memory block is the same format used to pass data to a DLL. This format allows you to have variable numbers of parameters, and parameters of different types.

See How Data is Passed to a DLL for details.

Notice that while OLE Automation passes data by mimicking C++ conventions (which are designed for a compiled language), Flute mimicks CleanSheet's passing conventions (which are parsed at runtime rather than compile time).

Flute functions are designed to be polymorphic - they often have two or three actions they perform depending on the data passed. You are encouraged to take advantage of this freedom when writing Window Acts - **if there is a sensible action you can perform with the given data then perform that action**.

Who owns the Data

When a block of data is passed to your window procedure or class method, you do not need to free it. It is the calling applications responsibility to free this block.

When you return a data block, again you do not own it. The controlling program that requested the data will free this block for you.

Stopping Message Deadlocks

The Window Act message is sent to a window using the SendMessage function. This has an unfortunate side effect under Windows 3.1 - if you go into a message loop while processing one of these messages, Windows will hang due to a message deadlock between your application and Flute. Your application can go into a message loop when you don't expect it, for example if you create a dialogue using Windows DialogBox function, or display a message with Windows MessageBox function.

However Windows provides a simple way around this deadlock:

        ReplyMessage(LRESULT lResult);

This function returns the result lResult to the calling application. If you go into a message loop that calls PeekMessage, GetMessage, Yield, DialogBox, MessageBox or any other Windows function that might release control of messages - prepare the result of the call beforehand and call ReplyMessage.

Once you have gone into a message loop, the data block passed to you will no longer be valid, so ensure you get every parameter you need *before* you go into the loop.

Further Improvement - Supporting the Browser

Registering the functions as Window Messages is enough to support Window Acts. You could then provide printed documentation for these functions to allow programmers to operate your windows.

However, Flute provides a Window Act browser to allow the programmers to browse through your windows and see which Acts they support. It also permits programmers to retrieve help on each of the Acts.

> // declare your variable with global scope
>
> UINT   uActListMsg;
>
>
> // initialise it once somewhere early on in your program
>
> uActListMsg = RegisterWindowMessage("ActList");

How the Browser Work

The browser sends an "ActList" message to a window to decide if it supports any Window Acts. The wParam and lParam parameters of the message are both 0.

If the window supports any Acts it should return a fluid array listing the functions. Using array notation our colour selection box would look like this:

> {{"SetColour", "SetColour(NewColour) - Sets the colour given the number of a colour n."} ,
>
> {"GetColour","GetColour - return the colour currently selected"}}

Notice that each Window Act has two string objects:
- The name of the function (not case sensitive)
  - A short description of the function

This is used by the browser to display a list of the available Window Acts. The function ProduceActList provides a much easier way of generating these lists directly from strings in your resource table.

► It is recommended that functions that take parameters should specify those parameters. For example in SetColour, our description started with "SetColour(NewColour)". This tells the browser that we to pass one parameter "NewColour". The browser will place a short reminder of this parameter into the array.

How the Browser Provides Help on a Function

When the browser needs to provide help on a particular function it sends the same "ActList" message but with different parameters.

> message =   RegisterWindowMessage("ActList");
>
> wParam =   id of window Act to provide help on.
>
> lParam =   NULL

Here wParam contains the id of the Window Act that you should provide help on. This is the same id returned when you registered the function using RegisterWindowMessage.

Typical Browser Support Code

```
switch (message) {
      ... other message handlers
      default:
            if (message == uActListMsg) {          // browser message?
                if ( ! wParam)
                    // wParam is zero, therefore return the list of supported acts
                    return (LRESULT)ProduceActList(hInst, IDS_COLOURACTLIST);
                else {
                    DWORD          ulHelpContext;
                    // wParam is not zero, so provide help on the function
                    ReplyMessage(NULL); // stop message deadlocks

                    // convert the window act id to a help context id
                    if ( wParam == Acts[0] ) ulHelpContext = ID_HELP_SETCOLOUR;
                    else if ( wParam == Acts[1]) ulHelpContext = ID_HELP_GETCOLOUR;
                    else ulHelpContext = ID_HELP_CONTENTS;

                    ReplyMessage(NULL); // see stopping message deadlocks
                    WinHelp(mainhWnd, WINHELPFILE,HELP_CONTEXT, ulHelpContext);
                    return NULL;
                };
            } else if (message==acts[0]) {
                // handle   window act 0
            ... further window acts follow
};


/////////////////////////////////////////////////////////////////////////////////////////
// in the message map section, add a ON_REGISTERED_MESSAGE declaration
        //{{AFX_MSG_MAP(CMyNewClass)
        ON_WM_CREATE()
        ... further message maps added by Class Wizard
        //}}AFX_MSG_MAP
        // uActListMsg is a global UINT containing the Registered message "ActList"
        ON_REGISTERED_MESSAGE(uActListMsg, OnActList)

/////////////////////////////////////////////////////////////////////////////////////////
// in the header file declare the handler function after the Class Wizard functions
        //{{AFX_MSG(CMyNewClass)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        .... further declarations follow
        //}}AFX_MSG
        // declare your ActList function
        afx_msg LRESULT OnActList(WPARAM wParam, LPARAM lParam);

/////////////////////////////////////////////////////////////////////////////////////////
//   finally add the code to handle your act list function
```

```
// Provide a list of supported Window Acts, or help on a single window act
// Input: wParam = the id of the registered window message to provide help on
//         wParam = NULL if we should return an ActList instead.
LRESULT OnActList(WPARAM wParam, LPARAM )
{
        if ( ! wParam)
                // wParam is zero, therefore return the list of supported acts
                return (LRESULT)ProduceActList(hInst, IDS_COLOURACTLIST);
        else {
                DWORD          ulHelpContext;
                // wParam is not zero, so provide help on the function
                ReplyMessage(NULL);   // stop message deadlocks

                // convert the window act id to a help context id
                if ( wParam == m_Acts[0] ) ulHelpContext = ID_HELP_SETCOLOUR;
                else if ( wParam == m_Acts[1]) ulHelpContext = ID_HELP_GETCOLOUR;
                else ulHelpContext = ID_HELP_CONTENTS;
                ReplyMessage(NULL);   // see stopping message deadlocks

                WinHelp(mainhWnd, OURHELPFILE,HELP_CONTEXT, ulHelpContext);
                return NULL;
        };
};
```

In both the MFC and SDK examples we converted from the Act id (as returned by RegisterWindowMessage) into the Help Context id by a series of IF statements. If your window supports a lot of Window Acts then this is inefficient.

