

# **FTP Scripting Manual**

---

# Table of Contents

Introduction .....	iv
1. Basic syntax .....	1
1.1. Commenting your script .....	2
1.2. Script commands .....	2
1.3. Predefined Status flags and constants .....	3
1.4. Strings, numbers, variables and lists .....	3
1.5. Conditional execution .....	4
1.6. Loops .....	6
1.7. List manipulation loops .....	6
1.8. Error checking .....	7
1.9. Adding execution delays .....	8
1.10. Forced exit from loops .....	8
1.11. Script termination .....	9
2. Working with string and numeric variables .....	10
2.1. Setting and updating variables .....	11
2.2. Printing variables .....	11
2.3. Incrementing and decrementing numeric variables .....	12
2.4. Converting strings to uppercase or lowercase .....	12
2.5. Modifying string contents .....	13
2.6. Comparing strings .....	15
3. Working with lists of files and folders .....	18
3.1. Creating a list .....	19
3.2. Accessing items in a list .....	19
3.3. Appending to lists .....	20
3.4. Comparing lists .....	21
4. File transfer commands .....	22
4.1. Connecting to remote hosts .....	23
4.2. Local and remote paths .....	25
4.3. Obtaining folder listings .....	26
4.4. Transferring files and folders .....	27
4.5. Modification time for remote file .....	28
4.6. Dealing with duplicate files .....	29
4.7. Passive mode transfers .....	29
4.8. ASCII and binary transfer types .....	30
4.9. Non RFC959 compliant FTP servers .....	31
5. File system commands .....	32
5.1. Renaming files and folders .....	33
5.2. Deleting files and folders .....	33
5.3. Creating folders .....	34
5.4. Executing custom remote FTP commands .....	34
5.5. Executing local programs .....	35
5.6. Compressing and uncompressing local files and folders .....	36
5.7. Copying and moving local files and folders .....	37

6. Recursive File Transfers .....	39
7. Comparing Files and Folders .....	42
8. Synchronizing Files and Folders .....	44
9. Backup Files and Folders .....	47
10. Reading and writing local files .....	50
10.1. Opening and closing local files .....	51
10.2. Reading lines .....	52
10.3. Writing lines .....	52
11. Email notification .....	53
11.1. Creating email messages .....	54
11.2. Multi line messages and attachments .....	54
11.3. Sending email .....	55
12. OpenPGP Encryption and Decryption .....	58
12.1. OpenPGP automation concepts .....	59
12.2. Generating OpenPGP key pairs .....	59
12.3. Keyrings and key management .....	60
12.4. Encrypting files with OpenPGP .....	63
12.5. Decrypting OpenPGP encrypted files .....	65
13. Error reporting .....	67
13.1. Setting program exit codes .....	68
13.2. Writing out error files .....	68
13.3. Email notifications .....	68
14. Working with timestamps .....	70
14.1. Generating timestamps .....	71
14.2. Formatted Timestamps .....	71
14.3. Using timestamps .....	72
15. Accessing system information .....	74
15.1. Reading environmental variables .....	75
15.2. Getting the local computer name .....	75
15.3. Getting username running the script .....	76
15.4. Reading windows registry values .....	76
16. Command line script parameters .....	77
16.1. Running a script from the command line .....	78
16.2. Specifying log files .....	78
16.3. Passing Environmental variables .....	78
16.4. Protecting passwords and other strings .....	79
Index .....	80

---

# Introduction

FTP scripts enable the automation of file transfers and many other file processing activities. The scripting language is simple yet expressive and provides for error checking, conditional execution, looping, list handling, wildcard matching, and variable manipulation. Secure file transfers using both FTPS (SSL/TLS) and SFTP (SSH2) are supported in addition to regular FTP. Execution status can be communicated through exit codes, log files, error files, or even by sending a status email from within the script. Scripts may be executed from the command line, scheduled as one-time/recurring tasks, triggered when the contents of a monitored folder changes, or called from other scripts. These scripts may also be executed by Sysax Multi Server in response to server connection events.

This document contains the scripting language guide, a cookbook of frequently scripted operations, and sample FTP scripts. You may also contact our support team by filing a support ticket online at [www.sysax.com](http://www.sysax.com) for questions and help on writing FTP scripts.

---

# 1

## Basic syntax

1.1. Commenting your script .....	2
1.2. Script commands .....	2
1.3. Predefined Status flags and constants .....	3
1.4. Strings, numbers, variables and lists .....	3
1.5. Conditional execution .....	4
1.6. Loops .....	6
1.7. List manipulation loops .....	6
1.8. Error checking .....	7
1.9. Adding execution delays .....	8
1.10. Forced exit from loops .....	8
1.11. Script termination .....	9

## 1.1. Commenting your script

Comments may be inserted into FTP scripts by preceding it with the "#" character. All text after "#" is ignored till the end of the current line.

