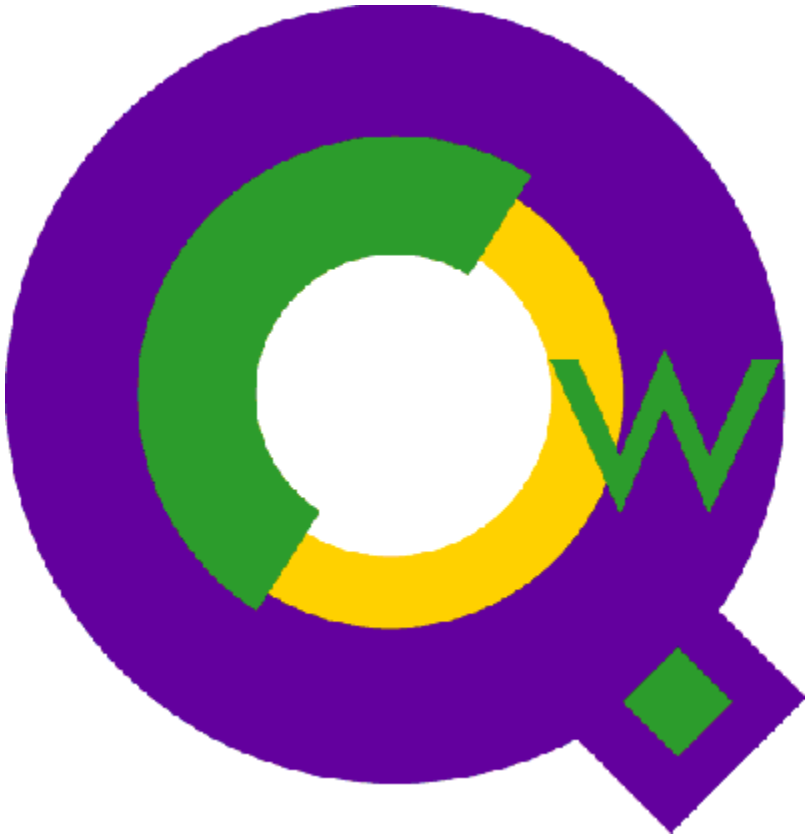
Qedit Scripting Language 5.8.10

User Manual

by Robelle Solutions Technology Inc.



Updated Monday, December 15, 2008

Robelle Solutions Technology Inc.
7360 – 137 Street, Suite 372
Surrey, BC, Canada V3W 1A3
Toll-free: 1.888.robelle
Tel: 604.501.2001
Fax: 604.501.2003
E-mail: support@robelle.com
Web: www.robelle.com

Contents

Introduction

What is Qedit Scripting Language?

Qedit Scripting Language (QSL) is a programming language that you can use to control Qedit for Windows objects. You can write programs in QSL to:

- automate repetitive file manipulation tasks
- perform complex file edits under program control
- automate Qedit's file handling
- add new commands to Qedit.
- customize Qedit for your environment.

You can use one of the scripts supplied by Robelle or you can write your own scripts. Scripts can be automatically loaded, ready to be executed, they can be loaded as you need them or you can simply execute them one by one. You can also write scripts and make them appear as commands on Qedit's **Script** menu.

Who Can Use QSL?

Anyone who has a need for additional functionality to become more productive with Qedit for Windows can use QSL. Enthusiastic end-users could probably find their way in and learn to write simple QSL scripts. However, because it is a programming language, people with programming knowledge such as Visual Basic application developers, would probably be more at ease writing QSL scripts. They could write them and make them available to other users.

How Do I Write a Script?

Any local or host file can be a script. There is nothing special about them. If you have a file that contains some text, you can try to run it as a script. Of course, if the file is not a script, you will likely get all kinds of compile errors.

If you want to start a new script, simply create a new file and start typing QSL statements.

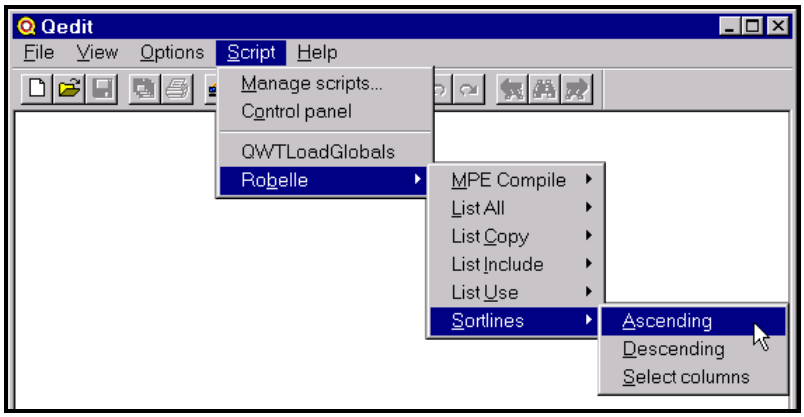
How Do I Run a Script?

Run tool on the Script Control dialog box.

There are a number of ways you can execute a script. All the commands you need are on the **Script** menu. If you want to execute the active document, you can use the **Run** command on the menu or bring up the **Script control** dialog box and use the **Run** tool.

Run tool on the Script Control dialog box.

If the script has been pre-loaded, its name should appear on the **Script** menu. The example below shows how you can access QSL scripts we provide as part of the **Robelle** submenu: **MPECompile**, **ListAll**, **ListCopy**, **ListInclude**, **ListUse** and **Sortlines**. Notice that all these scripts are also a submenus. This means they all have commands at a lower level. They do not have to be. **ListCopy** could have been a command in its own right.



Script menu with pre-loaded scripts

Scripts can be loaded automatically when Qedit for Windows starts up. You simply need to put these scripts in pre-assigned directories on your PC. Once they are loaded and if they contain `On command` statements, they become Qedit commands.

If the script you want is not loaded automatically, you can also use the **Manage scripts** command to display the **Scripts** dialog box. From there, you can add a script to the list. If they contain `On command` statements, next time you display the **Script** menu, the added script should appear in the list. If they do not contain `On command` statements, the subroutines inside the script become available to other scripts.

You can also run scripts indirectly using the `Invoke()` statement or by calling other scripts' methods.

What's New in This Version

The current version of Qedit Scripting Language is 5.8.10. There has been no changes to the language since 5.0.10.

Highlights in Version 5.8.10

Starting with version 5.8.10, the Qedit Scripting Language documentation (this manual) is available in HTML Help formats. The documentation is still available in Windows Help (winhelp) format.

For a history of development highlights, see "Appendix A - Earlier Highlights" on page 173.

QSL Foundations

Overview of Qedit Scripting Language

This chapter describes the basics of QSL: the language syntax and the structure of QSL programs.

Writing a Script

Qedit can execute any ordinary text file as a script. It does not matter whether the script is local or on a server, and you do not need a separate script editor application. Let's see how easy it is to create and execute a script:

1. Select the **File** menu, point to **New**, and click **Local**
2. Type in

```
result = dialog("This is my first script!");
```

3. Select the **Run** command on the **Script** menu.

While the simple script above may not seem very useful, it illustrates several QSL characteristics.

First, QSL consists of statements terminated by semicolons. The language is not case-sensitive, except in quoted strings where case is preserved. Spaces and line breaks may be used freely for readability. A QSL script may be a stand-alone series of statements, or may be part of a larger system of subprograms. This is covered in more detail later.

Saving a Script

If you have written a script and you want to preserve it for future use, simply write it to disk using Qedit for Windows' **Save** or **Save As** commands. The file can reside on the local PC or any of the connections available. For PC files and UNIX hosts, you can specify a file extension to easily identify them. We suggest you use a *.qsl* extension. However, Qedit does not enforce this so you can pick your own extension.

If you decide to create private scripts, you have to use the **Save compiled script** command of the **Script** menu. Private scripts can only be saved as local files. These files must have a *.qsc* extension. Because these are saved in compiled form, they can not be read nor modified by other users. This is a protection when you have to

distribute scripts and want to ensure the original source can not be tampered with. You should also look at the

QSL Language Elements

The examples illustrating QSL language elements have been kept as simple as possible, but you may want to refer to the complete description of QSL statements in "Reference" on page 87 while reading this section.

Statements

QSL scripts consist of one or more statements contained in a text file. The section called "Reference" on page 87 lists all QSL statement types. QSL includes assignment statements, as well as statements for program control, program structure, and debugging.

Statements are generally separated by semicolons. Hitting the ENTER key does not indicate the end of a statement. This means that a single statement can span multiple lines without causing a compile error.

There are instances where QSL assumes a semicolon. Some statements come in pairs to indicate the start and end of a logic block. These statements are:

IF and ENDIF

REPEAT and ENDREPEAT

SUB and ENDSUB

ON and ENDON

TRY and ENDTRY

Because `ENDIF`, `ENDREPEAT`, `ENDSUB`, `ENDON` and `ENDTRY` are sort of terminators themselves, a semicolon is optional.

Since the semicolon explicitly indicates the end of a statement, you can also type multiple statements on a single line as in:

```
stringmsg = "No selected text"; result = dialog(stringmsg);
```

Although this is perfectly legal, this coding style reduces readability a lot.

Basically, all language elements except strings enclosed in quotes, are not case sensitive. This means that `MyVariable` and `MYVARIABLE` are really pointing at the same data. Similarly, `dialog` and `DiaLog` both refer to Qedit's `Dialog()` function .

Character Set

QSL programs can be written using any supported language character set. Strings are stored in the Windows character set and are translated when entered into documents or sent to remote hosts.

Syntax Elements

Keywords

Keywords are reserved words that have special meaning to the QSL compiler. QSL does not prevent you from using a keyword as an identifier. However, doing so can

be confusing for the developer. Keywords contain only ASCII alphabetic characters. Here is the current list of reserved words.

And	For	Property	To
By	From	Recover	Try
Else	If	Repeat	Until
Endif	In	Return	While
Endrepeat	Name	Stop	With
Endsub	Not	Sub	Xor
Endtry	On	Then	
Error	Or	Timeout	

Identifiers

Identifiers must start with an alphabetic character and may contain any sequence of alphanumeric characters and underscores. Note that hyphens are **not** allowed. Identifier names can have up to 255 characters.

Comments

QSL scripts may contain comments for documentation and readability. The language supports both line-terminating comments and in-line comments. A pair of hyphens, "--", in a line of a script causes the remainder of the line to be treated as a comment, unless the double hyphen is inside a string constant. For example,

```
string = "Hello World!"; -- Here is a comment
string = "This is not a -- comment!";
```

In addition, Qedit ignores text between pairs of angle brackets, "<<" and ">>", which can be used for intra-line and multi-line comments.

```
area <<Inline comment>> = height * width;

<<
*****
this is a multi-line comment
*****
>>
```

Constants and Values

QSL supports different types of data. Any variable can take on values of different data types over time. It acquires the type of whatever is assigned to it. Qedit takes care of type coercion transparently. However, there are cases where you might have correctly assign a value to be certain the variable ends up with the correct type.

Data Types

You do not have to declare variables ahead of time. Qedit declares a variable when you first reference it. The assignment operator is an equal sign "=". The data type is based on the assigned value. Qedit supports 6 different data types:

Type	Description
Undefined	Variable does not have a type
Integer	Integer value, no decimals

Float	64-bit IEEE floating point
String	Windows character string
Record	Data structure or list
Object	Qedit objects

You can check a variable's data type using the `Typeof()` function. In most cases, you do not have to worry about data type conversion. Qedit changes data type dynamically. If you wish to explicitly change the type of a variable, you can use one of the functions to do type coercion. The `string()` function is probably the one you will use most often.

Numbers

Numbers are sequences of digits, which may optionally contain a decimal point. QSL allows only decimal numbers.

Named Constants

We have assigned names to constants that were frequently used. `True` is a named constant and can be used to check for the value 1. If used in a condition criterion, it also designates any non-zero value. `False` is the named constant for the value 0.

String Constants

A string constant is introduced by either a single quote (apostrophe) or a double quote. The same delimiter must be used to terminate the character string. If it is necessary to include the string delimiter within the string, it should be doubled:

```
"""What?"" she exclaimed."
```

or alternatively, the string should be started with a different delimiter as in

```
'"What?" she exclaimed.'
```

String constants may include tabs and nonprintable characters by using the special escape sequences shown below. The escape sequences allow you to place any possible character value in a character string. An escape sequence is introduced by a pair of question marks.

Sequence	Meaning
??a	Alert character, decimal value 7
??e	Escape character, binary value 27
??n	Line feed, decimal value 10
??r	Carriage return, decimal value 13
??t	Tab, decimal value 9
??$*dd*	A character with the hexadecimal value *dd*
??%*ddd*	A character with the octal value *ddd*
??*ddd*	A character with the decimal value *ddd*
??^*c*	The character obtained by using the control key with character c. For example, the sequence ??^y yields a decimal value of 25. The case of c does not affect the binary value of the sequence

??*c*	For any character not described above, the sequence ??*c* is equivalent to that character. This can be used to insert a quotation delimiter

Escape sequences containing numeric values are terminated by any character that is not valid for that type of number. For example, the octal escape sequence `??%59` produces a single character with a decimal value of 5, because 9 is not a legal octal character. Terminating a numeric escape sequence in this manner is not considered an error.

Decimal escape sequences always represent characters in the Windows character set even if the script resides on a non-Windows connection.

To code a string with two or more question marks in a row, all but the first or last must be escaped; for example, `????` yields two question marks.

Extended Characters

String constants may contain extended characters, that is characters not in the basic ASCII character set. When a script resides on a local disc drive or is executed from a new local file, extended characters are assumed to be in the Windows character set. When a script resides on a host computer, Qedit translates character strings from the host computer's character set to the Windows character set. This implies that any character used in a remote script must have a representation in the Windows character set. If a script contains characters that can't be represented in Windows, the results are undefined.

Records and Lists

Many operations and functions in QSL use records to specify input and to contain output. Records are lists of values which may optionally be named. Constant lists are written by enclosing them in braces "{" and "}". Individual elements in the list are separated by commas.

A list can be made up of constant values only:

```
myList = {1, 4, 9, 16, 25}    -- A simple list of 5 elements
```

Elements in this list can only be referred to using a subscript or using the `Repeat...In` *`List`* statement. Subscripts in QSL normally start at 1 and are enclosed in square brackets "[" and "]".

```
newVar = myList[2];        -- returns the value 4
```

A list can also be constructed with named values. The name and its corresponding value are separated by a colon. The general syntax to define a named element is:

{*name1*: *value1, name2*: *value2*, ...}

For example,

```
myList = {connection: "System 5", filename: "FOO"}
       -- A list of named values
myList2 = {start: {line: 5, column: 1}, end: {line: 15, column: 12} }
       -- A List with nested lists
```

If an element in a record has a name, QSL statements can refer to it by name. A period is used to identify named elements within records.

```
myList = {connection: "Omaha", filename: "TXMAY"};
myConn = myList.connection;   -- Returns "Omaha"
myVar = myList2.start.line;   -- Returns 5
```

Otherwise, the element can only be referred to by its subscript as in simple lists.

```
myconn = myList[1];           -- Also returns "Omaha"
myList.connection = "Paris";  -- Sets first element
myList[1] = "Paris";          -- Also sets first
myVar = myList2[2][2];        -- Returns 12. Equivalent to End.Column
```

Initializing Records And Lists

In all cases, when you initialize a record, the element list must include constants only. It can not contain variable names as in:

```
LineNumber = 1;
ColumnNumber = 20;
mySelection = { line: LineNumber, column: ColumnNumber };   -- Invalid
```

This is the correct way to do it:

```
mySelection = {};          -- Specifies that this is a record
mySelection.line = 1;      -- Initializes first named element
mySelection.column = 20;   -- Initializes second named element
```

Empty records are allowed; they are represented by an empty pair of braces. In addition to numbers, strings and other records, records may contain objects. A record can contain elements of different types.

```
myList = {};           -- Create an empty list
myList[1] = 1;         -- First element is numeric
myList["TWO"] = "two"; -- Second element is a named string value
coordinates = { line: 1, column: 20 };
myList[3] = coordinates;  -- Third element is a nested list
result = dialog(string(myList));
```

The dialog box would show:

```
{ 1, TWO: "two", {line: 1, column: 20}}
```

Using Nested Records

As shown in "Initializing Records And Lists" on page 10, a record can contain elements of any type including other records. However, the record structure will be different depending on how you add record elements. Let's say you have 2 record variables:

```
MainRecord = {};   -- Main empty record
NestedRecord = { item1: 1, item2: 2 };
```

The concatenation operator "+" adds elements to a record. It does not matter if there are elements with the same name already in the record. Thus, after executing the following statement,

```
MainRecord = MainRecord + NestedRecord;
```

MainRecord contains:

```
{ item1: 1, item2: 2 }
```

Executing the same statement a second time would set **MainRecord** to:

```
{ item1: 1, item2: 2, item1: 1, item2: 2 }
```

There are 2 ways to create nested records. The first option is to qualify the target record with a subscript.

```
MainRecord[1] = NestedRecord;
MainRecord[2] = NestedRecord;
```

MainRecord then contains:

```
{ {item1: 1, item2: 2}, {item1: 1, item2: 2} }
```

Notice the main record now contains 2 record elements which contain 2 elements each. To reference individual elements, you would have to use subscripts as in:

```
newVar = MainRecord[2].item2;
```

When using subscripts, you have to ensure there is no gap in the subscript sequence. For example, the following code segment would not create nested records #3 or #4. MainRecord would still contain only 2 nested records.

```
MainRecord[1] = NestedRecord;
MainRecord[2] = NestedRecord;
MainRecord[4] = NestedRecord; -- Invalid. Subscript 3 is missing.
```

Note that no error is reported in this case. The information is simply not stored.

The second option is to assign a name to nested records instead of subscripts.

```
MainRecord["Nest1"] = NestedRecord;
MainRecord["Nest2"] = NestedRecord;
```

MainRecord would then contain:

```
{ Nest1: {item1: 1, item2: 2}, Nest2: {item1: 1, item2:
2} }
```

The names can be anything you want and you do not have to worry about gaps. However, if a name already exists, its value is going to be replaced.

Objects

Most of the things that you can manipulate with QSL take the form of objects. An object can have both data (**properties**) and code (**methods**) associated with it. For example, when you open a new file with QSL, the function which creates the file returns a file object:

```
resultFile = newfile();              -- Returns a file object
resultFile.Insert("First line.");    -- Add text
```

Objects have properties that can be examined and set, as well as methods that can be called. The example above calls the file object's **Insert** method to insert text. The section called "Reference" on page 87 provides a complete list of the methods and properties for all QSL objects.

Scripts themselves are objects and can have properties and methods. These become important when you instruct Qedit to load scripts and make them available to other scripts.

Expressions

Qedit expressions can involve not only numeric quantities but strings, lists and records as well. Some operators apply only to certain data types, but others can be used with any Qedit data type.

Predicate results

Some operations, such as comparisons, are *predicates*: they apply a test and return a boolean, `True` or `False`, value. In a Qedit script, an expression returning a value of `TRUE` yields a numeric value of 1, while a `FALSE` result yields a numeric value of 0. Statements, such as `IF` and `REPEAT` that test for boolean values, can test any numeric expression; a non-zero value is equivalent to `TRUE`. However, it is good practice to make comparisons to 0 explicit:

```
if lineCount <> 0 then     -- preferred
if lineCount then          -- works, but less clear
```

Arithmetic Expressions

Number Format

Qedit manipulates all numbers using the highest-precision data type available on the client platform. For most Qedit clients, this is a 64-bit IEEE real number. Qedit number objects provide an `INTEGER` function in order to extract only the integer part of a number, when you need to do so explicitly. Qedit automatically converts numbers to integers by rounding when a number is used in a context requiring an integer.

Arithmetic Operations

In addition to the standard arithmetic operators, you can take advantage of Qedit's object orientation by manipulating various properties of numeric objects.

Operator	Operation
Plus sign "+"	Addition
Hyphen "-"	Subtraction
Asterisk "*"	Multiplication
Slash "/"	Division
Two asterisks "**"	Exponentiation

Arithmetic expressions are evaluated from left to right. Multiplication and divisions are executed first. Additions and subtractions are done next. Finally, exponentiation is done last. You can use parentheses to change the order of precedence.

Qedit also provides error handling for numeric expressions, provided that the client platform offers suitable numeric library facilities. Numeric underflow causes Qedit to substitute a zero value, while division by zero causes script execution to halt unless a `RECOVER` clause is in effect. A numeric overflow returns, once converted to a string, a value of *`1.#INF`*.

Boolean Expressions

Boolean expressions consist of boolean or numeric values and the operations `AND`, `OR` , `XOR` and `NOT`. The construction

`NOT` *`expression`*

negates the boolean value of the expression. The expression

expr1 AND *expr2*

is true if and only if both *expr1* and *expr2* are true, while the expression

expr1 OR *expr2*

is true if either *expr1* or *expr2* is true. Qedit also provides an *exclusive or* operation, so that

expr1 XOR *expr2*

is true if only one of *expr1* and *expr2* is true.

QSL uses short-circuit evaluation when checking conditions. This means that in complex conditions, control is transferred as soon as Qedit is able to determine the outcome. It also means that not all conditions will be checked every time. In the case of an AND condition, if the first expression is false, Qedit knows the overall condition will be false, no matter what the other expression evaluates to. In the case of an OR condition, if the first expression is true, Qedit knows the overall condition will be true, no matter what the other expression evaluates to.

String and List Operations

Qedit provides only one string operation, concatenation, indicated by the plus sign "+" operator. Concatenation can also be used to join two lists, or to add an item to a list. Strings and lists both support many compound item references, providing a rich variety of string manipulations.

```
myString = "ab";   -- Initialize string variable
myString = myList + "CD"; -- Appends new string, result is "abCD"

myList = { 1, 2 }; -- Initialize list variable
myList = myList + { 3, 4 };   -- Appends new elements
       -- Result is  { 1, 2, 3, 4 }
```

Compound Item References

Compound objects such as strings and lists respond to subset selection messages, which permits the script to reference individual parts of the compound item. Several other messages perform simple searches. Other kinds of objects may also respond to these messages where appropriate.

Subscripting

A QSL script uses subscripts to select portions of compound items such as strings or records. Subscripts have the form

[*element*]

or

[*startelement*:*endelement*]

where *value* is a numeric expression giving the ordinal number of the element to extract. Subscripts start at 1. If you specify only one element number, Qedit extracts only one character or list element. If you specify a start and end element, Qedit extracts all the elements between the 2 numbers including the start and end elements.

```
filename[2]           -- extract the second character
filename[1:8]         -- Characters 1 through 8 as a string
connectionList[1]     -- returns one item
connectionList[2:5]   -- returns a list of elements 2 to 5 inclusive
```

The value can also be a string to select a field from a list:

```
editTarget = {name: "S979", file: "EXTR001"} ;
fileName = editTarget["file"];    -- Gets EXTR001
```

Selection on lists

When you apply a range selection to a list, Qedit returns another list containing only those items in the range. For example:

```
{"Peter", "John", "Paul", "Mary", "Rafael"} [2:4]
```

yields the list

```
{"John", "Paul", "Mary"}
```

which contains elements 2, 3 and 4 of the original list.

Finding an Index

A script that needs to find the position of an element in a list or string can use the `POS` function. `POS` returns the index, relative to 1, of a substring or list item, or 0 if the substring or list item isn't found:

```
POS("Quick brown fox", "Quick")          -- Returns 1
POS("Quick brown fox", "fox")            -- returns 13
POS({"Quick", "brown", "fox"} , "fox")   -- returns 3
POS({2, 3, 4} , 5)                       -- returns 0
```

Comparison Operators

Qedit provides the usual collection of comparison operators. Most of the comparisons are meaningful only on numbers and strings, but you can test any two objects for equality.

Operator	Applies to	True when
Greater than ">"	Numbers, strings	Left operand is greater than right operand
Greater than or equal to ">="	Numbers, strings	Left operand is greater than or the same as right operand
Less than "<"	Numbers, strings	Left operand is less than right operand
Less than or equal to "<="	Numbers, strings	Left operand is less than or the same as right operand
Equal to "="	Any pair of items of similar type	Left and right operands are the same
Not equal to "<>	Any pair of items of similar type	Left and right operands are not the same

Type Coercion

Sometimes, it's convenient to change the representation of a value during script execution. A common example is coercing some non-string value to a string in order to insert it in a file or use it in a string comparison. Qedit uses the type name as a function to introduce a type coercion:

`type(`*`expression`*`)`

For example, to convert a number to a string, use

```
string(5)       -- returns "5"
```

Any object can be coerced to a list, in which case it becomes a one-element list with that object as the only element.

```
localfile = open("C:\personal\diary.txt");
locallist = {} + localfile;
```

Script Structure

All but the very simplest scripts are usually broken up into subroutines. Subroutines are sequences of statements that have a name, can have local variables, and can return a value. In addition, any script intended for other than one-time use probably has a name, and may have properties as well.

Script Name

Every script has a name by which commands and other scripts can refer to it. If you don't specify a script name, Qedit uses the name of the script's source file instead. Specify the script name by using the **Name** statement:

```
name Utilities;
```

Script names follow the same rules as other QSL identifiers: they must begin with an ASCII letter, and consist of ASCII letters, digits, and underscores.

Properties and Methods

Since they are objects, scripts can have properties and methods. A script property is a variable that is global not only within the script, but outside the script as well. Specify properties using the **Property** statement:

```
property resultDirectory = "c:";
```

A property may have an initial value, as shown above. The initial value must be a constant. Each script may have any number of properties.

Methods associated with a script are all the subroutines declared within it.

Subroutines and Handlers

Subroutines are sequences of statements that can be referred to by name, and that can return a value. A handler is a special kind of subroutine that Qedit runs when certain events occur.

```
sub Quadratic(a, b, c)
-- Compute roots of equation
--    a*x**2 + b*x + c = 0
-- Return a list of all real roots

returnValue = {} ; -- Variable for return value is a list

discrim = b**2 - 4 * a * c;
if discrim  = 0 then
    returnValue = returnValue + 0;
else
    returnValue = returnValue + ((-b + sqrt(discrim)) / (2 * a);
    returnValue = returnValue + ((-b - sqrt(discrim)) / (2 * a);
endif

return returnvalue;

endsub
```

This subroutine returns a list that may be empty, or may contain one or two elements. The caller must be prepared to deal with each of these cases.

A handler is also a subroutine, but is introduced with the **On** statement instead of a **Sub** statement:

```
on command "Update Change Log"
    whichFile = Currentfile();
    whichFile.Insert
        (text: {"Last changed " + string(datetime()) + " by TLA", , ""}
        at: {Line: 1, Column: 1} );
endon
```

This handler is added to Qedit's **Script** menu and appears as "`Update Change Log`".

Subroutines may be located anywhere in a QSL script, though they are traditionally placed together at the beginning or the end of the script. If you choose to intersperse subroutines and main script code, Qedit collects all the main script code together and treats it as if it were a single block of code.

All subroutines in a script are visible from other scripts as script methods. External scripts may invoke those methods by using the external script's name as the name of the object. For example if a script named `Utilities` has a subroutine named `Quadratic`, a second script can access `Quadratic` by calling `Utilities.Quadratic(1, 2, 3)`.

Controlling the Flow

Qedit normally executes script statements linearly, from beginning to end. However, several script statements can change this behavior.

Conditionals

Scripts use the `IF/THEN/ELSE/ENDIF` construct to execute certain statements conditionally. For example, the following subroutine opens a file that is to be used as a change log, or creates a new one if none exists:

```
if result.EnteredText = "START" then
    changeLog = newfile(connection: whichConnection);
    changeLog.Insert(list("Change log started on " +
        string(datetime), ""));
    changeLog.SaveAs("CHANGES.log");
else
    changeLog = open(connection: whichConnection,
        filename: "CHANGES.log");
endif
```

Every `IF` statement must have a corresponding `ENDIF`.

Iteration

Qedit's `REPEAT` statement causes a statement or a group of statements to be executed repeatedly until some condition is met, or until a list is exhausted. The group of statements to be repeated must end with an `ENDREPEAT` statement. Here are some of the possible syntax for `REPEAT`.

REPEAT WHILE Construct

Syntax:

REPEAT WHILE *condition*

The block of statements is repeated while the condition is met. If the condition is false to start with, the block is not executed.

```
resultlist = {};   -- Create a record variable

findresult = localfile.find(regexp: "G...n");
repeat while findresult   -- True is assumed
    resultlist = resultlist + localfile.getselectedtext();
        -- Add the string to the list
    findresult = localfile.find(regexp: "G...n");
        -- Search for next occurrence
endrepeat

result = dialog("List of matches:" + string(resultlist));
```

REPEAT UNTIL Construct

Syntax:

`REPEAT UNTIL` *condition*

The block of statements is repeated until the condition is met. The condition can be a simple or complex expression. The block always executes at least once.

```
resultlist = {};   -- Create a record variable

repeat until findresult = false   -- True is assumed
    findresult = localfile.find(regexp: "G...n");
        -- Search for next occurrence
    if findresult then
        resultlist = resultlist + localfile.getselectedtext();
            -- Add the string to the list
    endif
endrepeat

result = dialog("List of matches:" + string(resultlist));
```

REPEAT FOR Construct With Numbers

Syntax:

REPEAT FOR *variable* FROM *start* TO *end*

REPEAT FOR *variable* FROM *start* TO *end* BY *step*

Variable must be a name. The variable does not have to exist already. *Start*, *End* and *Step* can numeric constants, numeric expressions or predefined numeric variables.

Variable is set to the initial value specified by *Start*. The block of statements is executed and *Variable* is incremented by the value of *Step*. If *Step* is not specified, a value of 1 is used. A step value of 0 causes a run-time error.

If *Start* is less than *End*, the block is repeated until the value in *Variable* is greater than *End*. Thus, the value of *Step* should be positive. If *Start* is greater than *End*, the block is repeated until the value in *Variable* is less than *End*. Thus, the value of *Step* should be negative.

```
localfile = newfile();
lines = {};

repeat for linenumber from 1 to 10
    lines = lines + ("Line #" + string(linenumber));
endrepeat

localfile.insert(lines);
```

This script creates a new local file with 10 lines formatted as:

```
Line #1
Line #2
...
Line #10
```

REPEAT FOR Construct With a Record

Syntax:

REPEAT FOR *variable* IN *list*

The block of statements repeats for each item in the *list*. *Variable* is set to each list element in turn.

```
localfile.select(startline: 1, startcolumn: 1);  -- Start of file

findlist = {1, 3, 5, 7, 9};   -- List of values to search for

repeat for linenum in findlist
    localfile.find(string: string(linenum));
    result = dialog("Found " + string(linenum) + " at " +
                    string(localfile.selection));
endrepeat
```

This script searches for the odd numbers contained in the list, one by one, and displays a message if the string is found.

Exception Handlers

When script execution results in an error, Qedit normally stops executing the script and displays an error dialog. A script can trap the error, however, and begin executing special code to handle it. In addition, when a script is attached to an object, it receives events from the user interface (and from other scripts) and can execute groups of statements in response.

To "protect" a group of statements from halting script execution in the event of an error, use a TRY/RECOVER block. Qedit executes statements from the TRY block in the normal fashion until an error occurs, at which point script execution continues

with the statements in the RECOVER block. If no error occurs during the TRY, the statements in the RECOVER block are ignored:

```
try
    file = open(connection: connection, filename: filename);
recover
    result = dialog("Sorry, unable to open file " + filename +
                " on connection " + connection);
    stop;
endtry
```

In this example, Qedit tries to open the specified host file. If, for any reason, the open() does not succeed, an error message is displayed using the Dialog() function and the script execution is interrupted.

Catching specific errors

In the example above, the statements in the RECOVER block are executed for any kind of error. If you want to check for specific errors, you can specify a variable name on the RECOVER statement. The variable is then created as a record and contains information about the error. You can check the contents of that variable in the RECOVER block and make decisions based on that.

Error Recovery Scope

TRY blocks have dynamic scope rather than lexical scope. If a script is being executed as the result of a subroutine call from another script, and that call is within a TRY block, Qedit transfers control to the calling script's RECOVER block in the event of an error not handled in the called script. Normally, this is exactly what is required, but it means that if you write a script that is intended to be called as a subroutine, you should include enough error handling to clean up any loose ends in the subroutine before passing the error on to Qedit.

Subroutines

Qedit scripts can contain subroutines. They can be declared anywhere within a script. A subroutine can even be declared after calls to it. However, for clarity and readability, it is recommended that they all be declared either at the beginning or at the end.

In the current implementation, however, the only exception to this would be when there are global variables. These variables have to be declared or initialized before the subroutines that use them. For example, the following script fails with a run-time error, "unitialized value encountered", on the writelog() call.

```
sub testSub()
    writelog(GlobalText);
endsub

GlobalText = "Global text";
testSub();
```

To fix this, the script should be coded this way:

```
GlobalText = "Global text";
testSub();

sub testSub()
    writelog(GlobalText);
endsub
```

Subroutines are in effect event handlers for the script, and a subroutine call is an event. A subroutine can also be called from other scripts. Subroutines become methods for the script.

Parameters

As in C or Pascal, subroutines can contain local variables and can receive parameters passed from their caller. The parameter list, if any, is enclosed within parentheses. If a subroutine does not require any parameters, the parentheses are also optional. The parameter list only contains names. Parameters do not have data types. They take on the type of the original variables they represent.

```
sub QualifyFilename(theName)
    groupName = "LIBS";
    accountName = "SOURCE";

    return theName + "." + groupName + "." + accountName;
endsub

sub ErrorSub
    result = dialog("Invalid operation encountered!");
endsub
```

In this example, `theName` is a parameter passed by the calling statement to the `QualifyFilename` subroutine. On the other hand, the `ErrorSub` subroutine does not expect any parameters.

Parameters are passed by value. This means that the content of the original variable can not be changed. If you need to change the value of a parameter, you should use the `Return` statement. If you need to return multiple values, specify a record as the returned value.

Qedit does not perform type checking on parameters. This means that the caller could pass a numeric value while the subroutine expects a string. To avoid possible problems, you can use the `Typeof()` function to validate the parameter types. This also allows for parameter overloading. The same subroutine could perform different operations if the parameter is numeric or a string.

Calling a Subroutine

QSL uses the parentheses to differentiate a subroutine name from a variable. When calling a subroutine with no required parameters, the parentheses are optional. These subroutines can be called with an empty parameter list as in:

```
MySubroutine();
```

Alternatively, you can use the `call` statement to explicitly execute a subroutine. In this case, the parentheses do not have to specified for a subroutine with no parameters. Keep in mind that using the `call` statement does not allow the capture of a return value.

```
call MySubroutine; -- Valid subroutine call
returnValue = call MySubroutine;  -- Invalid. Can not assign return.
```

Parameters can be passed by position or by name. You can not mix these options on the same call. That is, if you want to use positional parameters, you cannot have

named parameters. If you use named parameters, you can not use positional parameters.

Positional parameter values are matched from left to right with the names in the subroutine declaration. For example,

```
sub testsub(theFile, theLine, theColumn);

    returnValue = false;

    if typeof(theFile) = qedit.typeundefined then
        result = dialog("You MUST pass a file object to TESTSUB");
        return returnValue;
    endif

    returnValue = true;

    if typeof(theLine) = qedit.typeundefined then
        -- theLine parameter is missing. Use default value.
        theLine = 1;
    endif

    if typeof(theColumn) = qedit.typeundefined then
        -- theColumn parameter is missing. Use default value.
        theColumn = 1;
    endif

    -- subroutine logic
    localFile = open(theFile);
    localFile.select(startline: theLine, startcolumn: theColumn);

    return returnValue;

endsub

testsub("c:\personal\diary.txt", 10, 20);
```

This call to **testsub** passes three parameters: "c:\personal\diary.txt" is assigned to **theFile**, the value 10 is assigned to **theLine** and 20 to **theColumn**.

If you use positional parameters, you can not omit parameters in the middle of the parameter list. However, you can omit parameters at the end of the list. Here are some sample calls:

```
testsub("c:\personal\diary.txt");
    -- Valid. theLine and theColumn parameters are undefined

testsub("c:\personal\diary.txt", 10);
    -- Valid. Only theColumn is undefined

testsub("c:\personal\diary.txt", , 20);
    -- Invalid. theLine must be specified
```

If you choose to omit positional parameters, they will be undefined. This will cause a runtime error on the first statement that references the parameter name. The subroutine should be coded to handle these situations. As shown in the sample subroutine above, you can use the `Typeof()` function to determine the presence of optional parameters.

If you want get around these restrictions, you can use a named parameter list. You would qualify each value with the name of the corresponding parameter. This feature allows you to omit parameter values anywhere in the list. If you omit parameters, you still have to write code to handle these situations. It also allows you to specify the parameters in any order. Here are some examples:

```
testsub(theFile: "c:\personal\diary.txt");
    -- theLine and theColumn parameters are undefined

testsub(theFile: "c:\personal\diary.txt", theLine: 10);
    -- Only theColumn is undefined

testsub(theFile: "c:\personal\diary.txt", theColumn: 20);
    -- theLine is undefined

testsub(theColumn: 20, theFile: "c:\personal\diary.txt", theLine: 10);
    -- All parameters are present but in a different order
```

If you need to allow a variable number of parameter values, you can also define the subroutine with a single parameter and pass a `record` structure. The subroutine should then be coded to "unpack" the structure.