An alternative method of providing help on a Window Act is to convert the Registered Window message id back into a text string and then search for help on that subject.

The function FindActName is a complementary function to RegisterActList. It uses the same resource data to convert back from a previously registered Act message id into the name of the Act.

A typical block of code using this method:
```
        if ( ! wParam)
                // wParam is zero, therefore return the list of supported acts
                return (LRESULT)ProduceActList(hInst, IDS_COLOURACTLIST);
        else {
                char    ContextName[128];
                FindActName(hInst, wParam, ContextName,sizeof(ContextName));
                WinHelp(mainhWnd, OURHELPFILE, HELP_PARTIALKEY, ContextName);
        }
```

Further Improvement - Identifying Windows

Flute identifies windows using the best description it can manage. It is however a best guess rather than a perfect method.

You can improve the way Flute locates your window by giving the window a special meaningful *hidden* name that uniquely identifies it from its siblings.

Consider the Button Style dialogue within Flute:
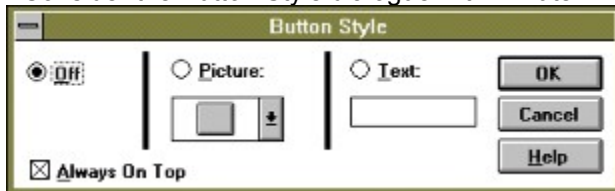
Here the 'Off' button, 'Picture' button and 'Text' button are easily identified by Flute - but what about the drop down list and the text edit field?
It would be better if we could give the drop down list the name "ChosenButtonPicture" and the text edit field "ButtonText" so that Flute can identify these items in a meaningful way.

Assigning Names to Windows

Names are stored as properties within each window. A windows name should be registered using GlobalAddAtom and assigned to the window using SetProp under the (global atom) name "FluteName".

Fortunately we provide functions to simplify this process.

The following code illustrates how to assign names to dialogue box items:

```
// in the INITDIALOG handler, assign atoms to each window using SetDlgItemFluteName
// function
case WM_INITDIALOG:
     // on INITDIALOG, set the names of special items on the dialogue
     SetDlgItemFluteName(hDlg,IDC_BUTTONPICTURECOMBO, "ChosenButtonPicture");
     SetDlgItemFluteName(hDlg,IDC_BUTTONTEXT, "ButtonText");
     break;
... other message handlers

case WM_DESTROY:
     // when the dialogue is destroyed, destroy any special names
     ClearDlgItemFluteNames(hDlg);
     return FALSE;
```

The code within C++/MFC is similar to the SDK version. Place the code to set the names of items on the dialogue within your OnInitialUpdate or OnInitDialog. Place the code to tidy up in the OnDestroy method.

Functions Provided to Simplify Window Acts

The following functions and definitions can be found in WindActs.c and WindActs.h. These provide helper functions to implement Window Acts within your program. They may be included freely within your code but are provided as is.

Definitions

DataObj   Declares a union which contains all possible types of data passed to your application.

General Functions

RegisterActList   Registers a list of window acts from strings within your resource string tables. This is done when your window is first created.

Functions for Windows that Support Browsing

ProduceActList   Produces a data array used by type browsers to display the list of acts supported by a window. This is optional, but if implemented allows you programs to be examined by browsers.

FindActName   Given a Window Act id, return the name of the corresponding Act. This can be used when displaying help on a particular Window Act. This is optional, but allows browsers to display detailed help.

Functions that Give Meaningful Names to Windows

SetWindowFluteName Sets the 'Flute Name' for a single window. This is optional, but helps Flute identify difficult to describe windows.

ClearWindowFluteName Clears the 'Flute Name' from a window. If SetWindowFluteName was called for a window, this function should be called when the window is destroyed to tidy up.

SetDlgItemFluteName Set the 'Flute Name' for an item on a dialogue. This is optional, but helps Flute identify difficult to describe windows on dialogues.

ClearDlgItemFluteNames Clears the 'Flute Name' for every item on a dialogue. If SetDlgItemFluteName is called for any item on a dialogue, this should be called when the dialogue is destroyed to tidy up.

Data Construction Functions

These simplify construction of the most common Flute Data Objects.

GeneralError   Converts a string in the resource table into an Error Object.

MakeString   Makes a string object from a ptr to a null terminated string

MakeInt   Makes an integer object (Flute integers are 32 bit numbers)

| MakeDouble | Makes a double object from a double. |
| NullObj | Returns a Null Object |

Data Extraction Functions

These functions get data out of an object

GetALong	Given a ptr to an object, extracts a long value (converting as necessary)
GetADouble	Given a ptr to an object, extracts a double value (converting as necessary)
GetAString	Given a ptr to a string object or error object, extracts the string into a null terminated buffer.

Array Assembly Functions

These assemble arrays from given objects.