### Exhibit 1.1. Example for using the command for end

```
#this is a comment  
  
endscript; #this is comment
```

## 1.2. Script commands

All predefined script commands must be terminated with a semicolon. Parenthesis and commas between parameters are optional. Parenthesis are also optional for commands with no arguments.

### Exhibit 1.2. Examples for using the connect and disconnect commands for ftp

```
ftpconnect "127.0.0.1", 21, "anonymous", "anon@anon.com"; #command without  
parenthesis  
  
ftpconnect "127.0.0.1" 21 "anonymous" "anon@anon.com"; #command without  
parenthesis and commas  
  
ftpconnect("127.0.0.1", 21, "anonymous", "anon@anon.com"); #command with  
parenthesis and commas  
  
ftpdisconnect(); #command with no arguments with parenthesis  
  
ftpdisconnect; #command with no arguments without parenthesis
```

## 1.3. Predefined Status flags and constants

The **ftpreresult** predefined status flag contains the result of commands that begin with the phrase *ftp*. It is set to the predefined constant **success** if the command completed successfully.

The **mailresult** predefined status flag contains the result of commands that begin with *mail*. It is set to the predefined constant **success** if the command completed successfully.

The **fileresult** predefined status flag contains the result of commands that begin with *file*. It is set to the predefined constant **success** if the command completed successfully.

Other predefined constants are **true** and **false** that denote the corresponding conditions.

## 1.4. Strings, numbers, variables and lists

A string is any group of characters enclosed in quotes. Strings may also contain wildcards such as **\*** and **?**. A number is any value between 0 and  $2^{32}$ .

A variable is represented by a **~** character followed by alphanumeric or underscore characters. A variable may contain either string or numerical values and will be treated as a string or number depending on the context in which it is used. A variable is automatically created and initialized to an empty string at the point of first use. Protected variables may be used to hide passwords and other strings. The contents of a protected variable will be automatically decrypted when the variable is passed to the **ftpconnect\*** group of commands, the **certload** command or the **pkeyload** command. In all other cases, including printing of the variable, only the encrypted value is made available. The encrypted string for a protected variable should be generated from the command line using the **-protectstring** option.

### Exhibit 1.3. Examples for using the strings, numbers, and variables in script

```
"this is a string" #string  
45 #number  
~my_variable #variable
```

A list is represented by a **@** character followed by alphanumeric or underscore characters. Lists represent the contents of a folder and can contain zero or more items. Each item represents a specific file or folder. Lists can be set using the **ftpgetlist** command and each item in a list can be accessed using the **foreach** loop. Two lists can be compared using the **listcompare** command. The contents of an entire list or a single list item can be appended to another list using the **listappend** command. Each list also contains the following property:

.count        number of items contained in the list

Each list item is represented by a **\$** character followed by alphanumeric or underscore characters and contains the following properties:

.name        file or folder name string  
.isfolder    flag set to false for file and true for folder  
.size        size of the file in bytes  
.modtime    modification time string in YYYYMMDDhhmmss format (Y=year,  
            M=month, D=day, h=hour, m=minute, s=second)

### **Exhibit 1.4. Example for using the lists in script**

```
@my_list #list  
  
@my_list.count #count property of a list  
  
$list_item #list item  
  
$list_item.size #size property of list item
```

## **1.5. Conditional execution**

A conditional statement consists of the keyword **if** followed by the test condition and a statement block. The statement block is mandatory. An **else** keyword may also be used to specify an alternate condition.

String comparison operators are:

eq        (equality)  
ne        (inequality)  
lt        (less than)  
le        (less than or equal to)  
gt        (greater than)  
ge        (greater than or equal to)

Numeric comparison operators are:

```
==      (equality)
!=      (inequality)
<       (less than)
<=      (less than or equal to)
>       (greater than)
>=      (greater than or equal to)
```

Arguments are automatically treated as strings or numbers based on the comparison operator that is used. Strings may also contain wildcards such as \*.

### **Exhibit 1.5. Syntax of conditional execution statement**

```
if <test condition> begin
end

if <test condition> begin
end else begin
end
```

### **Exhibit 1.6. Examples for using the conditional execution statement**

```
if ~numvar < 5 begin
end

if ~strvar eq "*.txt" begin
end else begin
end
```

## 1.6. Loops

A loop statement consists of the keyword **loop**, followed by an integer number or a variable, followed by a statement block. A statement block begins with the **begin** keyword and ends with the **end** keyword. The statement block is mandatory.