Return values

Optionally, subroutines can return values to their callers. A call to a subroutine that returns a value might look like this:

```
fullFileName = QualifyFilename("IOSUBS");
```

After control is returned from the `QualifyFilename` subroutine, the variable `fullFileName` will contain "IOSUBS.LIBS.SOURCE".

If the call statement does not expect a return value but the subroutine has a `return` statement, the value is simply discarded.

If the call statement expects a return value but the subroutine does not return one, the receiving variable is of type `Undefined` and does not contain any value.

Local variables

Any new variable defined within the body of a subroutine are local to that subroutine, meaning that their values aren't visible outside the `SUB/ENDSUB` block. Their initial value is undefined upon entry in the subroutine and their value becomes undefined when the subroutine returns.

Global Variables

Variables that are declared outside subroutine blocks are global to the script. This means they are recognized in all internal subroutines. Once declared, global variables retain When global variables are declared with the `Property` keyword, their values are exported to external scripts.

Recursion

Script subroutines can call themselves recursively. This may be useful for processing lists (or nested lists) in subroutine parameters. Recursive calls, like normal calls, get a complete new set of local variables. Recursion depth is limited to around 2000, depending on other characteristics of the script.

Timeouts

Since Qedit clients communicate with servers to get part of their work done, the client must wait for the result from the server to know whether a particular request succeeded or failed. And while in normal operation there's a thinking person in charge of Qedit's actions, a script can't make complicated decisions. So while the Qedit is operating under control of a script, it sends only one command at a time, and waits for the response from the server before continuing.

By default, Qedit will wait 60 seconds for a server to respond before deciding that no response is forthcoming, and stopping the script with a timeout error (-11). You can change that for specific statements by using a `WITH TIMEOUT` block around statements that you know might take a long time, such as searching a very large file:

```
with timeout 300
    logFile.Find("SNEAKY.PAYROLL")
endwith
```

If you need to do something special in the event of a timeout, be sure to use a `TRY/ON ERROR` block to trap the timeout error.

Stop button on the Script Control Panel

You can also use `WITH TIMEOUT 0` or `WITH NO TIMEOUT` if you want Qedit to wait forever. If you become impatient before forever expires, you can stop execution of the script using the **Stop** button on the **Script control** panel.

Stop button on the Script Control Panel

Stopping Execution

The `stop` statement halts execution of the current script. Control is returned to the calling script or to Qedit depending on how the script was invoked.

The Script Environment

The collection of scripts loaded at any one time, either via the **Manage Scripts** dialog box or from another script using the `Loadscript()` application method, is an environment which newly-loaded or transient scripts must interact with. This means that you can put commonly-used utility routines into scripts by themselves, and then refer to those utility routines in other scripts you write. If you wish to execute a complete script, you can use the `Invoke()` statement.

Methods and Subroutines

When Qedit executes a method call in a script, it searches for the method in a well-defined order:

1. The script containing the method call.
2. Previously-loaded scripts, starting with the most-recently loaded script.
3. The Qedit application object.

This means that scripts you write can override Qedit application methods, and that scripts can contain overrides of routines in earlier scripts. You can always refer to a specific subroutine or method by qualifying the name with the script that you want Qedit to search. The special qualifier `QEDIT` tells Qedit to search the application object as in `Qedit.NewFile()`.

Loading Scripts When Qedit Starts

When Qedit starts up, it automatically loads scripts from predefined locations. First, it loads all scripts from the `AUTOLOAD` subdirectory of the `SYSTEM` directory in the directory containing the Qedit application. These scripts are supplied by Robelle. You should not change the scripts in any of the subdirectories under `SYSTEM`.

Next, Qedit scans the AUTOLOAD subdirectory of the USER directory in the directory containing the Qedit application. These scripts are custom to your site.

Finally, Qedit checks your personal script directory.

Running Scripts from the Command Line

You can also ask Qedit to run a script from the MS-DOS command line using different arguments.

Execute Only Argument

If you use the -r argument, Qedit executes the script specified on the command line after it has finished loading all of the scripts in the two default script directories, but before it opens any text files specified in the command. If the command-line script doesn't call the Qedit EXIT() method, Qedit remains open after the script terminates.

```
C:\QWIN32.EXE -r MYSCRIPT.QSL
```

The "-r" argument and the script name should be the last arguments on the line. If the script name contains embedded spaces e.g. My Script.qsl, the name should be enclosed in quotes as in:

```
C:\QWIN32.EXE -r "My Script.QSL"
```

Execute and Terminate Argument

If you want to run Qedit from the DOS command line, have it execute the script and terminate even if the script does not call the EXIT() method, use "-q" along with the "-r" (execute only) the argument. The "-r" argument and the script name should be the last arguments on the line.

```
C:\QWIN32.EXE -q -r MYSCRIPT.QSL
```

Prevent Autoloading

By default, Qedit loads all the scripts in the **Autoload** directories even if it is run from the DOS command line. If that's not desirable, you can use the "-n" argument. This way, Qedit runs the specified script only as indicated by the other arguments.

```
C:\QWIN32.EXE -n -r MYSCRIPT.QSL
```

Using Scripts to Add Commands

When Qedit loads a script, it scans the script for commands to add to the **Script** menu. These commands are defined by ON COMMAND statements. If the script contains more than one command, Qedit adds a submenu to the **Script** menu and lists all of the commands in the script.

If the script contains an "outer block", executable statements that are not part of a subroutine or a handler, Qedit also adds the script name to the **Script** menu as a command. Choosing this command runs the outer block of the script.

Command names can be the script's name or names associated with On command statements. A submenu name can be the script's name or the name defined in the **Group** parameter of the **Name** statement.

Installing Scripts

Script Libraries

It's fine to write scripts for yourself but it's even nicer to share scripts with others and make them more productive too. In some cases, you might want specialized scripts and their methods to be readily available to other scripts. Qedit is designed to facilitate these type of operations.

Robelle Public Library

Robelle has written a number of scripts already. Most of these scripts are distributed with Qedit for Windows and are stored the **System** directory. However, the script library is continuously enhanced with new ones and existing ones are improved. If you have access to the World Wide Web (WWW), you can download the latest versions directly from our web site. To get a script on your PC, you can copy the script using the clipboard:

1. point your browser to http://www.robelle.com/support/qwin/scripts.
2. browse through the list of available scripts
3. when you have found the one you want, click on the script name to display it
4. hit CTRL+A to select all the text
5. hit CTRL+C to copy the text to the clipboard
6. switch to Qedit for Windows and create a new local file
7. hit CTRL+V to paste the clipboard
8. save the file in the appropriate directory. The script should contain recommendation about it.

If you prefer, you can download the file directly instead of using the clipboard.

1. point your browser to http://www.robelle.com/support/qwin/scripts.
2. browse through the list of available scripts
3. when you have found the one you want, right-click on the script name
4. select the **Save Target as** command from the shortcut menu
5. save the file in the appropriate directory. The script should contain recommendation about it.

Contributed Library

We encourage users and customers to write their own scripts. We are also encouraging people to make their scripts available to other users in the community. These scripts can be of general interest or they can be useful to other people using the same software package. It doesn't really matter. Robelle is ready to setup a contributed script library area on the Web site as a distribution mechanism.

Where Are They?

The Qedit installation procedure creates a number of directories specifically to store scripts. These directories are located under the directory where the Qedit application program file resides. Typically, this would be `\robelle\bin`.

Scripts From Robelle

It is our intention to write scripts to provide additional functionality to Qedit or solutions to common editing tasks. For this purpose, the Qedit installation procedure creates the **System** directory, e.g. `\robelle\bin\system`. This directory is owned by Robelle and you should not change anything in it. We are likely to change the scripts we provide. When you install a new version of Qedit, the content of the **System** directory is going to be refreshed. If you have made changes to it, they will probably be lost.

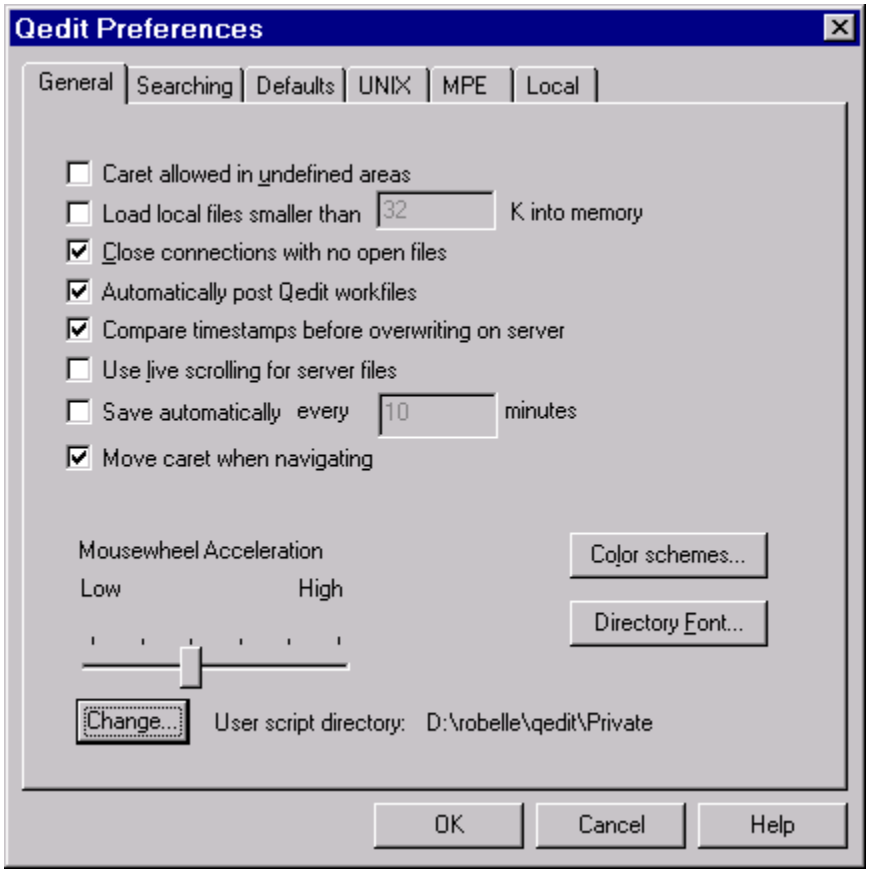
A description of these scripts is provided in "Robelle Script Library" on page 82.

Company-wide Scripts

We expect customers to write their own scripts or to customize scripts that we have distributed. In order for you to distribute your versions, the installation procedure creates a directory called **User** under the install directory e.g. `\robelle\bin`. This is your area. You can make all the changes you want.

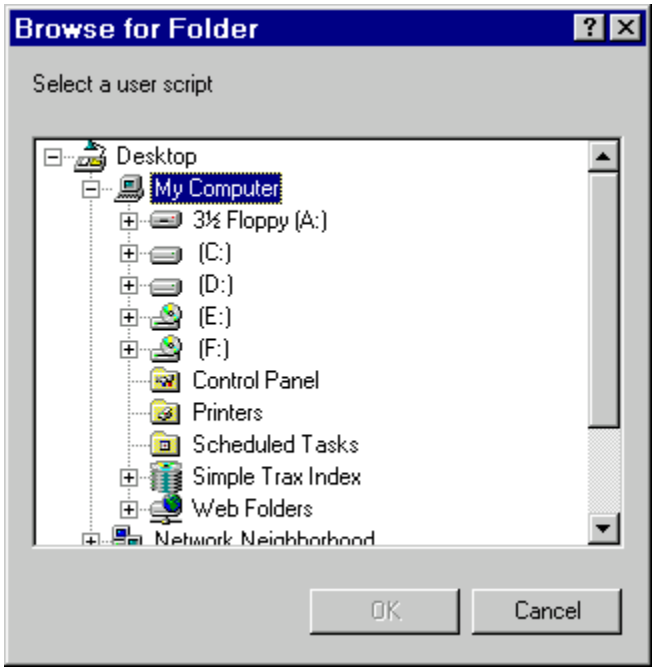
Personal Scripts

Qedit allows you to have personal scripts. These scripts do not have to reside in a specific location. You can specify the location in the **User script directory** box on the **Preferences** dialog box.



User script directory option in the Preferences dialog box

You can store your personal scripts anywhere you like. In the example above, the scripts are in the `\robelle\qedit\Private` directory on the `D:` drive. To point to a different directory, click on the **Change...** button. From the **Browse for Folder** dialog box, select the location where the scripts are residing.



Browse for Folder dialog box

Specialized Subdirectories

Each of these directories, **System**, **User** and the personal directory, has subdirectories for specific purposes. Qedit assumes that all the files in these subdirectories are scripts, no matter what they are called or which attributes they might have.

Autoload Subdirectory

This subdirectory contains scripts that are automatically loaded when Qedit starts. Loading does **not** mean execute. Qedit scans all the scripts in the sorted order. It compiles each script and reports compile errors, if any. Then, it searches for subroutines and `On command` statements. Statements in these blocks are compiled and stored in memory, ready to be used.

Subroutines become methods and, as such, can be called from any other scripts. Script names, if applicable, and `On command` statements are added to the **Script** menu. Users can then execute them from the menu.

Qedit scans the **Autoload** subdirectory in each script library directory.

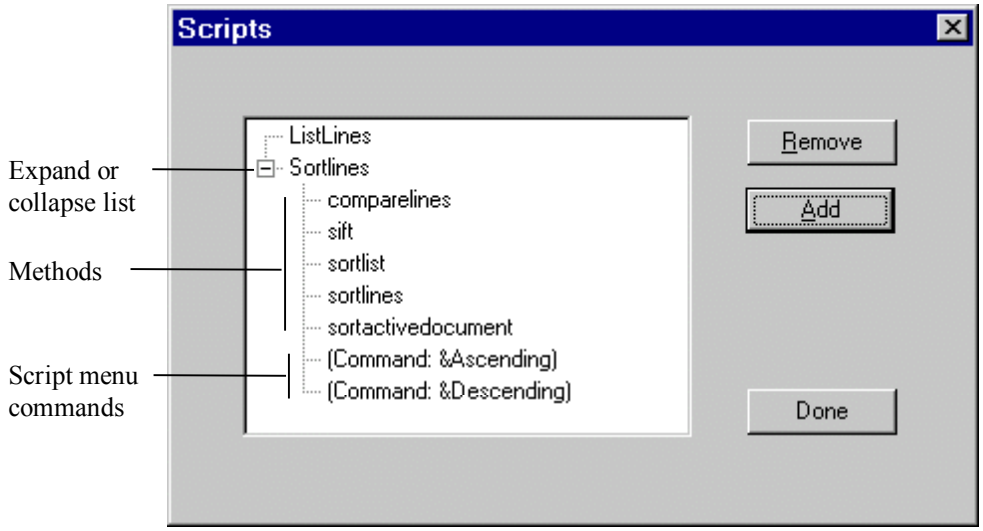
Scripts Subdirectory

This subdirectory is used to store all other scripts. These scripts are typically run or loaded manually as needed.

Although Qedit does not look for this subdirectory in particular, by convention, each script library directory should have one called **Scripts**.

Managing Loaded Scripts

Qedit automatically loads all the scripts stored in the **Autoload** directories. However, there are times where you will want to manually load other scripts or even remove scripts that were auto-loaded. You can manage the list of loaded scripts with the **Manage scripts** dialog box. This dialog box is available from the **Manage scripts** command of the **Script** menu.



Manage scripts dialog box

Collapsed list

Expanded list

Qedit displays all the loaded scripts in the list window. The list is arranged as an hierarchical structure. The top levels are the script names. If a script contains methods (subroutines) or commands (`On command` statements), the name is preceded by a plus or minus sign. A plus sign (collapsed structure) indicates that the lower levels are not visible. To expand the structure and make the lower-level entries visible, click on the plus sign. The sign changes to a minus (expanded structure).

Collapsed list

Expanded list

With an expanded structure, you can see all the callable methods (subroutines). **Script** menu commands appear in parentheses with the **Command:** prefix. The ampersand, if any, indicates the mnemonic. A mnemonic is an alternative way of invoking the command. In the example above, you can sort the lines in ascending order by entering ALT+A.

If you wish to unload a script, simply select its name by clicking on it and select **Remove**. If you want to load a new script, select **Add** and find the script file. Note that you can only load local scripts with the **Manage Scripts** dialog box. Once added, methods and commands in the script become available.

Macros and Common Functions

Macro commands are found in a lot of software today. QSL also allows you to create your own set of commands for things that are not implemented in Qedit or that require a number of steps. The sample script "Insert a Signature or a Timestamp" on page 71 is a good example. It contains 2 `On command` statements, each on corresponding to a macro. To make these macros available on the **Script** menu, you need to load the script. You can manually load the script using the **Manage scripts** dialog box, the first time you need one of the functions. You can automatically load the script simply by saving it in one of the `Autoload` directories. Or you can have a special script that will load other scripts for you.

Another use for loaded scripts is to create a library of general-purpose functions. These functions would be available to all other scripts.

For example, here is how we combine all these things to manage the Qedit test suite.

First, we create an autoload script to add a command on the **Script** menu. Let's call it `QWTLoadGlobals.qsl` and it is going to reside in the `User/Autoload` directory. The script only contains a few lines:

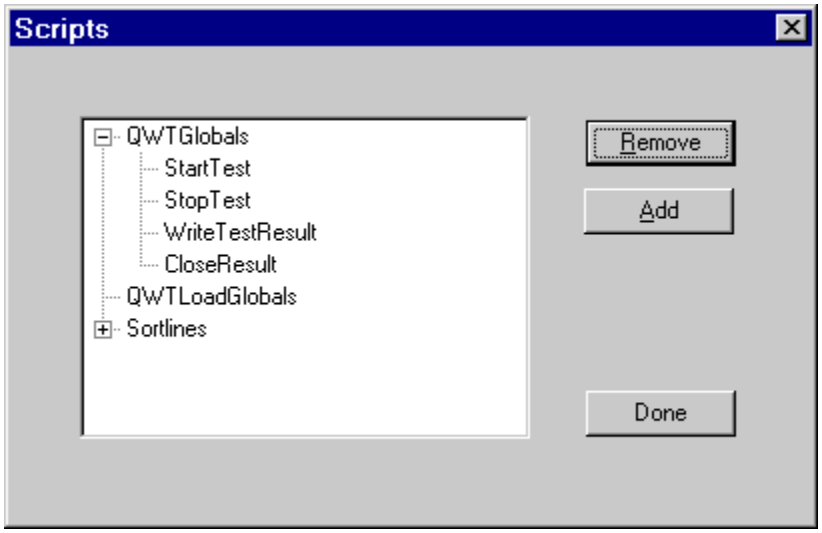
```
name QWTLoadGlobals;

loadscript(filename: "robelle/qedit/user/scripts/globals.qsl");
```

Because it's in the `Autoload` directory, the **QWTLoadGlobals** command automatically appears on the **Script** menu. It is important to remember that the script is not executed at this point.

This script simply loads another script called `globals.qsl`. This script contains a number of generic methods to manage a result file. Each method is defined as a subroutine. We also assign a name to the script: `QWTGlobals`.

```
name QWTGlobals;

property resultFile;
property fileResultFilename = "c:\robelle\qedit\testing\results.txt";

sub StartTest(testSet, file)
    resultFile = open(fileResultFilename);
    theMessage = "Start " + testSet + " on file ";
    theMessage = theMessage + file.title;
    theMessage = theMessage + " at ";
    theDate = datetime();
    theMessage = theMessage + theDate.FmtShortDateTime();
    aLine = {};
    aLine = aLine + theMessage;
    aLine = aLine + "";
    theSpot = {line: 0, column: 1};
    theSpot.line = resultFile.linecount;
    resultFile.Insert(at: theSpot, text: aLine);
endsub

sub StopTest(testSet, result)
    theMessage = "Stop " + testSet;
    if result then
        theMessage = theMessage + " Okay";
    else
        theMessage = theMessage + " Failure";
    endif

    aLine = {};
    aLine = aLine + theMessage;
    aLine = aLine + "";
    theSpot = {line: 0, column: 1};
    theSpot.line = resultFile.linecount;
    resultFile.Insert(at: theSpot, text: aLine);
endsub

sub WriteTestResult(theMessage)
    resultFile.Insert(theMessage);
endsub

sub CloseResult()
    resultFile.Close();
endsub
```

These methods are not available immediately when Qedit starts. You have to execute the **QWTLoadGlobals** command first. If you have not done so, scripts that try to call one of the methods simply aborts with an error. Once the command is executed, the methods become available. Note that these do not appear as additional commands on the **Script** menu. If you want to display their names, you have to use the **Manage scripts** dialog box.



Loaded scripts after executing QWTLoadGlobals

A sample script could then use the methods as follows:

```
mpefile = open(connection: "MPE", filename: "file01.test");
QWTGlobals.starttest("TEST01", mpefile);
QWTGlobals.writetestresult("Step 1: file opened successfully");
QWTGlobals.stoptest("TEST01", True);  -- Success
mpefile.close();
```

The `results.txt` file would then contain:

```
Start TEST01 on file MPE: FILE01.TEST.ACCT at 10/29/1999
11:31:07 AM
Step 1: file opened successfully
Stop TEST01 Okay
```

Some Basic Scripting Operations

Where To Start

We have seen how Qedit works with objects. Each object has attributes (properties) and functions (methods). In this section, you will find how these all hang together and what is the logical sequence of dealing with objects.

The QEDIT Application Object

The `QEDIT` application object is always there. It is the foundation for all other operations because the application methods allow you to create and manage other objects such as documents.

File Operations

Document objects have a one-to-one relationship with an instance of a file. We are talking about instances because one physical file can be accessed multiple times. The first thing you have to do is to create a document object. This can be done with the `Newfile()` or `Open()` application methods.

Creating a File

Let's start by creating a new local file.

```
newlocal = newfile();
```

This statement creates a new document object called `newlocal`. There are no parameters because the default is to create a local file. You do not have to provide a filename at this point. If you want to create a host file, you have to specify the connection name where the file should go. The connection name must already exist.

```
newhost = newfile(connection: "Production UX");
```

`Newhost` is also a document object. Both `newlocal` and `newhost` take on default properties for the particular platform as defined on the corresponding page in the **Preferences** dialog.

Opening a File

If you want to work on an existing file, you have to open it first. Similar to creating a file, you have to create a document object.

```
oldlocal = open("C:\personal\diary.txt");
oldhost = open(connection: "Production UX", filename: "cobolsrc");
```

In this example, `oldlocal` and `oldhost` are document objects linked to existing files. Note that you do not have to specify the **Filename** parameter keyword when opening a local file.

Each document object, no matter how it was created, comes with document methods. These methods are functions to alter the object. When you call a method, you have to qualify it with the object name so Qedit knows which file you want to work on.

Printing a File

QSL allows you to send the content of a file to the local printer attached to your PC or on the network. For host files, you can also send the output to a system printer on the host. Both document methods, **PrintOnLocal()** and **PrintOnHost()**, assume that you want to print the whole file unless you have specified **StartLine** or **Endline**.

The **PrintOnLocal** method sends the output to the default printer configured on your PC. It uses all the default settings. You do not have any way of changing these settings.

The **PrintOnHost** method provides all the controls that you have from the **Print on host** dialog box.

In this example, the script prints the last 20 lines of the local file and prints the first 20 lines of the host file using the **Shift** option.

```
localfile = open("c:\personal\diary.txt");
last20 = localfile.linecount - 19;
localfile.printonlocal(startline: last20);
localfile.close();

mpefile = open(connection: "Development MPE", filename="prog1.src");
mpefile.printonhost(endline: 20, shift: true);
mprefile.close();
```

Saving a File

If you want to save the changes made to a file, you can use one of two methods: `Save()` or `SaveAs()`. `Save()` is used to save changes to an existing file, overwriting the original. `SaveAs()` must be used to record changes for a new file. It can also be used for an existing file if you want to save the changes under a different name.

```
newlocal.saveas("C:\personal\newdiary.txt");
oldhost.save();
```

Closing a File

If you want to close either types of files, you should call the `close()` method. For example, to close a host file you would use:

```
newhost.close();
```

There are no parameters in this call because we want to use default settings.

Editing Files

Once you have a file opened, you likely want to modify its contents. Again, this is done using document methods.

Adding Text

Let's put some text in a new local file. You use the `Insert()` method to add text to a file.

```
newlocal = newfile();
newlocal.insert(at: {line: 1, column: 1},
       text: "This is the first line.");
newlocal.saveas("C:\personal\newdiary.txt");
newlocal.close();
```

In this example, the location defined with the **at** keyword is not important since the file is empty. If you want to insert text in the middle of a line, simply provide a different column number. If you want to insert a new line, simple use a different line number. If you want to insert a line at beginning of the file, use the coordinates from the sample script.

Calling the `Insert()` method multiple times is not an efficient way to load a file. If you need to create many lines, you can create a record where each element represents a line. For example, to create a file with 3 lines of text each separated by an empty line, you could use the following:

```
newlocal = newfile();
textlines = {};    -- Create an empty record
textlines = textlines + "This is the first line";    -- Append text
textlines = textlines + "";   -- Append an empty line
textlines = textlines + "This is the second line";
textlines = textlines + "";
textlines = textlines + "This is the third line";
newlocal.insert(at: {line: 1, column: 1}, text: textlines);
newlocal.close();
```

Deleting Text

To remove unwanted text, all you have to do is call the `Delete()` method with the coordinates. If you know exactly what you want to remove, you can hard-code the start and end coordinates.

```
textlines = {};    -- Create an empty record
textlines = textlines + "This is the first line";    -- Append text
textlines = textlines + "";   -- Append an empty line
textlines = textlines + "This is the second line";
textlines = textlines + "";
textlines = textlines + "This is the third line";
newlocal.insert(at: {line: 1, column: 1}, text: textlines);

newlocal.delete(startline: 2);    -- Delete line #2
newlocal.delete(startline: 4, startcolumn: 5, endcolumn: 7);
       -- Delete text in column 5 through 7 of line #4
```

If you don't know the coordinates ahead of time, you can use the information from the `selection` document property. This property contains the cursor's current

position whether it's a simple caret location or an actual selection. The selection might be the result of a previous call to the `select()` or `find()` methods.

```
textlines = {};    -- Create an empty record
textlines = textlines + "This is the first line";    -- Append text
textlines = textlines + "";   -- Append an empty line
textlines = textlines + "This is the second line";
textlines = textlines + "";
textlines = textlines + "This is the third line";
newlocal.insert(at: {line: 1, column: 1}, text: textlines);

newlocal.delete(startline: 2);    -- Delete line 2
newlocal.delete(startline: 4, startcolumn:5, endcolumn: 7);
       -- Delete text in column 5 through 7 of line 4

if newlocal.find(string: "third", entirefile: true) then
    newlocal.delete(range: newlocal.selection);
endif
```

In this example, the script searches for the string "third" from the beginning of the file. If it finds it, it deletes it.

Replacing Strings

The string replace operation is implemented in the `Find()` method. All you need to do is specify the **ReplaceWith** parameter. If you specify this parameter, all the strings found within the range are changed.

By default, the operation starts at the current cursor location and goes to the end of the file. You can perform a change on a selection only. To define the selection, you can use the `Select()` document method before calling `Find()` or specify the coordinates right in the `Find()` call.

```
oldlocal = open("C:\personal\diary.txt");
oldlocal.select(startline: 1, startcolumn: 1);
    -- Places cursor at the beginning of the file
replaceresult = oldlocal.find(string: "J. Doe",
       replacewith: "John Doe");

oldlocal.select(startline: 1, endline: 5);   -- Only lines 1 to 5
replaceresult = oldlocal.find(string: "Joe Doe",
       replacewith: "Joe Doe",
       selectiononly: true);
if replaceresult then
    result = dialog(string(replaceresult) + " occurrences replaced");
else
    result = dialog("NOT FOUND!");
endif
oldlocal.close();
```

Copying, Cutting and Pasting Text

You can `Copy()`, `Cut()` and `Paste()` text inside the same document or from one document to another just like you would using the mouse and keyboard. All you have to do is

1. select the text that you want to copy or cut.
2. call `cut()` or `copy()`
3. move the cursor to the destination location
4. paste the clipboard with the `paste()` method

```
oldlocal = open("C:\personal\diary.txt");
oldlocal.select(startline: 1, endline: 5);
    -- Selects the first 5 lines
oldlocal.copy();   -- Send the selection to the clipboard
oldlocal.select(startline: oldlocal.linecount, startcolumn: 1);
oldlocal.paste();  -- Paste the clipboard
oldlocal.close();
```

To paste into another file, just use the object name of the other file when calling the `Paste()` method.

```
oldlocal = open("C:\personal\diary.txt");
oldlocal.select(startline: 1, endline: 5);
    -- Selects the first 5 lines
oldlocal.copy();   -- Send the selection to the clipboard

newlocal = newfile();
newlocal.select(startline: 1, startcolumn: 1);
newlocal.paste();  -- Paste the clipboard
oldlocal.close();
newlocal.close();
```

Selecting and Retrieving Text

In many cases, you have to be able to move the cursor or select existing text. There are a few document methods available for this purpose.

Retrieving Text in Known Location

If all you want to do is retrieve some text, you can use the `GetText()` method. As long as you know the text coordinates, you can extract that portion of the file and copy it to a variable.

To select one whole line, simply specify the start line. If you want to select multiple complete lines, specify a start and end line.

```
oldlocal open("C:\personal\diary.txt");
firstline = oldlocal.gettext(startline: 1);
    -- Puts only the first line in firstline
lines1to5 = oldlocal.gettext(startline: 1, endline: 5);
    -- Puts lines 1 to 5 in lines1to5
oldlocal.close().
```

To select one or characters on a single line, specify a start line and a start and end column. To select text on different lines, add an end line.

```
oldlocal open("C:\personal\diary.txt");
column3to12: oldlocal.gettext(startline: 1,
           startcolumn: 3, endcolumn: 12);
    -- Copies text from columns 3 to 12 on line 1 into column3to12
multilines = oldlocal.select(startline: 1, startcolumn: 3,
              endline: 5, endcolumn: 12);
    -- Copies text from column 3 to the end of line 1 up to
    -- column 12 on line 5 into multilines
oldlocal.close().
```

If you want to perform a columnar operation, you need to specify the **Rectangular** keyword.

```
oldlocal open("C:\personal\diary.txt");
rectangle = oldlocal.gettext(startline: 1, startcolumn: 3,
               endline: 5, endcolumn: 12,
               rectangular: true);
    -- Copies text between columns 3 and 12 on lines 1 through 5
    -- into rectangle
oldlocal.close().
```

The target variable can be a simple variable or a record. The only time it's single is when you are retrieving characters from a single line. If the retrieved text spans multiple lines, the target variable is a record. A selected line is stored as 2 elements:

- the text on the line
- an empty element to indicate the start of a new line

For example, if the file contains:

```
First line.
Second line.
```

Retrieving these 2 lines would create a record with the following elements:

```
{ "First line", "Second line.", "" }
```

Moving the Cursor

If you know where you want to move the cursor, you can use the `Select()` method. By itself, the `Select()` method does not do anything with the text. It simply moves the cursor to the specified location and, if applicable, highlights the selected text.

The `Select()` method is typically used to prepare another operation. Here are examples showing how you would use this method to:

- start a search operation at a precise location instead of searching through the entire file
- do a replace operation in a selection only
- retrieve part of a file

If you want to move the caret to a different location but not select any text, you should specify a start line and start column only.

```
oldlocal = open("C:\personal\diary.txt");

oldlocal.select(startline: 1, startcolumn: 10);
    -- Places the insertion point in column 10 of the first line

if oldlocal.find(string: "John Doe") then
    result = dialog("Found at " + string(oldlocal.selection));
else
    result = dialog("NOT FOUND!");
endif
oldlocal.close();
```

If you want to put the caret at the beginning of the file, set **StartLine** and **StartColumn** to 1. If you want to put the caret at the end of a line, use the **RecordLength** document property as the starting column as in:

```
file.select(startline: 2, startcolumn: file.recordlength);
```

Of course, the record length is likely going to be greater than the actual line length. In this situation, Qedit puts the caret after the last significant character on the line.

If you want to put the caret at the end of the file, you can use the **LineCount** document property as the starting line and the **RecordLength** property as the starting column as in:

```
file.select(startline: file.linecount, startcolumn: file.recordlength);
```

If you want to select some text, you have to use different start and end combinations. When the `Select()` method is executed, the corresponding text appears highlighted in the document window.

To select one whole line, simply specify the start line. If you want to select multiple complete lines, specify a start and end line.

```
oldlocal open("C:\personal\diary.txt");
oldlocal.select(startline: 1);    -- Selects only the first line
oldlocal.select(startline: 1, endline: 5);   -- Selects lines 1 to 5

replaceresult = oldlocal.find(string: "John Doe",
       replacewith: "Jane Dover",
       selectiononly: true);
if replaceresult then
    result = dialog(string(replaceresult) + " occurrences replaced");
else
    result = dialog("NOT FOUND!");
endif
oldlocal.close().
```

To select one or characters on a single line, specify a start line and a start and end column. To select text on different lines, add an end line.

```
oldlocal = open("C:\personal\diary.txt");
oldlocal.select(startline: 1, startcolumn: 3, endcolumn: 12);
    -- Selects text in columns 3 to 12 on line 1

columns3to12 = oldlocal.getselectedtext();
    -- Copies the selection into columns3to12

oldlocal.select(startline: 1, startcolumn: 3,
               endline: 5, endcolumn: 12);
    -- Selects text from column 3 to the end of line 1 up to
    -- column 12 on line 5
oldlocal.close().
```

If you wish to select all the lines in a file, you can use the `Select()` method where **StartLine** is 1 and **EndLine** is the **LineCount** property. A simpler way is to use the `SelectAll()` method.

```
oldlocal.select(startline: 1, endline: oldlocal.linecount);

oldlocal.selectall();  -- Same result as previous Select call
```

If you want to perform a columnar operation, you need to specify the **Rectangular** keyword.

```
oldlocal = open("C:\personal\diary.txt");
oldlocal.select(startline: 1, startcolumn: 3,
               endline: 5, endcolumn: 12,
               rectangular: true);
    -- Selects text between columns 3 and 12
    -- on lines 1 through 5
oldlocal.close().
```

If you wish to return to the current position, you should save the **Selection** property into a variable. You can continue to process the file. When you are ready to go back, you have to use the `Select()` method specifying the variable as the value for the **Range** keyword.

```
saveCurrent = file.selection;
-- Perform other edit operations
file.select(range: saveCurrent);  -- Go back to the saved position
```

Finding Text

The `Select()` and `GetText()` methods are fine if you know where the text you need is located. If you don't know where it is, you can use the `Find()` method. You might still have to use `Select()` to control the range of the search.

In its simplest form, a call to `Find()` requires a search string. It can be a string of characters, a pattern or a regular expression. The default is to start the search from the current cursor location. If the string is found, the method returns `True`. It also updates the following document properties:

- **LastFoundLength**
- **LastFoundLine**
- **LastFoundColumn**

You can also access the **Selection** document property to determine the coordinates.

```
oldlocal = open("C:\personal\diary.txt");
oldlocal.select(startline: 1, startcolumn: 1);
    -- Make sure the cursor is the beginning of the file

-- Search for an exact string
findresult = oldlocal.find(string: "John Doe");
if findresult then
    result = dialog("String found at " + string(oldlocal.selection));
else
    result = dialog("String NOT FOUND");
endif

-- Search for a pattern e.g. John Doe, J. Doe, Jane Doe
findresult = oldlocal.find(pattern: "@J@Doe@");
if findresult then
    result = dialog("Pattern found at " + string(oldlocal.selection));
else
    result = dialog("Pattern NOT FOUND");
endif

-- Search for a pattern e.g. John Doe, J. Doe, Jane Doe
findresult = oldlocal.find(regexp: "J[a-zA-Z .]*Doe");
if findresult then
    result = dialog("Regexp found at " + string(oldlocal.selection));
else
    result = dialog("Regexp NOT FOUND");
endif
```

Retrieving Selected Text

If you know there is an active selection, either explicitly set with `Select()` or after a successful search operation, you can retrieve the selected text using the `GetSelectedText()` method. It's a very simple method call because there are no parameters. All it does is copy the selection into a variable.

```
oldlocal = open("C:\personal\diary.txt");
-- Search for a pattern e.g. John Doe, J. Doe, Jane Doe
findresult = oldlocal.find(regexp: "J[a-zA-Z .]*Doe");
if findresult then
    stringfound = oldlocal.getselectedtext();
else
    result = dialog("Regexp NOT FOUND");
endif
if stringfound = "John Doe" then
    result = dialog("Found the full name");
else
    result = dialog("Found someone named J Doe:" + stringfound);
endif
oldlocal.close();
```

The target variable can be a simple variable if the selected text is on a single line. The result of a `Find()` can never span multiple lines so `GetSelectedText()` would create a simple variable. However, if `GetSelectedText()` is used after a call to `Select()`, it could retrieve text on multiple lines. In that case, a record would be created. A selected line is stored as 2 elements:

- the text on the line
- an empty element to indicate the start of a new line

For example, if the file contains:

```
First line.
Second line.
```

Retrieving these 2 lines would create a record with the following elements:

```
{ "First line", "", "Second line", "" }
```

Text Within Text

Qedit provides various methods to retrieve text from a document into variables where it's going to be further processed. There are other features such as the `POS()` built-in function and subscripts allowing you to work with text retrieved previously. The following code segment first retrieves some text from a document and then checks to see if the text contains the word *`gadget`*.

```
foundText = file.getselected();
wordPosition = pos(foundText, "gadget"); -- return starting position
```

This code would work fine as long as *`gadget`* is always spelled that way. This would not work if the word contained uppercase letters. To work around this possibility, you can use the `Downshift()()` or `Upshift()()` built-in functions to force casing in one direction i.e. all lowercase or all uppercase. The revised segment could look like this:

```
foundText = file.getselectedtext();
wordPosition = pos(downshift(foundText), "gadget");
```

The retrieved text would first be changed to lowercase characters before being compared.