MakeArray2	Assembles a two element array
MakeArray3	Assembles a three element array
MakeArray4	Assembles a four element array
MakeArrayN	Assembles an N element array

Miscellaneous Functions

GetVarSize	Given a ptr to a data object, returns it size in bytes
NextDataObj	Given a ptr to a data object in a fluid array, this returns a ptr to the next object in the array.
PtrContent	Given a ptr to an array object, this returns a ptr to the first element in that array

DataObj

Defined in WindActs.h

Declares a union which contains all possible types of data passed to your application. If you examine the definition of DataObj (below), you can see that it contains structures that represent each object. Each of these structures contains a short value called vtype as their first entry. This is used to decide what type of object is contained within DataObj.

```
typedef union   varObj {
        struct   t_int   tint;              // integer and null
        struct   t_float tfloat;            // floating point number
        struct   t_double tdouble;          // double precision number
        struct   t_string tstring; // string object
        struct   t_boolean tboolean;        // boolean object
        struct   t_error terror;            // error object
        struct   t_date tdate;              // date object
        struct   t_hms thms;                            // hours minutes seconds object (time)
        struct   t_complex tcomplex;        // complex number object
        struct   t_equation tequation;      // equation object
        struct   t_array tarray;            // array of DataObjs
        } DataObj;
```

RegisterActList

Registers a list of window acts from strings within your resource string tables. This is done when your window is first created. This function simplifies the mechanics of registering multiple Window Acts.

void   RegisterActList(HINSTANCE hInst, UINT baseID,UINT FAR * pmessID)

hInst	The instance handle of the module that contains the resouce strings.
baseID	The id of the first string in the table that describes the Acts
pmessID	Ptr to the table of UINTs that will receive the result

Notes

To register a list of Window Acts for a particular window class. Construct a set of strings within your resources that have sequential id numbers. For example:

IDS_CACT1	"SetColour"
IDS_CACT2	"Sets the currently selected colour."
IDS_CACT3	"GetColour"
IDS_CACT4	"Gets the currently selected colour."
IDS_CACT5	"GetVersionNumber"
IDS_CACT6	"Returns the version number of this window."
IDS_CACT7	"!"

Each Window Act has two strings, the first string contains the name of the Window Act and the second string contains a brief description.

The last entry in the table contains an exclamation mark used to indicate the end.

The ID's IDS_CACT1, IDS_CACT2...IDS_CACT7 must have sequential ids (e.g. 1000,1001,1002,1003...1006).

```
{
    UINT    Acts[3];
    RegisterActList (hInst, IDS_CACT1, Acts);
}
```

This fragment of code will register the three Window Acts and place the resulting ids in the Acts array.

ProduceActList

Produces a data array used by type browsers to display the list of acts supported by a window.

HGLOBAL ProduceActList(HINSTANCE hInst, UINT baseID)

hInst   Instance handle of this code module.

baseID   ID of the first resource string in the Act table.

Notes

When a window class is sent the message "ActList" it should return a list of Window Acts that it supports. This function simplifies construction of this list.

To make use of it, you should declare a set of strings within your resource, such as the example that follows. The format for these strings is the same used by all the Window Act helper functions.

IDS_CACT1	"SetColour"
IDS_CACT2	"Sets the currently selected colour."
IDS_CACT3	"GetColour"
IDS_CACT4	"Gets the currently selected colour."
IDS_CACT5	"GetVersionNumber"
IDS_CACT6	"Returns the version number of this window."
IDS_CACT7	"!"

Each Window Act has two strings, the first string contains the name of the Window Act and the second string contains a brief description.

The last entry in the table contains an exclamation mark used to indicate the end.

The ID's IDS_CACT1, IDS_CACT2...IDS_CACT7 must have sequential ids (e.g. 1000,1001,1002,1003...1006).

```
if (message == uActListMsg) {
    if ( !wParam)
        return ProduceActList( hInst, IDS_CACT1);
};
```

This block of code uses ProduceActList to return the information about Window Acts that this window supports.

FindActName

Given a Window Act id, return the name of the corresponding Act.

void FindActName(HINSTANCE hInst, UINT ActId, UINT baseID, LPSTR pBuff, int len)

hInst	Instance handle for the module of code containing the strings.
ActId	The id of the Act to look for
baseID	The id of the first string containing the Act definitions
pBuff	Ptr to the buffer to receive the name
len	Size of the buffer

Notes

The is the reverse of RegisterActList, it converts back from an Act ID to the name of the Act.

As with the other helper functions, you should declare a set of strings for the Acts supported by your window class.

IDS_CACT1	"SetColour"
IDS_CACT2	"Sets the currently selected colour."
IDS_CACT3	"GetColour"
IDS_CACT4	"Gets the currently selected colour."
IDS_CACT5	"GetVersionNumber"
IDS_CACT6	"Returns the version number of this window."
IDS_CACT7	"!"

Each Window Act has two strings, the first string contains the name of the Window Act and the second string contains a brief description.

The last entry in the table contains an exclamation mark used to indicate the end.

The ID's IDS_CACT1, IDS_CACT2...IDS_CACT7 must have sequential ids (e.g. 1000,1001,1002,1003...1006).

```
{
    UINT    Acts[3];
    char    text[256];

    RegisterActList (hInst, IDS_CACT1, Acts);
    FindActName (hInst, Acts[1], IDS_CACT1, text, sizeof(text));
}
```

// This example registers all the window acts and then searches for Acts[1]. After the FindActName call, the text buffer contains the string "GetColour".

SetWindowFluteName

Sets the 'FluteName' for a single window. This name is optional, but helps Flute identify difficult to describe windows. These names are hidden from the outside world and do not affect the application. They have no connection to the windows text.

FluteNames should differentiate one sibling from another. For example, if an application has three button bars, you might want to give them FluteNames "ButtonBar1", "ButtonBar2", "ButtonBar3".

It is not necessary to give a window a Flutename if it has a window name that identifies it.

void SetWindowFluteName(HWND hWnd, LPCSTR lpBuff)

hWnd   Handle of the window

lpBuff   Ptr to the name to set

ClearWindowFluteName

Clears the 'Flute Name' from a window. If SetWindowFluteName was called for a window, this function should be called when the window is destroyed to tidy up.

void ClearWindowFluteName(HWND hWnd)

SetDlgItemFluteName

Set the 'Flute Name' for an item on a dialogue. This is optional, but helps Flute identify difficult to describe windows on dialogues.

void SetDlgItemFluteName(HWND hDlg, UINT id, LPCSTR lpBuff)

hDlg	Handle of the dialog window
id	The id of the dialogue item
lpBuff	Ptr to the name to set

ClearDlgItemFluteNames

Clears the 'Flute Name' for every item on a dialogue. If SetDlgItemFluteName is called for any item on a dialogue, this should be called when the dialogue is destroyed to tidy up.

void ClearDlgItemFluteNames(HWND hDlg)

hDlg   Handle of the dialog window

GeneralError

Converts a string in the resource table into an Error Object held in a HGLOBAL. This is a useful function for returning errors.