### Exhibit 1.7. Syntax of loop statement

```
loop <integer number> begin  
end
```

### Exhibit 1.8. Examples for using the loop statement

```
loop 25 begin  
end  
  
loop ~my_variable begin  
end
```

## 1.7. List manipulation loops

A list based loop statement consists of the keyword **foreach** followed by a list item name, the keyword **in**, the list name, and a statement block. The statement block is mandatory.

### Exhibit 1.9. Syntax of list manipulation loop statement

```
foreach <list item name> in <list name> begin  
end
```

The list is filled using the **ftpgetlist** command and the foreach loop is used to access each file or folder item on the list.

### Exhibit 1.10. Example for using the list manipulation loop statement

```
foreach $list_item in @my_list begin
    print "item name is ", $list_item.name, " and size is ",
    $list_item.size;
end
```

## 1.8. Error checking

The **ftpresult**, **mailresult**, and **fileresult** predefined status flags can be used to determine the result of the last executed command. The flags will be set to **success** if the command completed successfully. Errors may be reported by setting a program exit code, printing an error message, writing out an error file, or even sending out an email message.

### Exhibit 1.11. Examples for using some commands for error checking

```
ftpconnect "127.0.0.1", 21, "anon", "anon@anon.com";
if ftpresult eq success begin
end else begin
end

mailcreate "My name", "me@mydomain.com", "Test mail", "This is a test
mail";
if mailresult eq success begin
end else begin
end

fileopen read, "myfile.txt";
if fileresult eq success begin
end else begin
end
```

## 1.9. Adding execution delays

Script execution can be temporarily paused using the **waitsecs** command. The time to wait is specified in seconds.

### Exhibit 1.12. Syntax of command for execution delay

```
waitsecs <number of seconds>;
```

### Exhibit 1.13. Example for using the command for execution delay

```
waitsecs 30; #pause script execution for 30 seconds
```

## 1.10. Forced exit from loops

The **exitloop** command can be used to immediately exit from a loop. If this command is called from nested loops, the innermost loop is exited.

### Exhibit 1.14. Syntax of command for exit from loops

```
exitloop;
```

### Exhibit 1.15. Example for using the command for exit from loops

```
loop 100 begin
  ftpconnect "ftp.ftpsite.com", 21, "ftp", "ftp@site.com";
  if success eq ftpresult begin
    exitloop; #exit loop immediately if connected
  end
end
```

## 1.11. Script termination

A script will stop execution after the last command in the file is executed. The script can be terminated at any other execution point by calling the **endscript** command.

### **Exhibit 1.16. Example for how the script can be terminated at any other execution point**

```
if ftpresult ne success begin
    endscript;
end
```

---

# 2

## Working with string and numeric variables

2.1. Setting and updating variables .....	11
2.2. Printing variables .....	11
2.3. Incrementing and decrementing numeric variables .....	12
2.4. Converting strings to uppercase or lowercase .....	12
2.5. Modifying string contents .....	13
2.6. Comparing strings .....	15

## 2.1. Setting and updating variables

A variable is automatically created and set to an empty string when first used. The **setvar** command or the **strprint** command can be used to set or update the values stored in a variable. A variable holds both numbers and strings and can be treated as a number or string based on the context. The **setprotectedvar** command is used to hide passwords and other strings. The contents of a protected variable will be automatically decrypted when the variable is passed to the ftpconnect\* group of commands, the certload command or the pkeyload command. In all other cases, only the encrypted value is made available. The encrypted string for a protected variable should be generated from the command line using the -protectstring option.

### Exhibit 2.1. Syntax of commands for setting and updating variables

```
setvar <variable>, <string or number>;

strprint <variable>, <sequence of comma separated strings, numbers, and
variables>;

setprotectedvar <variable>, <encrypted string generated from the command
line using the -protectstring option>;
```

### Exhibit 2.2. Examples for using the commands for setting and updating variables

```
setvar ~my_number, 5;

setvar ~my_string, "this is a string";

strprint ~my_value, "the value is ", 5, " bytes";

setprotectedvar ~my_value,
":#!FEC016d09ab332ff7edfdbe90dd212c8b0e37dd033bc6cd7ad3d31f5a4075e94d1f1c0ef8
ZGCM8g5Z08ASGrddPLDDux2QA/6wttec7/yUg1mmVUAdLeWE=";
```

## 2.2. Printing variables

The **print** command is used to print variables, strings, and numbers to the log file.