Navigating Through Directories

When working with files in a directory tree structure, it is important to know where you are and how to move around in the structure. That's true for local files as well as host files. QSL provides the information and facility in different ways.

Local Directories

If you wish to see what is the local current working directory, simply query the **LocalCWD** application property. The information is returned as string and contains the fully-qualified pathname, including the drive letter.

If you wish to move to a different directory, assign the destination to **LocalCWD**. Qedit takes care of switching to the specified directory. The pathname can contain a drive letter. It can also be an absolute pathname e.g. `c:\robelle` or relative to the current location e.g. `bin\user\scripts`.

```
currentCWD = qedit.localcwd;  -- e.g. returns c:\robelle\bin

qedit.localcwd = "d:\personal";   -- changes CWD
localfile = open("diary.txt");    -- opens a file in new CWD
                                  -- perform edits on the file
localfile.save();                 -- save changes

qedit.localcwd = currentCWD;      -- returns to original CWD
```

If the pathname entered as the destination directory is incorrect e.g. does not exist, Qedit returns an error at run time:

```
Line n: a property request failed.
```

This only thing you can do at this point is fix the pathname and re-run the script.

Host Directories

When it comes to communication with host computers, Qedit takes care of everything. It establishes the connection when the first file is opened. There are times when you do not know which file to open at first. You might want to select files from a directory listing instead. In these situations, you can use the `Openconnection()` application method to create a new connection object. Qedit does all the work to connect you to the host without actually opening a file.

If you wish to see the host current working directory, you can query the **HostCWD** connection property.

```
currentHostCWD = mpeconn.hostcwd;
```

Once a connection is opened, you can enable the **ShowBrowser** connection property to display the **Directory** dialog box.

You can use the `ChangeCWD()` connection method to navigate through the directory tree on that host. That same method also allows you to get a subset of files in a directory.

The following script opens a connection to a UNIX host, changes the current working directory and displays all the files starting with the letter "s".

```
uxconn = openconnection(connection: "Development UX");
uxconn.changecwd(pathname: "/usr/include/sys",   -- Change the CWD
                 wildcard: "s*");   -- Select file subset
uxconn.showbrowser = true;          -- Display the Directory dialog box
```

If you have a file already opened, you can get the connection information from the **Connection** document property. This way, you can create a connection object and work with the connection methods and properties.

```
uxfile = open(connection: "Development UX", filename: "program1.c");
uxconn = uxfile.connection;
uxconn.changecwd(pathname: "/usr/include/sys");
uxconn.showbrowser = true;
```

If the pathname entered as the destination directory is incorrect e.g. does not exist, Qedit returns an error at run time. The error message itself is going to be different depending on the host it's coming from. On a UNIX host, for example, the error could be:

```
Unable to get the directory listing. No such file or
directory
```

On an MPE host, the error could be:

```
Unable to get the directory listing. A component of the
pathname "" does not exist. (CIERR 9039).
```

If you want to protect the script from such errors, you can use a `TRY/RECOVER` block as in:

```
try
    uxconn.changecwd(pathname: "/home/unknown/directory");
recover
    result = dialog("Could not change CWD");
    return false;
endtry
```

Using the Directory Iterators

Scripts can be written to access specific files at specific locations on specific machines simply by hard-coding everything. Scripts can also be more flexible by accepting file-related information in parameters or global variables. Typically, these methods are appropriate for a small number of files or a pre-determined set of files. If you wish to write a script to process all the files stored in a group or directory, you should use directory iterator objects.

These objects are created with the `GetDirectoryIterator()` methods. The application method is used to process local directories. The connection method is used to process host directories. The method name is the same in both cases. The only thing that differentiates them is the prefix. The application method does not have a prefix whereas the connection method is prefixed with the name of a connection object previously created.

Directory Iterators Are Dynamic Objects

Iterator objects are really a list of records. Each record containing information about a file or subdirectory. For a description of these records, see "Local Directory Iterator" on page 160 or "Host Directory Iterator" on page 161 . To access entries in a directory iterator, you have to use a `REPEAT` statement.

Iterator are dynamic objects. They are really a snapshot of the directory structure and files at a particular point in time. As soon as you start processing an iterator, the information in it is removed and no longer available. So, after you process all the entries of an iterator using a `REPEAT` block, the iterator object will be empty. If you want to preserve the information, you can save it in a temporary record as the `REPEAT` loop executes. Keep in mind that this information might not be up-to-date anymore.

```
saveDirList = {};  -- Create a temporary variable
index=1;

localdir = getdirectoryiterator( "c:\personal" );    -- Get information
repeat for direntry in localdir       -- Save iterator information
    saveDirList[index] = direntry;    -- As nested records
    index = index + 1;
endrepeat

    -- At this point, localdir is empty.
repeat for direntry in saveDirList
    writelog( "--" + direntry.name );
endrepeat
```

See "Using Nested Records" on page 10 on how to create and access nested records.

Local Directory Iterator

Let's say you want to get a list of all the subdirectories in the "c:\personal" local directory. You could use the following script:

```
localdir = getdirectoryiterator("c:\personal");
        -- localdir is the name of the directory iterator

repeat for direntry in localdir
    if direntry.canonicaltype = "directory" then
        writelog(direntry.name);
    endif
endrepeat
```

The log window might contain something like:

```
.
..
Expenses
Agenda
Pictures
```

If you do not specify a directory name in the call to `GetDirectoryIterator()`, Qedit uses the current working directory. The parent directory ".." and the current directory "." are always there. Note also that subdirectory names do not contain any special characters or separators. The **canonicaltype** element is the only way to identify them.

Host Directory Iterator

To get a list of subdirectories on a host connection, you need a connection object. If one does not exist, you can use the `OpenConnection()` application method to create it. If the connection is already opened, you can create the connection object using the **Connection** document property.

Host directory iterators are created with the `GetDirectoryIterator()` connection method. Thus, you have to qualify the call with the connection object name. The directory name passed in as parameter for an MPE connection must use the POSIX notation on MPE as in:

/DEVACCT/SRC

For example, you could get the list of subdirectories from a UNIX connection using the following script:

```
uxconn = openconnection("Production UX");-- Create connection object

uxdir = uxconn("/home/clerk");
    -- Create directory iterator based on specified directory

repeat for direntry in uxdir
    if direntry.canonicaltype = "directory" then
        writelog(direntry.name);
    endif
endrepeat
```

The log window might show something like:

```
.elm/
.vue/
Mail/
data/
programs/
personal/
```

Like local directory iterators, if you do not specify a directory name on the call to `GetDirectoryIterator()`, the current working directory is used. Note that the parent directory ".." and current directory "." do not appear in the list. Subdirectory names end with a slash character "/".

Moving To Different Levels

Directory iterators contain information on files and subdirectories located at the specified level only. To get at the information on different levels, you have to code the script accordingly. See "Displaying Information From Directory Iterators" on page 80 for a sample script.

Executing Host Commands

At times, it is necessary to go back to the host to execute other types of commands. Qedit can handle all the file editing functions but these files are created for a purpose. These can be program source files that need to be compiled into executable files. These executable files need to be run and tested.

QSL provides a set of application methods allowing you to execute host commands directly from inside Qedit for Windows.

Host Commands Environment

Important: This feature is not meant to be a replacement terminal emulator. It's not interactive. It's more like a batch-oriented type of environment. This means that it can not handle user input in the middle of the execution. All the commands must be *self-contained* i.e. once started they can complete without further interaction from the user.

Qedit connections start a new network session on MPE hosts. However, these sessions do not acquire the same environment as terminal sessions. MPE does not execute logon UDCs for network sessions. If you use logon UDCs to set global settings like HPPATH, these will not be set when execute host commands from Qedit for Windows. This can cause execution errors as unqualified program names or command files might not be found.

There are a couple of ways you can work around this limitation. You can write scripts such that there are no ambiguity, making sure the environmnent is set up properly. If you have many scripts with host commands, this can get complicated to

maintain, The Qedit server can automatically execute commands found in configuration files on the host. Please refer to "Customizing the server" in the Qedit for Windows User Manual for more details.

If you set the host environment the way you want, it's going to remain that way until you change it or until you close the connection. For example, if you change the HPPATH variable, it is going to retain this new value for subsequent host command execution.

Starting Execution

In order to execute host commands, you first have to establish a connection. You can use a connection that is already opened or, if there aren't any, you can initiate one explicitly with the `OpenConnection()` application method.

If you have a single command to execute, you can include it as a parameter directly in the call to `HostCommand()` application method.

```
mpeConn = openconnection(connection: "Prod MPE");
hostCmdResult = hostcommand(connection: mpeConn,
                command: "listf ,2");
```

If you wish to execute more than one command, you can put them all in a record variable, each command being an element.

```
mpeConn = openconnection(connection: "Prod MPE");
cmdList = {"Showme", "Listf ,2", "Showvar"};
hostCmdResult = hostcommand(connection: mpeConn,
                command: cmdList);
```

Checking Results

The `HostCommand()` application method returns information on the execution outcome in a record variable. The number of elements in the record is variable. There is always an element called **ExitCode**. It indicates whether the host was able to start the execution. It does not indicate the success or failure of the actual commands. A value of zero indicates the command was started successfully. Any non-zero value indicates that the host has been unable to start command execution.

Typically, host commands return information on the outcome. MPE hosts use Job Control Word (JCW). By convention, there is a JCW called **JCW**. This particular variable is typically reflects the overall execution status. The value can be set automatically by executing programs or manually using the `Setjcw` command. On UNIX, most programs, commands and scripts use the `return` statement to indicate success or failure. From a UNIX shell, the return value is stored in the **$?** shell variable. Upon host command completion, the server returns the value of the **JCW** variable or the **$?** variable as the **JCW** element of the return record.

On a UNIX host, the **JCW** element can have different meanings. If the value is positive, it represents the return code. If the value is negative, it represents a signal. This means the host command has been interrupted by someone else or encountered a serious error. For example, if the host command has been interrupted with

`kill <processID>`

the value of **JCW** would be -15. If, for some reason, the server can not find the return code or a valid signal, the **JCW** element will contain -1000.

The QhostResult feature is only available on MPE hosts.

Optionally, the MPE server provides a simple way to communicate information from your host commands. Upon host command completion, the server checks the existence of a user-defined variable called **QHostResult**. If it exists, the server transmits its value back to the client. The variable then becomes an element in the return record. The variable can be a numeric value or a string. Numeric values can range between -2147483648 and 2147483647. Boolean values are transmitted as numeric values where 0 is `False` and 1 is `True`. A string value can contain up to 255 characters. You decide which type serves you best.

Another element in the result record is called **CommandOutput**. It's another record containing the actual output generated by the host commands. Each element of this record represents one line of output. Here is an example on how to display the result of a host command in the log window of the **Script Control** dialog box.

```
mpeConn = openconnection(connection: "Prod MPE");
hostCmdResult = hostcommand(connection: mpeConn,
                command: "showme");
if hostCmdResult.ExitCode = 0
    repeat for outputline in hostCmdResult.CommandOutput
        writelog(outputline);
    endrepeat
endif
```

The log window would contain something like:

```
MPE/iX CI C.16.01 Copyright (C) Hewlett-Packard 1987. All Rights
Reserved.
_&dJ****************   Production MPE ************************
_&dJ***   This is a private system operated for ACME Widget  ***
_&dJ***   company business.                                   ***
_&dJ***   Use by unauthorized persons is prohibited.         ***
_&dJ*************************************************************
:showjob job=@j
FRI, MAR 17, 2000, 11:50 AM

JOBNUM  STATE IPRI JIN  JLIST    INTRODUCED  JOB NAME
#J11263 EXEC QUIET  10S LP       FRI  1:35A  JHTTPD,MGR.APACHE
#J11276 EXEC        10S LP       FRI  1:38A  JINETD,MANAGER.SYS

2 JOBS (DISPLAYED):
    0 INTRO
    0 WAIT; INCL 0 DEFERRED
    2 EXEC; INCL 0 SESSIONS
    0 SUSP
JOBFENCE= 5; JLIMIT= 9; SLIMIT= 60

CURRENT:  3/17/00 11:50

JOBNUM  STATE IPRI JIN  JLIST    SCHEDULED-INTRO   JOB NAME

#J11259 SCHED   15  10S LP        3/18/00  1:00    BACKUP,OPERATOR.SYS

1 SCHEDULED JOB(S)
```

Redirecting Results

Getting the host command output in a record makes it easy to process and react on their execution. However, if you need to process these results further, it might not be flexible enough. You can use the **Output** parameter and save the output into a local file. You have to open the file before calling the `HostCommand()` application method.

```
mpeConn = openconnection(connection: "Calvin");
localFile = newfile();

hostCmdResult = hostcommand(connection: mpeConn,
                command: "showjob job=@j",
                output: localFile);
```

In this case, the `hostCmdResult` return variable is still a record but contains only the **ExitCode** element. The host command output is automatically inserted in the local file specified in the **Output** parameter.

To Wait or Not To Wait

By default, host command execution is synchronous. This means that the QSL script suspends until the output is received from the host. For commands that may take a long time to execute, you might want to continue editing other documents. In order to do this, you need asynchronous command execution. This is done using the **Wait** parameter of the `HostCommand()` application method.

By default, this parameter is set to `True` i.e. QSL waits for the command to complete. If you wish to use asynchronous execution, simply set **Wait** to `False`. Qedit immediately resumes script execution.

How do you know when the host command is finished executing? You have to use the `HostCommandStatus()` application method. This method also uses the **Wait** parameter to indicate if you want to wait until the execution is complete before returning to the next statement in the script. By default, the method does not wait i.e. the parameter is `False`.

```
mpeConn = openconnection(connection: "Calvin");
localFile = newfile();

hostCmdResult = hostcommand(connection: mpeConn,
                command: "longcmd.cmd.devacct",
                output: localFile,
                wait: false);      -- Does not wait for completion

otherfile.activate();  -- Switch to other opened document

cmdIsDone = false;
repeat until cmdIsDone     -- Loop until the command is done
    checkStatus = hostcommandstatus();
    cmdIsDone = checkStatus.finished;     -- Save Finished value
endrepeat

result = dialog("Longcmd has completed!");
localfile.activate();     -- Switch to host command results file
```

`HostCommandStatus()` sets and returns the **Finished** record element to indicate whether the command is still executing or not. In the previous example, the `Repeat` block executes until **Finished** is `True`.

If the `HostCommandStatus()` method is called in an undefined context, you should also check the **Running** record element. A value of `False` indicates there are no host commands currently executing. A modified code segment would look like:

```
checkStatus = hostcommandstatus();

if checkStatus.Running then   -- Is there an active host command?
    cmdIsDone = false;
    repeat until cmdIsDone    -- Loop until the command is done
        checkStatus = hostcommandstatus();
        cmdIsDone = checkStatus.finished; -- Save Finished value
    endrepeat
    result = dialog("Host command has terminated!");
else
    writelog("No host command currently executing!");
endif
```

Is It Really Executing?

Even though the `HostCommandStatus()` method tells you the host command is currently executing, it does not necessarily mean it is actually doing useful work. As far as Qedit is concerned, the host command has not terminated but the command might not be doing anything on the host. To verify this, `HostCommandStatus()` returns 2 other variables that might help: **ProgressTime** and **Step**.

ProgressTime is the CPU time used by the host command. It represents the number of milliseconds of CPU.

```
hostresult = hostcommand(connection: "Prod MPE",
           command: "longcmd.cmd.devacct",
           wait: false);

lastProgress = 0;
cmdIsDone = false;
repeat until cmdIsDone
    checkStatus = hostcommandstatus();
    cmdIsDone = checkStatus.Finished;
    if lastProgress <> checkStatus.ProgressTime
        writelog("CPU=" + string(checkStatus.ProgressTime));
        lastProgress = checkStatus.ProgressTime;
    else
        writelog("No CPU used. Stuck at " + string(lastProgress));
    endif
endrepeat
```

Step contains the command currently executing. In a multi-command request, this would be the last command encountered in the list.

```
hostresult = hostcommand(connection: "Prod MPE",
           command: "longcmd.cmd.devacct",
           wait: false);

lastStep = "";
cmdIsDone = false;
repeat until cmdIsDone
    checkStatus = hostcommandstatus();
    cmdIsDone = checkStatus.Finished;
    if lastStep <> checkStatus.Step
        writelog("Executing command: " + checkStatus.Step);
        lastStep = checkStatus.Step;
    else
        writelog("Still executing: " + string(lastStep));
    endif
endrepeat
```

Stopping Execution

If there are situations when you would like to interrupt an asynchronous host command, you can use the `HostCommandAbort()` application method. This

method sends an abort request to the server. The server takes appropriate action to terminate execution. The method has to be called from the same script as the `HostCommand()` method.

By default, `HostCommandAbort()` is asynchronous. This means it does not wait for an acknowledgement from the server and returns to the script immediately. In the mean time, the host process might still be running for a few seconds. All host output is discarded. Rather the result list contains the **Running** and **Finished** elements, both set to `True`, to indicate that the process has terminated. The result list also contains the **ProgressTime**, **Step** and **ProcessID** elements to provide information of where the process was at when the abort occurred.

You can get `HostCommandAbort()` to work synchronously using the **Wait** parameter set to `True`. The method would then waits for server confirmation that the process has indeed terminated. The result list contains the **Running** and **Finished** elements, both set to `True`, to indicate that the process has terminated. It would also contain the **ExitCode** element to indicate the host command execution outcome and the **CommandOutput** element which contains host command output received up to the point of the abort. Of course, the output is likely to be different than a succesful uninterrupted execution of the same commands.

Stop button

If the script does not call `HostCommandAbort()` and is still executing, you can interrupt the host command by clicking on the **Stop** button of the **Script Control** dialog box.

Stop button

If the script has already stopped executing but the host command is still going, the only way you can interrupt it is by breaking the connection using the **Disconnect** command of the **File** menu. Remember that this operation automatically closes all files currently opened on this connection. If file changes have not been saved, Qedit prompts for save confirmation.

Dealing with Connection Templates

In many cases, connection templates are created permanently in the connection template files using the **Connection List** dialog box. You can reference these templates in all your scripts. To write faul-tolerant scripts, you might want to check the existence a particular connection template and, if it does not exist, create one for the duration of the script execution. Similarly, there are times when you might need to create connections on-the-fly using different logon information. QSL provides a number of application and connectiotemplate methods to help you deal with these situations.

Find a Connection Template

The `FindConnectionTemplate()` application method retrieves information about a connection template. It needs the name of the connection you are looking for and, if it finds it, creates a ConnectionTemplate object. If it cannot find the connection, it returns an undefined variable.

The parameter is a string. QSL does a caseless match i.e. uppercase and lowercase letters are treated the same. Except for this, the name must match exactly.

```
connobject = findconnectiontemplate("Prod UX");
if typeof(connobject) = qedit.typeundefined then
    writelog("The connection does not exist!");
else
    writelog("The connection has been found.");
endif
```

Once the object is created, you can view its properties and modify them with the `SetLogonInformation()` ConnectionTemplate method. For security reasons, QSL always resturns the passwords as null strings. So, there is no way to determine what the current passwords are. However, nothing prevents you from entering new passwords.

Create a Connection Template

If the script requires a specific connection, it can create it using the `NewConnectionTemplate()` application method. In order to do this, you have to supply individual elements making up a valid connection. This includes a connection name, a host name and valid logon information.

```
sub createconn();
    connname="New UX Conn";       -- Assign a name
    hostname="Prod UX";           -- Which host?
    logoninfo={};                 -- Setup logon information
    logoninfo.ConnectionType = "unix";
    logoninfo.Username = "pgmr";
    logoninfo.Password = "hispass";
    newconn = newconnectiontemplate(Name: connname,
                                    Host: hostname,
                                    LogonInformation: logoninfo);
    return newconn;        -- Return the new connection template object
endsub

connobject = findconnectiontemplate("Prod UX");
if typeof(connobject) = qedit.typeundefined then
    writelog("The connection does not exist!");
    connobject = createconn();        -- Create a new one
else
    writelog("The connection has been found.");
endif
```

Delete a Connection Template

Once you are done with a connection template, you can remove it from the connection template file using the `DeleteConnectionTemplate()` application method. The method accepts the name of a connection template passed as a string or a connection template object created by a call to `FindConnectionTemplate()`, `NewConnectionTemplate()` or `GetConnectionTemplateIterator()` application methods.

```
sub createconn();
    connname="New UX Conn";       -- Assign a name
    hostname="Prod UX";           -- Which host?
    logoninfo={};                 -- Setup logon information
    logoninfo.ConnectionType = "unix";
    logoninfo.Username = "pgmr";
    logoninfo.Password = "hispass";
    newconn = newconnectiontemplate(Name: connname,
                                Host: hostname,
                                LogonInformation: logoninfo);
    return newconn;        -- Return the new connection template object
endsub

existingConnection = true;

connobject = findconnectiontemplate("UX Conn");
if typeof(connobject) = qedit.typeundefined then
    writelog("The connection does not exist!");
    connobject = createconn();               -- Create a new one
    existingConnection = false;
else
    writelog("The connection has been found.");
endif

-- Use the new connection template
performactions();
-- Processing has completed

if not existingConnection then
    deleteconnectiontemplate(connobject);    -- Done. Get rid of it
endif
```

Getting All Connection Templates

The `GetConnectionTemplateIterator()` application method retrieves all existing connection templates into an iterator object. The iterator object can then be scanned to perform global searches and changes. For example, let's say the password for the `pgmr` user on all Unix hosts has expired. Instead of manually modifying all existing connections, you can write a script to perform a global change.

```
conniterator = getconnectiontemplateiterator();

repeat for connobject in conniterator
    if connobject.logoninformation.connectiontype = "Unix" and
        connobject.logoninformation.username = "pgmr" then
        templogon = connobject.logoninformation;
        templogon.password = "newpass";
        connobject.setlogoninformation(templogon);
    endif
endrepeat
```

Clone a Connection Template

If you wish to make an exact copy of an existing connection template quickly and efficiently, you can use the **FromTemplate** parameter on the `NewConnectionTemplate()` application method. The parameter requires a connection object created with a previous call to `NewConnectionTemplate()`, `FindConnectionTemplate()` or `GetConnectionTemplateIterator()`. You only need to specify a new connection name on the `NewConnectionTemplate()` call.

```
sub copyconn()
    testconn = findconnectiontemplate("test");   -- Get existing info
    if typeof(testconn) = qedit.typeobject then
        writelog("Test connection found");
    else
        writelog("Test connection NOT found");
    endif

    result = newconnectiontemplate(name: "Copy UX",  -- Create new conn
                fromtemplate: testconn);
    if typeof(result) = qedit.typeobject then
        writelog("Connection Copy UX copied");
    else
        writelog("Connection Copy UX NOT copied");
    endif
endsub

findconn = findconnectiontemplate("Copy UX");
if typeof(findconn) = qedit.typeundefined then
    writelog("Connection does not exist");
    copyconn();        -- Create a copy
else
    writelog("Connection already exists. Deleting!");
    deleteconnectiontemplate(findconn);      -- Delete existing
    copyconn();        -- Create a copy
endif
```

Executing and Testing Scripts

The Script Menu

The **Script** menu provides all the commands you need to work with scripts. The **Compile** command analyzes the active document and reports any QSL syntax errors. The **Run** command compiles the script and, if there are no syntax errors, executes it.

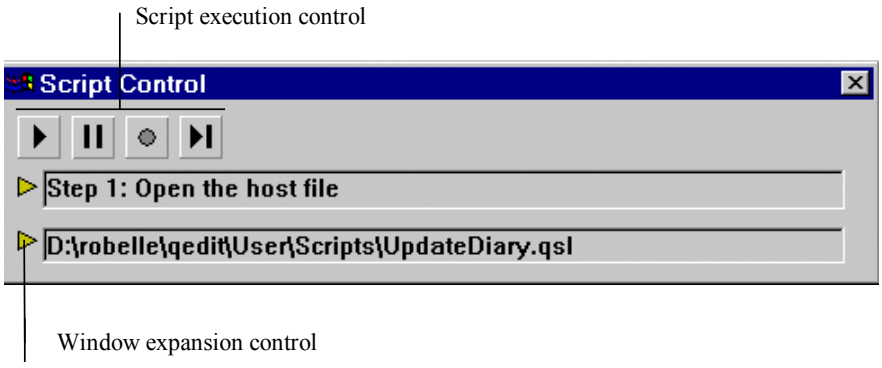
The **Save compiled script** command allows you to save the current script in compiled form. Compiled scripts are not readable and can only be executed. You have to explicitly assign a name before you can use this command.

The **Manage scripts** command brings up the corresponding dialog box. This dialog box allows you to maintain the list of loaded scripts. The **Script control** dialog box is another dialog box that provides control over script execution. It also contains list boxes for easier script debugging. You can display the **Script control** dialog box with the **Control panel** command.

Controlling Script Execution

There are different ways to execute a script. You can use the **Run** command on the **Script** menu to execute the script in the active document. If you have pre-loaded scripts with `On Command` statements, these commands also appear on the **Script** menu. Selecting a command starts the execution of the corresponding `On command` section. Once a script is started, you do not really have direct control over its execution. The **Script control** dialog box has been designed to fill that gap.

The **Script Control** dialog box is often referred to as the **Script Control** Panel or SCP.



Script control dialog box

The dialog box is independent from the document windows. You can leave it open at all times even while making changes to documents. This way, you can move it out of the way to see text changes made by the script.

You can display the dialog box using the **Control panel** command on the **Script** menu. You can also display it or hide it dynamically right from the script itself. To display the dialog box, enter:

```
qedit.showscp = true;
```

To hide it, enter:

```
qedit.showscp = false;
```

This dialog box gives controls similar to a tape recorder. There are 4 control buttons at the top, one on the of the log window and one on the left of the source code window.

Run Button

Run button

The **Run** button begins execution of the currently active script document.

Run button

Pause Button

Pause button

The **Pause** button allows you to temporarily suspend execution. To resume execution, click on the **Run** or **Step-through** button.

Pause button

Stop Button

Stop button

The **Stop** button interrupts the execution immediately. You can only continue execution by starting from the beginning using the **Run** button. If the dot is gray, the script is not executing thus the button has no effect when clicked. If the dot is red, the script is executing.

If a host command is executing at that time, it is also stopped. The output received might be incomplete.

Stop button

Step-through Button

Step-through button

The **Step-through** allows you to step through the script, one statement at a time. The execution starts the first time you click on it. You need to click twice on each statement in the script:

1. the first click moves the current statement indicator in front of the next executable statement.
2. the second click actually executes the statement.

Step-through button

Source Code Window Expansion

When you first start up Qedit and open the **Script control** dialog box, the source code window is minimized. That is, the source window is reduced to a single line. If you have run a script, it displays the name of the script. The **Source Code Window Expansion** control button at that point appears as a right-pointing arrowhead.

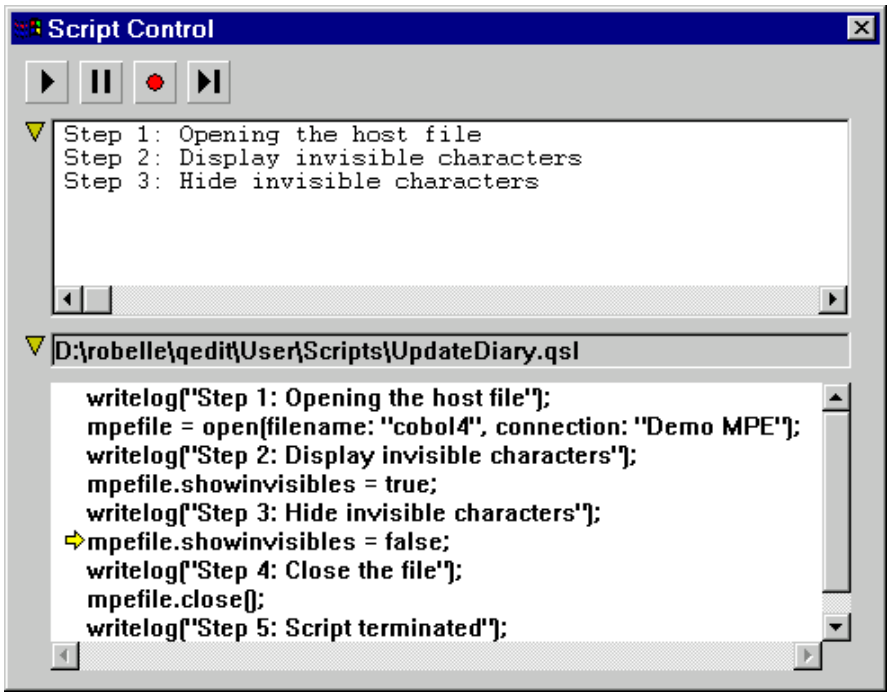
To expand the window, simply click on the **Source Code Window Expansion** control button. The arrowhead now points downward and the window displays a set of script statements.

Current statement indicator

If you use the **Step-through** button to execute the script, the current statement is identified by a small arrow.

Current statement indicator

Here is a sample **Script control** dialog box with information in the log and script windows:



Script Control panel with expanded log and script windows

Testing Your Scripts

Like any programming language, writing a program, in our case a script, is only part of the work. In a lot of instances, it is probably a small part of the overall process. Getting the script to do the work it is designed to do is a big part of the process. You should test your scripts using all kinds of scenarios. Since you are possibly dealing with critical information, you have to make sure that you are fabricating invalid data or wiping out valid data. Qedit provides a few simple tools to test and debug scripts.

Because of the nature of QSL, you could even write scripts to test other scripts, making sure the results are the ones you expected.

Interactive Debugging

The `Dialog()` built-in function allows scripts to interact with users as the scripts execute. It can be used to give feedback to the user and get dynamic information. It is also very useful to debug scripts.

The function requires at least one parameter, a string. It allows for 2 other optional parameters: a **Cancel** button and a text box for user input. It always returns a record showing which button has been used and, optionally, the text entered by the user. The element names are respectively, **Button** and **EnteredText**.

Displaying Informative Messages

In its simplest form, the `Dialog()` function displays some text in a message dialog box along with an **OK** button. The script's execution is suspended until the user hits the **OK** button. In this case, the returned value only contains the **Button** element and it is always set to 1.

The following function call:

```
result = dialog("Step 1 of 10: Opening the host file");
```

displays this dialog box:

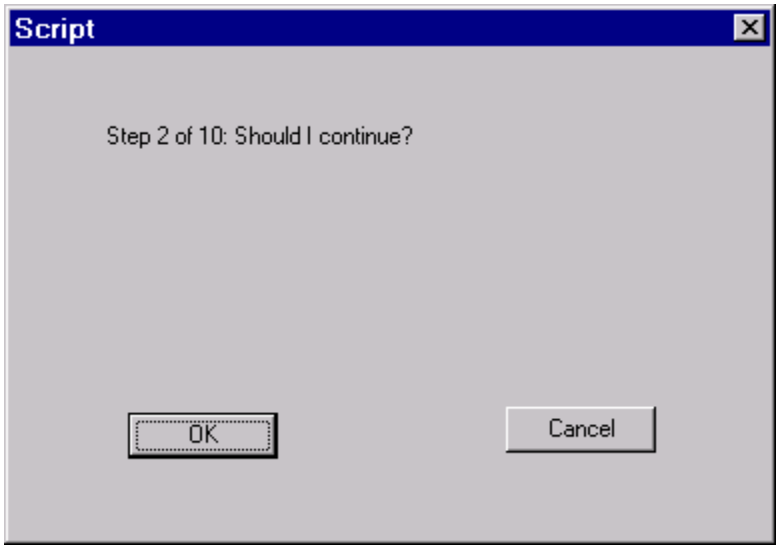
Simple message from the Dialog() function

Using a Cancel Button

You can add a **Cancel** button to the message dialog box simply by specifying a non-zero value as the second parameter to `Dialog()`. The returned value only contains the **Button** element. It is set to 1 if the user hit the **OK** button. It is set to 2 if the user hit the **Cancel** button. For example, the following code,

```
result = dialog("Step 2 of 10: Should I continue?", 1);
if result.button = 2 then     -- Cancel button
    stop;                     -- Stop execution
endif
```

produces the following dialog box:



Dialog box with Cancel button

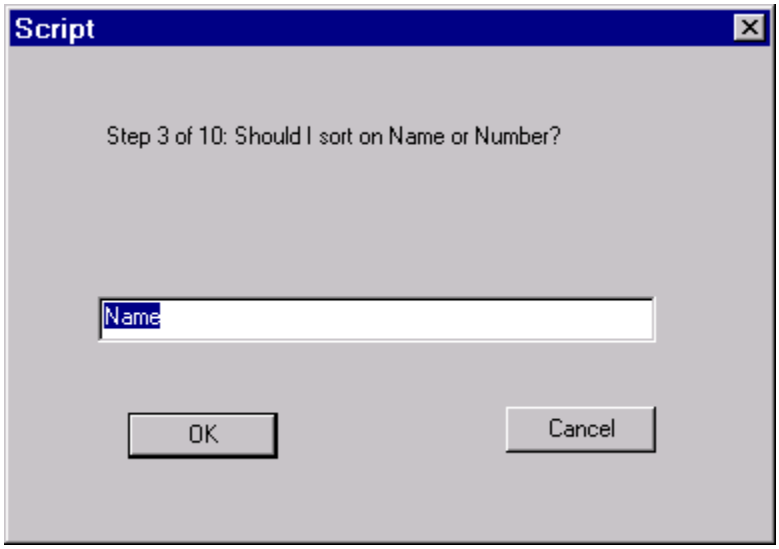
Prompting For Input

You can also get the `Dialog()` function to prompt the user for some input. The function only allows one text box. You only have to specify a third parameter on the function call. This parameter must be a string but it can be an empty string. This string represents the default value in the text box. You can still choose to have only the **OK** button or have both buttons. The returned record contains the **Button** element to indicate which button has been pressed. The record also contains the text entered in the **EnteredText** element.

If the user clicks **Cancel**, **EnteredText** always contains the default value. For example,

```
result = dialog("Step 3 of 10: Should I sort on Name or number?",
           1, "Name");
if result.button = 2 then     -- Cancel button
    stop;       -- Don't sort after all
else
    sortOn = pos( {"Name", "Number"}, result.enteredtext );
    if sortOn = 1 then     -- Sort on Name
        -- call name sort subroutine
    else
        if sortOn = 2 then     -- Sort on Number
            -- call number sort subroutine
        else
            result1 = dialog("Invalid sort option");
            stop;
        endif
    endif
endif
```

the dialog box looks like this:



Dialog box prompting for user input

Logging Messages

Using the `Dialog()` function is nice because it provides immediate feedback. These advantages are also its disadvantages. The user has to be aware of these dialogs otherwise the script might be sitting there waiting for someone to click a button. The dialog box also hides the document windows, or parts of them, and it has to be moved out of the way if the user wants to visually inspect the text.

An alternative to the `Dialog()` function is to use `Writelog()`. This function allows you to write messages to the log window of the **Script control** dialog box**.**

The **Script control** dialog box does not have to be opened in order to use `Writelog()`. Messages are written to the log and are displayed next time you open the dialog box.

Keep in mind that the log window is cleared automatically when a script starts executing.

Log Window Expansion

When you first start up Qedit and open the **Script control** dialog box, the log window is minimized. That is, the log window is reduced to a single line. If you have run a script, the log window displays the last message, if any, sent to it via `Writelog()`. The **Log Window Expansion** control button at that point appears a right-pointing arrowhead.

To expand the log window, simply click on the **Log Window Expansion** control button. The arrowhead now points downward and the log window displays the first set of messages. If there are additional logged messages, you can use the scroll bar to move up or down the list. The log window can hold approximately 500 lines.

Debugging Tips

Here are a few tips that can help you be more efficient at writing and testing scripts.

Undo Your Changes

If the script is opened, Qedit always executes the script as it currently appears in the document window. It does not use the version saved on disc. This means that you can make changes to a script and test them right away without doing a **Save**. If the changes you have made do not work, you can make more changes or you can back them out one at a time using the **Undo** tool. If you prefer to go back to the original version, you can re-open the file or use the **Revert** command on the **File** menu.