HGLOBAL   GeneralError(HINSTANCE hInst, UINT id)

hInst	Handle of the instance that has the resource string
id	The strings ID number

MakeString

Makes a string object held in a HGLOBAL from a ptr to a null terminated string.

HGLOBAL MakeString(LPCSTR lpBuff)

lpBuff   Ptr to the text string (null terminated)

MakeInt

Makes an integer object (Flute integers are 32 bit numbers). The integer object is held inside a HGLOBAL.

HGLOBAL MakeInt(long lval)

lval   The value to produce an integer object from

MakeDouble

Makes a double object from a double. The returned object is stored in a HGLOBAL

HGLOBAL MakeDouble(double dval)

dval   The double value to wrap in a double object

NullObj

Returns a Null Object - the null object is wrapped in a HGLOBAL.

HGLOBAL   NullObj( );

GetALong

Given a ptr to an object, extracts a long value (converting as necessary from other numeric types).

long GetALong(DataObj FAR *optr, BOOL FAR *pTrans)

optr   ptr to the object to extract a long value from

pTrans   ptr to a Boolean that is filled on exit. If a long value was successfully extracted
                this result is set TRUE, otherwise the Boolean is set FALSE to indicate failure.

Notes

Consider the following fragment of code:

```
{
    HGLOBAL result;
    BOOL      b;
    DataObj   FAR *pdat;
    long    lval;

    result = MakeDouble (100.04);     // make a double object
    pdat = DataGlobalLock(result);    // lock it to access it
    lval = GetALong(pdat, &b);        // get a long from the data
    GlobalUnlock(result);             // unlock the data

    // after this code, lval contains 100 and b is TRUE
}
```

GetADouble

Given a ptr to an object, extracts a double value (converting as necessary from other numeric objects).

double GetADouble(DataObj FAR *optr, BOOL FAR *pTrans)

optr   ptr to the data object to extract a double from

pTrans   ptr to a Boolean that is filled on return. The result is TRUE if successfully translated or FALSE if failed.

GetAString

Given a ptr to a string object or error object, extracts the string into a buffer and null terminate the string.

void GetAString(DataObj FAR *optr, BOOL FAR *pTrans, char FAR *pBuff, UINT len)

optr	ptr to the data object to extract a string from
pTrans	Ptr to Boolean that is filled on exit, the value is TRUE if string or error text was successfully extracted from the object.
pBuff	Ptr to the buffer to receive the string
len	length of the buffer

Note

Consider the following code fragment

```
{
    HGLOBAL result;
    char text[256];
    DataObj    FAR *optr;
    BOOL   b;

    result = GeneralError(hInst, IDS_PARAMETERERROR);
    optr = DataGlobalLock(result);
    GetAString(optr,   &b, text, sizeof(text));
    GlobalUnlock(result);
    // after this code, text contains the error string that corresponds to
    // resource string IDS_PARAMETERERROR, and b is TRUE
}
```

MakeArray2

Assembles a two element array given two objects encapsulated in HGLOBALs. After the call, the two HGLOBALs are freed.

HGLOBAL MakeArray2 (HGLOBAL obj1, HGLOBAL obj2)

    obj1   HGLOBAL to first element

    obj2   HGLOBAL to second element

Returns a HGLOBAL containing an array {obj1, obj2}

Example

return MakeArray2 (MakeDouble(100.102), MakeString("Sample Text"));

This function returns a fluid array   {100.102, "Sample Text"}

MakeArray3

Assembles a three element array from 3 objects provided in HGLOBALs. After the call, the input HGLOBALs are freed.

HGLOBAL MakeArray3 (HGLOBAL obj1, HGLOBAL obj2, HGLOBAL obj3)

    obj1   HGLOBAL to first element

    obj2   HGLOBAL to second element

    obj3   HGLOBAL to third element

Returns a HGLOBAL containing an array {obj1, obj2, obj3}

MakeArray4

Assembles a three element array from 4 objects provided in HGLOBALs. After the call, the input HGLOBALs are freed.

HGLOBAL MakeArray3 (HGLOBAL obj1, HGLOBAL obj2, HGLOBAL obj3, HGLOBAL obj4)

    obj1   HGLOBAL to first element

    obj2   HGLOBAL to second element

    obj3   HGLOBAL to third element

    obj4   HGLOBAL to fourth element

Returns a HGLOBAL containing an array {obj1, obj2, obj3, obj4}

MakeArrayN

Assembles an N element array from an array of HGLOBALs. Each HGLOBAL should contain a Flute data object. After the call the HGLOBALs in the array area freed.

HGLOBAL MakeArrayN(HGLOBAL FAR *pHands, UINT N)

pHands   Ptr to an array of HGLOBALs that are to be assembled

N   The number of elements in the array

Example:
```
{
    HGLOBAL tabs[2];
    HGLOBAL result;

    tabs[0] = MakeDouble(1234.567);
    tabs[1] = MakeString("Example Text");
    result = MakeArrayN(tabs, 2);
    // after the call, result contains a Flute array {1234.567, "Example Text"}
    // note also that the HGLOBALs tabs[0] and tabs[1] are no longer
    // valid
}
```

GetVarSize

Given a ptr to a data object, returns it size in bytes.

long GetVarSize(DataObj   FAR *op1)

op1   ptr to the data object

NextDataObj

Given a ptr to a data object in a fluid array, this returns a ptr to the next object in the array. This can be used to step over objects in the array.

ptr = NextDataObj (ptr);

This is defined in "windacts.h" as a macro

Example

```
{
    // result is a HGLOBAL containing an array of 3 numbers {200.5, 100.4, 192.4}
    DataObj   FAR *pob;
    UINT      elements, i;
    double    total;
    BOOL      b;

    pob = DataGlobalLock(result);
    total=0e0;
    elements = pob->tarray.elements;
    pob = PtrContent (pob);
    for (i=0; i<elements; i++) {
        total += GetADouble ( pob,&b);
        pob = NextDataObj (pob);
    };
    GlobalUnlock(result);

    // after this code total contains 493.3
}
```

PtrContent

Given a ptr to an array object, this returns a ptr to the first element in that array.