### Exhibit 2.3. Syntax of command for printing variables

```
print <variable>, <sequence of comma separated strings, numbers, and
variables>;
```

### Exhibit 2.4. Example for using the command for printing variables

```
print ~my_value, "the folder contains ", 17, " files with a combined size
of ", ~my_size, " Megabytes";
```

## 2.3. Incrementing and decrementing numeric variables

The **increment** command and **decrement** commands can be used to increment or decrement a numeric variable by 1.

### Exhibit 2.5. Syntax of commands for incrementing and decrementing numeric variables

```
increment <variable>;
decrement <variable>;
```

### Exhibit 2.6. Example for using the commands for incrementing and decrementing variables

```
setvar ~my_num, 4;
increment ~my_num; #value of ~my_num is now 5
decrement ~my_num; #value of ~my_num is now 4
```

## 2.4. Converting strings to uppercase or lowercase

The **strupper** and **strlower** commands are used to string variables into upper or lower case.

**Exhibit 2.7. Syntax of commands for converting strings to uppercase or lowercase**

```
strupper <variable>;  
strlower <variable>;
```

**Exhibit 2.8. Example for using the commands  
for converting strings to uppercase or lowercase**

```
setvar ~my_var, "NaMe.Txt";  
strupper ~my_var; #convert to uppercase  
strlower ~my_var; # convert to lowercase
```

## 2.5. Modifying string contents

The **strslice** command can be used to extract part of a string. The **left** and **right** keywords are used to specify the start direction for the extracted string slice. The offset specifies the number of characters to skip from the start direction. The character count specifies the number of characters to extract. A value of 0 for the character count extracts the entire remaining portion of the string. The extracted string replaces the original string contained in the variable.

**Exhibit 2.9. Syntax of command for modifying string contents**

```
strslice <variable>, <keyword: left, right>, <offset from start  
direction>, <character count>;
```

**Exhibit 2.10. Examples for using the command for modifying string contents**

```
setvar ~my_var, "my_string_value";

strslice ~my_var, left, 4, 0; #start from left, skip 4 chars and extract
rest of the string. ~my_var contains "tring_value"

strslice ~my_var, left, 0, 4; #start from left, extract 4 chars. ~my_var
contains "my_s"

strslice ~my_var, right, 4, 0; #start from right, skip 4 chars and extract
rest of the string. ~my_var contains "my_string_v"

strslice ~my_var, right, 0, 5; #start from right, extract 5 chars. ~my_var
contains "value"
```

The **regexreplace** command replaces parts of a string with another string based on a regular expression pattern. This is similar to the substitution operation in Perl. The regex pattern specifies the regular expression pattern to search for in the original string. The replacement text specifies the string to substitute when a match is found. The match options specify other conditions that affect a match. The available match options are:

```
i    perform a case insensitive match
g    match all occurrences of a pattern
```

Note that for the regular expression pattern string, a \ character must be preceded by another \.

**Exhibit 2.11. Syntax of command for modifying string contents**

```
regexreplace <variable>, <regex pattern string>, <replacement text
string>, <match options>;
```

**Exhibit 2.12. Examples for using the command for modifying string contents**

```
setvar ~my_var, "xyz_file_050502out.txt";  
  
regexreplace ~my_var, "[0-9]+", "", "ig"; #remove all numbers in string.  
~my_var contains "xyz_file_out.txt"  
  
regexreplace ~my_var, "[0-9]+", "00000", "ig"; #replace numbers in string  
with 00000. ~my_var contains "xyz_file_00000out.txt"
```

The replacement text can also contain backreferences. Backreferences are specified using \$1 for the subexpression matched in the first parenthesis, \$2 for the subexpression matched in the second parenthesis and so on.

**Exhibit 2.13. Examples for using command for modifying string contents**

```
setvar ~my_var, "dat_file.txt";  
  
regexreplace ~my_var, "([\_a-zA-Z0-9]+\)\.\.[\_a-zA-Z0-9]+", "$1.log", "ig";  
#rename extension from txt to log. ~my_var contains "dat_file.log";  
  
regexreplace ~my_var, "([\_0-9a-zA-Z]+)[.][\_0-9a-zA-Z]+", "$1_050402.txt",  
"ig"; #append a timestamp. ~my_var contains "dat_file_050402.txt"
```

## 2.6. Comparing strings

Strings can be directly compared using the string comparison operators. Additionally, strings can also be matched using wildcards such as \*, and ?. This is known as wildcard matching or pattern matching. \* matches 0 or more characters while ? matches exactly one character.