Checking Identifiers

Misspelling an identifier can cause some grief when trying to debug a script. Because of a typing mistake, Qedit might decide to create a new variable but one that is of undefined type or not initialized. Use the `Typeof()` function to check the data type of a variable. This way, you can be sure that the variable is defined the way you expect it to be.

Displaying Invisibles

If you are working with tab characters, you can not distinguish an empty area filled with spaces from another one with tab characters. The **Show invisibles** command on the **View** menu can be enabled manually to display small symbols to represent spaces, tab characters and end-of-line markers. If you want, you can enable that same function from a script using:

```
file.showinvisibles = true;
```

If you want to hide them again, use:

```
file.showinvisible = false;
```

Infinite Loops

If you think a running script is in a loop, there are two ways to stop it. Both solutions are available on the **Script Control** dialog box. To display the control panel, select the **Control panel** command on the **Script** menu. The menu commands are available even though the script is still executing.

If you want to terminate the script immediately, use the **Stop** control button.

If you want to see what part of the script is executing, use the **Pause** control button. You can then use the **Step-through** control button to go through the script, one statement at a time. You should be able to determine if the script really in a loop or simply taking a long time to complete.

Getting the Most Out of Scripting

Coding Tips and Techniques

As with any programming language, there are certain things you should do to write a script that will compile and run. This section contains a number of simple but valuable tips and techniques to help you become an expert script developer.

Checking Results of a Search

The `Find()` document method always returns a value to indicate success or failure. You should always check this result before doing anything with the found string. If the search fails i.e. no matched string, the cursor does not move. This means the **LastFoundLength** and **LastFoundColumn** properties are still pointing the last cursor position or selection. However, **LastFoundLine** is automatically set 0 after an unsuccessful search.

Checking the Cursor

The insertion point can take many forms. It can be a simple caret where the cursor sits between 2 characters in the text. It can be a simple selection where one or more characters on the same line have been selected. It can be a complex selection where many characters have been selected on multiple lines. It can also be a complex selection making up a rectangle.

You can check the **Selection** document property to determine exactly what the insertion point looks like. This property is record containing line and column coordinates and optionally a boolean value in the case of a rectangular selection. The property can have up to 3 elements:

- **Start**: a record containing the starting location coordinates. The coordinates are stored as Line and Column elements.
- **End**: a record containing the end location coordinates. The coordinates are stored as Line and Column elements.
- **Rectangular**: a boolean. If `True`, indicates a rectangular selection.

Caret Only

If there is no selection, the **Selection** property contains the **Start** element only. In this case, the length of the **Selection** property is 1.

```
{Start: {Line: 1, Column: 1}}
```

Single-line Selection Without End-of-line

If the selection is one or more characters on a single line but does not include the end-of-line marker, both **Start** and **End** elements are present. The length of the **Selection** property is 2. Of course, the **Start.Line** and **End.Line** elements are the same.

```
{Start: {Line: 1, Column: 1}, End: {Line: 1, Column: 3}}
```

Line Including End-of-line

If the selection is one or more lines and includes the end-of-line marker of the last line, the **Selection** property contains the **Start** and **End** elements. The length of the **Selection** property is 2. The **Start.Line** and **End.Line** elements are different. The **Start.Column** element always contains 1 and the **End.Column** element always contains 0.

```
{Start: {Line: 1, Column: 1}, End: {Line: 3, Column: 0}}
```

Multi-line Selection

If the selection has many characters and spans more than one line, the record contains both **Start** and **End** elements. The length of the **Selection** property is 2. The **Start.Line** and **End.Line** elements are different. In this case, however, the **Start.Column** can be 1 but does not have to. The **End.Column** should never be 0.

```
{Start: {Line: 1, Column: 1}, End: {Line: 3, Column: 3}}
```

Rectangular Selection

If the selection is rectangular, the **Selection** property contains all 3 elements: **Start**, **End** and **Rectangular**. The length of the **Selection** property is 3. The **Start.Line** and **End.Line** elements are different. The **Start.Column** can be 1 but does not have to. The **End.Column** should never be 0. The **Rectangular** element is set to `True`.

```
{Start: {Line: 1, Column: 1}, End: {Line: 3, Column: 3},
Rectangular: 1}
```

Writing For Reusability

It is good practice to write scripts in such a way that you can reuse them (or parts of them) in other scripts. Here are a few tips that will help reusability. To make a subroutine available to other scripts, you just have to load it ahead of time.

Always Name Your Scripts

Although QSL assigns default names to all the scripts, we recommend that you always assign names yourself. This way, you can pick names that are meaningful to you and your staff. With the **Group** keyword on the **Name** script attribute, you can also group scripts and subroutines under explicit categories e.g. TextUtil, FileUtil, etc.

Separate Functions From User Interface

You should separate functional sections from user interface sections. Let's use the sample script "Append Text at End of Lines" on page 79 as an example. In this

script, there are 3 subroutines. The `AppendActiveDocument()` subroutine is really the main logic. The `GetText()` subroutine (not to be confused with the `GetText()` document method) is used to prompt the user for some text. And that's all it does. The `AppendText()` subroutine is then called to do the actual insert into the current file. They are sort of specialized subroutines. This way, they all can be called independently. For example, another script could retrieve text from another source and call `AppendText()` to perform the insert operation. The latter does not mind where the text is coming from as long as it receives enough information as parameters.

Directory Iterators

Directory iterators are processed using `REPEAT` statements. A call to a `GetDirectoryIterator()` method temporarily changes the current working directory to the directory specified in the call. The information is retrieved and the iterator object is built. Then, the current working directory is changed back to the value just before the call.

Because of this, it is recommended that you do not change the current working directory, the **LocalCWD** application property for local files or call the `ChangeCWD()` connection method for host files, in the middle of the `REPEAT` block.

Limiting Random Number Range

You can use the `randseed()` and `rand()` built-in functions to get random numeric values. Both functions return a random number between 0 and 32,767. However, if you want to narrow down this range to values between 1 and *n* where *n* is a user-defined limit, you can use the following algorithm.

```
upperLimit = 25;       -- Requests values between 1 and 25
randomResult = mod(rand(), upperLimit) + 1;
```

Is the File Opened?

If you wish to determine if a particular file is already opened, you can use the `FindOpenFile()` method. The method searches the list of all currently opened files and selects the ones matching the criteria you specified. This method always returns a record. If no matching files were found, the record is empty.

```
fileOpened = findopenfile(matches: "diary.txt");
if fileOpened = {} then   -- Record is empty. No match
    result = dialog("Could not find the file!");
    stop;
endif
```

Another way to check for the same condition is to look at the number of elements in the record using the `Length()` built-in function.

```
fileOpened = findopenfile(matches: "diary.txt");
if length(fileOpened) = 0 then    -- Record is empty. No match
    result = dialog("Could not find the file!");
    stop;
endif
```

If there were one or more matching files, the record contains the corresponding file objects. However, even if the record contained only one entry, you can not use the information with document methods. For example, this would be invalid:

```
fileOpened = findopenfile(matches: "diary.txt");
if length(fileOpened) = 1 then    -- Record has 1 element
    findResult = fileOpened.find(string: "something");
endif
```

You have to extract the file object from the record into a new variable. This variable would then have the proper type. The revised code would be:

```
fileOpened = findopenfile(matches: "diary.txt");
if length(fileOpened) = 1 then    -- Record has 1 element
    fileObject = fileOpened[1];
    findResult = fileObject.find(string: "something");
endif
```

Has the File Been Opened?

If you want to control script errors like file open errors, you can use a Try-Recover block. However, there are cases where part of script has to know about the success or failure but does not have control over the Try-Recover block. For example, let's say a script calls a generic subroutine to open files. The subroutine returns an empty string in case of failure or a file object in case of success.

Checking for an empty string on a file object is invalid.

```
returnValue = openSub("diary.txt");
if returnValue <> "" then -- Invalid if file has been opened (object)
    -- Some code
endif
```

You could use workarounds like coercing the return value as in:

```
returnValue = openSub("diary.txt");
if string(returnValue) <> "" then -- Works but not elegant
    -- Some code
endif
```

This construct would not cause any script error but is not very elegant and might fail if non-empty string is returned. A better solution is simply to test for the type of return value.

```
returnValue = openSub("diary.txt");
if typeof(returnValue) = qedit.typestring then   -- String is failure
    -- Some code
    stop;
else
    if typeof(returnValue) = qedit.typeobject then   -- Object is OK
        findResult = returnValue.find(string: "something");
    endif
endif
```

With Performance in Mind

Given a specific problem to solve, you can probably write a number of different scripts that would all give the same results. However, some of these scripts might execute very slowly while others might be lightning fast. Like all programming languages, QSL has statements that require more resources than others. This section includes information on things you should do, try to avoid or not do at all. All this to get the most out of QSL.

Is There a Selection?

You should not use the following construct to see if there is a selection:

```
If file.getselectedtext() = {} then
```

This statement retrieves the selection and compares it to an empty list. If there is no selection, there is no penalty. However, if the selection included thousands of lines on a host file, Qedit would retrieve all these lines before doing the comparison. This simple statement might take a long time to complete.

The recommended approach is to check the length of the **Selection** object. This can be done with the following:

```
If length(file.selection) = 1 then    -- No text selected
    Result = dialog("No text has been selected");
else
    theSelection = file.getselectedtext();    -- Retrieves selected text
```

How Long Is The Selection?

If you expect the selection to have special characteristics like having only a few characters on one particular line, check the line coordinates stored in the `selection` document property. By looking at the start and end coordinates, you can make sure the selection you are going to be working on seems reasonable for the application.

```
If length(file.selection) = 2 then    -- Some text selected
    theSelection = file.selection;
    startOnLine = theSelection.start.line;
    endOnLine = theSelection.end.line;
    if startOnLine <> endOnLine then
        result = dialog("The selection spans more than one line!");
    else
        result = dialog("The selection is OK!");
    endif
endif
```

Working With Single Characters

You should avoid writing scripts that perform operations on single characters. Let's say you want to insert a line with an exact number of asterisks. Your first inclination might be to use:

```
Repeat for inx from 1 to n
    File.insert("*");
endrepeat
```

This would cause the `Insert()` document method to be invoked *n* times. That's very inefficient. A better approach is to build the object with all the characters you need and do a single call to the method. It would be something like this:

```
Asterisks = "";        -- Create a new string variable
Repeat for inx from 1 to n
    Asterisks = Asterisks + "*";      -- Fill up the variable
Endrepeat
File.insert(Asterisks);
```

The `Repeat` statement simply fills up the `Asterisks` string variable with the desired number of characters. All this happens in memory and is extremely fast. A single call to `Insert()` is required to create the line, no matter the length of the string.

Overloading Parameters

Qedit does not check parameter data types automatically. The subroutine can explicitly check the data types using the `Typeof()` function and determine a different course of action based on that.

```
sub mySub(parm1)
    if typeof(parm1) = qedit.typestring then
        return parm1 + "XYZ;
    endif
    if typeof(parm1) = qedit.typeinteger or
        typeof(parm1) = qedit.typefloat then
        return parm1 * 100;
    endif
    return "UNDEFINED";
endsub
```

You could then call it with:

```
returnString = mySub("ABC");  -- returns "ABCXYZ"
returnNumber = mySub(123);    -- returns 12300
```

Variables Versus Properties

QSL has been designed with performance in mind and, as such, uses different methods to get at the information in the most efficient way possible. Even with that though, there are certain operations that are more efficient than others. When dealing with properties, you have the choice to access the information directly or to copy the information into a variable which will subsequently be used for processing. Access to variables is inherently faster than direct access. If the same property value is going to be used repeatedly, you should consider copying the information into a variable. For example, the following code segments perform the same operations.

```
if file.lastfoundline <> "" then
    writelog (file.lastfoundline);
endif
```

The first segment is slightly less efficient, even though it contains a smaller number of lines, than the second.

```
lastline = file.lastfoundline ;
if lastline <> "" then
    writelog (lastline);
endif
```

Of course, there is no noticeable difference in execution time between these 2 segments. However, in a more complex script, using the second approach might make a difference.

Short-circuit Evaluation

QSL uses short-circuit evaluation when checking conditions. This means that in complex conditions, control is transferred as soon as Qedit is able to determine the outcome. It also means that not all conditions will be checked every time. In the case of an AND condition, if the first expression is false, Qedit knows the overall condition will be false, no matter what the other expression evaluates to. In the case of an OR condition, if the first expression is true, Qedit knows the overall condition will be true, no matter what the other expression evaluates to.

You can use this feature to improve performance of resource-intensive scripts. You would put the simpler conditions first in the condition list. You should also put the conditions that are more likely to fail or succeed first.

Off-the-Shelf Solutions

In this section, you will find a selection of sample scripts. These are fairly simple scripts but they should give you a good feel for the power of QSL. You will find these scripts and others in the `c:\robelle\qedit\system` directory and subdirectories. You can also connect to our web site at www.robelle.com to download all the latest scripts.

Initializing a Test File

Here is an example of the `FillFile()` subroutine used throughout the Qedit test suite. It demonstrates how you can use the `Insert()` method to add lines to a file.

```
sub FillFile(file)

    lines = {};    -- creates an empty list

    file.delete(startline: 1, endline: file.linecount);  -- empty file
    file.tabs(ClearAll: true);
    file.tabs(SetEvery: 5);

    lines = lines + "";    -- Empty line
    lines = lines + "A";
    lines = lines + "";    -- Empty line
    lines = lines + "bb";
    lines = lines + "";    -- Empty line
    lines = lines + "CCC";
    lines = lines + "";    -- Empty line
    lines = lines + "dddd";
    lines = lines + "";    -- Empty line
    lines = lines + "EEEEE";

    file.insert(at:{line: 1, column: 1}, text: lines);

endsub
```

Comparing Two Files

Below is the `CompareFile()` subroutine from the Qedit test suite. It is a general purpose subroutine that compares the contents of two files. If there are any differences, it writes an error message to the log window. `CompareFile()` returns `True` if the files are identical and `False` otherwise.

```
sub comparefile(file1, file2, testname)
returnvalue = false;
if file1.linecount <> file2.linecount
    message = "Number of lines mismatch:  ";
    message = message + file1.title;
    message = message + " has ";
    message = message + string(file1.linecount);
    message = message + " line(s)";
    message = message + " and ";
    message = message + file2.title;
    message = message + " has ";
    message = message + string(file2.linecount);
    message = message + " line(s)";
    writelog("Failure in " + testname);
    writelog(message);
else
    same = true;
    lineNum = 1;
    repeat while same and lineNum <= file1.linecount
        lineFile1 = file1.gettext(startline: lineNum,
                    endline: lineNum);
        lineFile2 = file2.gettext(startline: lineNum,
                    endline: lineNum);
        lineFile1 = lineFile1[1];
        lineFile2 = lineFile2[1];
        if lineFile1 <> lineFile2
            finished = false;
            inx      = 1;
            if length(lineFile1) <> length(lineFile2)
                message = "Files at line ";
                message = message + string(lineNum);
                message = message + " are not the same length. ";
                message = message + file1.title;
                message = message + " is '";
                message = message + lineFile1;
                message = message + "' and ";
                message = message + file2.title;
                message = message + " is '";
                message = message + lineFile2;
                message = message + "'";
                writelog("Failure in " + testname);
                writelog(message);
                finished = true;
            endif
            repeat while not finished
                if inx > length(lineFile1)
                    finished = true;
                endif
                if lineFile1[inx] <> lineFile2[inx]
                    message = "Mismatch at column " + string(inx);
                    message = message + " File1 character = '";
                    message = message + lineFile1[inx];
                    message = message + "' (";
                    message = message + string(code(lineFile1[inx]));
                    message = message + ") File2 character = '";
                    message = message + lineFile2[inx];
                    message = message + "' (";
                    message = message + string(code(lineFile2[inx]));
                    message = message + ")";
                    writelog("Failure in " + testname);
                    writelog(message);
                    finished = true;
                endif
                inx = inx + 1;
            endrepeat
            same = false;
        endif
        lineNum = lineNum + 1;
    endrepeat

    if same then
```

```
        returnvalue =  true;
    endif
endif

return returnvalue;

endsub
```

Insert a Signature or a Timestamp

This script is an easy way to insert some predefined text into a file. The script adds a submenu called **Macros** to the **Script** menu. On that submenu, there are 2 commands:

- **Signature**: inserts 3 lines at the cursor position in the document window
- **Timestamp**: inserts the current date and time at the cursor position in the document window into a file

The **Signature** command inserts lines of text that you have typed right in the script. It can be as long as you need and in the format that you want.

After running the **Timestamp** command, you should see the PC's current date. The date and time are in the short format.

The script should be saved in the `c:\robelle\qedit\user\autoload` directory e.g. `macros.qsl`. If you are already in Qedit for Windows, you should exit and restart it.

```
name macros;

Property CurrentFile = 0; -- shared global without a mainline

sub CheckFileOpen
    flag = false;
    file = qedit.activefile;  -- we need an object to insert into
    if typeof(file)=qedit.typeundefined then
        result=dialog("You must have a file open");
    else
        CurrentFile = file;    -- update shared Global variable
        flag = true;
    endif;
    return flag;
endsub

sub timestamp
    if CheckFileOpen() then
        timestamp = datetime();
        timestamp = timestamp.fmtshortdatetime();
        CurrentFile.insert(timestamp);    -- CurrentFile is global
    endif
endsub

on command "&Timestamp"
    timestamp();
endon

sub signature
    if CheckFileOpen() then
        sig = {"Ralph J. Smith",
               "Email: smithr@ourfriendlyfirm.com,
               "Telephone: 345-678-9012"};
        -- create a list of 3 strings
        CurrentFile.insert(sig);  -- CurrentFile is shared global file
    endif
endsub

on command "&Signature"
    signature();
endon
```

Insert a Rectangular Selection

In Qedit for Windows, there is no single command to insert a rectangular selection. You have to create an empty space using the **Insert column** command**,** for example, select the empty area and then paste the clipboard. That's tedious. The following script allows you to do the same operation with one command.

If you want to see the **PasteInBox** command on the **Script** menu, put the script file in the `c:\robelle\qedit\user\autoload` directory.

```
name PasteInBox;
-- insert Clipboard into an area the size of the rectangular selection

Property CurrentFile = 0; -- shared global variable
Property CurrentSel = 0;

sub CheckFileOpen
    flag = false;
    file = qedit.activefile;  -- we need an object to insert into
    if typeof(file)=qedit.typeundefined then
        result=dialog("You must have a file open");
    else
        CurrentFile = file;    -- update shared Global variable
        flag = true;
    endif
    return flag;
    endsub

sub CheckSelection
    flag = false;
    if CheckFileOpen() then
        s = CurrentFile.selection;
        if length(s) = 1 then  -- Oops, only a caret is active
            r = dialog("You must have an active selection to fill.");
        else
            CurrentSel = s;
            flag = true;
        endif
    endif
    return flag;
endsub

-- Mainline

if CheckSelection() then
    startsel = CurrentSel.start;
    startl = startsel.line;
    startc = startsel.column;
    endsel = CurrentSel.end;  -- unknown property if no selection
    endl = endsel.line;
    endc = endsel.column;
    if length(CurrentSel) = 2 then    --- oops, not rectangular
        r = dialog("PasteInBox only works on rectangles.
                Use Control-Drag.");
        stop;
    endif
    width = endc - startc + 1;    -- insert some spaces to replace
    spaces = "";
    repeat for x from 1 to width by 1
        spaces = spaces + " ";
    endrepeat
    -- insert a new rectangle of same size, consisting of spaces
    CurrentFile.insertcolumn(startline: startl, endline: endl,
            atcolumn: startc, text: spaces);
    -- rectangle should still be selected, so now paste-replace
    CurrentFile.Paste();
    -- Cancel selection, Position caret at upper left corner:
    CurrentFile.select(startline: startl, startcolumn: startc);
endif
```

Fill a Rectangular Area With Asterisks

The following scripts allow you to fill a predefined rectangular area with asterisks. The first one inserts a block of asterisks as long and wide as the currently selected area.

```
name InsertStars;
-- insert a rectangle of stars

Property CurrentFile = 0; -- shared global variable
Property CurrentSel = 0;

sub CheckFileOpen
    flag = false;
    file = qedit.activefile;  -- we need an object to insert into
    if typeof(file)=qedit.typeundefined then
        result = dialog("You must have a file open");
    else
        CurrentFile = file;    -- update shared Global variable
        flag = true;
    endif
    return flag;
endsub

sub CheckSelection
    flag = false;
    if CheckFileOpen() then
        s = CurrentFile.selection;
        if length(s) = 1 then  -- Oops, only a caret is active
            r = dialog("You must have an active selection to fill.");
        else
            CurrentSel = s;
            flag = true;
        endif
    endif
    return flag;
endsub

-- Mainline
if CheckSelection() then
    startsel = CurrentSel.start;
    startl = startsel.line;
    startc = startsel.column;
    endsel = CurrentSel.end; -- unknown property if no selection
    endl = endsel.line;
    endc = endsel.column;
    if length(CurrentSel) = 2 then --- oops, not rectangular
        r = dialog("STARS only works on rectangles.
                Use Control-Drag.");
        stop;
    endif;
    width = endc - startc + 1;
    stars = "";
    repeat for x from 1 to width by 1
        stars = stars + "*";
    endrepeat
    r = CurrentFile.insertcolumn(startline: startl, endline: endl,
                atcolumn: startc, text: stars);
    -- Cancel selection, Position caret at upper left corner:
    CurrentFile.select(startline: startl, startcolumn: startc);
endif
```

This next script also fills an area with asterisks but this one replaces the contents of the selected area.

```
name ReplaceStars;
-- replace a rectangular selection with asterisks

Property CurrentFile = 0; -- shared global variable
Property CurrentSel = 0;

sub CheckFileOpen
    flag = false;
    file = qedit.activefile; -- we need an object to insert into
    if typeof(file)=qedit.typeundefined then
        result = dialog("You must have a file open");
    else
        CurrentFile = file; -- update shared Global variable
        flag = true;
    endif
    return flag;
    endsub

sub CheckSelection
    flag = false;
    if CheckFileOpen() then
        s = CurrentFile.selection;
        if length(s) = 1 then -- Oops, only a caret is active
            r = dialog("You must have an active selection to fill.");
        else
            CurrentSel = s;
            flag = true;
        endif
    endif
    return flag;
endsub

-- Mainline
if CheckSelection() then
    startsel = CurrentSel.start;
    startl = startsel.line;
    startc = startsel.column;
    endsel = CurrentSel.end;  -- unknown property if no selection
    endl = endsel.line;
    endc = endsel.column;
    if length(CurrentSel) = 2 then --- oops, not rectangular
        r = dialog("STARS only works on rectangles.
                Use Control-Drag.");
        stop;
    endif
    CurrentFile.delete(); -- delete current selection
    width = endc - startc + 1;
    stars = "";
    repeat for x from 1 to width by 1
        stars = stars + "*";
    endrepeat
    -- insert a new rectangle of same size, consisting of stars
    r = CurrentFile.insertcolumn(startline: startl, endline: endl,
            atcolumn: startc, text: stars);
    -- Cancel selection, Position caret at upper left corner:
    CurrentFile.select(startline: startl, startcolumn: startc);
endif
```

If you want to see the **InsertStars** or **ReplaceStars** commands on the **Script** menu, put the script files in the `c:\robelle\qedit\user\autoload` directory.

Draw a Box

This script draws a box with asterisks around a rectangular selection. It actually replaces the characters that make up the top, bottom, left and right edges.

```
name box;
-- draw a box of asterisks around a rectangular selection

Property CurrentFile = 0; -- shared global variable
Property CurrentSel = 0;

sub CheckFileOpen
    flag = false;
    file = qedit.activefile; -- we need an object to insert into
    if typeof(file)=qedit.typeundefined then
        result = dialog("You must have a file open");
    else
        CurrentFile = file; -- update shared Global variable
        flag = true;
    endif
    return flag;
endsub

sub CheckSelection
    flag = false;
    if CheckFileOpen() then
        s = CurrentFile.selection;
        if length(s) = 1 then -- Oops, only a caret is active
            r = dialog("You must have an active selection to fill.");
        else
            CurrentSel = s;
            flag = true;
        endif
    endif
    return flag;
endsub

sub make
    if CheckSelection() then
        startsel = CurrentSel.start;
        startl = startsel.line;
        startc = startsel.column;
        endsel = CurrentSel.end; -- unknown property if no selection
        endl = endsel.line;
        endc = endsel.column;
        if length(CurrentSel) = 2 then --- oops, not rectangular
            r = dialog("Box only works on rectangles.
                    Use Control-Drag.");
            stop;
        endif
        width = endc - startc + 1;
        stars = "";
        repeat for x from 1 to width by 1
            stars = stars + "*";
        endrepeat
        -- delete the top of the box
        CurrentFile.delete(startline: startl, endline: startl,
                startcolumn: startc, endcolumn: endc,
                rectangular: true);
        -- insert the new top of the box
        CurrentFile.insertcolumn(startline: startl, endline: startl,
        atcolumn: startc, text: stars);
        -- delete the bottom of the box
        CurrentFile.delete(startline: endl, endline: endl,
                startcolumn: startc, endcolumn: endc,
                rectangular: true); -- insert the new bottom of the box
        CurrentFile.insertcolumn(startline: endl, endline: endl,
                atcolumn: startc, text: stars);
        height = endl - startl + 1;
        stars = "*";
        -- delete the left side of box
        CurrentFile.delete(startline: startl, endline: endl,
                startcolumn: startc, endcolumn: startc,
                rectangular: true);
        -- insert left side of box
        CurrentFile.insertcolumn(startline: startl, endline: endl,
```

```
                atcolumn: startc, text: stars);
        -- delete the right side of box
        CurrentFile.delete(startline: startl, endline: endl,
                startcolumn: endc, endcolumn: endc, rectangular: true);
        -- insert right side of box
        CurrentFile.insertcolumn(startline: startl, endline: endl,
                atcolumn: endc, text: stars);
        -- Cancel selection, Position caret at upper left corner:
        CurrentFile.select(startline: startl, startcolumn: startc);
    endif
endsub

make(); -- dummy mainline. make is exported as a callable method
```

If you want to see the **Box** command on the **Script** menu, put the script file in the `c:\robelle\qedit\user\autoload` directory.

Copying Files Between Systems

It's easy to automate copying files from one machine to another. This script can be very useful if you have to copy files on a regular basis.

```
name CopyFiles;

sub docopy(localfilename, serverconnection, serverfilename)
    localfile = open(localfilename);
    localfile.select(startline: 1, endline: localfile.linecount);
    localfile.copy();
    serverfile = open(connection: serverconnection,
            filename: serverfilename);
    serverfile.select(startline: 1, endline: serverfile.linecount);
    serverfile.delete();
    serverfile.paste();
    localfile.close();
    serverfile.close(forceoverwrite: true);
endsub

docopy("diary.txt", "Production UX", "Diary");
docopy("cobolsrc", "Source MPE", "cobol.src");
```

Prompt Before Replacing

Currently, Qedit replaces all occurrences of a string in the specified range. You can use the following script to confirm which string occurrence to replace.

```
name ReplacePrompt;

filename = "";
connection = "";
file = false;

sub getFile ()
    returnValue = False;
    result = dialog("Which file to search?",2,"");
    if result.button = 1 then
        filename = result.enteredtext;
        result = dialog("Host Connection, if not local?",2,"");
        if result.button = 1 then
            connection = result.enteredtext;
            returnValue = True;
        endif
    endif
    return returnValue;
endsub

result = dialog("String to Find and Replace?",2,"");
if result.button = 2 then
    stop;
endif
target = result.enteredtext;
if (target = "") then
    stop;
endif;

result = dialog("Replacement string?",2,"");
if result.button = 2 then
    stop;
endif
replacement = result.enteredtext;
if (replacement = "") then
    stop;
endif;

repeat while (typeof(file)<>qedit.typeobject)
    if not getFile()
        stop;
    endif;

    try
        file = open(connection: connection, filename: filename);
    recover
        result = dialog("Unable to open that file. Try again.");
    endtry
endrepeat

file.activate(); -- bring the file to the front
file.select(line:1, column:1);    -- may be at previous location
if (file.find(regexp: target)) then
    goFlag = true;
else
    goFlag = false;
    result = dialog("No matches in this file");
endif

repeat while (goFlag <> False)
    result = dialog("Replace this one with " + replacement + "?",2);
    if result.button=1 then   -- Do this update
    result = file.find(regexp: target, selectiononly: true,
            replacewith: replacement);
        if not (result) then
            result = dialog("Error doing replacement!");
            goFlag = false;
        endif
    endif
    if (goFlag) and not (file.find(regexp: target)) then
        goFlag = false;
```

```
        result = dialog("End of file.");
    endif
endrepeat;

file.select(line: 1, column: 1);  -- cancel last selection
file.close(forceoverwrite: true);
```

Append Text at End of Lines

This script prompts the user for a text string. Then, the script proceeds to append the entered text at the end of each selected line in a file. If there are no selection, the text is appended at the end of all the lines.

```
name QSLUtil Append group "Utility" command "Append Text";

sub appendtext(file, startline, endline, text)

    repeat for lineno from startline to endline

        -- Position the caret at the end of the line:
        file.select(startline: lineno, startcolumn: file.recordlength);
        -- Insert the new text at the current position
        file.insert(text);
    endrepeat

endsub

sub gettext

    result = dialog("Enter the text to append", 1, "");
    if result.button = 1 then
        return result.enteredtext;
    else
        return ""
    endif

endsub

sub appendactivedocument

    appendfile = qedit.activefile;
    if not exists(appendfile) then
        result = dialog("There is no document to append lines to");
    else
        thetext = gettext();
        if length(thetext) 0 then
            theselection = appendfile.selection;
            startcolumn = theselection.start.column;
            if length(theselection) = 1 then
                -- Append to the entire file
                startline = 1;
                endline = appendfile.linecount;
            else
                startline = theselection.start.line;
                if theselection.end.column = 0 then
                    endline = theselection.end.line - 1;
                else
                    endline = theselection.end.line;
                endif
            endif
            appendtext(appendfile, startline, endline, thetext);
            -- Restore the selection
            appendfile.select(range: theselection);
        endif
    endif
endsub

appendactivedocument();
```

Displaying Information From Directory Iterators

This script displays the directory tree structure starting from a specific directory. The script prompts for a connection name and a start directory. It then proceeds to display all subdirectories at lower levels from that point. The script uses the recursive functionality of QSL subroutines to walk up and down the structure.

If the connection name is empty, it assumes the directory is on a local drive.

If the start directory is empty, Qedit starts from the current working directory. The start directory can be an absolute or relative path. If you use a relative path, you have

to know what is the current working directory. Failing that, you might get a listing for another directory or get an error if the directory does not exist at that location.

```
sub getinitialinfo(promptmsg)

    result = dialog(promptmsg, 1, "");

    if result.button = 1 then
        return result.enteredtext;
    else
        return ""
    endif

endsub

sub getdirlisting(connection, startdir, levelindicator)

    levelindicator = "   " + levelindicator;

    if string(connection) = "" then
        localdir = getdirectoryiterator(startdir);
        dirseparator = "\";
    else
        localdir = connection.getdirectoryiterator(startdir);
        dirseparator = "/";
    endif

    repeat for direntry in localdir
        if direntry.canonicaltype = "directory" and
            direntry.name <> "." and  -- Otherwise goes into a loop
            direntry.name <> ".." then
            writelog(levelindicator + direntry.name);
            nextlevel = direntry.path + dirseparator + direntry.name;
            getdirlisting(connection, nextlevel, levelindicator);
        endif
    endrepeat

endsub

-- Mainline

startconn = getinitialinfo("Enter connection name:");

startdir = getinitialinfo("Enter starting directory:");

if startconn = "" then
    writelog ("Listing for local " + startdir);
    connobject = "";
else
    writelog ("Listing for " + startdir + " on " + startconn);
    connobject = openconnection(startconn);
endif

getdirlisting(connobject, startdir, "");
```

Directories at lower levels are automatically indented. For example, the output for a local directory would appear as:

```
Listing for c:\personal
    Expenses
        1999
            January
            February
    Agenda
    Pictures
        Family
        Work
        Travel
```

Robelle Script Library

Robelle provides utility scripts to expand Qedit's built-in feature set. These scripts are stored in the **System** directory.

Sort Lines

This script is in the **Autoload** directory thus it is automatically loaded and available from the **Robelle** submenu of the **Script** menu. It can be used to arrange lines of text in sorted order.

The script sorts the selected lines. If it's a regular selection, whole lines are used as the sort keys. If it's a rectangular selection, the selected columns only are used as the sort keys.

The name of the script is `QSLUtilSort`. Since the script is automatically loaded, all its subroutines are available to other scripts as methods.

SortActiveDocument() Method

The `SortActiveDocument()` method is the main entry point to the `QSLUtilSort` script. This entry point assumes that you want to sort the active document.

`SortActiveDocument()` accepts only one parameter. The parameter is a boolean to indicate whether you want the lines to be sorted in ascending or descending order. The default value is `True` and corresponds to ascending order. A value of `False` sorts in descending order.

`SortActiveDocument()` does some assertion checks before calling the subroutines that do the actual work. For example, `SortActiveDocument()` ensures there is actually an active document and that some lines have been selected. If everything looks fine, it calls the `SortLines()` method.

```
QSLUtilSort.SortActiveDocument(false);   -- Sort in descending order
```

SortLines() Method

This method is a secondary entry point to the `QSLUtilSort` script. It can be used to sort files other than the active document. It assumes the calling script has taken all the necessary steps to ensure the information is correct e.g. the start and end line numbers are correct.

`SortLines()` accepts 4 parameters:

- **FileHandle** (object): this is the file object corresponding to the file you want to sort
- **StartLine** (integer): this is the line number of the first line to sort

- **EndLine** (integer): this is the line number of the last line to sort
- **Ascending** (boolean): this indicates the desired sort order. It is optional and the default is `True` (ascending).

```
QSLUtilSort.SortLines(file, 1, file.linecount, true);
    -- Sort all lines in the file in ascending order

selectionStarts = file.selection.start.line;
selectionEnds = file.selection.end.line;
QSLUtilSort.SortLines(file, selectionStarts, selectionEnds, false);
    -- Sort all lines in the selection in descending order

file.select(startline: selectionStarts, startcolumn: 10,
            endline: selectionEnds, endcolumn: 20);
QSLUtilSort.SortLines(file, selectionStarts, selectionEnds, true);
    -- Sort all lines in the selection in ascending order
    -- Sort on text in columns 10 to 20
```

List Lines

There are 4 scripts in the **Autoload** subdirectory performing similar operations. They are loaded automatically out of the **System** directory. They can be used to list lines which contain a specified string, regular expression or pattern.

The `ListAll` script searches the current document only. The ListInclude, ListCopy and ListUse scripts also scan referenced files. Referenced files can be Include files. These are identified by `$include`, `!include`, `#include` or `.include` statements. The `ListInclude` script is used for this type of search. Other referenced files can be Use files. These are identified by `Use` statements and can be searched using the `ListUse` script. Referenced files in COBOL source files are identified on `Copy` statements. The `ListCopy` script can be used to scan them.

All these scripts use similar logic to extract and display the information. Informative messages are listed in the log window of the **Script Control** dialog box. These messages include the version number, error messages (if any) and a termination message. The actual results are displayed in a local file. The scripts open a new file for each execution. The scriptnames are `QSLUtilListInclude`, `QSLUtilListUse` and `QSLUtilListCopy` respectively.

The `ListAll` script contains the main search logic. Its scriptname is QSLUtilListAll. Once the script is loaded, all its subroutines are available to other scripts as methods. The main subroutine is called `ProcessListRequest`. The other 3 scripts are really shell scripts and are calling this method to perform the actual search.

ProcessListRequest() Method

This is the main entry point to the `QSLUtilListInclude` script. It requires 2 parameters:

- **SearchType** (integer): this indicates the type of search to use: simple string, regular expression or pattern.
- **IncludeFile** (string): this indicates whether or not to scan referenced files. This parameter is required. The default is a null string (do not search referenced files). The other possible values are "Copy", "Include" and "Use".

Possible values for **SearchType** are:

- **1**: String

- **2**: Regular expression
- **3**: Pattern

`ProcessListRequest()` performs some assertion checks e.g. there is an active document, before calling the appropriate methods. If the **IncludeFile** parameter is not supplied or contains a null string, only the active document is searched. If the **IncludeFile** parameter is supplied, the active document and all related referenced files, if any, are searched. The script always performs a caseless search no matter which **SearchType** is requested. The subroutine then calls the `ListAll()` method.

```
localFile = open("C:\personal\diary.txt");
QSLUtilListAll.ProcessListRequest(1, "Include");
    -- Search for a string in Diary.txt
    -- Scan referenced $Include files, if any
```

ListAll() Method

This is a secondary entry point to the `QSLUtilListAll` script. It can be used by other scripts to start a search without user's intervention. The `ListAll()` method has 5 parameters:

- **FileHandle** (object): file object corresponding to the document you want to start searching from
- **IncludeOption** (integer): indicates whether Include files should be scanned. Should match **SearchReferenced** parameter of `FindAll()` document method.
- **SearchString** (string): the search string you want to search for
- **SearchType** (integer): the type of search string passed in **SearchString**
- **OutputFile** (object): a file object where the search results should go. This parameter is optional. If it is not specified, the script opens a new file automatically.