It is defined as a macro in "windacts.h"

optr = PtrContent (optr)

Example

```
{
    // result is a HGLOBAL containing an array of 3 numbers {200.5, 100.4, 192.4}
    DataObj    FAR *pob;
    UINT       elements, i;
    double     total;
    BOOL       b;

    pob = DataGlobalLock(result);
    total=0e0;
    elements = pob->tarray.elements;
    pob = PtrContent (pob);
    for (i=0; i<elements; i++) {
        total += GetADouble ( pob,&b);
        pob = NextDataObj (pob);
    };
    GlobalUnlock(result);

    // after this code total contains 493.3
}
```

Samp_Act

In the folder Samp_Act.cpp on your Flute master disk you will find a null application that has had Window Acts added to it. This application was produced using AppWizard and displays a window (actual a CView) showing a number and a string. The following describes the steps used to add two Window Acts to this application: SetValues and GetValues.

All the modifications to the files are marked with the comment "WINDACTS" to enable you to easily find the changes.

Registering the Acts

The first step is to construct your list of supported Acts using App Studio. The following strings were added to the resources:
```
IDS_ACTSTRING1 "SetValues"
IDS_ACTSTRING2 "SetValues{IntObj, StringObj} - Sets the value of the integer and strings
                        displayed in this window."
IDS_ACTSTRING3 "GetValues"
IDS_ACTSTRING4 "Gets the value displayed in this window and returns them as an array"
IDS_ACTSTRING5 "!"
```

Including the Header File

Include the header file "windacts.h" as an external C header file and add the file windacts.c to your project.
```
extern "C" {
        #include        "windacts.h"
};
```

Register the Window Acts

In the header file for our CView (Samp_vw.h) we need to declare the Acts variables and in the Samp_vw.cpp file we need to declare space for these variables.

In Samp_vw.h:
```
        private:
                static UINT     Acts[2];
```

In Samp_vw.cpp
```
        UINT    CSamp_actView::Acts[2];
```

In the Samp_Vw.cpp file we add the command to register the Window Acts in our OnCreate method.
```
        RegisterActList(AfxGetInstanceHandle(), IDS_ACTSTRING1,Acts);
```

This register the Acts that we support using the resource information we have already created.

Declaring our Acts to MFC

Now that we've registered our WindowActs, we need to declare them in the message maps of our window.

Inside the BEGIN_MESSAGE_MAP / END_MESSAGE_MAP macros in the file Samp_vw.cpp we need to add calls to out two new functions:

```
ON_REGISTERED_MESSAGE(CSamp_actView::Acts[0], OnSetValues)
ON_REGISTERED_MESSAGE(CSamp_actView::Acts[1], OnGetValues)
```

Inside the Samp_vw.h file we also need to add the definitions of our window acts.:

```
//}}AFX_MSG
afx_msg LRESULT OnSetValues(WPARAM wParam, LPARAM lParam);
afx_msg LRESULT OnGetValues(WPARAM wParam, LPARAM lParam);
DECLARE_MESSAGE_MAP()
```

Notice that the declarations are added *after* AFX_MSG and *before* DECLARE_MESSAGE_MAP.

Adding Our Code

Now we can add the code to handle OnSetValues and OnGetValues to the file Samp_vw.cpp:

```
/////////////////////////////////////////////////////////////////////
// Window Acts message handlers
LRESULT CSamp_actView::OnSetValues(WPARAM wParam, LPARAM lParam)
{
        DataObj         *pob;
        BOOL    b1,b2;
        long    lval;
        CString sval;

        if (!wParam) return 0;           // there is a problem

        pob = DataGlobalLock((HGLOBAL)wParam);
        if (pob->tint.vtype!=ARRAYTYPE || pob->tarray.elements!=2) {
                // we haven't been given the correct parameters - report an error
                GlobalUnlock((HGLOBAL)wParam);
                return (LRESULT)
                        GeneralError(AfxGetInstanceHandle(),IDS_PARAMETERERROR);
        };

        pob = PtrContent(pob);
        lval = GetALong(pob,&b1);
        pob = NextDataObj(pob);
        GetAString(pob, &b2, sval.GetBufferSetLength(256),256);
        sval.ReleaseBuffer();
        GlobalUnlock((HGLOBAL)wParam);

        // if the extraction of data failed, then report an error
        if (!b1 || !b2)
                return (LRESULT)
                        GeneralError(AfxGetInstanceHandle(),IDS_PARAMETERERROR);

        // set the values and update the window
        m_lValue1 = lval;
        m_sValue2 = sval;
        Invalidate();

        return NULL;
```

```
}

LRESULT CSamp_actView::OnGetValues(WPARAM wParam, LPARAM lParam)
{
        return (LRESULT)MakeArray2(MakeInt(m_lValue1), MakeString(m_sValue2) );
}
```

We have implemented the Window Acts, and at this point we could stop. However it would be nice to be able to browse this window.

See Also

Adding Browser Support to Samp_Act

Adding Window Naming to Samp_Act

Adding Browser Support

We are going to add support to allow the Flute browser to examine Acts supported by our window.

In the application we need a UINT to hold the id of the "ActList" message. This is the message used by the browser and since this same id is used by all the classes that support Window Acts, we need to register it once and give it global scope.

In Samp_Act.cpp (the Application object's file) declare the variable (note it is a near variable)
```
UINT    near uActListMsg;
```

and in the header file Samp_Act.h, add a definition
```
extern UINT near uActListMsg;
```

Here we've made it global, you could place it in the Application object if you wished.

We need to register this message when the application first starts, so we add the line
```
uActListMsg = RegisterWindowMessage("ActList");
```

inside the InitInstance method.

Handling the Browser Message

We need to declare the ActList message handler in our message map and the header file for our Window.

In Samp_vw.cpp:
```
ON_REGISTERED_MESSAGE(uActListMsg, OnActList)
```

In Samp_vw.h :
```
afx_msg LRESULT OnActList(WPARAM wParam, LPARAM lParam);
```

And finally add the code to support the browser:
```
LRESULT CSamp_actView::OnActList(WPARAM wParam, LPARAM)
{
        CString sres;

        // if not wParam then return a list of Acts
        if (!wParam) return (LRESULT)
                        ProduceActList(AfxGetInstanceHandle(), IDS_ACTSTRING1);

        // else provide help - find the act name
        FindActName(AfxGetInstanceHandle(), (UINT)wParam,
                        IDS_ACTSTRING1, sres.GetBufferSetLength(128), 128);
        sres.ReleaseBuffer();
        ReplyMessage(NULL);   // see stopping message deadlocks

        ::WinHelp(m_hWnd,"SampAct.hlp",HELP_PARTIALKEY, (DWORD)(LPCSTR)sres);
        return NULL;
}
```

Notice that we've used ReplyMessage to avoid a message deadlock in case WinHelp dispatches messages.

Naming the SampAct Window

It would be nice for our window to have a meaningful name. To do this, a single call to SetWindowFluteName is all that is required.

The best place for this is in OnCreate:

```
SetWindowFluteName(m_hWnd, "My_Sample_Window");
```

The name should be meaningful, unique and constant so that each time the program starts this window is always called by the same name.

As we only have one window of this class, we only need to use a single name. If we had two windows of the same class that are siblings, we would need need to set different names for each sibling (this only applies to siblings, Flute Names only need to differentiate between siblings).

Samp_Act.C

Adding WindActs to a C/SDK Program. In the folder Samp_act.c on your Flute master disk you will find a simply C/SDK program that displays a window and draws two values - a number and a string.

Adding the Window Acts

The first step is to include the "windacts.h" file in our Window Class source file (samp_act.c) and add the "WindActs.c" file to our project.

Next we need to register our window acts messages.

```
static   UINT  Acts[2];
```

In the window class - declare a static array of UINTs used to hold the ids of the Window Acts.

In the resources, we need to add strings that list and describe all our Window Acts:

```
IDS_ACTSTRING1          "SetValues"
IDS_ACTSTRING2          "SetValues{IntObj, StringObj} - Sets the value of the integer and strings
displayed in this window."
IDS_ACTSTRING3          "GetValues"
IDS_ACTSTRING4          "Gets the value displayed in this window and returns them as an array"
IDS_ACTSTRING5          "!"
```

Notice that the list ends in a string containing an exclamation mark and that each Window Act has a name and a description string.

```
RegisterActList(hInst, IDS_ACTSTRING1,Acts);
```

In the WM_CREATE section of the window procedure, register these Window Acts using the RegisterActList function.

Adding the Acts Code

In the default section of our window procedure, we need to test for the two Window Acts and call the appropriate handler. Refer to Samp_Act.c for the code to handle these functions (SetValues and GetValues).

```
default:
     if (message == Acts[0]) {
          return SetValues( (HGLOBAL) wParam, hWnd);
     } else if (message == Acts[1]) {
          return GetValues();
     } else return DefWindowProc(hWnd,message,wParam,lParam);
```

At this point we can stop - we have implemented Window Acts. However it would be better if the browser could list our functions and if our single window had a meaningful name.

Supporting the Browser

<u>Naming the Window</u>

Adding Browser Support to Samp_act.c

The browser sends an "ActList" message to the window to request a list of functions supported and ask for help on individual functions.

The first step in supporting this message is to register it.
- Declare a Global Variable UINT uActListMsg - this will hold the id of this message.
- In our initialisation code in the WinMain function, register the message:
  uActListMsg = RegisterWindowMessage("ActList");

Next we add support for this message in the default section of our window procedure:
default:
 if (message == uActListMsg) {
     // a browser message has been received
     if (wParam) {
         // help on a function
         char   actname[128];
         FindActName(hInst, (UINT)wParam, IDS_ACTSTRING1, actname, sizeof(actname));
         ReplyMessage(NULL);   // IMPORTANT
         WinHelp(hWnd,"SampAct.hlp",HELP_PARTIALKEY, (DWORD)actname);
         return NULL;
     } else {
         // return a list of functions
         return (LRESULT)ProduceActList(hInst, IDS_ACTSTRING1);
     };

 } else .... further tests follow

Notice that before WinHelp is called, we call "ReplyMessage". This is to avoid a message deadlock under Windows 3.1x (see Stopping Message Deadlocks)

Our window class now fully supports the browser.

Naming The Window

It would be nice for our window to have a meaningful name. To do this, a single call to SetWindowFluteName is all that is required.

The best place for this is in WM_CREATE section of our window:

```
SetWindowFluteName(hWnd, "My_Sample_Window");
```

The name should be meaningful, unique and constant so that each time the program starts this window is always called by the same name.

As we only have one window of this class, we only need to use a single name. If we had two windows of the same class that are siblings, we would need need to set different names for each sibling (this only applies to siblings, Flute Names only need to differentiate between siblings).

OLE Automation

OLE Automation is a way of controlling one application from another through the OLE system designed by Microsoft. To use these functions the target application must support OLE **Automation**, supporting OLE alone is not enough.

Automation is build around objects (for example a Worksheet is treated as an object). Objects are either created new, or loaded from files and released when no longer needed.

CreateObject   Creates a new object

GetObject   Gets an object from a file

ReleaseObject   Releases an object when it is not longer needed

Function Calls in OLE Automation

In object oriented programming, you ask the object to perform a function rather than applying a function on an object. This makes an important difference to the syntax of function calls. Function calls are renamed *methods* and any method that sets or returns a value is renamed a *property*. To call a method within an object we follow the object with a decimal point, followed by the name of the method, followed by any parameters.

For example, if variable 'AnObj' contains an object and that object has a method called 'SetDefaultName' then to set the default name to be "MyName" we would use:

        AnObj.SetDefaultName("MyName");

Notice that there are no spaces between the decimal point and the method name 'SetDefaultName' - this lets Flute recognise that the decimal point is not from a number.