Possible values for **SearchType** are:

- **1**: String
- **2**: Regular expression
- **3**: Pattern

```
localFile = open("C:\personal\diary.txt");
QSLUtilListAll.ListInclude(localFile, 1, "Dentist", 1);
    -- Search for the string "Dentist" in Diary.txt and
    -- Scan referenced $Include files, if any.
    -- Results are sent to a new file.
```

MPE Compilers

This script is in the **Scripts** directory. It is designed to be loaded and then be available from the **Robelle** submenu of the **Script** menu. You can load it manually using the **Manage Scripts** command of the **Script** menu. If you wish to have it loaded automatically, copy it to the **Autoload** subdirectory of the **User** directory. It can be used to compile source programs written in most programming languages supported on MPE.

Informative messages are listed in the log window of the **Script Control** dialog box. These messages include the version number, progress messages, error messages (if any) and a termination message.

The output from the compilers is written to a host-temporary file called `QSLCOMPO`. When the compilation terminates, the script opens the file and selects the last line.

The name of the script is `QSLMPECompile`. Once the script is loaded, all its subroutines are available to other scripts as methods.

DoMPECompile() Method

This is the main entry point to the `QSLMPECompile` script. It requires only one parameter:

- **CompileCommand** (string): MPE command name used to invoke the compiler.

The script does not validate the information. It uses the parameter value to build the complete the MPE command. That command has the following syntax:

```
CompileCommand currentFilename,,outputFilename
```

where:

- **CompileCommand** is the compiler command past as parameter. For example, to compile a Cobol source file using Cobol85, the parameter would be `COB85XL`.
- **currentFilename** is the name of the currently active document. The script fills that in automatically.
- **outputFilename** is the name of the file that will receive the compilation results. The script uses `QSLCOMPO`.

Let's assume the active document is COBPRG1.SRC.DEVACCT and we want to use the Cobol85 compiler, you would call DoMPECompile with:

```
mpesrc = open(connection: "Dev MPE", filename: "cobprg1.src.devacct");
QSLMPECompile.DoMPECompile("cob85xl");
```

Reference

Overview

This section gives a list of all the statements that are part of the scripting language itself. They are mostly used for controlling the flow of a script, handling exceptions and setting script attributes.

Script Attributes

Name

Assigns a name to the script. If not specified, Qedit uses the filename.

```
name scriptname command "commandString" group "groupName";
```

The script name is case-sensitive and the only required parameter. The `Command` keyword allows you to specify exactly how the command is going to appear on the **Script** menu. You specify a mnemonic by prefixing one of the characters with an ampersand. For example,

```
name Exercise04 command "Exercise &4";
```

The script name is `Exercise04` but the **Script** menu will show:

Exercise 4

Notice the number 4 is underlined. This means you can invoke the script simply by typing ALT+S, 4 on your keyboard.

If you want a script to appear as a command with other scripts on a submenu, use the `Group` keyword.

Option Private

When you create a compiled script using the **Save compiled script** command, by default, Qedit saves the script code in compiled form, ready for execution and also saves the source lines in encoded form so you can still use the **Step-through** debugging feature in the **Script Control** dialog box.

If you do not want to get the encoded source lines, simply enter the `Option Private` attribute anywhere in the script mainline (outer block).

```
option private;    -- Do not save source code
name "Test Script";
```

Property

Defines global variables that can be used in all scripts. If the `Property` keyword is not specified and the variable is declared outside a subroutine block, the variable is global to the script but can not be accessed by external scripts.

General syntax:

```
property variableName = initialValue;
```

A variable name must be provided but the initial value is optional.

Control Statements

Break

Interrupts a `REPEAT` loop. Normally, all statements in the `REPEAT` block are executed until the condition is met. If you wish to terminate the loop before this happens, enter a `BREAK` statement.

General syntax:

```
break;
```

Call

Explicitly execute a subroutine or method. Normally, subroutines and methods can be executed implicitly and not mistaken for variable names if they are followed by parentheses. If you do not wish to specify the parentheses, you can use the `CALL` statement. A CALL statement can also be used to explicitly ignore a possible return value.

General syntax:

```
call subroutine(parameter);
```

Error

Forces Qedit to report an error and take appropriate actions.

General syntax:

```
Error (errorvariable);
```

The parameter is optional. If specified, it can be a string or a record. If it's a string, Qedit simply displays a message dialog box with the string. For example, if you code:

```
Error ("You should not have hit that key!");
```

the message dialog box would show:

User-generated error dialog box

If the parameter is a record, it has to have the same structure as for Qedit errors returned by a `TRY/RECOVER` block. The line number and script name fields are replaced by the actual line number and script name, so the information provided by the user in these 2 fields is ignored.

IF, Else and Endif

Indicates statements that need to be executed under certain conditions.

General syntax:

```
if condition then
    -- statements to be executed if condition is true
else
    -- statements to be executed if condition is false
endif
```

The `then` keyword is optional. The `else` keyword and block are optional. `Endif` is required.

Invoke

Executes another script's mainline.

```
invoke "scriptName";
```

The called script must have mainline code. The **scriptName** parameter is actually the filename. If it is a local file, it only needs to follow the file system's pathname syntax. If it's a host file, **scriptName** has 2 parts separated by a colon:

- the connection name
- the actual filename

Of course, the connection name must exist in your current Qedit configuration.

For example, to execute the `globals.qsl` script on the `Dev UX` connection, the command would be:

```
invoke "Dev UX:/home/dev/scripts/globals.qsl";
```

On Command and Endon

Executes statements when certain events occur.

Adds *commandname* to the **Script** menu and executes the block if the menu command has been selected:

```
on Command commandname
    -- statements to be executed
endon
```

Repeat and Endrepeat

Execute statements repeatedly until a condition is met.

Repeat statements while the condition is true:

```
repeat while condition
    -- statements to be executed
endrepeat
```

Repeat statements until the condition is true (and always executes the block at least once):

```
repeat until condition
    -- statements to be executed
endrepeat
```

Repeat statements setting *variable* to *start*, and then adding *step* after every iteration until the value of *variable* is greater than *end*:

```
repeat for variable from start to end
    -- statements to be executed
endrepeat
repeat for variable from start to end by step
    -- statements to be executed
endrepeat
```

Repeat statements for each item in *list:*

```
repeat for variable in list
```

All keywords are required. The *list* can also be an iterator object such as a directory iterator.

Return

Used inside subroutines to return a value to the calling script. The subroutine stops executing immediately and the specified value is returned to the calling statement.

```
return value;
```

The returned value is optional. If not specified, QSL returns a variable of undefined type.

Stop

Terminates execution of the current script. Control returns to the calling script or to Qedit depending on how the script was invoked.

Sub and Endsub

Define a subroutine. Subroutine can receive parameter, execute some code and return a result.

General syntax:

```
Sub subroutineName (parameterlist)
    -- statements to be executed
    return value
endsub
```

The parameter list is optional. For a subroutine without parameters, do not include the parentheses. The `return` statement is optional.

Try and Recover

Executes statements and, in case of problems, executes a series of statements.

General syntax:

```
try
    -- statements to be executed
recover (errorrecord)
    -- statements to be executed if an error is detected
endtry
```

In case of an error, the variable **errorrecord** is created. The variable is a record and contains information about the error.

- **ErrorNumber**: an error number
- **LineNumber**: the line number in the script where the error occurred. It is zero if it is unknown.
- **NumericParameter**: Numeric parameter of the error.
- **StringParameter**: String parameter of the error.
- **MessageText**: The localized error message that you would normally see if you did not trap the error.
- **Script**: Script name where the error occurred. It is blank if it is unknown.

See "Exception Handlers" on page 18 for details and examples of `Try/Recover` blocks.

Built-in functions

Built-in functions look like subroutines or application methods because they have parentheses with parameters after them. However, built-in functions can not be prefixed with an object name such as a script name. For example, `open()` is an application method so you can use `qedit.open(file)`. On the other hand, `exists()` is a built-in function and `qedit.exists(variable)` is invalid.

Also, unlike method calls, you can not use named parameters in a built-in function call. Functions used positional parameter lists only. For example, when calling a method, you could write:

```
file.insert(at: {line: 1, column: 10}, text: "This is new text");
```

At and **Text** are parameter names. When calling a built-in function such as `dialog()`, you can not write:

```
result = dialog(message: "This is my message", buttons: 2)
```

You have to use:

```
result = dialog("This is my message, 2);
```

Character()

This function allows you to get the character corresponding to a numeric value. The value is taken from the Windows code representation.

```
tabCharacter = character(9);
line = tabCharacter + "This line starts with a tab";
file.insert(line);
```

Code()

If you need to see what is the numeric equivalent of a character, use the `Code()` function. This is always the numeric equivalent in the Windows character set.

```
firstCharacter = line[1];
thisCharacter = code(firstCharacter);
result = dialog("The numeric value is:" + string(thisCharacter));
```

Dialog()

The dialog function allows you to perform simple interaction with the user. You can display a message and request some information from the user. You must provide a variable to receive the result. The `Dialog()` function calls can have up to 3 parameters:

- **Message text**: enter the text you want to display in the message dialog box.
- **Number of buttons**: the default is to display only the **OK** button. Enter a value greater than 0 to get the **OK** and **Cancel** buttons.
- **Default answer**: if you wish to prompt the user for some information, specify something as the third parameter. This parameter then appears as the default value.

If you specified only the first or second parameters, the returned value is a number representing the button clicked by the user. If the user chose the **OK** button, the value is 1. If the user chose the **Cancel** button, the value is 2.

If you prompted the user for something, the function returns a list which includes the button (**Button**) and the user input (**EnteredText**). If the user clicked the **Cancel** button, the default value is returned.

To display a message, simply use:

```
result = dialog("This is my message!");
```

To display a message with both buttons and a text box with a default value, use:

```
result = dialog("What is your name?", 1, "John Doe");
if result.button = 1 then     -- OK button used
    userName = result.enteredText;
else
    userName = "NO ENTRY";    -- User clicked on Cancel
endif
```

See "Interactive Debugging" on page 58 for details on how to use the `Dialog()` function.

Downshift()

This function changes all uppercase characters in a string to the equivalent lowercase characters. Qedit takes into account the PC's local configuration and handles extended characters such as accents in French.

```
myString = "AbCdE";
lowerMyString = downshift(myString);  -- returns "abcde"
```

Exists()

This function tells you if a variable exists or not. This function can not be used to check for the existence of a file.

```
if exists(myVar) then
    result = dialog("The variable myVar exists");
else
    result = dialog("The variable myVar does not exist!");
endif
```

Integer()

Numeric variables are typically stored as 64-bit IEEE floating point values. If you need to get at the integer portion of these values, you can convert them to a whole number using this function. This is actually a type coercion operation. The new variable is of Integer type, not float. Also note that the integer part is not rounded during this operation i.e. 123.999 becomes 123.

```
myNumber = 123.456789;
myInteger = integer(myNumber);
```

Length()

The `Length()` function returns the number of characters in a string or the number of elements in a record. This function can not be used on other data types such file objects.

```
myName = "John Doe";
lengthMyName = length(myName);    -- returns 8

myList = { 1, 3, 5, 7, 9 };
lengthMyList = length(myList);    -- returns 5
```

LTrim()

This function removes all leading spaces from a string. See also "RTrim()" on page 94 and "Trim()" on page 94.

```
strvar = "   abc   ";
writelog(trim(str));      -- returns "abc   "
```

Num()

In most cases, Qedit converts string variables to numbers whenever appropriate. There might be times when this does not happen. In these situations, you can use the `Num()` function to explicitly convert a string to a number. This is a type coercion operation.

```
myString = "123";
myNumber = num(myString) * 1000;
```

If the string starts with a numeric digit but contains invalid digits after that e.g. 123abc456, the `Num()` function converts as many characters as it can. With this sample value, **myNumber** would contain 123. If the string starts with a non-numeric digit, an error is returned.

Pos()

This function returns the position of string or value within a character string or a record.

```
position = pos("Quick brown fox", "Quick")          -- Returns 1
position = pos("Quick brown fox", "fox")            -- returns 13
position = pos({"Quick", "brown", "fox"} , "fox")   -- returns 3
position = pos({2, 3, 4} , 5)                       -- returns 0
```

RTrim()

This function removes all trailing spaces from a string. See also "LTrim()" on page 93 and "Trim()" on page 94.

```
strvar = "   abc   ";
writelog(rtrim(str));     -- returns "   abc"
```

String()

Some methods and functions such as `dialog()` require a string variable to operate on. Use the `String()` function to convert variables of any other type. This is a type coercion operation. The `String()` function can also take objects as a parameter. This is sometimes useful when debugging a script.

```
myNumber = 123;
result = dialog("This is the value of myNumber: " + string(myNumber));
```

Trim()

This function removes all leading and trailing spaces from a string. See also "LTrim()" on page 93 and "RTrim()" on page 94.

```
strvar = "   abc   ";
writelog(trim(str));      -- returns "abc"
```

Typeof()

If you need to determine the data type of a variable, you can use the `Typeof()` function. It returns a number corresponding to the various types supported in Qedit. To make it easier to identify each type, Qedit provides application constants . You can use the application constants or the numeric representation interchangeably in QSL statements. See "Application Constants" on page 101.

```
if typeof(myVar) = qedit.typeundefined then  -- 0
    writelog("The variable MyVar has an undefined type");
else
    if typeof(myVar) <> 3 then    -- qedit.typestring
        writelog("MyVar is a string:" + myVar);
    else
        writelog("Myvar is not a string:" + string(myVar));
```

Upshift()

This function changes all lowercase characters in a string to the equivalent uppercase characters. Qedit takes into account the PC's local configuration and handles extended characters such as accents in French.

```
myString = "AbCdE";
upperMyString = upshift(myString);    -- returns "ABCDE"
```

Writelog()

This function sends messages to the log window of the **Script control** dialog box. You can pass any kind of parameter. `Writelog()` converts everything to a string before sending it to the control panel.

```
writelog("Script XYZ is starting");
-- Other statements to execute
writelog("Script XYZ has terminated");
```

Built-in Arithmetic Functions

Qedit offers a number of built-in functions to perform arithmetic operations.

** (exponentiation)

This function works like an arithmetic operator. It raises the value on the left to the power of the value on right.

```
float = 2.0;   -- raise 2 to the power 16
result = dialog("**   = " + string(float ** 16));    -- returns 65536
```

Abs()

This function returns the absolute value of the parameter. If the parameter value is positive, nothing changes. If the value is negative, the value becomes positive.

```
float = -1.0;
result = dialog("abs   = " + string(abs(float)));    -- returns 1
```

Acos()

This function calculates the arccosine value of the parameter.

```
float = 0.12;
result = dialog("acos  = " + string(acos(float)));   -- returns 1.45051
```

Asin()

This function calculates the arcsine value of the parameter.

```
float = 0.12;
result = dialog("asin  = " + string(asin(float)));   -- returns 0.12029
```

Atan()

This function calculates the arctangent value of the parameter. Both single and double argument forms are supported.

```
float = 0.12;
result = dialog("atan(one argument)   = " + string(atan(float)));
    -- returns 0.119429
result = dialog("atan(two arguments)  = " + string(atan(float, 5.0)));
    -- returns 0.0239954
```

Ceil()

This function finds the smallest integer that is greater than or equal to the parameter. Note, this is not the same as rounding up.

```
float = 1.1;
result = dialog("ceil  = " + string(ceil(float)));   -- returns 2
float = -1.1;
result = dialog("ceil  = " + string(ceil(float)));   -- returns -1
```

Cos()

This function calculates the cosine value of the parameter.

```
float = 12.0;
result = dialog("cos  = " + string(cos(float))); -- returns 0.843854
```

Floor()

This function returns the largest integer value that is less than or equal to the parameter. This is the same as truncate in other programming languages.

```
float = 1.6;
result = dialog("floor = " + string(floor(float)));  -- displays 1
float = -1.6;
result = dialog("floor = " + string(floor(float)));  -- displays -2
```

Fp()

This function returns the fractional portion of the parameter. The fraction value is stored in a new variable of type float.

```
float = 1.6;
result = dialog("fp   = " + string(fp(float)));  -- returns 0.6
```

Integer()

This function returns the integer portion of the parameter. This is a type coercion operation so the value is stored in a new variable of type integer.

```
float = 1.6;
result = dialog("integer = " + string(integer(float)));  -- returns 1
```

Ip()

This function returns the integer portion of the parameter. The integer value is stored in a new variable of type float.

```
float = 1.6;
result = dialog("ip   = " + string(ip(float)));  -- returns 1
```

Ln()

This function calculates the natural logarithm value of the parameter.

```
float = 180.0;
result = dialog("ln    = " + string(ln(float)));     -- returns 5.19296
```

Log()

This function calculates the logarithm value of the parameter.

```
float = 100.0;
result = dialog("log   = " + string(log(float)));    -- returns 2
```

Mod()

This function needs 2 parameters. It divides the first parameter by the second and returns the remainder.

```
float = 20.0;
result = dialog("mod  = " + string(mod(float,3)));   -- returns 2
```

Randseed() and Rand()

These functions are used to generate random numbers. Random values are between 0 and 32,767. `randseed()` can be used first to seed the random number generator. The function also returns the first random number.

Subsequently, you can call the `rand()` function to get the next random number.

```
float = randseed(123);
result = dialog("randseed(123)   = " + string(float));  -- returns 440
result = dialog("rand()   = " + string(rand())); -- returns 19053
```

If you call `randseed()` with the same value, you will repeatedly get the same sequence of random numbers. If you do not call `randseed()`, the seed itself will be random. This causes the random sequence from `rand()` to be different every time.

Sin()

This function calculates the sine value of the parameter.

```
float = 12.0;
result = dialog("sin  = " + string(sin(float))); -- returns -0.536573
```

Sqrt()

This function calculates the square root of the parameter.

```
float = 144.0;
result = dialog("sqrt  = " + string(sqrt(float)));   -- returns 12
```

Tan()

This function calculates the tangent value of the parameter.

```
float = 12.0;
result = dialog("tan  = " + string(tan(float))); -- returns -0.63586
```

Objects, Methods and Properties

Overview

Most of the things you manipulate with QSL are "objects". An object is a special type of variable that you normally must create in your script. There is only one Application object called `qedit`, which is created automatically for you. For example, a Document (a.k.a. File) is an object. You create a new document object in QSL using `Newfile()` or `Open()`. These return an object that you store into a variable.

```
fileA = newfile();
```

Even though an object is an internal structure of QSL, you can convert an object to readable characters with the `string()` built-in function and display it with the `writelog()` or `dialog()` functions.

```
fileC = open("c:\autoexec.bat");
result = dialog(string(fileC));
       -- displays <Object: File c:\autoexec.bat>
```

What Are Properties?

Objects have Properties that can be examined such as the **ShowInvisibles** or **Linecount** property. You access a property or method of the object by qualifying the name with a period "." followed by the property/method name.

Some properties such as `Connection.Hostcwd` are read-only but others such as `File.ShowInvisibles` may be set also.

```
fileA = newfile();
fileA.Insert("First line of text");
fileA.ShowInvisibles = true;  -- Display Tab, etc on screen
```

What Are Methods?

The things you can do to and with the **fileA** object, such as `Insert()` text, are called Methods.

Typically, a method is followed by a parameter list enclosed in opening and closing parentheses. Some methods do not have parameters. In this case, the parentheses must be specified but do not contain anything as in:

```
file.cut();
```

Other methods accept one or more parameter values within the parentheses. Parameters are passed by value. This means that the content of the original variable can not be changed. Most methods return a value using the assignment operator. The type of the return value is different for each method. Some returns an object, some return a record and others simple variables.

Parameters can be passed by position or by name. You can not mix these options on the same call. That is, if you want to use positional parameters, you cannot have named parameters. If you use named parameters, you can not use positional parameters.

Positional parameter values are matched from left to right with the names in the method declaration. For example, the `Open()` application method expects the following parameters:

- **Filename** (string)
- **Connection** (string)
- **ReadOnly** (Boolean)
- **OpenACopy** (Boolean)
- **ForceUnnumbered** (Boolean)

This means you could call `Open()` with three parameters: "program1.src" as **Filename**, "Production MPE" as **Connection** and `True` as **ReadOnly**. This call opens the specified file with Read-only access on the specified connection.

```
mpefile = open("program1.src", "Production MPE", True);
```

If you use positional parameters, you can not omit parameters in the middle of the parameter list. However, you can omit parameters at the end of the list. Here are some sample calls:

```
localfile = open("c:\personal\diary.txt");
    -- Valid. All other parameters get default values
mpefile = open("program1.src", "Production MPE");
    -- Valid. All other parameters get default values
localfile = open("c:\personal\diary.txt", , True);
    -- Invalid. Connection parameter must be specified
```

If you want get around these restrictions, you can use a named parameter list. You would qualify each value with the name of the corresponding parameter. This feature allows you to omit parameter values anywhere in the list. It also allows you to specify the parameters in any order. Here are some examples:

```
localfile = open(Filename: "c:\personal\diary.txt");
    -- Open a local file with default access options
mpefile = open(connection: "Production MPE, readonly: true,
               Filename: "program1.src");
    -- Open a host file as readonly. Parameters in different order

fileB = open(filename: "c:\personal\diary.txt", readonly: true);
       -- fileB is a Document object
writelog("lines=" + string(fileB.linecount)));   -- LineCount property
fileB.close();     -- Document method
```

In the last example, the `open()` application method is called with 2 parameters: **Filename** and **Readonly**. You could reverse the parameters' order without affecting the result.

The method creates a document object called **fileB**. The **LineCount** property for **fileB** is accessed in the `writelog()` function. Finally, the document is closed using the `close()` document method.

Making copies of an object?

If you assign an object variable to another variable, you do **not** have two independent objects. You still have one object, but with two names for it. For example, if you run this script you will see that `timestamp` and `anotherdate` both get added to, since there really is only one object:

```
timestamp = datetime();

anotherdate = timestamp;  -- aha, is this the same object??
anotherdate.adddays(1);   -- increments both names
if (timestamp.daysbetween(anotherdate)) = 0
    result = dialog("These are still the same date!");
endif
```

Application Object

QSL has a single application object, `qedit`, but this object has seven methods and many properties. The properties record the state of the current execution of Qedit and the configuration options that have been enabled. The methods create and find objects of the other types. For example, `qedit.activefile` is the document object that is currently open and active, if any.

To avoid any confusion with ordinary variables, you should always qualify application constants and properties with the `qedit` object name.

```
if currentFile = qedit.typeundefined then    -- Constant
    currentFile = qedit.activefile;          -- Property
endif
```

Application Constants

Remember to qualify them with the "qedit" object name.

Application constants are read-only properties that always return the same value. They are used for defining values for specific application properties and results.

Data Type Application constants

The following constants are used to describe variable data types. These can be used to test the results of a call to the `typeof()` function.

Name	Value	Compare to
qedit.typeundefined	0	Result of TypeOf()
qedit.typeinteger	1	Result of TypeOf()
qedit.typefloat	2	Result of TypeOf()
qedit.typestring	3	Result of TypeOf()
qedit.typerecord	6	Result of TypeOf()
qedit.typeobject	10	Result of TypeOf()

File Language Application Constants

Another set of application constants is defined for the languages supported in Qedit. These constants can be used to check the value of the Language document property. These constants are:

Name	Language Code	Description
QeditLanguageSPL	0	HP's Systems Programming Language
QeditLanguageFTN	1	Fortran
QeditLanguageCOBOL	2	Cobol
QeditLanguageCOBOLX	4	Cobol with comments
QeditLanguageJOB	5	Job stream
QeditLanguageRPG	6	RPG
QeditLanguageTEXT	7	Free-format text up to 256 bytes
QeditLanguagePASCAL	8	Pascal
QeditLanguageDATA	9	Free-format text up to 8172 bytes
QeditLanguageCC	10	C language
QeditLanguageCPP	11	C++ language
QeditLanguagePH	12	Powerhouse
QeditLanguagePASCX	13	Pascal with long lines
QeditLanguageCOBFREE	14	Free-format Cobol

Boolean Application Constants

Similarly, Qedit provides 2 named constants to be used in boolean comparisons: `True` and `False`. Reference to these constants do not require the `Qedit` prefix.

SearchReferenced Application Constants

The `FindAll()` document method allows you to search for string in a file and, optionally, other files referenced in it. The main file can reference Include files, Use files or COBOL Copy libraries. To specify which related files you want to search, you set the **SearchReferenced** parameter of the document method. Qedit provides application constants to make it easier to set the parameter value.

Name	Value	Search in
SearchFile	0	Current file only
SearchInclude	1	Current file and Include files
SearchUse	2	Current file and Use files
SearchCopylib	3	Current file and COBOL Copy libraries

Line Termination Application Constants

On local files, you can retrieve or set information about the line termination characters used for that particular file. This information is an integer value and can be referenced using the following constant names:

Name	Value	File Format
LineTerminationUnix	0	Unix
LineTerminationDos	1	DOS
LineTerminationMacintosh	2	Macintosh

Application Properties

Below is list of all application properties. They are associated with the `qedit` object and should always be qualified with the object name.

```
if currentFile = qedit.typeundefined then    -- Constant
    currentFile = qedit.activefile;          -- Property
endif
```

The QSL code below stores each application property value into its own variable. To see what the property looks like, so you can test for it in your script, you convert it into a string and display it with `dialog()` or `writelog()`.

Most properties correspond directly to options on dialog boxes or menu commands. Here is a summary:

Property	Option / Command	Found In
Autopush	Automatically push caret location on search	Searching tab of Qedit Preferences
AutoWorkfilePost	Automatically post Qedit workfiles	General tab of Qedit Preferences
CaretAllowedOutsideText	Caret allowed in undefined areas	General tab of Qedit Preferences
CheckServerTimestamps	Compare timestamps before overwriting on server	General tab of Qedit Preferences
LiveScrolling	Use live scrolling for server files	General tab of Qedit Preferences
MPEServerName	MPE Server Name	Configure Server Settings
Overwrite	Insert / overwrite mode	Insert key
ReleaseOnClose	Close connections with no open files	General tab of Qedit Preferences
ShowSCP	Control panel command	Script menu
UseRulerBar	Open files with a ruler bar	Defaults tab of Qedit Preferences

Some properties are initialized when Qedit starts up but retain their values throughout the edit session. These properties are:

- **CommandLine**: returns the command line used to invoke Qedit. For example, if you enter this at the DOS prompt:

  ```
  c:\>qwin32.exe -r myScript.qsl
  ```

  the **CommandLine** propery contains:

  ```
  -r myScript.qsl
  ```

- **VersionNumber**: returns Qedit's current version number.

Other properties are changing dynamically as you work in Qedit. For example:

- **Activefile**: returns the file object of the currently active document. This is a standard QSL file object.
- **File**: returns a list of all currently opened files. These are fully-qualified filenames but the connection name is not included. You can also use **Files** to get the same list.
- **LocalCWD**: when used on the right side of an assignment, the local current working directory is returned. If you use it on the left side of an assignment, Qedit saves the new value and actually changes the CWD to the new location.
- **OpenConnections**: returns a list of all currently opened connections.

If you wish to display the current values for all the application properties, you can use the following script:

```
-- script to display all Application Properties

activefile            = qedit.activefile;       --  object
autopush              = qedit.autopush;         --  boolean
autoworkfilepost      = qedit.autoworkfilepost; --  boolean
caretallowedoutsidetext   = qedit.caretallowedoutsidetext;  --  boolean
checkservertimestamps  = qedit.checkservertimestamps;    --  boolean
commandline           = qedit.commandline;      --  string readonly
file                  = qedit.file;             --  same as files
files                 = qedit.files;            --  list readonly
livescrolling         = qedit.livescrolling;    --  boolean
localcwd              = qedit.localcwd;         --  string
mpeservername         = qedit.mpeservername;    --  string readonly
openconnections       = qedit.openconnections;  --  list readonly
overwrite             = qedit.overwrite;        --  boolean
releaseonclose        = qedit.releaseonclose;   --  boolean
showscp               = qedit.showscp;          --  boolean
userulerbar           = qedit.userulerbar;      --  boolean
versionnumber         = qedit.versionnumber     --  string readonly

-- convert each property to a string and display it

writelog("activefile             = " + string(activefile));
writelog("autopush               = " + string(autopush));
writelog("autoworkfilepost       = " + string(autoworkfilepost));
writelog("caretallowedoutsidetext = " +
       string(caretallowedoutsidetext));
writelog("checkservertimestamps   = " + string(checkservertimestamps));
writelog("commandline            = " + string(commandline));
writelog("file                   = " + string(file));
writelog("files                  = " + string(files));
writelog("livescrolling          = " + string(livescrolling));
writelog("localcwd               = " + string(localcwd));
writelog("mpeservername          = " + string(mpeservername));
writelog("openconnections        = " + string(openconnections));
writelog("overwrite              = " + string(overwrite));
writelog("releaseonclose         = " + string(releaseonclose));
writelog("showscp                = " + string(showscp));
writelog("userulerbar            = " + string(userulerbar));
writelog("versionnumber          = " + string(versionnumber));
```

If you execute the script above in an unnamed text window with one other host file open, the log window of **Script Control** dialog box should contain something like this:

```
activefile             = <Object: File >
autopush               = 0
autoworkfilepost       = 1
caretallowedoutsidetext   = 0
checkservertimestamps  = 1
commandline            = -r myScript.qsl
file                   =
       {<Object: File /home/billsmith/diary.txt>, <Object: File >}
files                  =
       {<Object: File /home/billsmith/diary.txt>, <Object: File >}
livescrolling          = 0
localcwd               = C:\personal
mpeservername          = qedit.pub.robelle
openconnections        = {<Object: Connection uxserver>}
overwrite              = 0
releaseonclose         = 0
showscp                = 0
userulerbar            = 0
versionnumber          = 4.8.04
```

This result shows that the File and Files properties are identical.

Application Methods

Because there is only one application object instance, it is not necessary to always qualify application methods as in `qedit.open()`. You can simply use `open()`. You can tell if something is an application method by putting an optional 'qedit.' in front of the invocation. If that gives an error, you are working with a built-in function and not an application method. Here are all the Qedit application methods:

`DateTime()`	`HostCommandAbort()`
`DeleteConnectionTemplate()`	`HostCommandStatus()`
`DOSCommand()`	`LoadScript()`
`Exit()`	`NewConnectionTemplate()`
`FindConnectionTemplate()`	`NewFile()`
`FindOpenFile()`	`Open()`
`GetConnectionTemplateIterator()`	`OpenConnection()`
`GetDirectoryIterator()`	`ShellCommand()`
`HostCommand()`	`UnloadScript()`

DateTime() Application Method

The method does not require any parameter and returns an object.

This method creates a DateTime object and initializes it with the current date and time. This sample script inserts the timestamp in the active file.

```
file = qedit.activefile; -- we need an object to insert into
if typeof(file) = qedit.typeundefined then
    result = dialog("You must have a file open");
    stop;       -- terminates the script, but not Qedit
else
    timestamp = datetime();    -- Create a DateTime object
    timestamp = timestamp.fmtshortdatetime();
        -- a method, not a function
    file.insert(timestamp);       -- insert the formatted timestamp
endif
```

DeleteConnectionTemplate() Application Method

This method has only one parameter:

- **ConnectionTemplate** (any)

The return value is a boolean.

This method removes the connection template specified in the **ConnectionTemplate** parameter. The parameter can be a string containing the name of an existing connection. The name must match exactly but is not case-sensitive. This means that "`PROD UX`" and "`prod ux`" are equivalent and would match the same connection. The parameter can also be a ConnectionTemplate object found with the `FindConnectionTemplate()` application method, selected from a list created with the `GetConnectionTemplateIterator()` application method or created with the `NewConnectionTemplate()` application method.

If the connection template is successfully removed, the method returns true. If the operation fails, the method returns false. This would happen if the connection does not exist.

```
result = deleteconnectiontemplate("Prod UX");
if result then
    rd = dialog("Connection has been removed");
else
    rd = dialog("Connection has NOT been removed");
endif

connobject = findconnectiontemplate("Developer UX");
result = deleteconnectiontemplate(connobject);
if result then
    rd = dialog("Connection has been removed");
else
    rd = dialog("Connection has NOT been removed");
endif
```

DOSCommand() Application Method

Here are the positional parameters:

- **CommandName** (string)
- **Arguments** (string)
- **Wait** (boolean)

The return value is a record.

If you wish to execute commands on a host, check out the `HostCommand()` application method. If you wish to execute other programs using program names or associated files on your PC, check out the `ShellCommand()` application method.

This method allows the execution of commands in the Windows 95/NT shell. The difference is that `DosCommand()` can be an asynchronous or synchronous

operation as specified by the **Wait** parameter. Synchronous mode is the default. This means execution of the QSL script suspends until the DOS command has terminated. Asynchronous mode means the command is launched and executes on a different thread. Qedit does not wait for the command to complete and continues the script execution immediately.

The **CommandName** parameter must be a string and should contain an executable statement. This could be the name of an executable file such as a program file (`.exe`) or batch file (`.bat`).

The **Arguments** parameter is used to specify command arguments required by the program to execute.

The **Wait** parameter indicates whether the DOS command should execute asynchronously or synchronously. The parameter is optional and the default is `True`.

If the parameter is `True`, the command execution is synchronous. This means that the script is suspended until Qedit receives a signal from the DOS shell.

The `DOSCommand()` method returns a record to indicate the operation's success or failure. The record contains only one element:

- **ExitCode** (integer)

ExitCode contains 0 if the execution was successful. It contains a value other than 0 if the command itself reported a problem.

If the **Wait** parameter is set to `False`, the execution is asynchronous. This means Qedit immediately resumes execution of the script and does not wait for a response. Since the `DOSCommand()` method does not wait for the task to complete, it's unable to determine the outcome. In this case, it always returns a record where **ExitCode** is set to 0.

```
doscommand(commandname: "c:\robelle\bin\extractinfo.exe",
           arguments: "c:\personal\diary.txt");
    -- Executes the ExtractInfo program on the diary file
    -- Uses synchronous mode i.e. QSL waits for it to complete

doscommand(startname: "c:\robelle\bin\extractinfo.exe",
           arguments: "c:\personal\otherdiary.txt",
           wait: false);
    -- Executes the ExtractInfo program on another diary file
    -- Uses asynchronous mode i.e. control returns to QSL immediately
```

Exit() Application Method

The method does not require any parameter and does not return anything.

This method forces the termination of Qedit. Qedit then goes through its regular process as if you had selected the **Exit** command from the **File** menu.

FindConnectionTemplate() Application Method

This method has only one parameter:

- **Name** (string)

This method returns a ConnectionTemplate object corresponding to the name specified as the parameter value. If the connection name does not exist, the result variable is of undefined.

```
connobject = findconnectiontemplate("Developer UX");
if typeof(connobject) = qedit.typeundefined then
    writelog("Connection does not exist!");
else
    writelog(connobject.Name);        -- Display the connection name
    writelog(connobject.HostName);    -- Display the host name
endif
```

FindOpenFile() Application Method

The positional parameters are:

- **Pathname** (string)
- **Connection** (any type)
- **Matches** (string)

The return value is a record.

This method returns a list of files that match some search criteria. It searches through all of the open files and returns a file object if it finds a match. Qedit uses slightly different rules when searching on filenames and pathnames. The rules are:

- local files: caseless match
- MPE/iX host files: caseless match
- HP-UX host files: case-sensitive match

Two forms of comparison are done. If you specify a `Pathname`, the full pathname must match. If you specify a **Matches** parameter, then there will be a match if any substring of the full pathname matches the fragment that you specify using the case comparison rules form above.

The meaning of each parameter is:

- **Connection**: Search for files on the specified Connection.
- **Matches**: Search for this string anywhere in the pathname of a file.
- **Pathname**: Fully-qualified file pathname, must match exactly.

The `Findopenfile()` method always returns a list, even if there is only a single match. But it can return more than one file object. If you specify the **Connection** parameter only, QSL returns the names of all the files opened on the connection. If you specify the **Matches** parameter, any pathname that match what you specify qualifies the file and return it as part of the object list.

If you enter a string without a parameter name, **Pathname** is assumed.