Just as with other Flute function, to call a method that requires several parameters we use an array of parameters (signified with curly brackets).

        AnObj.SetDefaultNames{"MyName1" , "MyName2"};

Assigning Values to Properties

If an object has a 'Name' property, then we can set the value of 'Name' by using the normal Flute assignment operator :=

        AnObj.Name := "This is some text";

If the object has a matching property get function, we can use the complex assignment operators in Flute, such as += -= *= etc (see Complex Assignments to a Variable for a list of what's available).

The += assignment adds the new value to the present value of a property.

        AnObj.Value += 3;

Is equivalent to

        AnObj.Value := AnObj.Value +3;

Methods that take Optional Arguments

If a method takes 4 parameters, 3 of which are optional, you can omit the last 3 parameters. For example:

AnObj.Method {"Param1"};

By supplying only one parameter, the remaining three are treated by the method as being omitted. However if you want to supply the first and third parameter, omitting the second and fourth you need to place a NULL in place of the second parameter.

AnObj.Method {"Param1", NULL, "Param3"};

Flute will inform the method that parameter 2 was not supplied.

Methods That Take Named Arguments

Many methods refer to parameters by name. For example the Cells method within Microsoft Excel takes two named arguments *RowIndex* and *ColumnIndex*. To call this function you can either list the parameters in the correct order:

AnObj.Cells {4,2}   Indicates row 4, column 2.

Alternatively you can refer to the parameters by name:

AnObj.Cells {RowIndex := 4, ColumnIndex :=2};

Inside the array of parameters for a method, the := operator is used to indicate the name of a parameter. The order you supply named parameters is unimportant the following line of code is equivalent:

AnObj.Cells {ColumnIndex :=2, RowIndex:=4};

If a method takes unnamed and named parameters, the unnamed parameters must be listed first:

AnObj.ExampleMethod { 1, "Param2", Param3 := "hello"};

In the above example, the method takes 2 regular parameters and a single named parameter.

Methods That Return Values by Reference

Automation is build around the language C++. C++ can return only single values from functions, so to overcome this shortcoming, C++ functions can store return values into addresses passed as function parameters.

Quite literally you tell the function where to store the result.

Automation has adopted this exact same system to return multiple values - a system known as passing by reference. If a method takes 4 values, two of which are also return values then these parameters must be marked with the 'ByRef' function to indicate that they should be passed by reference:

a := 3;

b: = 4;

AnObj.PassingMethod {"Param1", "Param2", ByRef (a), ByRef (b) };

Notice that the last two parameters are taken from variables (in this case variables 'a' and 'b'). When the method is called, variables 'a' and 'b' will contain return values. It is important to initialise any variables you intend to pass by reference as the method will expect sensible values passed to it.

ByRef (a)

The ByRef function is only used to indicate parameters passed by reference to OLE Automation, it has no other purpose within Flute. Also since ByRef only takes one parameter the variable is surrounded by round brackets ( ) rather than the curly brackets {} used to indicate an array of parameters.

As well as storing the results in variables, we can also store the results in parts of variables. Form example:

> a := {3,4};

> AnObj.PassingMethod { ByRef (a[0]) , ByRef (a[1]) };

Here we are passing a[0] and a[1] by reference. When the method returns element 0 and 1 of the array will contain the return values.

Methods that Return Objects as Result

Some objects have methods that return objects as results. For example, the Cells method within Excel actually returns a Range object.

> a:=getobject{"d:\msoffice\excel\examples\nigel.xls"};

> a.Cells{2@,4@}.Value := "Sample Text";

> ReleaseObject(a);

In this example, we first get an excel file as an object. The resulting object is stored in variable 'a'.

> a.Cells{2@, 4@}

returns a range object for cell row 2 column 4

> .Value := "Sample Text";

stores the text "Sample Text" into the Value property of this range object.

Methods that Return Collections

A collection is an array object that is accessed through methods. Typically a Collection object has a Count property to return the number of items in the collection and an Item method to return a single item. Collections are a hangover from OLE, OLE Automation supports arrays as return types and does not need collections.

Flute provides a function called Collect to convert a collection into a normal Flute array. Using this function avoids the 'key hole' surgery that is usually involved in collections.

Active Objects

You can create new objects using CreateObject, and obtain objects from files using GetObject, but how can you tell which objects are already running?

Flute provides the function GetActiveObjects to return an array of the names (actually the Monikers) of objects running in the system. For a valid moniker it is possible to get the object associated with it by using the GetObject function.

Browsing for Objects

Flute provides an Object Browser which allows you to see the objects and their associated methods, properties and parameters and construct the method calls automatically. There are several pitfalls associated with this.

Firstly, the browser displays information provided by Type Libraries. Type Libraries were an afterthought by Microsoft and are often either not provided or are inaccurate. For example version 5 of Excel provides a type library that specifies no parameters for the Atan2 function in the application object. Atan2 actually takes two parameters, because of the fault in the type library it is not possible to call Atan2 from Flute with the correct number of parameters because Flute uses the Type information available.

Secondly, the browser only display information for libraries that have type information. There may be more objects that support Automation but Flute does not know about these.

See Paste OLE Automation

Paste Ole Automation Function / Object

This dialogue displays the objects and functions available to OLE Automation programs. To use this dialogue you must have an understanding of the workings of OLE Automation.
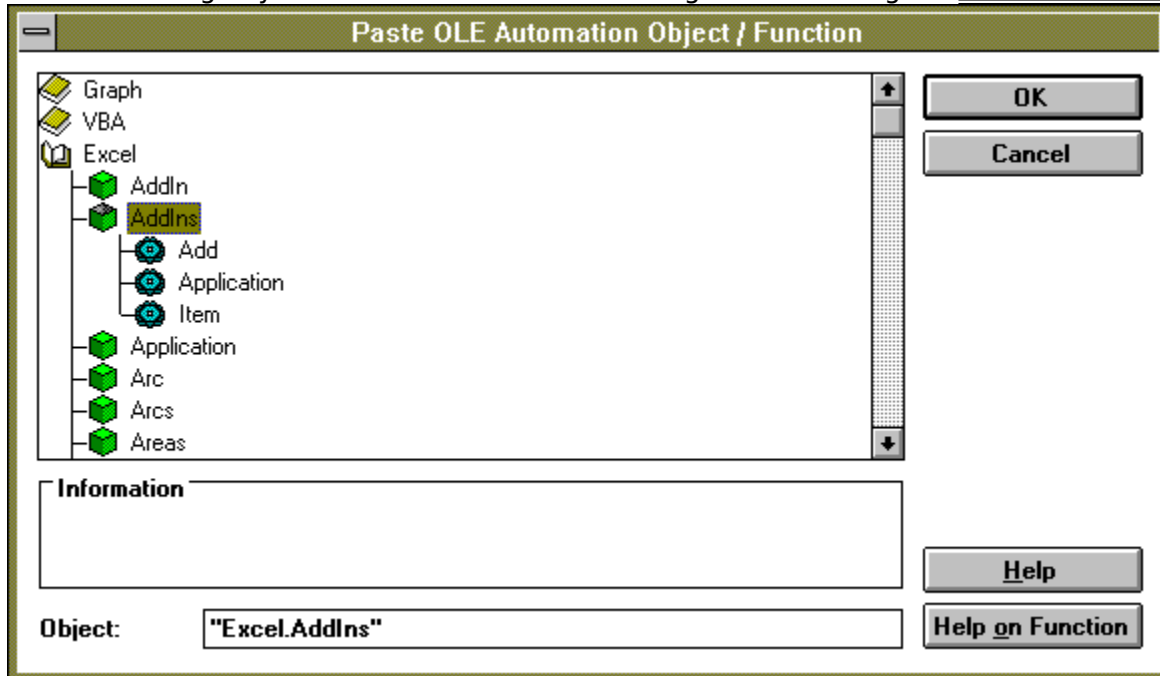
Click on any part of the dialogue you need help with.

When the dialogue first opens it contains a list of the type libraries available on the system.
- Type library
- Objects within a type library
- Methods and Properties within the Object

To open a type library double click on it. Inside the type library are a list of the objects supported by that library. If you select one of these objects the paste text contains the specification for that object, for example "Excel.Sheet". When you create objects using the CreateObject function, it requires this object specification. Note that just because an object is listed in this dialogue does not mean that you can use CreateObject to make one. Some of these objects can only be returned from function calls.

To open an object to see the methods and properties it contains double click on it.

Select a method or property and the appropriate construction needed to access this method is shown in the edit field. If you need help on either a function or an object, select the 'Help on Function' button.

This section shows a hierarchical view of the libraries, objects and methods available.

If the object, method or library has information available on it, that information is displayed in this section.

This line of text displays 'Object' for objects, and 'Method' or 'Property' as appropriate to indicate what type of function is selected.

This edit field displays the function call together with any named parameters it requires. This is the text that will be pasted into your program.

To obtain help on a particular function, click on this button.

CreateObject

AnObj := CreateObject (ObjectSpec)

Creates a new object given an object specification. When an application is started using OLE Automation it is usually hidden from the user, do not be surprised if you cannot see the application on screen.

ObjectSpec   A specification for an object as a string, for example "Word.Basic" is a WordBasic object within Microsoft Word.

The return from this function is an OLE Object if successful or an appropriate error object. An OLE Object is actually a special kind of array that contains all the information needed to access the object.

Example:

AnObject := CreateObject("Word.Basic");

See Also

GetObject

GetObject

Loads an object from a file, or returns the active object if the object is already running. When an application is started using OLE Automation it is usually hidden from the user.

AnObj := GetObject { Pathname, ClassName };

PathName   A Pathname to the file, for example "d:\flute\example.xls"

ClassName   The type of object to create from the file. This is normally not omitted which tells get object to return the default object for that type of file.

Example:

AnObj := GetObject ("d:\flute\example.xls");

Loads the example Excel sheet into memory and stores the object for it in the variable 'AnObj'. If this sheet is already loaded AnObj returns an object for the presently loaded sheet.

AnObj := GetObject {NULL, "Word.Basic"};

Returns the first object of type Word.Basic that is presently running. Note that this only works if application correctly register their running objects.

AnObj := GetObject {"d:\flute\example.xls", "Excel.Chart"};

Attempts to create an object of type Excel.Chart from the specified file.

See Also

CreateObject

GetActiveObjects

ReleaseObject

Releases an object that is no longer required.

  ReleaseObject (AnObj);

  AnObj   An object as returned by GetObject, CreateObject or as returned from an OLE Automation method call that returns objects.

Note:

Just because the object is released does not mean that the application that provided the object will be closed.

GetActiveObjects

Returns an array of the names (the Monikers in OLE parlance) of the running objects. Running or active objects are the objects that already exist on the system in a executing state.

  RunningObjects := GetActiveObjects;

A typical return from this function is:

  { "{00020841-0000-0000-c000-000000000046}", "Book1"}

This indicates that there are two objects running, "Book1" and another object described by its ID number.

Notes

The function only works if the objects register themselves correctly with the system. It is quite common for objects to fail to register or to register incorrectly.

If the object is a file, for example if the array contained an item "d:\example.xls", you can get access to this object using the <u>GetObject</u> function (e.g. GetObject("d:\example.xls"))

ByRef

Used to indicate that a parameter is passed by reference to OLE Automation.

  ByRef (Var)

The parameter inside the brackets must be a variable or part of a variable. When the method call returns this variable is also used to hold a result.

Collect

Converts a collection object into a normal Flute array.

   Result := Collect (AnObj);

   AnObj   An OLE Automation collection object that contains one or more items.

   Result   A Flute array containing the same items as the original collection.

Notes

A collection is a special kind of object used to group other data together. For example, Excel has a 'Sheets' collection which contains an object for each sheet.

Collect does not release the Collections object, you still have to call ReleaseObject to free the OLE collection when you no longer need it. e.g. ReleaseObject(AnObj). Once the object has been converted to an array, it is usually no longer needed.