Here is an example for some of the common ways of using `findopenfile()`:

```
--  The first three examples show full pathname searching:

--  Pathname is assumed.
file01 = findopenfile("c:\Personal\Diary.txt");
if file01 = {} -- No match. Not already open
    file01 = open("c:\Personal\Diary.txt");
endif
result = dialog("Full match: " + string(file01));

-- MPE/iX file:
file02 = findopenfile(connection:"mpe03", pathname:"prog1.src.dev");
if file02 = {} -- No match. Not already open
   file02 = open(connection:"mpe03", filename:"prog1.src.dev");
endif
result = dialog("Full match: " + string(file02));

-- HP-UX file:
file03 = findopenfile(connection:"hpux02",
                      pathname:"/home/dev/src/test.c");
if file03 = {} -- No match. Not already open
   file03 = open(connection:"hpux02", filename: "test.c");
endif
result = dialog("Full match: " + string(file03));

--  The next three examples show substring matching for each file:
file04 = findopenfile(matches: "Diary");
result = dialog("Partial match: " + string(file04));

--  MPE/iX file:
file05 = findopenfile(matches:"prog1.src");
result = dialog("Partial match: " + string(file05));

--  HP-UX file:
file06 = findopenfile(matches: "test.c");
result = dialog("Partial match: " + string(file06));

--  Multiple files on a connection:
file07 = open(connection:"mpe03", filename:"prog2");
file08 = findopenfile(connection:"mpe03");
result = dialog("Connection mpe03: " + string(file08));
    -- Displays prog1 and prog2 file objects

--  Multiple matches:
file09 = findopenfile(matches: "src");
result = dialog("Matches src: " + string(file09));
```

GetConnectionTemplateIterator() Application Method

This method does not have any parameter.

It returns an iterator object which contains ConnectionTemplate objects for existing connections.

```
conniterator = getconnectiontemplateiterator();
repeat for connitem in conniterator
    writelog(connitem);
endrepeat
```

This script produces something like:

```
<Object: Connection template Prod UX>
<Object: Connection template Prod MPE>
<Object: Connection template Dev UX>
<Object: Connection template Dev MPE>
```

GetDirectoryIterator() Application Method

The positional parameters are:

- **Directory** (string)

The return value is an iterator object.

For examples, see "Navigating Through Directories" on page 41.

This application method only works with local directories. It is also available as a connection method to handle host directories.

This method returns an directory iterator object which contains information about all the files and subdirectories in the specified directory. Each entry is a record with a number of elements describing the file or subdirectory. For a detailed description of each element, see "Local Directory Iterator" on page 160. For local directories, the elements are:

- **Name** (string)
- **Path** (string)
- **OpenName** (string)
- **Size** (integer)
- **ModificationTimestamp** (date object)
- **CreationTimestamp** (date object)
- **AccessTimestamp** (date object)
- **CanonicalType** (string)

Individual entries can be accessed using a `REPEAT` statement.

```
localdir = getdirectoryiterator("c:\personal");

subdircount = 0;
filecount = 0;
repeat for direntry in localdir
    if direntry.canonicaltype = "directory" then
        subdircount = subdircount + 1;
    else
        filecount = filecount + 1;
    endif
endrepeat

writelog("Number of subdirectories=" + string(subdircount));
writelog("Number of files=" + string(filecount));
```

The meaning of each parameter is:

- **Directory**: Retrieve the list of files and subdirectories stored at that location.

The `GetDirectoryIterator()` method itself is not recursive. This means that it returns information on subdirectories in the requested directory but it does not go down the subdirectories. If you need to see what is stored at other levels in the directory tree, the script has to do it.

HostCommand() Application Method

Here are the positional parameters:

- **Connection** (any type)
- **Command** (any type)
- **Output** (object)
- **Wait** (boolean)

The return value is a record.

For some examples, see "Executing Host Commands" on page 45. If you wish to execute a command on the PC, check out the `ShellCommand()` or the `DOSCommand()` application methods.

This method requests that the server associated with the **Connection** executes the commands specified in the **Command** parameter.

The Connection parameter is mandatory.

The **Connection** parameter can be of any type. It can be a connection object previously opened with a call to the `OpenConnection()` application method. It can also be a string constant or variable containing an actual connection name. In this case, the connection must already be opened.

The **Command** parameter can also be of any type. If it is a string, it should be a single host command. If you wish to execute multiple commands, you can put them in a record variable and use that variable as the parameter value.

*The output file **must** be a local file.*

The **Output** parameter is optional. It is used to specify an alternate destination for the command results. If specified, the **Output** parameter must be a file object i.e. the file must be opened. With this option, the host command output is written to the file.

If the **Output** parameter is not specified, the results are returned in a record variable. The structure of this record is determined by the **Wait** parameter.

The **Wait** parameter indicates whether the host command should execute asynchronously or synchronously. The parameter is optional and the default is `True`.

If the parameter is `True`, host command execution is synchronous. This means that the script is suspended until Qedit receives the output from the server. If the **Output** parameter is not specified, the return variable is a record of 3 elements:

- **ExitCode** (integer)
- **CommandOutput** (record)
- **JCW** (integer)

ExitCode contains 0 if the execution was successful. It contains a value other than 0 if there has been a problem. **CommandOutput** is a record that contains all the lines produced by the host command. Each element in the record represents one line of output. **JCW** is an integer which represents the overall execution status. Check "Checking Results" on page 46 for details on how to access and interpret these elements.

If the **Wait** parameter is set to `False`, the execution is asynchronous. This means Qedit immediately resumes execution of the script and does not wait for a response. The `HostCommand()` method then returns a record with a single element in it:

- **Running** (boolean)

The element is typically initialized to 1. The script can check the host command execution status using the `HostCommandStatus()` application method.

HostCommandAbort() Application Method

There is only one positional parameter:

- **Wait** (boolean)

The return value is a record.

This method is used to abort an asynchronous host command. The **Wait** parameter is optional and the default is `False`.

If the **Wait** parameter is `True`, `HostCommandAbort()` waits until the host command completes. The return variable then contains:

- **ExitCode** (integer)
- **CommandOutput** (record)
- **Running** (boolean)
- **Finished** (boolean)

ExitCode contains 0 if the execution was successful. It contains a value other than 0 if there has been a problem. **CommandOutput** is a record that contains all the lines produced by the host command *up to the abort*. Thus **CommandOutput** might not contain everything sent during a normal execution. Each element in the record represents one line of output. **Running** and **Finished** both contain `True`.

If the **Wait** parameter is `False` or is omitted, `HostCommandAbort()` sends the abort request to the host and returns control to the script immediately. The return variable is a record containing these elements:

- **Running** (boolean)
- **Finished** (boolean)
- **ProgressTime** (integer)
- **Step** (string)
- **ProcessID** (string)

Running is set to `True` to indicate that command execution has started. **Finished** is set to `True` to indicate the command has finished executing. In reality, the host process might still be executing but Qedit simply discards any output received from that point.

ProgressTime indicates the CPU time used since the start of the command. **Step** contains the last command encountered in a multi-command record sent to `HostCommand()`. This actually indicates which command is actually executing. **ProcessID** is the Command Interpreter (CI) process id or shell process id currently used to execute the commands.

HostCommandStatus() Application Method

There is only one positional parameter:

- **Wait** (boolean)

The return value is a record.

For examples, see "Executing Host Commands" on page 45.

This method is used to check the status of an asynchronous host command. The **Wait** parameter is optional and the default is `False`.

If no host command is in progress, the return variable is a record with only one element set to `False`:

- **Running** (boolean)

If a host command was in progress and is now finished, the return variable is a record which contains:

- **ExitCode** (integer)
- **CommandOutput** (record)

- **Running** (boolean)
- **Finished** (boolean)

ExitCode contains 0 if the execution was successful. It contains a value other than 0 if there has been a problem. **CommandOutput** is a record that contains all the lines produced by the host command. Each element in the record represents one line of output. **Running** and **Finished** both contain `True`.

If a host command is in progress and the **Wait** parameter is `True`, `HostCommandStatus()` waits until the host command completes. The result has the same record structure: **ExitCode**, **CommandOutput**, **Running** and **Finished**.

If a host command is in progress and the **Wait** parameter is `False` or is omitted, the return variable is a record containing these elements:

- **Running** (boolean)
- **Finished** (boolean)
- **ProgressTime** (integer)
- **Step** (string)
- **ProcessID** (string)

Running is set to `True` to indicate that command execution has started. **Finished** is set to `False` to indicate the command is still executing. **ProgressTime** indicates the CPU time used since the start of the command. **Step** contains the last command encountered in a multi-command record sent to `HostCommand()`. This actually indicates which command is actually executing. **ProcessID** is the Command Interpreter (CI) process id on MPE or shell process id on UNIX currently used to execute the commands.

Loadscript() Application Method

The positional parameters are:

- **Filename** (string)
- **Connection** (any type)

The return value is a string.

This method dynamically loads a QSL file into your script environment. The script is reloaded each time `Loadscript()` is called. If the script contains `On command` statements, the command names are added to the **Script** menu. If the script does not contain any `On command` but contains a mainline, the name of the script is added to the menu. If the script does not contain `On command` statements nor a mainline, the **Script** menu is unchanged.

You can check the presence of the loaded script using the **Manage scripts** dialog box. There is no error message if the script is already loaded. Instead, Qedit replaces the currently loaded script with the script code again. This is handy when developing an external script. You can make changes to the external script and test them simply by running another script calling this built-in function. Once loaded, the script can be accessed via the **Script** menu (if it contains `On Command` statements or has a mainline) or via an external reference.

You can assign the result of a call to `Loadscript()` to a variable. If the call is successful, it returns the name of the script and the resulting variable is created as a string type.

```
loadedscript = loadscript(connection: "Production HPUX",
           filename: "/opt/robelle/scripts/globals.qsl");
```

NewConnectionTemplate() Application Method

The positional parameters are:

- **Name** (string)
- **Host** (string)
- **LogonInformation** (record)
- **ColorScheme** (string)
- **Autologon** (boolean)
- **FromTemplate** (object)

The method returns a ConnectionTemplate object.

This method creates a new connection based on the information in the parameters. The **Name** parameter is required in all cases. The name of the new connection must not exist in the connection template file.

If the **FromTemplate** parameter is specified, it has to be a ConnectionTemplate object created with the `FindConnectionTemplate()` application method or extracted from a ConnectionTemplate iterator object created with the `GetConnectionTemplateIterator()` application method. When used, the new connection acquires all attributes from the selected ConnectionTemplate object.

If the **FromTemplate** parameter is omitted, the **Host** and **LogonInformation** parameters are required. The **Host** paramater contains the hostname or IP address of the host. The **LogonInformation** parameter is a record and all elements in it are required. See the "ConnectionTemplate Properties" on page 163 for details on the **LogonInformation** record.

The **ColorScheme** and **Autologon** parameters are optional.

```
connname="New UX";     -- Create a new connection from scratch
hostname="Prod UX";
logoninfo={};
logoninfo.ConnectionType = "unix";
logoninfo.Username = "pgmr";
logoninfo.Password = "hispass";
logoninfo.InitialDirectory = "/home/pgmr";
newconn = newconnectiontemplate(Name: connname,
                                Host: hostname,
                                LogonInformation: logoninfo);

findconn = findconnectiontemplate("Dev UX");
connname="Copy UX";    -- Create a connection based on an existing one
newconn = newconnectiontemplate(Name: connname,
                                FromTemplate: findconn);
```

Newfile() Application Method

The positional parameters are:

- **Connection** (string)
- **Recordlength** (number)
- **QeditLanguage** (number)
- **Minimize** (boolean)

- **DiscardOnClose** (boolean)

The return value is an object.

For examples, see "Creating a File" on page 33.

This method creates a new host or local document and returns it as a document object.

All parameters are optional. The meaning of each parameters is:

- **Connection**: Specifies the host connection as a string. Default is a local file, which can also be specified explicitly as an empty string.
- **QeditLanguage**: Specifies what type of file you wish to create, selecting from a list of "language codes".
- **RecordLength**: Specifies the maximum number of columns in a line for this file.
- **Minimize**: A boolean value. `True` causes the file to be opened in a document window minimized to an icon.
- **DiscardOnClose**: A boolean value. `True` means the file can be closed and its contents automatically discarded. `False` is the default and forces Qedit to ask for confirmation to discard when the file is closed.

The **Connection** parameter must contain the name of an existing connection. The **RecordLength** can be any value between 1 and 8,172. **QeditLanguage** can take any one of the values defined in the QeditLanguage application constants set.

Remember that **RecordLength** is tightly tied in with **QeditLanguage**. Currently, **QeditLanguage** is only valid for host files.

Here is a fragment from a sample script which creates a new file on the same server as the current file:

```
connectionname = qedit.activefile.connectionname;
duplicate = newfile(connection: connectionname);
```

It is important that you specify the desired destination connection when you create a `newfile()` since there is no method to save an open file on a different connection (although you can use the sample script "Copying Files Between Systems" on page 77 to copy a file).

Open() Application Method

The positional parameters are:

- **Filename** (string)
- **Connection** (string)
- **ReadOnly** (boolean)
- **OpenACopy** (boolean)
- **ForceUnnumbered** (boolean)
- **Minimize** (boolean)
- **UpdateRecentFiles** (boolean)

The return value is an object.

For examples, see "Opening a File" on page 34.

This method opens a host or local document and returns a Document Object. The cursor position or selection at the time of the last close is restored. If none, the cursor is at the start of the file i.e. line 1, column 1.

The meaning of each parameter is:

- **Connection**: A string with the Connection name. If the parameter is omitted or contains blanks, QSL assumes it is a local file.
- **Filename**: A string with the file name. This parameter is required.
- **ForceUnnumbered**: A Boolean. `True` means ignore sequence numbers in the file.
- **Minimize**: A boolean value. `True` causes the file to be opened in a document window minimized to an icon.
- **OpenACopy**: A Boolean value. `True` causes a copy to be made of an existing Qedit workfile. You will need to do a **SaveAs** to give it a name if you want to save it.
- **ReadOnly**: A Boolean value. `True` means you should not be allowed to modify the file.
- **UpdateRecentFiles**: A Boolean value. `False` means the file will not be inserted in the **Recent Files** list of the **File** menu.

If the `Open()` fails, the script normally prints an error message and stops. However, you can use a `Try-Recover` block to handle errors in a more graceful way. Here is a sample script showing various forms of the `Open()` method:

```
--  Open a local file and detect errors:
try
    file01 = open("c:\autoexec.bat");
recover
    result = dialog("Sorry, could not open the autoexec file.");
    stop;
endtry
file01.close();

--  Open a local file using a "named" filename parameter:
file02 = open(filename: "c:\autoexec.bat");
file02.close();

--  Open a local file as read-only:
file03 = open(filename: "c:\autoexec.bat", readonly: true);
file03.close();

--  Open a host file:
file04 = open(filename: "diary", connection:"mpe05");
file04.close();

--  Open a host file as read-only:
file05 = open(filename: "diary", connection:"mpe05", readonly: true);
file05.close();

--  Open a host file as openacopy:
file06 = open(filename: "diary", connection:"mpe05", openacopy: true);
file06.close();

--  Open a host file as forceunnumbered:
file07 = open(filename: "diary", connection: "mpe05",
              forceunnumbered: true);
file07.close();

--  Open with all possible parameters:
file08 = open(filename: "k",
              connection: "mpe05",
              readonly: true,
              openacopy: true,
              forceunnumbered: true);
file08.close();
```

OpenConnection() Application Method

The method has only one positional parameter:

- **Connection** (string)

The return value is an object.

This method is used to establish a connection to a host. The connection name must exist in the connection list. The method does not open any file but opens the connection and validates all security settings.

The parameter must be a string and should contain the name of an existing connection.

```
mpeconn = openconnection(connection: "Production MPE");
```

ShellCommand() Application Method

Here are the positional parameters:

- **Startname** (string)
- **Arguments** (string)

The return value is a boolean.

If you wish to execute commands on a host, check out the `HostCommand()` application method. If you wish to execute a local command in synchronous mode, check the `DOSCommand()` application method.

This method allows the execution of commands in the Windows 95/NT shell. This is an asynchronous operation. This means that the command is launched and executes on a different thread. Qedit does not wait for the command to complete and continues the script execution immediately.

The **Startname** parameter must be a string and should contain an executable statement. This could be the name of a program file or a file with an associated extension. For example, the extension `.doc` could be associated in Windows with Microsoft Word. Passing a filename ending in `.doc` to the `Shellcommand()` method would automatically bring up MS-Word.

The **Arguments** parameter is used to specify command arguments required by the program to execute.

```
shellcommand(startname: "c:\robelle\bin\qwin32.exe",
           arguments: "c:\personal\diary.txt");  -- Executes Qedit
shellcommand(startname: "http://www.abcwidget.com");
       -- Connects to this URL in the default browser
```

UnloadScript() Application Method

The method has only one positional parameter:

- **ScriptName** (string)

The return value is boolean.

This method is used to remove a script from the script environment.

The parameter must be a string and should contain the name of a previously loaded script.

```
UnloadScript(scriptname: loadedscript);
```

Document Objects

A Document is an instance of a file, either local or host-based, that is open in Qedit. There are two application methods for creating a document object: `Newfile()` and `Open()`.

Document Constants

Currently, there are no document constants.

Document Properties

Qedit keeps tracks of many attributes and settings that relate to a specific document. The document properties are:

- **AutoIndent**: If `True`, indicates that the auto-indent option is enabled. This is equivalent to the **Auto-indent** option on the **Options** tab of the file **Properties** dialog box.
- **CacheMaxLines**: Indicates the maximum number of lines that can be stored in the cache.

- **Connection**: If the active document is a host file, this property contains a connection object. If the active document is a local file, the target variable is not initialized. Check the status with the `exists()` function.
- **ConnectionName**: If the active document is a host file, this property contains the connection name.
- **ConvertTabsToSpaces**: If True, indicates that tab characters are converted to spaces when entering text in the file. This is equivalent to the **Use spaces for tabs** option on the **Options** tab of the file **Properties** dialog box.
- **DisplayDetabbedColumn**: If True, indicates that the display detabbed column option is enabled. This is equivalent to the **Display detabbed column** option on the **Options** tab of the file **Properties** dialog box.
- **FullFilename**: This property contains the fully-qualified file name. It does not include the connection name.
- **IsNew**: If `True`, indicates this is a new file that has not been named yet.
- **IsModified**: If `True`, indicates the file has been modified.
- **IsOnHost**: If `True`, indicates the file resides on a host.
- **IsQedit**: If the opened file is a Qedit workfile, this property is set to `True`. Otherwise, it is set to `False`.
- **IsReadOnly**: If `True`, indicates that the file has been opened in read-only mode.
- **IsSaveable**: If `True`, indicates the file can be saved. Typically, this is applicable to host files. For example, it would be `False` for spoolfiles on MPE hosts.
- **KeepTrailingBlanks**: If `True`, indicates that the option to preserve trailing blanks is enabled. This is equivalent to the **Preserve trailing blanks** option on the **Options** tab of the file **Properties** dialog box.
- **LastFoundColumn**: After a successful search, this property contains the column number where the found string starts.
- **LastFoundLength**: After a successful search, this property contains the number of characters in the found string.
- **LastFoundLine**: After a successful search, this property contains the line number where the string has been found.
- **LastSearchString**: This property contains the last search string used in the **Find** dialog box. The value is **not** changed by the `Find()` method i.e. it is not the last string searched from within a script.
- **LineCount**: This is the number of lines currently in the file.
- **LinesTruncated**: Contains the number of lines that have been truncated during the last **Paste** or **Insert** operation.
- **LineTermination**: On local files, indicates the type of line termination character used. This is an integer value corresponding to UNIX format (0), DOS format (1) or Macintosh format (2). For easy manipulation, see "Application Constants" on page 101.
- **OriginalCurrentLine**: It contains the current line number at the time of the last close on this file.

- **RecordLength**: This is the maximum line length for this file. This is equivalent to the **Record Length** option on the **Options** tab of the file **Properties** dialog box.
- **Selection**: This is a record with information about the cursor. Looking at this property, you can determine if the cursor is a simple caret, a regular selection or a rectangular selection.
- **ShowInvisibles**: If set to `True`, displays invisible characters in the document.
- **Title**: This is a string that contains the document title as it appears on the title bar.
- **WorkfileName**: If the host file has been copied into a Qedit workfile, this property contains the name of that workfile. If no workfile is used, this is blank.

Here is a script that writes all the document properties to the log window:

```
--  show all document properties
file = open("c:\autoexec.bat");

fullfilename           = file.fullfilename;          -- string
linecount              = file.linecount;             -- integer
isqedit                = file.isqedit;               -- boolean
workfilename           = file.workfilename;          -- string
recordlength           = file.recordlength;          -- integer
isreadonly             = file.isreadonly;            -- boolean
autoindent             = file.autoindent;            -- boolean
displaydetabbedcolumn  = file.displaydetabbedcolumn; -- boolean
converttabstospaces    = file.converttabstospaces;   -- boolean
lastfoundlength        = file.lastfoundlength;       -- integer
keeptrailingblanks     = file.keeptrailingblanks;    -- boolean
originalcurrentline    = file.originalcurrentline;   -- integer
isnew                  = file.isnew;                 -- boolean
linestruncated         = file.linestruncated;        -- integer
cachemaxlines          = file.cachemaxlines;         -- integer
connection             = file.connection;            -- object
connectionname         = file.connectionname;        -- string
issaveable             = file.issaveable;            -- boolean
lastfoundcolumn        = file.lastfoundcolumn;       -- integer
lastfoundline          = file.lastfoundline;         -- integer
lastsearchstring       = file.lastsearchstring;      -- string
selection              = file.selection;             -- record
title                  = file.title;                 -- string
ismodified             = file.ismodified;            -- boolean
isonhost               = file.isonhost;              -- boolean

file.close();

writelog("fullfilename        = " + string(fullfilename));
writelog("linecount           = " + string(linecount));
writelog("isqedit             = " + string(isqedit));
writelog("workfilename        = " + string(workfilename));
writelog("recordlength        = " + string(recordlength));
writelog("isreadonly          = " + string(isreadonly));
writelog("autoindent          = " + string(autoindent));
writelog("displaydetabbedcolumn   = " + string(displaydetabbedcolumn));
writelog("converttabstospaces = " + string(converttabstospaces));
writelog("lastfoundlength     = " + string(lastfoundlength));
writelog("keeptrailingblanks  = " + string(keeptrailingblanks));
writelog("originalcurrentline = " + string(originalcurrentline));
writelog("isnew               = " + string(isnew));
writelog("linestruncated      = " + string(linestruncated));
writelog("cachemaxlines       = " + string(cachemaxlines));
if exists(connection) then    -- Does not exist for local files
    writelog("connection          = " + string(connection));
else
    writelog("connection          = <none for local file>");
endif
writelog("connectionname      = " + string(connectionname));
writelog("issaveable          = " + string(issaveable));
writelog("lastfoundcolumn     = " + string(lastfoundcolumn));
writelog("lastfoundline       = " + string(lastfoundline));
writelog("lastsearchstring    = " + string(lastsearchstring));
writelog("selection           = " + string(selection));
writelog("title               = " + string(title));
writelog("ismodified          = " + string(ismodified));
writelog("isonhost            = " + string(isonhost));
```

If you execute this script to open `c:\autoexec.bat`, this is what you might see in the log window:

```
fullfilename           = c:\autoexec.bat
linecount              = 4
isqedit                = 0
workfilename           =
recordlength           = 8172
isreadonly             = 0
autoindent             = 1
displaydetabbedcolumn  = 0
converttabstospaces    = 0
lastfoundlength        = 0
keeptrailingblanks     = 0
originalcurrentline    = 0
isnew                  = 0
linestruncated         = 0
cachemaxlines          = 4200
connection             = <none for local file>
connection name        =
issaveable             = 0
lastfoundcolumn        = 1
lastfoundline          = 1
lastsearchstring       = something
selection              = {Start: {Line: 1, Column: 1}}
title                  = autoexec.bat
ismodified             = 0
isonhost               = 0
```

If you try it on a host file, this is what you might see:

```
fullfilename           = /home/dev/escape.html
linecount              = 73
isqedit                = 0
workfilename           = /var/tmp/qscr.UAAa27026
recordlength           = 8172
isreadonly             = 0
autoindent             = 1
displaydetabbedcolumn  = 0
converttabstospaces    = 0
lastfoundlength        = 0
keeptrailingblanks = 0
originalcurrentline    = 59
isnew                  = 0
linestruncated         = 0
cachemaxlines          = 4200
connection             = <Object: Connection UX Scripts>
connection name        = UX Scripts
issaveable             = 0
lastfoundcolumn        = 1
lastfoundline          = 1
lastsearchstring       = something
selection              = {Start: {Line: 60, Column: 7}}
title                  = UX Scripts: escape.html
ismodified             = 0
isonhost               = 1
```

Here is a subroutine called `CheckLastFound()` that we use in the Qedit test suite. The subroutine checks three document properties. It is called after a call to `Find()` to verify that the found string was the one expected by the test:

```
sub CheckLastfound(file, startLine, startCol, theString)

returnValue = true;

if file.lastfoundline <> startLine then
    writelog( "Last found line " +  string(file.lastfoundline) +
            " not the expected " + string(startLine));
    returnValue = false;
    endif

    if file.lastfoundcolumn <> startCol then
        writelog("Last found column " + string(file.lastfoundcolumn) +
            " not the expected " + string(startCol));
    returnValue = false;
endif

if file.lastfoundlength <> length(theString) then
    writelog("Last found length " + string(file.lastfoundlength) +
        " does not match " + string(length(theString)));
    returnValue = false;
    endif

return returnValue;

endsub
```

Document Methods

There are a lot of things you can do with a document and for each operation, there is a document method.

`Activate()`	`Entab()`	`Insert()`	`SaveAs()`
`Close()`	`Find()`	`InsertColumn()`	`Select()`
`Copy()`	`FindAll()`	`Paste()`	`SetWidth()`
`Cut()`	`GetSelectedText()`	`PrintOnHost()`	`ShiftLeft()`
`Delete()`	`GetText()`	`PrintOnLocal()`	`ShiftRight()`
`Detab()`	`Guides()`	`Save()`	`Tabs()`

Most document methods return `True` or `False` to indicate success or failure. The returned value should always be checked before proceeding to the next step. If there is a possibility of failure, you might want to use a Try-Recover block.

Activate() Document Method

This method does not require parameters and the return value is an integer.

This method moves the focus to the specified document and brings the window to the front. The purpose of the `Activate()` method is to make the document the "active" one. The active document is the document window that you see in the foreground. You may want to bring a specific window to the front before asking the user for input with `Dialog()`.

Here is an example from one Robelle's test scripts. In this script, there are several documents open and we want a specific one on top at each debugging `Dialog()` call:

```
file.select(startline: 1, endline: file.linecount);
file.copy();

comparefile = newfile();
comparefile.paste();

--  Loop through each line of the copy deleting the column range
repeat for inx from firstline to lastline
    comparefile.delete(startline: inx,
        endline: inx,
        startcolumn: firstcolumn,
        endcolumn: lastcolumn);
endrepeat

comparefile.activate();   -- Brings <comparefile> to the front
result = dialog("examine comparison file after rectangular delete.");

deleteresult = file.delete(startline:   firstline,
        endline:     lastline,
        startcolumn: firstcolumn,
        endcolumn:   lastcolumn,
        rectangular: true);

file.activate();   -- Brings <file> to the front
result = dialog("examine original file after rectangular delete.");
```

Close() Document Method

The positional parameters are:

- **DiscardChanges** (boolean)
- **ForceOverwrite** (boolean)
- **IgnoreErrors** (boolean)

The return value is boolean.

This method closes an open document, releasing resources and ending all operations on that document. See also the "Save() Document Method" on page 142 and "SaveAs() Document Method" on page 142.

Here are all of the parameters to the `close()` method:

- **DiscardChanges**: `True` means you want to discard changes instead of asking if you want to save them.
- **ForceOverwrite**: `True` means you want to automatically save the changes to the document, overwriting any existing file without asking if it is okay.
- **IgnoreErrors**: `True` means you want to ignore possible errors during the close operation i.e. the file is closed anyway.

It is good programming practice to explicitly `close()` the files you do not need anymore. This way you can be sure to close the files with the options you prefer.

Copy () Document Method

There are no parameters to `Copy()` and the return value is boolean.

This method copies the current selection to the clipboard. The previous contents of the clipboard is erased. This is equivalent to having a document active and doing CTRL+C on the keyboard or using the **Copy** command on the **Edit** menu. You can code as if there were a single clipboard shared by the client PC and all host servers. See also "Cut() Document Method" on page 125 and "Paste() Document Method" on page 141.

For details on how to make a selection, see the "Select() Document Method" on page 143.

Below is a short example of a `Select()` and `Copy()` that writes the entire document to the clipboard. For a full example on how to copy text across documents, see "Copying Files Between Systems" on page 77.

```
localfile = open(localfilename);
localfile.select(startline: 1, endline: localfile.linecount);
localfile.copy();
```

Cut() Document Method

There are no parameters to `Cut()` and the return value is boolean.

This method copies the current selection to the clipboard, then deletes it from the file. The previous contents of the clipboard is erased. This is equivalent to having a document active and doing CTRL+X on the keyboard or using the **Cut** command on the **Edit** menu. You can code it as if there were a single clipboard shared by the client PC and all host servers. See also "Copy () Document Method" on page 124 and "Paste() Document Method" on page 141.

For details on how to make a selection, see "Select() Document Method" on page 143.

Below is the `CutRange()` subroutine we use in the Qedit test suite. It is a subroutine that does a rectangular cut from a file, then simulates the same operation using other methods and compares the results to ensure they are identical. This subroutine calls two other subroutines: `FillFile()` and `CompareFile()`. See "Initializing a Test File" on page 69 and "Comparing Two Files" on page 69.

```
<<
    The cutrange() subroutine cut the specified rectangle
    to the clipboard.
>>

sub cutrange(file, name, firstline, lastline, firstcolumn, lastcolumn)

    firsttest = name + "-paste";

    -- select the rectangle
    file.select (startline: firstline,
            endline: lastline,
            startcolumn: firstcolumn,
            endcolumn: lastcolumn,
            rectangular: true
            );

    file.cut (); -- writes this area to clipboard

endsub
```

Delete() Document Method

The positional parameters are:

- **StartLine** or **Line** (integer)
- **StartColumn** or **Column** (integer)
- **EndLine** (integer)
- **EndColumn** (integer)
- **Start** (record)

- **End** (record)
- **Range** (record)
- **Rectangular** (boolean)
- **FillWithSpaces** (boolean)

The return value is boolean.

This method deletes a portion of a document. This includes the ability to delete a column range. The area to be deleted is specified in the same manner in which `Select()` specifies a region.

`Delete()` returns `True` if the operation is successful, otherwise it returns `False`. Here are all possible parameters to the `Delete()` method:

- **StartLine**: An integer specifying the starting line of the region to delete. **Line** is a valid alias. As in `Select()`, if an explicit **StartColumn** is not specified, the entire start line is selected.
- **StartColumn**: An integer specifying the first column in **StartLine** of the region to be deleted. **Column** is a valid alias.
- **EndLine**: An integer specifying the last line of the region to be deleted; as in `Select()`, if an **EndColumn** is not specified, the entire last line is selected.
- **EndColumn**: An integer specifying the last column in **EndLine** of the region to be deleted.
- **FillWithSpaces**: A boolean that, if `True`, fills the selected region with spaces instead of removing it from the document.
- **Rectangular**: A boolean that, if `True`, specifies a columnar delete. In that case, the selected region is a rectangle, starting at an upper-left corner (specified by line and column) and ending at a lower-right corner. If `False`, the region deleted is a normal stream of characters instead of a columnar area.
- **Range**: A record that specifies the entire range to be delete, as in

  If you specify a **Range**, you don't specify any of the preceding coordinate parameters including **Rectangular**. This means that a complete **Range** value could be:

  ```
  {Start: {Line:5, Column:1}, End:{Line:10, Column:1},
  Rectangular: True}
  ```

- **Start**: A record that specifies the starting point of the region to be deleted, as in

  ```
  {Line:5, Column:1}
  ```

  Of course if you pass a **Start** record, you should not pass a **StartLine** and **StartColumn**.

- **End**: A record that specifies the ending point of the region to be deleted, as in

  ```
  {Line:10, Column:1}
  ```

  Of course, if you pass an **End** range, you should not pass an **EndLine** and **EndColumn**.

Here are some examples showing the various parameters that can be passed to `Delete()`:

```
--  delete a single line including end-of-line:
file = open("c:\temp\qwintest\select.txt");
file.delete(startline: 3);
result = dialog("Line 3 is deleted");
file.close(discardchanges: true);

--  delete multiple lines:
file = open("c:\temp\qwintest\select.txt");
file.delete(startline: 3, endline: 5);
result = dialog("Line 3 through 5 are deleted");
file.close(discardchanges: true);

--  delete a portion of a single line:
file = open("c:\temp\qwintest\select.txt");
file.delete(startline: 3, startcolumn: 2, endcolumn: 5);
result = dialog("Line 3, columns 2 through 5 are deleted");
file.close(discardchanges: true);

--  delete a multi-line and multi-column region:
file = open("c:\temp\qwintest\select.txt");
file.delete(startline: 3, startcolumn: 2, endline: 7, endcolumn: 5);
result = dialog(
       "Line 3, column 2 through line 7, column 5 are deleted");
file.close(discardchanges: true);

--  delete part of multiple lines using start/end record structures:
file = open("c:\temp\qwintest\select.txt");
startrec = {line: 2, column: 2};
endrec   = {line: 5, column: 3};
file.delete(start: startrec, end: endrec);
result = dialog("Start 2,2 through End 5,3");
file.close(discardchanges: true);

--  delete a selection consisting of one character:
file = open("c:\temp\qwintest\select.txt");
file.select(startline: 3, startcolumn: 6, endcolumn: 7);
file.delete();
result = dialog("Delete selection (last character of line 3)");
file.close(discardchanges: true);
```

If you specify

`rectangular: true`

as one of the parameters in your `Delete()` method calls, Qedit delete these columns only. Here are some typical rectangular `Delete()` calls:

```
-- Delete a rectangle:
-- Columns 15 through 20 of lines 11 through end of file
result = file.delete(startline:   11,
       startcolumn: 15,
       endline:     file.linecount,
       endcolumn:   20,
       rectangular: true);

-- Delete column 1 of all lines in the file
result = file.delete(startline:   1,
       startcolumn: 1,
       endline:     file.linecount,
       endcolumn:   1,
       rectangular: true);
```

Detab() Document Method

The positional parameters are:

- **StartLine** (integer)
- **EndLine** (integer)

The return value is boolean.

This method converts all tab characters to spaces in document lines, using the current tab stops settings as a guide. See also the "Entab() Document Method" on page 128, which is the opposite of `Detab()`. If a tab represents a single space, it is converted to a space by `Detab()`.

`Detab()` returns `True` if the operation is successful, otherwise it returns `False`. The two parameters specify the line range to "detab". If you do not specify any parameters, the entire file is detabbed. `Detab()` ignores selections.

- **StartLine**: An integer that specifies the first line to be detabbed. If you don't specify **StartLine**, then line 1 is assumed.
- **EndLine**: An integer that specifies that last line in the region. If you don't specify **EndLine**, then the last in the document is assumed i.e., file.linecount.

When working with tabs, you will find that enabling **file.ShowInvisibles** and **qedit.UseRulerBar** are handy for debugging. Here are some example calls to `Detab()`:

```
file = open("c:\robdev\manage.html");
-- Detab the Entire File
    file.Detab();
-- detab lines 2 through 9
    file.detab(startline:2, endline: 9);
```

Entab() Document Method

The positional parameters are:

- **StartLine** (integer)
- **EndLine** (integer)

The return value is boolean.

This method changes spaces to a combination of tab characters and spaces, using the document's tab stops settings as a guide. See also the "Detab() Document Method" on page 127, which is the opposite of `Entab()`. `Entab()` never converts single spaces to tabs.

`Entab()` returns `True` if the operation is successful, otherwise it returns `False`. The two parameters specify the line range to "entab". If you do not specify any parameters, the entire file is entabbed. `Entab()` ignores selections.

- **StartLine** An integer that specifies the first line to be entabbed. If you don't specify **StartLine**, then line 1 is assumed.
- **EndLine** An integer that specifies that last line in the region. If you don't specify **EndLine**, then the last in the document is assumed i.e., file.linecount.

When working with tabs, you will find that enabling **file.ShowInvisibles** and **qedit.UseRulerBar** are handy for debugging. Here are some example calls to `Entab()`:

```
file = open("c:\robdev\manage.html");
-- Entab Part of a File
    file.Entab(startline: 2, endline: 3);

-- Entab an entire file
    file.entab();
```

Find() Document Method

The positional parameters are:

- **String** (string)
- **Pattern** (string)
- **Regexp** (string)
- **Backwards** (boolean)
- **IgnoreCase** (boolean)
- **SelectionOnly** (boolean)
- **StartAtTop** or **EntireFile** (boolean)
- **Smart** (boolean)
- **LeftColumn** (integer)
- **RightColumn** (integer)
- **StartLine** (integer)
- **EndLine** (integer)
- **ReplaceWith** (string)

The return value is boolean.

This method has two major functions:

- Find an occurrence of a string in a document
- Replace one or more occurrences of a string with a different one.

There are many parameters to `Find()` but most of them are optional. `Find()` returns `True` if the operation is successful, otherwise it returns `False`. Certain document properties are updated by `Find()`: **LastFoundLength**, **LastFoundColumn**, and **LastFoundLine**. You can check these after a call to `Find()` to see what was found.

- **String**: A string to be searched for. Unless **IgnoreCase** or **Smart** is `True`, the string must be matched exactly, character by character.
- **Pattern**: A string that specifies a Qedit-style "match pattern". For example, "@bob@green@" matches any occurrence of the string "bob" anywhere in a line, followed by zero or more intervening characters and then the string "green". The pattern match control characters are the same as those used in the standard host-based Qedit. See the Qedit for Windows User Manual or online help for details.
- **Regexp**: A string containing a regular Expression to match. Regular expressions are much more powerful than patterns. For example, the expression "^..$" matches any line that contains exactly two characters. See the Qedit for Windows User Manual or online help for details.

Note that the **String**, **Pattern** and **Regexp** parameters are mutually exclusive that is, in a single call, you can only specify one of them.

- **SelectionOnly**: A boolean that, if `True`, causes the search to occur only in the current selection. See "Select() Document Method" on page 143 for details on how to select text.
- **EntireFile**: A boolean that, if `True`, causes the operation to occur from the start of the file. The default is to search from the current cursor position. **StartAtTop** is also accepted as an alias parameter.
- **StartLine**: An integer specifying the line to start searching from. If **EndLine** is not specified, the search continues until end of file. The default is to search from the current cursor position. You can also specify **SelectionOnly** to search within selected text instead.
- **EndLine**: An integer that specifies the last line to search for the string. If **StartLine** is not specified, the search starts from the current cursor position or at line 1, if **EntireFile** is `True`.
- **LeftColumn**: An integer specifying the leftmost column of the string match. Column numbers start at 1. If **LeftColumn** is specified but **RightColumn** is not, `Find()` searches to the end of each line.
- **RightColumn**: An integer that specifies the rightmost column in any line that can match the string. If **LeftColumn** is not specified, the string match can occur between the first column of each line and **RightColumn**. The first column depends on the **Language** of the file. For most languages, it is 1. For some languages such as Cobol, the first column is 7. See the Qedit for Windows User Manual for details. If both **LeftColumn** and **RightColumn** are specified, the string must match within those columns, inclusive.
- **Backwards**: A boolean that, if `True`, searches backward from the current position. The default is to search forward.
- **IgnoreCase**: A boolean that, if `True`, causes the case of alphabetic characters to be ignored in the search. The default is `False`.
- **Smart**: A boolean that, if `True`, matches a string only if it is not an embedded string i.e. it is considered a word. The definition of a word is dependent on the **Language** of the file. See the Qedit for Windows User Manual for details.
- **ReplaceWith**: A string that replaces each of the matches. When used, the method returns the number of occurrences replaced.

See "Prompt Before Replacing" on page 77 for an example that selectively replaces matches in any file.

Here are some examples on how you can use `Find()`:

```
-- Search for the next "Bob" in the file
findResult = file.find(string: "Bob");

-- Search for the first "Bob" within columns 21 to 32
findResult = file.find(string: "Bob", entirefile: true,
       leftcolumn: 21, rightcolumn: 32);

-- Search for a Regular Expression:
-- Find the next line containing only 2 characters
findResult = file.find(regexp: "^..$");

-- Replace 1 with 9 in the entire file
replaceResult = file.find(string:"1", entirefile: true,
       replacewith:"9");

-- search for a string within a column range (exactly in columns 1-3)
file.select(line: 1, column: 1);  -- Position caret at 1,1
findResult = file.find(string: "Bob", leftcolumn: 1, rightcolumn: 3);

-- search to the right of a column, starting from the cursor position
findResult = file.find(string:"5", leftcolumn: 20);

-- search within a selection only
file.select(startline: 1, endline: 13, startcolumn: 2, endcolumn: 2);
findresult = file.find(string:"1", selectiononly: true);

-- search 3 lines of the file for a string
findResult = file.find(startline: 13, endline: 15, string:"1");

-- search with the Smart Option, should ignore "Summary"
findResult = file.find(string: "Sum", entirefile: true, smart: true);

-- Ignore the Case and use Smart Option, matches "sum", ignores "sums"
findResult = file.find(string: "Sum", ignorecase: true, smart: true);

-- search backwards from a point
file.select(line: 13, column: 22);    -- Position caret at 13, 22
findResult = file.find(string: "Qedit", backwards: true);

-- replace "Bob" with "Robert" from current position on
replaceResult = file.find(string: "Bob", ignorecase: true, smart: true,
       replacewith: "Robert");

-- Search for "Bob" followed by "Green", anywhere in line, any case.
findResult = file.find(pattern: "@Bob@Green@", ignorecase: true);
```

After a successful string match, the current selection becomes the matched text. Here is the `CheckString()` subroutine from the Qedit test suite. It is called after a `Find()` to verify that the text matched is the expected text:

```
sub CheckString(file, testName, theString)

returnValue = true;

foundList = file.getselectedtext();
if foundList = {} then
    foundString = "";
else
    if length(foundList) > 1 then
        TestFailure(file, testName, "found more than one string");
    endif
    foundString = foundList[1];
endif

if foundString <> theString then
    TestFailure(file, testName, "File selection " +
            string(foundString) + " does not match " +
            string(theString));
    returnValue = false;
endif

return returnValue;

endsub
```

FindAll() Document Method

The positional parameters are:

- **String** (string)
- **Pattern** (string)
- **Regexp** (string)
- **IgnoreCase** (boolean)
- **Smart** (boolean)
- **LeftColumn** (integer)
- **RightColumn** (integer)
- **StartLine** (integer)
- **EndLine** (integer)
- **SearchReferenced** (integer)

The return value is a record.

This method is used to find all occurrences of a string. When executed on a host file, the search operation is performed by the server. Only the matching lines are transmitted back to the client.

There are many parameters to `FindAll()` but most of them are optional.

- **String**: A string to be searched for. Unless **IgnoreCase** or **Smart** is `True`, the string must be matched exactly, character by character.
- **Pattern**: A string that specifies a Qedit-style "match pattern". For example, "@bob@green@" matches any occurrence of the string "bob" anywhere in a line, followed by zero or more intervening characters and then the string "green". The pattern match control characters are the same as those used in the standard host-based Qedit. See the Qedit for Windows User Manual or online help for details.

- **Regexp**: A string containing a regular Expression to match. Regular expressions are much more powerful than patterns. For example, the expression "^..$" matches any line that contains exactly two characters. See the Qedit for Windows User Manual or online help for details.

Note that the **String**, **Pattern** and **Regexp** parameters are mutually exclusive that is, in a single call, you can only specify one of them.

- **StartLine**: An integer specifying the line to start searching from. If **EndLine** is not specified, the search continues until end of file. The default is to search from the current cursor position.
- **EndLine**: An integer that specifies the last line to search for the string. If **StartLine** is not specified, the search starts from the current cursor position.
- **LeftColumn**: An integer specifying the leftmost column of the string match. Column numbers start at 1. If **LeftColumn** is specified but **RightColumn** is not, `FindAll()` searches to the end of each line.
- **RightColumn**: An integer that specifies the rightmost column in any line that can match the string. If **LeftColumn** is not specified, the string match can occur between the first column of each line and **RightColumn**. The first column depends on the **Language** of the file. For most languages, it is 1. For some languages such as Cobol, the first column is 7. See the Qedit for Windows User Manual for details. If both **LeftColumn** and **RightColumn** are specified, the string must match within those columns, inclusive.
- **IgnoreCase**: A boolean that, if `True`, causes the case of alphabetic characters to be ignored in the search. The default is `False`.
- **Smart**: A boolean that, if `True`, matches a string only if it is not an embedded string i.e. it is considered a word. The definition of a word is dependent on the **Language** of the file. See the Qedit for Windows User Manual for details.
- **SearchReferenced**: An integer specifying the search range in terms of files referenced in the current file. By default, `FindAll()` scans only the current file. This is the equivalent of a value of 0. Instead of remembering the numeric values for each option, you can use the **SearchReferenced** application constants.

  If you wish to search files referenced in Include statements, set **SearchReferenced** to 1. Include statements must start with `$Include`, `!Include`, `#Include` and `.Include`. The leading characters, "$", "!", "#" or "." must be in the first column. The Include keyword does not have to start immediately. It can be separated from the leading character by spaces. Whatever follows the Include keyword is used as a filename.

  If the statement starts with `#Include`, Qedit assumes it's a C-type construct. If the filename is enclosed between a less-than "<" and a greater-than ">", Qedit assumes this is a system library file. Qedit then prepends the filename with `/usr/include/`. If the system library name starts with "`../h`", Qedit assumes it's a UNIX Kernel file and ignores it completely.

  If you wish to search files referenced in `Use` statements, set **SearchReferenced** to 2. Use statements must have the keyword Use starting in the first column. Whatever follows is used as a filename.

  If you wish to search files referenced in COBOL `Copy` statements, set **SearchReferenced** to 3. COBOL Copy libraries, also known as Copylib on MPE hosts, must follow COBOL syntax rules. On MPE hosts, a typical statement is:

```
Copy member-name Of library-name.
```

`Member-name` is mandatory and identifies the statements to be included. The `library-name` is the filename where the member is stored. It is optional and defaults to COPYLIB. On UNIX hosts, a typical statement is:

```
Copy library-name Of path-name.
```

`Library-name` is actually a filename on the UNIX host and is mandatory. The `path-name` is optional and represents the directory path where the `Library-name` resides. Thus, Qedit appends `library-name` to `path-name` to build the qualified filename.

Whichever option is used, the `FindAll()` method scans individual files only once, the first time they are encountered.

`FindAll()` returns a record containing of series of nested records. Each nested record contains a number of elements:

- **Filename**: (string) this element contains the fully-qualified filename to which the following elements are refering to.
- **ErrorCode**: (integer) this element indicates if there has been a problem opening the file. If the value is 0, the access was successful. If it's not 0, it gives some indication as to the cause of the problem.
- **ErrorString**: (string) this elements contains an error message describing the problem. It contains an empty string if ErrorCode is 0.
- **Lines**: (record) this element is a record containing all matching lines found in the file. Each record contains the following elements:

  Text: (string) this element contains the whole line

  Line: (integer) this element contains the relative record number for that line

  LineNumber: (integer) this element contains the absolute line number for that line. This number has 3 implied decimal places.

The sample script demonstrates a typical way of extracting the information returned by the `FindAll()` method. Error messages are written to the **Script Control** panel and actually lines of text are written to a new local file.

```
OutputFile = newfile();
findresult = fileH.FindAll(string: SearchString,
                           searchreferenced: qedit.SearchInclude,
                           ignorecase: true);

repeat for fileResult in findresult
    if fileResult.ErrorCode <> 0 then
        writelog ("Unable to open " + fileResult.Filename);
        writelog ("Error # " + string(fileResult.ErrorCode));
        writelog ("Message:" + fileResult.ErrorString);
    else
        if length(fileResult.Lines) <> 0 then
            fileHeader = "Found " + string(length(fileResult.Lines)) +
                         " lines in " + fileResult.Filename;
            OutputFile.Insert(fileHeader);
            OutputFile.Insert({"",""});   -- Skip to next line
            repeat for lineDescriptor in fileResult.Lines
                foundLine = string(lineDescriptor.Line) + ": " +
                            lineDescriptor.text;
                OutputFile.Insert(foundLine);
                OutputFile.Insert({"",""});   -- Skip to next line
            endrepeat
        else
            writelog ("No lines found in " + fileResult.Filename);
        endif
    endif
endrepeat
```

GetSelectedText() Document Method

There are no parameters and the return value is a record.

This method returns a list that represents the current selection. If the selection represents the caret i.e., there is no text selected, the returned list is empty. If the selection is on a single line, the list contains one element. If the selection spans multiple lines, each element in the list represents one line. See also the "GetText() Document Method" on page 135, which is similar but specifies the selection as parameters to the call.

There are no parameters to `GetSelectedText()`.

See the `CheckString()` subroutine shown in the section about "Find() Document Method" on page 129 for an example of how to analyze an unknown selection, looking for a specific result.

Here is an example of how to use `Select()` and `GetSelectedText()` to retrieve the first 2 lines of a file:

```
file = open("c:\personal\select.txt");

--  select the first 2 lines
file.select(startline: 1, endline: 2);

-- get the selected lines
textlist =  file.GetSelectedText();

-- step through the list, writing lines to log window
repeat for nextLine in textlist
   writelog(nextLine);
endrepeat
```

GetText() Document Method

The positional parameters are:

- **StartLine** or **Line** (integer)

- **StartColumn** or **Column** (integer)
- **EndLine** (integer)
- **EndColumn** (integer)
- **Start** (record)
- **End** (record)
- **Range** (record)
- **Rectangular** (boolean)

The return value is a record.

This method returns a list of lines from a document. Note that a list is always returned, even if only a portion of a single line was requested in the `Gettext()` method call. Each element in the returned list represents one line in the file. Empty lines are also represented by an element containing an empty string.

`GetText()` returns True if the operation is successful, otherwise False. The parameters to define the region to be copied are used in the same way as in the `Select()` method. Here are all of the parameters to the `GetText()` method:

- **StartLine**: An integer specifying the starting line of the region to get. Line is a valid alias. If **StartColumn** is not specified, the entire start line is selected.
- **StartColumn**: An integer specifying the first column to get from **StartLine**. **Column** is a valid alias.
- **EndLine**: An integer specifying the last line of the region to get. As in `Select()`, if an **EndColumn** is not specified, the entire last line is selected.
- **EndColumn**: An integer specifying the last column to get from **EndLine**.
- **Rectangular**: A boolean that, if `True`, specifies a columnar retrieval. In that case, the region is a Rectangle, starting at an upper-left corner specified by line and column, and ending at a lower-right corner. By default, the selected region is a normal stream of characters instead of a columnar area.
- **Range**: A record that specifies the entire range to get. If this parameter is used, you don't specify any of the preceding parameters. Thus the **Range** parameter could appear as:

  ```
  {Start: {Line:5, Column:1}, End:{Line:10, Column:1},
  Rectangular: true}
  ```

- **Start**: A record that specifies the starting point of the region, as in

  ```
  {Line:5, Column:1}
  ```

  Of course, if you pass a **Start** record, you should not pass **StartLine** and **StartColumn**.

- **End**: A record that specifies the ending point of the region, as in

  ```
  {Line:10, Column:1}
  ```

  Of course, if you pass an **End** range, you should not pass **EndLine** and **EndColumn**.

There are several forms of calls to `GetText()`, depending upon how you specify the range of text to be retrieved. Here are some examples:

```
file = open("c:\personal\select.txt");

--  get a single line including end-of-line:
buf = file.gettext(line: 3);

--  get multiple lines including end-of-line:
buf = file.gettext(startline: 3, endline: 8);

--  get part of a single line including end-of-line:
buf = file.gettext(startline: 3, startcolumn: 6);

--  get a portion of a single line:
buf = file.gettext(startline: 3, startcolumn: 3, endcolumn: 6);

--  get a single character:
buf = file.gettext(startline: 3, startcolumn: 6, endcolumn: 7);

--  get portions of multiple lines:
buf = file.gettext(startline: 3, startcolumn: 6,
           endline: 5, endcolumn: 2);

--  get part of multiple lines using start/end record structures:
startrec = {line: 2, column: 2};
endrec   = {line: 5, column: 3};
buf = file.gettext(start: startrec, end: endrec);

--  do a full get using the range record structure:
rangerec = {{line: 4, column: 1}, {line: 9, column: 2}};
buf = file.gettext(range: rangerec);
```

The `CompareFile()` subroutine is a general purpose subroutine that uses `GetText()` to compare the contents of two files. If there are any differences, it writes an error message to the log window. `CompareFile()` returns `True` if the files are identical and `False` otherwise. See "Comparing Two Files" on page 69.

Guides() Document Method

The positional parameters are:

- **SetAt** (integer)
- **SetEvery** (integer)
- **ClearAt** (integer)
- **ClearAll** (boolean)

The return value is boolean.

This method sets and clears guide lines in a document. When you `Close()` a document, the guide settings are not saved in the document itself. They are saved in your personal document database. When you `Open()` the file again, the guides are set based on the settings from the document database.

The purpose of the `Guides()` method is to either:

- Set a guide at a specific column
- Set guides at a fixed interval e.g. every 5 columns
- Clear a guide at a specific column
- Clear all guides

`Guides()` returns `True` if the operation is successful, otherwise it returns `False`. There are four parameters to `Guides()` and they are mutually exclusive i.e. you

can only specify one in a given call. Columns are numbered from one and match what you see on the ruler bar.

- **SetAt**: An integer that specifies a column number where a guide will be set.
- **SetEvery**: An integer that specifies guides at regular intervals.
- **ClearAt**: An integer that specifies a specific column where a guide should be removed.
- **ClearAll**: A boolean that, if `True`, clears all guides in the file. Passing `False` makes no sense and doesn't change any guides.

Column numbers start at 1. For most files, this number matches the number on the **Ruler bar**. However, the **Ruler bar** for Cobol source files starts in column 7. It is the first column where you can set a guide. This means that if you want to set a guide in column 12, you have to use:

```
file.guides(setat: 6);
```

If you want to set guides at regular interval, you can use the **SetEvery** keyword. For example, on files where the first column is 1, coding

```
file.guides(SetEvery:5);
```

sets guides at column 6, 11, 16, 21, etc. On a Cobol source file, the guides are at column 12, 17, 22, etc.

In all cases, **SetEvery** does not clear existing guides, so it is usually good practice to first call

```
file.guides(ClearAll);
```

Here are some example calls to `Guides()`:

```
file = newfile();
-- Use the SetAt parameter to set a guide at column 6
file.guides(setat: 6);

-- Use the ClearAt parameter to clear a single guide at column 5
file.guides(clearat: 5);

-- Use the ClearAll parameter to clear all guides.
file.guides(clearall: true);

-- Use the SetEvery parameter to set a guide every 6 (clear first)
file.guides(clearall: true);
file.guides(setevery: 6);
```

Insert() Document Method

The positional parameters are:

- **Text** (any type)
- **At** (record)

The return value is boolean.

This method inserts text into the document. You must do something special to create a dynamic (i.e., non-constant) place to insert the text. You can insert one or more lines in a single call because the **Text** parameter can be either a string constant or a list.

Here are the parameters to the `Insert()` method:

- **At**: A record specifying the line and column where the **Text** should be inserted, as in:

  `{line: 5, column: 10}`

- **Text**: A string or a list of strings to be inserted. Each element of the list creates a new line.

`Insert()` returns `False` if anything goes wrong, otherwise it returns `True`.

Here are two common ways to call `Insert()`:

```
file = newfile();

--  Insert a single line at the start of the file (without a NewLine):
file.insert(at:{line: 1, column: 1}, text: "Single line");
result = dialog("File with single line (no NewLine)");

--  Insert two lines after the one above (each with a NewLine):
file.insert(at:{line: 2, column: 1}, text: {"Line Two", "Line Three"});
result = dialog("File with three lines (all with NewLines)");
```

Here is an example showing how to specify a dynamic (non-constant) insertion point. It also contains a helper subroutine which creates the data structure needed to specify the insertion line and column point:

```
sub create_at (line, column)

    result = {};       --  Empty record structure

    result.line   = line;      -- "at" needs a line
    result.column = column;    -- and a column
    return result;             -- return the record structure

endsub;

-- Mainline

file = open("c:\personal\testdata.txt");
row = 5;    --  Insert a new line at line 5
where = create_at(row, 1);
file.insert(at: where, text: "New Text");

--  Or do it all in the Insert() call:
row = 6;    --  Insert at line 6 now
file.insert(at: create_at(row,1), text: "New Text");

file.Close();
```

Here is an example of the `FillFile()` subroutine used in the Qedit test suite. It initializes a file with a known set of text lines. Note that each concatenation operation, denoted with a plus sign "+", on the text list adds one more line. This is how you create a data structure so that a line is inserted **with** a NewLine.

```
sub FillFile(file)

    lines = {};    -- creates an empty list

    file.delete(startline: 1, endline: file.linecount);  -- empty file
    file.tabs(ClearAll: true);
    file.tabs(SetEvery: 5);

    lines = lines + "";    -- Empty line
    lines = lines + "A";
    lines = lines + "";    -- Empty line
    lines = lines + "bb";
    lines = lines + "";    -- Empty line
    lines = lines + "CCC";
    lines = lines + "";    -- Empty line
    lines = lines + "dddd";
    lines = lines + "";    -- Empty line
    lines = lines + "EEEEE";

    file.insert(at:{line: 1, column: 1}, text: lines);

endsub
```

This sample subroutine inserts 10 lines in a file. There will be five lines with letters and a blank line between each one.

InsertColumn() Document Method

The positional parameters are:

- **StartLine** or **Line** (integer)
- **EndLine** (integer)
- **AtColumn** (integer)
- **RightMargin** (integer)
- **Text** (string)

The return value is boolean.

This method inserts a string at a specified column in one or more lines in a document. `InsertColumn()` returns `True` if the insertion is successful, otherwise it returns `False`.

Here are the parameters to the `InsertColumn()` method:

- **StartLine**: An integer specifying the first line to be inserted into. A valid alias is **Line**.
- **EndLine**: An integer specifying the last line to be inserted into.
- **AtColumn**: An integer specifying the column to insert in front of. The number for the first column depends on the **Language** of the file. Column numbers start at 1 for most languages. In some languages such as Cobol, the first column is 7. See the Qedit for Windows User Manual for details.
- **RightMargin**: An integer specifying the column of the right margin. Any existing text in a line from this column on will not be changed by the insertion. If characters would normally have been right-shifted into that column, they will be discarded instead.
- **Text**: A string specifying the text to be inserted in each line. With `InsertColumn()`, it is only possible to insert the same text i.e., a constant, in each line. If you wish to copy some varying text and insert it as a column, see "Paste() Document Method" on page 141.

See "Draw a Box" on page 75 for uses of `InsertColumn()` in context.

Here is a subroutine to insert a rectangle of stars:

```
sub insertbox(file, startl, endl, startc, endc)

    width = endc - startc + 1;

    stars = "";
    repeat for x from 1 to width by 1
        stars = stars + "*";
    endrepeat

    file.insertcolumn(startline: startl,
            endline:   endl,
            atcolumn:  startc,
            text:      stars);
endsub
```

Paste() Document Method

There are no parameters and the return value is boolean.

This method pastes the entire contents of the clipboard into a document.

If the current cursor is a caret i.e. there is no selection, the text is inserted at that position. If there is a selection, `paste()` replaces the current selection.

If the current selection is rectangular, the rectangle is replaced with the clipboard, except that lines are truncated instead of folded. Normally you would do a rectangular `Copy()` or `Cut()` to the clipboard before doing a `Paste()` into a rectangular selection. If you want to insert a rectangle instead of replacing an existing one, see "Insert a Rectangular Selection" on page 72.

`Paste()` returns `True` if the operation is successful, otherwise it returns `False`. There are no parameters.

Here is a subroutine that pastes the clipboard into a rectangular region of the document:

```
sub InsertClipboard(file, startl, endl, startc, endc)

    file.select(startline:   startl,
            startcolumn: startc,
            endline:     endl,
            endcolumn:   endc,
            rectangular: true);
    file.paste();

endsub
```

PrintOnHost() Document Method

The positional parameters are:

- **StartLine** (integer)
- **EndLine** (integer)
- **DeviceName** (string)
- **Numbered** (boolean)
- **Shift** (boolean)

The return value is boolean.

This method sends one or more lines to a printing device on the host. If **StartLine** and **EndLine** are not specified, Qedit prints the whole file. `PrintOnHost()` returns `True` if the text has been successfully printed, otherwise it returns `False`.

Here are the parameters to the `PrintOnHost()` method:

- **StartLine**: An integer specifying the first line to print. If **Endline** is not specified, Qedit prints from **StartLine** to the end of the file.
- **EndLine**: An integer specifying the last line to print. If **StartLine** is not specified, Qedit prints from the start of the file up to and including **Endline**.
- **DeviceName**: The name of the print device on the host.
- **Numbered**: A boolean that, if `True`, requests that line numbers be included on the output.
- **Shift**: A boolean that, if `True`, shifts lines four characters to the right to provide a wider left margin.

PrintOnLocal() Document Method

The positional parameters are:

- **StartLine** (integer)
- **EndLine** (integer)

The return value is boolean.

This method sends one or more lines to the default local printer. If **StartLine** and **Endline** are not specified, Qedit prints the whole file. `PrintOnLocal()` returns `True` if the text has been successfully printed, otherwise it returns `False`.

Here are the parameters to the `PrintOnLocal()` method:

- **StartLine**: An integer specifying the first line to print. If **Endline** is not specified, Qedit prints from **StartLine** to the end of the file.
- **EndLine**: An integer specifying the last line to print. If **StartLine** is not specified, Qedit prints from the start of the file up to and including **Endline**.

Save() Document Method

There are no parameters and the return value is boolean.

This method saves changes to a document, whether local or host. You have to use `SaveAs()` on files created with the `Newfile()` application method. If you try to use `Save()`, an error is generated.

This example shows how you would use `Save()` after you `Open()` a file and replaced some strings:

```
file = open("c:\webpages\index.html");
file.find(entirefile: true, Smart: true, IgnoreCase: true,
          string: "Acme Software", replacewith: "IBM/Acme Inc.");
file.save();
file.close();
```

SaveAs() Document Method

The positional parameters are:

- **Filename** (string)

- **ForceOverwrite** (boolean)

The return value is boolean.

This method saves a document with a new filename without changing the connection, but does not `Close()` the file. You can use `SaveAs()` to make a `Newfile()` permanent or to rename an existing file opened with `Open()`.

Here are all of the parameters to the `Saveas()` method:

- **Filename**: A string specifying the pathname of the file.
- **ForceOverwrite**: A boolean. Specify `True` to purge an existing file with the same name. By default, a name collision causes an error.

The sample code below creates a file using `Newfile()`, then uses `SaveAs()` to save it.

```
tofile   = newfile(connection: connectionname);

tofile.saveas(filename: filename  -- give new file this name
           ,forceoverwrite: true  -- ok to purge old
           ); -- saves file, but does not close it

tofile.close();
```

Select() Document Method

The positional parameters are:

- **StartLine** or **Line** (integer)
- **StartColumn** or **Column** (integer)
- **EndLine** (integer)
- **EndColumn** (integer)
- **Start** (record)
- **End** (record)
- **Range** (record)
- **Rectangular** (boolean)

The return value is boolean.

This method creates a selection in the document. You can also use `Select()` to position the caret i.e., the logical insertion point.

`Select()` returns `True` if the selection is successful, otherwise it returns `False`. The many parameters provide alternative ways of specifying the selected region. If you do not specify an end point for your selection, your are positioning the caret i.e. the cursor, instead.

- **StartLine**: An integer specifying the starting line of the region to select. **Line** is a valid alias. If **StartColumn** is not specified, the entire start line is selected.
- **StartColumn**: An integer specifying the first column to be selected in **StartLine**. **Column** is a valid alias.
- **EndLine**. An integer specifying the last line of the region to be selected. If **EndColumn** is not specified, the entire last line is selected.
- **EndColumn**: An integer specifying the last column to be selected in **EndLine**.

- **Range**. A record that specifies the entire range to select, as in

  If you specify a **Range**, you don't specify any of the preceding parameters. Thus a complete **Range** parameter could be:

  ```
  {Start: {Line:5, Column:1}, End:{Line:10, Column:1},
  Rectangular: true}
  ```

- **Rectangular**: A boolean that, if `True`, specifies a rectangular selection. The region will be a Rectangle, starting at an upper-left corner specified by line and column and ending at a lower-right corner. You may `Paste()` into or `Delete()` a rectangular selection. If `False`, the selected region is a normal stream of characters instead of a columnar area.

- **Start**: A record that specifies the starting point of the region, as in

  ```
  {Line:5, Column:1}
  ```

  Of course if you pass a **Start** record, you should not pass **StartLine** and **StartColumn**.

- **End**: A record that specifies the ending point of the region, as in

  ```
  {Line:10, Column:1}
  ```

  Of `course`, if you pass an **End** range, you should not pass **EndLine** and **EndColumn**.

See the "Paste() Document Method" on page 141 for an example that selects a rectangle and pastes the clipboard into it. Below are some example calls to `Select()`. We use the `GetSelectedText()` method to show what has been selected. Notice that we convert the `buf` variable, the result from `GetSelectedText()`, into a string to format the logged information.

```
file = open("c:\personal\select.txt");

--  position the caret to 3,2 (there is no selection in this case)
file.select(line: 3, column: 2);
writelog("Caret at 3,2");

--  select a single line including end-of-line:
file.select(line: 3);
buf = file.getselectedtext();
writelog("Line 3: " + string(buf));

--  select multiple lines including end-of-line:
file.select(startline: 3, endline: 8);
buf = file.getselectedtext();
writelog("Line 3 through 8: " + string(buf));

--  select a portion of a single line:
file.select(startline: 3, startcolumn: 3, endcolumn: 6);
buf = file.getselectedtext();
writelog("Line 3, columns 3 through 6: " + string(buf));

--  select a single character:
file.select(startline: 3, startcolumn: 6, endcolumn: 6);
buf = file.getselectedtext();
writelog("Line 3,3 column 6: " + string(buf));

--  select portions of multiple lines:
file.select(startline: 3, startcolumn: 6, endline: 5, endcolumn: 2);
buf = file.getselectedtext();
writelog("Line 3, column 6 to line 5, column 2: " + string(buf));

--  select part of multiple lines using start/end record structures:
startrec = {line: 2, column: 2};
endrec   = {line: 5, column: 3};
file.select(start: startrec, end: endrec);
buf = file.getselectedtext();
writelog("Line 2,2 through line 5,3: " + string(buf));


--  do a full selection using the range record structure:
rangerec = {{line: 4, column: 1}, {line: 9, column: 2}};
file.select(range:rangerec);
buf = file.getselectedtext();
writelog("Line 4,1 through line 9,2: " + string(buf));
```

SelectAll() Document Method

There are no parameters and the return value is boolean.

This method selects all the lines in the file.

This example shows how you would use `SelectAll()` to copy a file to the clipboard and paste it in another file:

```
file = open("c:\webpages\index.html");
file.selectall();
file.copy();

otherfile.activate();  -- Switch to the other document
otherfile.paste();
```

SetWidth() Document Method

There is only one positional parameter:

- **NewWidth** (integer)

The return value is boolean.

This method specifies the maximum document width. You can only change the width of certain languages such as **Text** or **Data**. See the Qedit for Windows User Manual for details.

`SetWidth()` returns `True` if the operation is successful, otherwise it returns `False`. There is only one parameter to `SetWidth()`:

- **NewWidth**: An integer specifying the maximum document width. The largest size currently supported in Qedit is 8,172 characters.

This example shows how to create a new file and set the maximum width to 80 characters:

```
file = newfile();
file.setwidth(80);
```

ShiftLeft() Document Method

The positional parameters are:

- **Columns** (integer)
- **StartLine** (integer)
- **EndLine** (integer)

The return value is boolean.

This method shifts the specified lines left a specific number of columns. `ShiftLeft()` does not shift non-blank characters off the line. If the shift count is too high, the line ends up left justified. See also the "ShiftRight() Document Method" on page 146.

`ShiftLeft()` returns `True` if the shift is successful, otherwise it returns `False`.

Here are the parameters to the `ShiftLeft()` method:

- **Columns**: An integer specifying the number of columns by which the text should be shifted. If this parameter is not specified or if it is -1, the line is shifted to the next tab stop. Shift of a tab stop only works if there are tab characters in the file (it does not work on equivalent spaces).
- **StartLine**: An integer specifying the first line to shift. **StartLine** defaults to the first line of the file.
- **EndLine**: An integer specifying the last line to shift. **EndLine** defaults to the last line of the file.

Here are some examples of `ShiftLeft()`:

```
file = newfile();

-- Shift All Lines Left to the Next Tab Stop
file.shiftleft();

-- Shift lines 2 through 3 to the left by 5 columns
file.shiftleft(columns: 5, startline: 2, endline: 3);
```

ShiftRight() Document Method

The positional parameters are:

- **Columns** (integer)
- **StartLine** (integer)

- **EndLine** (integer)

The return value is boolean.

This method shifts the specified lines right by a specific number of columns. See also the "ShiftLeft() Document Method" on page 146.

`ShiftRight()` returns `True` if the shift is successful, otherwise it returns `False`.

Here are the parameters to the `ShiftRight()` method:

- **Columns**. An integer specifying the number of columns by which the text should be shifted. If this parameter is not specified or if it is -1, the line is shifted to the next tab stop. Shift of a tab stop only works if there are tab characters in the file (it does not work on equivalent spaces).
- **StartLine**: An integer specifying the first line to shift. **StartLine** defaults to the first line of the file.
- **EndLine**: An integer specifying the last line to shift. **EndLine** defaults to the last line of the file.

Here are some examples of `ShiftRight()`:

```
file = newfile();

-- Shift All Lines Right to the Next Tab Stop
file.shiftright();

-- Shift Lines 2 through 3 right by 5 columns
file.shiftright(columns: 5, startline: 2, endline: 3);
```

Tabs() Document Method

The positional parameters are:

- **SetAt** (integer)
- **SetEvery** (integer)
- **ClearAt** (integer)
- **ClearAll** (boolean)

The return value is boolean.

This method sets and clears tab stops in a document. When you `Close()` a document, the tab stops are not saved in the document itself. They are saved in your personal document database. When you `Open()` the file again, the tab stops are set based on the settings from the document database.

The purpose of the `Tabs()` method is to either:

- Set a tab stop at a specific column
- Set tab stops at a fixed interval e.g. every 5 columns
- Clear a tab stop at a specific column
- Clear all tab stops

`Tabs()` returns `True` if the operation is successful, otherwise it returns `False`. There are four parameters to `Tabs()` and they are mutually exclusive i.e. you can only specify one in a given call. Columns are numbered from one and match what you see on the ruler bar.

- **SetAt**: An integer that specifies a column number where a tab stop will be set.
- **SetEvery**: An integer that specifies tab stops at regular intervals.
- **ClearAt**: An integer that specifies a specific column where a tab stop should be removed.
- **ClearAll**: A boolean that, if `True`, clears all tab stops in the file. Passing `False` makes no sense and doesn't change any tab stops.

Column numbers start at 1. For most files, this number matches the number on the **Ruler bar**. However, the **Ruler bar** for Cobol source files starts in column 7. It is the first column where you can set a tab stop. This means that if you want to set a tab stop in column 12, you have to use:

```
file.tabs(setat: 6);
```

If you want to set tab stops at regular interval, you can use the **SetEvery** keyword. For example, on files where the first column is 1, coding

```
file.tabs(SetEvery:5);
```

sets tab stops at column 6, 11, 16, 21, etc. On a Cobol source file, the tab stops are at column 12, 17, 22, etc.

In all cases, **SetEvery** does not clear existing tab stops, so it is usually good practice to first call

```
file.tabs(ClearAll);
```

Here are some example calls to `Tabs()`:

```
file = newfile();
-- Use the SetAt parameter to set a tab stop at column 6
file.tabs(setat: 6);

-- Use the ClearAt parameter to clear a single tab stop at column 5
file.tabs(clearat: 5);

-- Use the ClearAll parameter to clear all tab stops.
file.tabs(clearall: true);

-- Use the SetEvery parameter to set a tab stop every 6 (clear first)
file.tabs(clearall: true);
file.tabs(setevery: 6);
```

DateTime Objects

The Qedit for Windows DateTime object is used for all date handling. The DateTime object stores dates and times in an internal format that supports dates from 1583 to 2583.

The date and time are treated separately within the object. But because all time values must relate to a specific date, you cannot deal with times without first having assigned a date to the DateTime object.

When the DateTime object is created, it is initialized with the PC's current date and time.

Creating a DateTime Object

You create a DateTime object by calling the `DateTime()` application method as in:

```
theDate = datetime();
```

You can now start using the variable `theDate` to obtain DateTime properties or invoke methods. For example, to get the year, month, and day you would use:

```
theYear  = theDate.Year;
theMonth = theDate.Month;
theDay   = theDate.Day;
```

To add one day to the date, you would do:

```
theDate.AddDays(1);
```

To format the date and time as a string, you would do something like:

```
currenttime = datetime();
timestamp   = currenttime.fmtshortdatetime();
       -- methods, not functions
```

You can also convert a DateTime object into a printable set of characters using the `String()` built-in function, but it will be formatted as:

```
<Object: DateTime 10/01/1999 11:53:45 AM>
```

See "Insert a Signature or a Timestamp" on page 71 for a sample script that inserts the current date and time into your file.

DateTime Constants

Some constants have been defined for DateTime objects. Note that these constants are specific to DateTime objects and are not global. Therefore, you precede them with the name of a DateTime object and a period.

Status Property Constants

The following constants are possible values for the **DateTime.Status** property. Please note that there are few cases where you would ever need to actually check the date status, since most DateTime objects are created by the system:

Name	Numeric Value
StatusNone	1
StatusYearLow	2
StatusYearHigh	3
StatusMonthLow	4
StatusMonthHigh	5
StatusDayLow	6
StatusDayHigh	7
StatusInitFormat	8
StatusHourLow	9
StatusHourHigh	10
StatusMinuteLow	11
StatusMinuteHigh	12
StatusSecondLow	13
StatusSecondHigh	14
StatusMonthChar	15

StatusDayChar	16
StatusYearChar	17
StatusDateInvalid	18

This is how you would check the status if you had to:

```
timestamp = datetime();
if (timestamp.status = timestamp.StatusNone)
    result = dialog("this timestamp is okay!");
endif
```

DayOfWeek Property Constants

The following constants are the possible values for the **Datetime.DayOfWeek** property:

Day	Numeric Value
Sunday	1
Monday	2
Tuesday	3
Wednesday	4
Thursday	5
Friday	6
Saturday	7

Here is how you would check the day of the week using one of these constants:

```
timestamp = datetime();
if (timestamp.dayofweek = timestamp.Sunday)
    result = dialog("Take the day off");
endif
```

DateTime Properties

The following are the properties of the Qedit datetime object.

Name	Type	Description
Year	Integer	4-digit year
Month	Integer	Month number, 1 to 12
Day	Integer	Day of the month, 1 to 31
Hour	Integer	Hour of the day, local time, 0 to 23
Minute	Integer	Minute of the hour, 0 to 59
Second	Integer	Second of the minute, 0 to 59
GMTHour	Integer	GMT hour, 0 to 23
GMTMinute	Integer	GMT minute of the hour
GMTSecond	Integer	GMT second of the minute
DayOfWeek	Integer	Day of the week, 1 to 7
DayOfYear	Integer	Day of the year, 1 to 366
Status	Integer	Date validity

For example, if you wanted to check whether today was in October, you would do:

```
timestamp = datetime();
if (timestamp.month = 10)
    result = dialog("this is October");
endif
```

DateTime Methods

Here are the methods for manipulating DateTime objects:

AddDays()	FmtShortDate()	SetCurrent()
Compare()	FmtShortDateTime()	SetDateTime()
DaysBetween()	FmtShortTime()	SetGMTDateTime()

Adddays() DateTime Method

There is only one positional parameter:

- **DaysToAdd** (integer)

There is no return value.

This method adds a specified number of days to a DateTime object. There is only one parameter to `Adddays()`:

- **DaysToAdd**: An integer that is added. If the value is negative, it subtracts from the date.

Here are some examples:

```
timestamp = datetime();
saveday = timestamp.day;

-- add 1 to today's date
timestamp.adddays(1);
result = dialog(string(timestamp.day));

-- subtract 1 from the revised date, should be back to today
timestamp.adddays(-1);
if (saveday = timestamp.day)
    result = dialog("adddays worked in both directions.")
endif
```

Compare() DateTime Method

There is only one positional parameter:

- **CompareDateTime** (object)

The return value is an integer.

This method compares a source DateTime object with another such object and tells you which is greater. The `Compare()` method returns one of three values:

Returned Value	Explanation
-1	The source date is less than the CompareDateTime parameter

0	The source date is equal to the CompareDateTime parameter
1	The source date is greater than the CompareDateTime parameter

There is one parameter to the `Compare()` method:

- **CompareDateTime**: Another DateTime object which is to be compared with the source object.

Here is an example:

```
timestamp = datetime();

anotherdate = datetime(); -- create a new object
anotherdate.adddays(1);   -- increments the date

qedit.showscp = true;  -- Display the Script Control Panel

compareresult = timestamp.compare(anotherdate);
if (compareresult=0)
    writelog("The timestamps are identical");
else
    if (compareresult=1)
        writelog("The object is greater than the parameter.");
    else
        if (compareresult=-1)
            writelog("The object is less than the parameter.");
        else
            result = dialog("Invalid value for compare result");
        endif
    endif
endif
```

DaysBetween() DateTime Method

There is only one positional parameter:

- **CompareDateTime** (object)

The return value is an integer.

This method computes the number of days between a source DateTime object and another such object. It basically subtracts the source object from the **CompareDateTime** parameter.

If `DaysBetween()` returns a positive integer, the **CompareDateTime** parameter is greater than the source object. If the return value is negative, the parameter is less than the source. If the result is zero, they are the same day but not necessarily the same time.

There is one parameter to the `DaysBetween()` method:

- **CompareDateTime**: Another DateTime object which is to be compared with the source object.

Here is an example of `DaysBetween()`. If you run it you should find that the objects have 0 days between them:

```
timestamp = datetime();
result = dialog("Wait a few seconds before getting another time");
anotherdate = datetime(); -- create a new object

qedit.showscp = true;  -- Displays the Script Control Panel

writelog(string(timestamp.daysbetween(anotherdate)));
if (timestamp.daysbetween(anotherdate)) = 0
    result = dialog("These are still the same day!");
endif
```

FmtShortDate() DateTime Method

There are no positional parameters and the return value is a string.

This method formats the date portion of a DateTime object into a string using the formatting conventions of the local computer. This is typically MM/DD/YYYY. You can change the format using the **Regional Settings** control panel in Windows.

`FmtShortDate()` returns a string and does not have any parameters.

Here is an example:

```
timestamp = datetime();

result = dialog("Today is " + timestamp.fmtshortdate());
```

FmtShortDateTime() DateTime Method

There are no positional parameters and the return value is a string.

This method formats the date and time into a string using the formatting conventions of the local computer. This is typically MM/DD/YYYY HH:MM:SS AM. You can change the format using the **Regional Settings** control panel in Windows.

`FmtShortDateTime()` returns a string and does not have any parameters.

Here is an example:

```
timestamp = datetime();

result = dialog("Date and time are " + timestamp.fmtshortdatetime());
```

FmtShortTime() DateTime Method

There are no positional parameters and the return value is a string.

This method formats the time portion into a string using the formatting conventions of the local computer. This is typically HH:MM:SS AM. You can change the format using the **Regional Settings** control panel in Windows.

`FmtShortTime()` returns a string and does not have any parameters.

Here is an example:

```
timestamp = datetime();

result = dialog("Now is " + timestamp.fmtshortTime());
```

SetCurrent() DateTime Method

There are no positional parameters and there is no return value.

This method sets the date and time of an object to the current date and time. It does not change the computer's date and time! It simply changes the date value inside the object.

There are no parameters to `SetCurrent()`.

Here is an example:

```
timestamp = datetime();
saveday = timestamp.day;

-- add 1 to today's date
timestamp.adddays(1);

-- reset to current date, should work unless midnight passed
timestamp.setcurrent();
if (saveday = timestamp.day)
    result = dialog("this is back to today.")
endif
```

SetDateTime() DateTime Method

The positional parameters are:

- **Year** or **YR** (number)
- **Month** or **MO** (number)
- **Day** or **DA** (number)
- **Hour** or **HR** (number)
- **Minute** or **MI** (number)
- **Second** or **SE** (number)

The return value is boolean.

This method sets the date and time of an object to specified values. It does not change the date or time of your PC! It simply changes the values inside the object.

The hour, minute, and second are assumed to be local time i.e., the time on your PC. You do not have to specify both a date and a time in a single method call. You can specify just the date or just the time. When setting the date, you must specify the year, month, and day. When setting the time you must specify the hour, minute, and second. You cannot set the time of an object that does not have a valid date.

`SetDateTime()` returns `True` if the operation is successful, otherwise it returns `False`. There are six parameters to the `SetDateTime()` method:

Parameter Name	Type
Year	Integer
Month	Integer
Day	Integer
Hour	Integer
Minute	Integer
Second	Integer

Here is an example:

```
timestamp = datetime();
result = dialog("Now is " + timestamp.fmtshortdatetime());

-- Set the Date, leave the time
timestamp.SetDateTime(year: 1947, month: 6, day: 13);
result = dialog("Same time on birthday is " +
           timestamp.fmtshortdatetime());

-- Set the Time, leave the Date unchanged
timestamp.SetDateTime(hour: 13, minute: 0, second:0);
result = dialog("Date and time of birth was " +
           timestamp.fmtshortdatetime());
```

SetGMTDateTime() DateTime Method

The positional parameters are:

- **Year** or **YR** (number)
- **Month** or **MO** (number)
- **Day** or **DA** (number)
- **Hour** or **HR** (number)
- **Minute** or **MI** (number)
- **Second** or **SE** (number)

The return value is boolean.

This method sets the date and time of an object to specified values. The hour, minute, and second are assumed to be GMT time i.e. the time in London, England, not the time on your PC. It does not change the date or time of your system or the host, only the values stored in the object.

You do not have to specify both a date and a time in a single method call. You can specify just the date or just the time. When setting the date, you must specify the year, month, and day. When setting the time you must specify the hour, minute, and second. You cannot set the time of an object that does not have a valid date.

`SetGMTDateTime()` returns `True` if the operation is successful, otherwise it returns `False`. There are six parameters to the `SetGMTDateTime()` method:

Parameter Name	Type
Year	Integer
Month	Integer
Day	Integer
Hour	Integer
Minute	Integer
Second	Integer

Here is an example:

```
timestamp = datetime();
result = dialog("Now is " + timestamp.fmtshortdatetime());

-- Set the GMT Date and time
timestamp.SetGMTDateTime(year: 1947, month: 6, day: 13,
           hour: 13, minute: 0, second:0);

-- Show the local time, if you are in GMT+4 it should be 13-4=9
result = dialog("Date and time of birth was is " +
           timestamp.fmtshortdatetime());
```

Connection Objects

Qedit provides ways to manage connection objects. These objects are used to control host connections and all related attributes.

Creating a Connection Object

There are a few ways you can create a Connection object. You can explicitly create one using the `OpenConnection()` application method. You don't have to open a file to open a connection. If you already have a file opened, you can create a connection object using the **Connection** document property.

Connection Properties

Here are the properties for Connection objects:

Name	Type	Description
HostCWD	string	Host current working directory
HostGroupAccount	string	Group and account if the CWD is an MPE group
HostOS	string	Name of the host operating system
HostPID	string	Process ID of the connected server process
MaxOpenFiles	integer	Maximum number of files opened concurrently on this connection
Name	string	Name of the connection
OpenCount	integer	Number of files currently opened on this connection
ServerVersion	string	Version number of the connected server
ShowBrowser	boolean	Show the directory browser window

- **HostCWD**: A string property containing the current working directory (CWD) using the absolute notation for that host. This property always contains a value for UNIX hosts and MPE hosts with Posix support. The property value changes as you navigate through the directory structure on that connection.

  `/home/clerk`

```
/usr/include/sys
```

```
/DEVACCT/SRC
```

- **HostGroupAccount**: A string property containing the group and account names if this is a connection to an MPE host and the current working directory is an MPE group. Otherwise, it is an empty string variable. The value changes as you navigate through the account structure on that connection. The format is `group.account`. For example:

  ```
  SRC.DEVACCT
  ```

- **HostOS**: A string property containing the host operating system (OS). Typically, this would be the OS name and version. The value remains the same for the duration of a connection but might change if the OS on the host itself is changed. Here are examples for an HP-UX and an MPE host:

  ```
  HP-UX B.10.20
  ```

  ```
  MPE/iX C.60.00
  ```

- **HostPID**: A string property showing the host process id (PID) for that instance of the server. The value remains the same as long as the connection exists but it can change if the connection is closed or you start a new Qedit session. Although this is a string, a PID is typically a number.

- **MaxOpenFiles**: An integer property showing the maximum number of files that can be opened concurrently on that connection. This information comes from the server and can not be changed.

- **Name**: A string property showing the name of the connection. This is the name originally used to establish the connection. The value remains the same throughout the duration of the connection.

- **OpenCount**: An integer property showing the number of files currently opened on that connection. The value changes as you open and close files.

- **ServerVersion**: A string property showing the version number of the Qedit server. This value remains the same for the duration of the connection but might change if the server program on the host is changed. Here are examples for a UNIX and MPE server:

  ```
  Qedit/UX (Version 4.8.10)
  ```

  ```
  Qedit/iX (Version 4.8.10)
  ```

- **ShowBrowser**: A boolean property used to display or hide the directory browser dialog box. It is the only property that can be changed explicitly. If set to `True`, the dialog box is displayed. The directory listing is refreshed only the first time the dialog box is enabled. To refresh the listing on subsequent calls, you have to use the **Refresh** button. If set to `False`, the dialog box is hidden.

```
mpeconn = openconnection(connection: "Production MPE");
mpeconn.showbrowser = true;   -- Display the Directory dialog box
```

To display all properties for a particular connection, you could use this script:

```
uxconn = openconnection(connection: "Production UX");
if exists(uxconn) then
    writelog ("UNIX Connection information");
    writelog("Name            =" + string(uxconn.name));
    writelog("ServerVersion   =" + string(uxconn.ServerVersion));
    writelog("HostOS          =" + string(uxconn.HostOS));
    writelog("HostPID         =" + string(uxconn.HostPID));
    writelog("MaxOpenFiles    =" + string(uxconn.MaxOpenFiles));
    writelog("OpenCount       =" + string(uxconn.OpenCount));
    writelog("HostCWD         =" + string(uxconn.HostCWD));
    writelog("HostGroupAccount=" + string(uxconn.HostGroupAccount));
    writelog("ShowBrowser     =" + string(uxconn.ShowBrowser));
else
    writelog("No ux connection");
endif

writelog(" ");
mpeconn = open(connection: "Production MPE");
if exists(mpeconn) then
    writelog ("MPE Connection information");
    writelog("Name            =" + string(mpeconn.name));
    writelog("ServerVersion   =" + string(mpeconn.ServerVersion));
    writelog("HostOS          =" + string(mpeconn.HostOS));
    writelog("HostPID         =" + string(mpeconn.HostPID));
    writelog("MaxOpenFiles    =" + string(mpeconn.MaxOpenFiles));
    writelog("OpenCount       =" + string(mpeconn.OpenCount));
    writelog("HostCWD         =" + string(mpeconn.HostCWD));
    writelog("HostGroupAccount=" + string(mpeconn.HostGroupAccount));
    writelog("ShowBrowser     =" + string(mpeconn.ShowBrowser));
else
    writelog("No MPE connection");
endif
```

The script control panel would show the following information:

```
UNIX Connection information
Name               = Production UX
ServerVersion      = Qedit/UX (Version 4.8.10)
HostOS             = HP-UX B.10.20
HostPID            = 10920
MaxOpenFiles       = 10
OpenCount          = 0
HostCWD            = /home/clerk
HostGroupAccount   =
ShowBrowser        = 0

MPE Connection information
Name               = Production MPE
ServerVersion      = Qedit/iX (Version 4.8.10)
HostOS             = MPE/iX C.60.00
HostPID            = 123
MaxOpenFiles       = 10
OpenCount          = 0
HostCWD            = /FINANCE/GL
HostGroupAccount   = GL.FINANCE
ShowBrowser        = 0
```

Connection Methods

Here are the Connection methods:

`ChangeCWD()`	`GetDirectoryIterator()`

ChangeCWD() Connection Method

The positional parameters are:

- **Pathname** (string)

- **Wildcard** (string)

The return value is boolean.

To change the local CWD, see the LocalCWD application property.

This method is used to change to a different working directory or to display a file subset. The value in **Pathname** has to be in the Posix notation, even on an MPE host. The value can be an absolute or relative path. If you wish to go to a different group on an MPE host but do not wish to use the Posix notation, enter a value with a group name in the **Wildcard** parameter.

By default, Qedit displays all the files in the specified pathname. If you wish to see only a subset, enter a value in the **Wildcard** parameter. The wildcard value must follow the syntax of the corresponding host. For example, an asterisk on a UNIX host or in the Posix namespace on an MPE host means one or more characters. To specify the same thing in the MPE namespace of an MPE host, you would use an at-sign. For details on how to use wildcards in the **Directory** dialog box, please refer to the Qedit for Windows User Manual.

The `ChangeCWD()` method returns `True` if the operation is successful. Otherwise, it returns `False`.

GetDirectoryIterator() Connection Method

The positional parameters are:

- **Directory** (string)

The return value is an iterator object.

This connection method only works with host directories. It is also available as an application method to handle local directories.

This method returns an directory iterator object which contains information about all the files and subdirectories in the specified directory. Each entry is a record with a number of elements describing the file or subdirectory. For a detailed description of each element, see "Host Directory Iterator" on page 161. For host directories, the elements are:

- **ConnectionName** (string)
- **Filecode** (integer)
- **Name** (string)
- **Path** (string)
- **OpenName** (string)
- **RecordLength** (integer)
- **Size** (integer)
- **ModificationTimestamp** (date and time object)
- **CanonicalType** (string)

Individual entries can be accessed using a `REPEAT` statement.

```
uxconn = openconnection("Production UX");
hostdir = uxconn.getdirectoryiterator("/home/clerk");

subdircount = 0;
filecount = 0;
repeat for direntry in localdir
    if direntry.canonicaltype = "directory" then
        subdircount = subdircount + 1;
    else
        filecount = filecount + 1;
    endif
endrepeat

writelog("Number of subdirectories=" + string(subdircount));
writelog("Number of files=" + string(filecount));
```

The meaning of each parameter is:

- **Directory**: Retrieve the list of files and subdirectories stored at that location.

The `GetDirectoryIterator()` method is not recursive. This means that it returns information on subdirectories in the requested directory but it does not go down the subdirectories. If you need to see what is stored at other levels in the directory tree, the script has to do it.

Iterator Objects

These objects are complex data structures created by a number of methods. Iterator objects usually contain information about similar items. Each entry in the object represents a single item and each entry is typically a record. Think of it as a list of records. Unlike other objects described so far, iterator objects do not really have properties and methods.

Currently, iterator objects are created with the `GetDirectoryIterator()` application method (for local directories), the `GetDirectoryIterator()` connection method (for host directories) and the `GetConnectionTemplateIterator()` application method (for connection templates).

Local Directory Iterator

A local directory iterator object is created by calling the `GetDirectoryIterator()` application method. It can only deal with local directories. It contains information on files and subdirectories stored in the directory specified on the call. Each file or subdirectory has a corresponding record in the object. Each record contains the following elements:

AccessTimestamp

This is a Datetime object containing the date and time when that file or subdirectory was last accessed.

CanonicalType

This is a string giving the type of item described in this record. Possible values are:

- **text**: if the filename ends with ".txt"
- **script**: if the filename ends with ".qsl"
- **program**: if the filename ends with ".exe"

- **directory**: if the entry is a subdirectory

The element contains an empty string if the file does not match any of these definitions.

CreationTimestamp

This is a Datetime object containing the date and time when that file or subdirectory was created.

ModificationTimestamp

This is a Datetime object containing the date and time when that file or subdirectory was last modified.

Name

This is a string containing the name of the file or subdirectory. The current directory itself contains a single dot, ".". The parent directory is identified with two dots, "..".

OpenName

This is a string containing the fully-qualified file or subdirectory name. Typically, this is the **Path** element concatenated to the **Name** element.

Path

This is a string containing the path for the file or subdirectory. Typically, this is the same as the directory specified in the call parameter.

Size

This a numeric variable of type float showing the number of bytes in the file. For subdirectories, this element typically contains 0.

Host Directory Iterator

A host directory iterator object is created by calling the `GetDirectoryIterator()` connection method. It can only deal with host directories. It contains information on files and subdirectories stored in the directory specified on the call. Each file or subdirectory has a corresponding record in the object. Each record contains the following elements:

CanonicalType

This is a string giving the type of item described in this record. Possible values are:

- **qedit**: if the file is a Qedit workfile.
- **cobol**: if the file is a Cobol source file. To see how the server identifies these, refer to the Qedit for Windows User Manual.
- **directory**: if the entry is a subdirectory

The element contains an empty string if the file does match not any of these definitions.

The above values are available on MPE hosts. For UNIX hosts, only "directory" is available.

ConnectionName

This is a string containing the name of the host connection where the directory is located.

FileCode

This is numeric variable of type integer containing the file code associated with the file. This only applies to MPE hosts. For UNIX hosts, the file code is always set to 0. For a comprehensive list of valid file codes, refer to "Appendix B - File Types" in the Qedit for Windows User Manual.

ModificationTimestamp

This is a Datetime object containing the date and time when that file or subdirectory was last modified. The date and time are automatically adjusted for different time zones. That is the information in the iterator object always appears as local time even though the file might actually be on host computer 3 time zones away.

Name

This is a string containing the name of the file or subdirectory.

On UNIX connections, the current directory itself contains a single dot, ".". The parent directory is identified with two dots, "..".

On both UNIX and MPE connections, directory names are terminated with a slash "/".

OpenName

This is a string containing the fully-qualified file or subdirectory name. Typically, this is the **ConnectionName** followed by a colon concatenated to **Path** element and the **Name** element.

Path

This is a string containing the path for the file or subdirectory. Typically, this is the same as the directory specified in the call parameter.

RecordLength

This is a numeric variable of type integer containing the record length for the file.

On UNIX hosts, the value represents the number of bytes allowed in each record. Because this concept is foreign, the value is always 0.

On MPE hosts, if the value is negative, it indicates the number of bytes allowed in each record. If the value is positive, it indicates the number of words (2-byte) allowed in each record.

Size

This a numeric variable of type integer showing the size of the file. On UNIX hosts, it represents the number of bytes. On MPE hosts, it represents the number of records.

Connection Template Iterator

A connection template iterator object is created by calling the `GetConnectionTemplateIterator()` application method. It contains

information on all connections currently defined in the connection template file. Each connection has a corresponding ConnectionTemplate object in the iterator object.

ConnectionTemplate Objects

A ConnectionTemplate object contains all information required to establish a connection to a host. These objects can be created (`NewConnectionTemplate()` application method), retrieved (`FindConnectionTemplate()` and `GetConnectionTemplateIterator()` application methods) and deleted (`DeleteConnectionTemplate()` application method).

ConnectionTemplate Properties

ConnectionTemplate objects have the following properties:

- **Name**: A string containing the name of the connection.
- **HostName**: A string containing the host name or IP address of the host to conect to.
- **LogonInformation**: A record containing the information required for a successful login.
- **ColorScheme**: A string containing the name of a color scheme to be used as the default.
- **Autologon**: A boolean value to indicate whether Qedit should establish this connection automatically at startup.

New values can be assigned to the **HostName**, **ColorScheme** and **Autologon** properties. If you wish to change the name of the connection, you have to use the `Rename()` ConnectionTemplate method. To change any of the elements in the **LogonInformation** property, you have to use of the `SetLogonInformation()` ConnectionTemplate method.

LogonInformation Property

The logon information is stored in a record variable called **LogonInformation**. The elements found in the record vary based on the host type. **ConnectionType** is the only common element to both types. The element contains a string and can be only one of two possible values: `Unix` or `MPE`. If **ConnectionType** is Unix, other elements in the record are:

- **UserName**: A string containing a user ID.
- **Password**: A string containing the password for the specified user ID. For security reasons, you can not see the current password. Qedit always returns a null string.
- **InitialDirectory**: A string containing the initial directory to be used at logon.

You can not assign a value directly to individual elements in that record. You have to use the `SetLogonInformation()` method.

If **ConnectionType** is MPE, other elements in the record are:

- **Hello**: A string containing the information normally provided in a `Hello` command.
- **UserPass**: A string containing the user password for the username specified in the `Hello` string. For security reasons, you can not see the current password. Qedit always returns a null string.
- **GroupPass**: A string containing the password for the group specified in the `Hello` string. For security reasons, you can not see the current password. Qedit always returns a null string.
- **AccountPass**: A string containing the password for the account specified in the `Hello` command. For security reasons, you can not see the current password. Qedit always returns a null string.
- **SessionPass**: A string containing the password for the sessionname specified in the `Hello` string. For security reasons, you can not see the current password. Qedit always returns a null string.
- **Firewall**: A boolean indicating whether Qedit should connect using the normal connection procedure or using the Firewall Protocol. A zero value or False indicates a normal connection. Any non-zero value or True indicates the Firewall Protocol.

ConnectionTemplate Methods

The name of a connection can not be changed with a simple assignment statement. It can only be changed using the `Rename()` method. ConnectionTemplate objects have the login information stored in a record variable. You can not assign a value directly to individual elements in that record. You have to use the `SetLogonInformation()` method.

Rename() ConnectionTemplate Method

The method has only one parameter:

- **NewName** (string)

Because a connection name has to follow certain rules, you have to use the `Rename()` ConnectionTemplate method to change a connection name. The method ensures the new name adheres to the rules and that the name does not already exist.

```
findconn = findconnectiontemplate("Prod UX");
if typeof(findconn) = qedit.typeundefined then
    writelog("Connection does not exist");
else
    newname = "Production UX";
    result = findconn.rename(newname);
endif
```

SetLogonInformation() ConnectionTemplate Method

This method accepts a record that adheres to the **LogonInformation** property definition. It replaces the current information with the updated version. If you wish to be prompted for one or more passwords, enter a question mark "?" instead of the actual password.

```
foundconn = findconnectiontemplate("Prod MPE");
newlogon = foundconn.logoninformation;
newlogon.hello = "clerk,user.acct";   -- Change the Hello string
newlogon.userpass = "NewPass";        -- Change the user password
newlogon.acctpass = "?";              -- Prompt for account password
foundconn.setlogoninformation(newlogon); -- Update the connection
```

`SetlogonInformation()` does not report any error if you mix elements for different host types. Extra elements are simply ignored. For example, if you retrieved logon information for a UNIX connection and assign a value to an MPE host element, the new element is simply added to the record but does not affect the operation. In the example below, none of the UNIX elements have been changed so the connection is still the same. The script executes without any error.

```
foundconn = findconnectiontemplate("Prod UX");   -- UNIX host
newlogon = foundconn.logoninformation;
newlogon.hello = "clerk,user.acct";   -- Hello string is ignored
newlogon.userpass = "NewPass";        -- User password is ignored
foundconn.setlogoninformation(newlogon); -- Update the connection
```

Properties and Methods Cross-Reference

The table below has a list of all properties and methods provided by Qedit. Entries in the table are sorted by their name. This should make it easier for you to identify the exact nature of a syntax element in a script.

For detailed information, see

- "Application Constants" on page 101
- "Application Methods" on page 105
- "Document Properties" on page 118
- "Document Methods" on page 123
- "DateTime Properties" on page 150
- "DateTime Methods" on page 151
- "Connection Properties" on page 156
- "Connection Methods" on page 158
- "Exception Handlers" on page 18

Name	Type	Description
Activate	Document method	Make a document the active window
ActiveFile	Application property	Name of the currently active document
AddDays	DateTime method	Add a number of days to a date
AutoIndent	Document property	Auto-indent option
Autologon	Connection template property	Connection autologon setting
Autopush	Application property	Automatically push caret location on search option
AutoWorkfilePost	Application property	Automatically post Qedit

		workfiles option
CacheMaxLines	Document property	Maximum number of lines allowed in the cache
CaretAllowedOutsideText	Application property	Caret allowed in undefined areas option
ChangeCWD	Connection method	Change to a different directory
CheckServerTimestamps	Application property	Compare timestamps before overwriting on server option
Close	Document method	Close a document
ColorScheme	Connection template property	Default color scheme for the connection
Compare	DateTime method	Compare 2 dates
ConnectionName	Document property	Name of the connection where the file resides
ConvertTabsToSpaces	Document property	Convert tabs to spaces option
Copy	Document method	Copy the selection to the clipboard
Cut	Document method	Cut the selection from the document and put in the clipboard
DateTime	Application method	Retrieves the current date and time
Day	DateTime property	Day of the month (0 to 31)
DayOfWeek	DateTime property	Day of the week (1 to 7)
DayOfYear	DateTime property	Day of the year (1 to 366)
DaysBetween	DateTime method	Calculates the number of days between 2 dates
Delete	Document method	Remove the selection
DeleteConnectionTemplate	Application method	Remove a connection template object
Detab	Document method	Replace tab characters with spaces
DisplayDetabbedColumn	Document property	Display detabbed column coordinates option
Entab	Document method	Replace spaces with tab characters
Exit	Application method	Terminates Qedit for Windows
File	Application property	List of opened files
Find	Document method	Search for a string
FindAll	Document method	Display all occurrences of a string in the current file and related files (Include, Use, COBOL Copy)
FindConnectionTemplate	Application method	Retrieve information and create a connection template object

FindOpenFile	Application method	Search for a document among all currently opened documents
FmtShortDate	DateTime method	Format date in short form
FmtShortDateTime	DateTime method	Format date and time in short form
FmtShortTime	DateTime method	Format time in short form
FullFileName	Document property	Fully-qualified name of the opened file
GetConnectionTemplateIterator	Application method	Create a connection template iterator object
GetDirectoryIterator	Application method	Create a local directory iterator object
GetDirectoryIterator	Connection method	Create a host directory iterator object
GetSelectedText	Document method	Retrieve the selection
GetText	Document method	Retrieve one or more lines
GMTHour	DateTime property	GMT hour (0 to 23)
GMTMinute	DateTime property	GMT minute (0 to 59)
GMTSecond	DateTime property	GMT second (0 to 59)
HostCWD	Connection property	Current working directory on the connection
HostGroupAccount	Connection property	Current working group and account on an MPE connection
HostName	Connection template property	Host name for the connection
HostOS	Connection property	Host operating system name and version
HostPID	Connection property	Current server process id
HostCommand	Application method	Execute host command
HostCommandAbort	Application method	Stop host command execution
HostCommandStatus	Application method	Check host command status
Hour	DateTime property	Hour, local time (0 to 23)
Insert	Document method	Insert new text
InsertColumn	Document method	Insert text at a specific column
IsModified	Document property	File has been modified
IsNew	Document property	File is new and has not been named yet
IsOnHost	Document property	File is on a host
IsQedit	Document property	File is Qedit workfile
IsReadOnly	Document property	File has been opened with read-only
IsSaveable	Document property	File can be saved
KeepTrainlingBlanks	Document property	Preserve trailing spaces in the file option

LastFoundColumn	Document property	Starting column of the matched string
LastFoundLength	Document property	Number of characters in the matched string
LastFoundLine	Document property	Line number where the matched string has been found
LineCount	Document property	Number of lines in the file
LinesTruncated	Document property	Number of lines truncated during a Paste or Insert operation
LiveScrolling	Application property	Use live scrolling for server files option
LoadScript	Application method	Loads a script in the environment
LogonInformation	Connection template property	Logon information e.g. user ID, passwords
MaxOpenFiles	Connection property	Maximum number of concurrently opened files
Minute	DateTime property	Minute, local time (0 to 59)
Month	DateTime property	Month (1 to 12)
MPEServerName	Application property	MPE Server Name
Name	Connection property	Name of the connection
Name	Connection template property	Name of the connection
NewConnectionTemplate	Application method	Create a new connection template object
NewFile	Application method	Creates a new file
Open	Application method	Opens an existing file
OpenConnection	Application method	Open a new connection
OpenConnections	Application property	List of opened connections
OpenCount	Connection property	Number of currently opened files on the connection
OriginalCurrentLine	Document property	Current line number at the time of the last close
Overwrite	Application property	Insert / overwrite mode
Paste	Document method	Paste the contents of the clipboard
PrintOnHost	Document method	Copy the file on a host printer
PrintOnLocal	Document method	Copy the file on a local printer
RecordLength	Document property	Maximum line length allowed in the file
ReleaseOnClose	Application property	Close connections with no open files option
Rename	Connection template method	Change the name of a connection template object

Save	Document method	Save the changes
SaveAs	Document method	Save the file with a different name or location
Second	DateTime property	Second, local time (0 to 59)
Select	Document method	Select some text
Selection	Document property	Coordinates of the current insertion point
ServerVersion	Connection property	Version of the currently running server
SetCurrent	DateTime method	Set object to current system local time
SetDateTime	DateTime method	Set object to specified local date and time
SetGMTDateTime	DateTime method	Set object to specified GMT date and time
SetLogonInformation	Connection template method	Change logon information for a connection template object
SetWidth	Document method	Change the maximum line length of a document
ShellCommand	Application method	Execute a command in the Windows 95/NT shell
ShiftLeft	Document method	Shift text to the left
ShiftRight	Document method	Shift text to the right
ShowBrowser	Connection property	Display or hide Directory browser dialog box
ShowSCP	Application property	Display or hide Script Control panel dialog box
Status	DateTime property	Date validity
Tabs	Document method	Set or clear tab stops
Typefloat	Application constant	Float data type
Typeinteger	Application constant	Integer data type
Typeobject	Application constant	Object data type
Typerecord	Application constant	Record data type
Typestring	Application constant	String data type
Typeundefined	Application constant	Undefined data type
UnloadScript	Application method	Removes a loaded script from the environment
UseRulerBar	Application property	Open files with a ruler bar option
WorkFilename	Document property	Name of the Qedit workfile used
Year	DateTime property	Four-digit year

Error Messages

Handling Errors

QSL can detect and report all kinds of errors. Most of these errors are critical and cause Qedit to stop script execution. However, you can write smart scripts to intercept errors and perform custom operations under certain conditions. This is done using `TRY/RECOVER` blocks. A TRY block returns information in record-type format. One of the elements in that record is the error number. That's probably the most convenient way to check for a specific error condition. See "Exception Handlers" on page 18 and "Try and Recover" on page 91.

Error Numbers

The following are the Qedit for Windows error numbers and the corresponding constant name:

Description	Numeric Value
ErrorDivisionByZero	200
ErrorRepeatByZero	201
ErrorSubscriptOutOfBounds	202
ErrorUnknownProperty	203
ErrorUnknownMethod	204
ErrorImproperCoordinate	205
ErrorInsertNotAtPoint	206
ErrorNameRequiredForSave	207
ErrorLineNumberOutOfRange	208
ErrorColumnNumberOutOfRange	209
ErrorSelectionIsEmpty	210
ErrorDomainError	211
ErrorModuloByZero	212
ErrorEditError	213
ErrorNonExistentFile	214
ErrorFileBusy	215

ErrorDuplicateFileName	216
ErrorFileOpenFailed	217
ErrorUserSpecifiedError	218
ErrorUnspecifiedUserError	219
ErrorTimeout	220
ErrorInvalidConnection	221
ErrorCannotPrintLocalOnHost	222
ErrorNoLocalPrinter	223

File Errors

When trying to deal with file-related errors, you have to be aware of the file type and where it resides. There are cases where QSL reports different error numbers for similar types of problems. For example, if a local file does not exist, Qedit reports **ErrorNonExistentFile**. If a host file does not exist, Qedit reports **ErrorFileOpenFailed**.

Appendix A - Earlier Highlights

Overview of Appendix A - Earlier Highlights

Here are some of the changes and enhancements made in earlier versions of Qedit for Windows.

Highlights in Version 5.0.10

- The MPE server now returns the path information with a trailing slash when filling out a request from a `GetDirectoryIterator()` connection method.
- Temporary files can be created with the **DiscardOnClose** argument of the `NewFile()` application method. If set to true, Qedit does not ask for save confirmation when the file is closed.

Highlights in Version 5.0

- A number of methods are available to manage connection templates. You can create, modify or delete connection templates directly from a script. Application methods are `NewConnectionTemplate()`, `DeleteConnectionTemplate()`, `FindConnectionTemplate()` and `GetConnectionTemplateIterator()`. These methods work with ConnectionTemplate objects.
- Three new application methods support execution of host commands . The methods are: HostCommand `()`, HostCommandStatus `()` and HostCommandAbort `()`.
- The `DOSCommand()` application method allows execution of local programs on your PC. This method offers synchronous and asynchronous execution modes.
- Directory iterator objects can now be created with the `GetDirectoryIterator()` methods. These objects allow you to get information on files and subdirectories within a specific directory. An iterator for a local directory is retrieved using the `GetDirectoryIterator()` application method. A host iterator is retrieved using the `GetDirectoryIterator()` connection method.
- The `FindAll()` document method finds all occurrences of a string in a file. It has an option to scan Include files, Use files or COBOL Copy libraries in a

single operation. This method offers great performance on host files as the search is entirely done by the server. Only the matching lines are sent back to the Qedit client.

- There are six scripts in the Robelle script library. Five of these scripts are automatically loaded when Qedit starts: **Sortlines**, **ListAll**, **ListInclude**, **ListUse** and **ListCopy**. The sixth script, **MPECompile**, can be loaded manually.
- The **Save compiled script** command on the **Script** menu saves scripts in compiled form. These scripts are known as private scripts and can only be executed.

Glossary of Terms

JCW

See Job Control Word

Job Control Word

(JCW) System variables used to pass information about process execution. Typically, these are used to indicates success or failure. They can be used for other purposes as defined by the application.

Asynchronous

Asynchronous execution means that the script can start a statement or task and immediately skip to the next. It does not wait for the task to complete. See Synchronous.

Autoload

When Qedit starts, it scans all the scripts in preconfigured directories. All subroutines in these scripts can now be called as methods. On Command statements add commands to the Script menu.

CanonicalType

Element returned in an entry inside iterator objects. It's a string that describes the type of current entry.

Current working directory

(CWD) Directory where Qedit is currently working out of. If a file name is not qualified, Qedit assumes the file resides there.

CWD

See Current Working Directory

Method

Functions associated with an object. A method allows manipulation of the object's properties.

Object

Piece of information that makes up an application like Qedit. Objects have properties (attributes) and methods (functions).

Properties

Object attributes. Properties are associated with an object. They describe the object.

SCP

See Script Control Panel

Script Control Panel

(SCP) Actually refers to the Script Control dialog box. This dialog allows the user to control the execution of a script. It can be started, paused, stopped or executed one statement at a time.

Synchronous

Synchronous execution means that the script waits for the completion of a statement or task before moving on to the next. See Asynchronous.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y