**Exhibit 2.14. Example for how to comparing strings**

```
if "index.txt" eq ~my_var begin
end

if "*.txt" eq ~my_var begin
end
```

The **ftwildcardmatch** command can be used to perform wildcard matching. Additionally, this command allows case insensitive pattern matching when **i** is passed as a match option. If the strings match, the predefined flag **ftpresult** is set to the predefined constant **success**.

**Exhibit 2.15. Syntax of command for performing wildcard matching**

```
ftwildcardmatch <string to match>, <wildcard pattern string>, <match
options>;
```

**Exhibit 2.16. Example for using the command for wildcard matching**

```
ftwildcardmatch ~my_var, "*.txt", "i"; #case insensitive wildcard match
if success eq ftpresult begin

end
```

The **ftregexmatch** command can be used to perform matching using regular expressions similar to that used in Perl. This command allows case insensitive regular expression matching when **i** is passed as a match option. If the strings match, the predefined flag **ftpresult** is set to the predefined constant **success**. Note that for the regular expression pattern string, a **\** character must be preceded by another **\**.

**Exhibit 2.17. Syntax of another string matching command**

```
ftregexmatch <string to match>, <regex pattern string>, <match options>;
```

**Exhibit 2.18. Example for using the command string matching**

```
ftpregexmatch ~my_var, "[a-zA-Z0-9]+\.\log", ""; #case sensitive regex
match
if success eq ftpresult begin

end
```

---

# 3

## Working with lists of files and folders

3.1. Creating a list .....	19
3.2. Accessing items in a list .....	19
3.3. Appending to lists .....	20
3.4. Comparing lists .....	21

## 3.1. Creating a list

The contents of a folder can be listed and stored into a user specified list name. Each item in the list can then be individually accessed. The **ftpgetlist** command can be used to obtain a listing of the current working path. The predefined keywords **local** or **remote** are used to specify either the local or remote system. The recurse level determines the number of levels of subfolders that need to be listed. If no recurse level is specified, it is set to 1 by default and lists only the top level files and folders in the current working path. Setting the recurse level to 0 will create a recursive listing of all subfolders in the current working path. A user specified list name is used to store the listing result. And the **.count** parameter is used to get the number of items in a list. And the The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 3.1. Syntax of command for obtaining folder listings

```
ftpgetlist <keywords: local, remote>, <list name>, <recurse level>,  
<.count>;
```

### Exhibit 3.2. Example for using the command for folder listing

```
ftpgetlist local, @local_list; #get the listing of the current local  
working path  
  
ftpgetlist remote, @remote_list; #get the listing of the current remote  
working path  
  
ftpgetlist remote, @remote_list, 0; #get recursive listing of the current  
remote working path  
  
ftpgetlist remote, @remote_list, 3; #get up to 3 levels of listings of the  
current remote working path
```

## 3.2. Accessing items in a list

A list based loop statement consists of the keyword **foreach** followed by a list item name, the keyword **in**, the list name, and a statement block. The statement block is mandatory.

### Exhibit 3.3. Syntax of list manipulation loop statement

```
foreach <list item name> in <list name> begin
end
```

The list is filled using the **ftpgetlist** command and the foreach loop is used to access each file or folder item on the list.

### Exhibit 3.4. Example for using the list manipulation loop statement

```
foreach $list_item in @my_list begin
    print "item name is ", $list_item.name, " and size is ",
    $list_item.size;
end
```

## 3.3. Appending to lists

The **listappend** command can be used to append a single list item or an entire list to an existing list.

### Exhibit 3.5. Syntax of command for appending list

```
listappend <targetlist>, <list item or list to be appended>
```

### Exhibit 3.6. Examples of command for appending single list item or list

```
listappend @biglist, $myitem; #append a list item to a list
listappend @biglist, @mylist; #append a list to another list
```

## 3.4. Comparing lists

A Folders can be **compared** between a local and remote computer, two remote computers, or even two local folders by first obtaining a listing using the **ftpgetlist** command and then using the **listcompare** command to compare the two lists. **Comparisons** can be performed based on size or date. The four output lists contain items in the first list items that are not present in the second list, identical to those in the second list, newer or larger compared to the second list, and older or smaller compared to the second list.

### Exhibit 3.7. Syntax of command for comparing folder listings

```
listcompare <keywords: bydate, bysize>, <list 1>, <list 2>, <items only in  
list 1>, <identical items>, <newer or larger items>, <older or smaller  
items>;
```

### Exhibit 3.8. Example of command for comparing folder listings

```
listcompare bydate, @list1, @list2, @list1only, @identical, @newerlist1,  
@olderlist1;  
  
listcompare bysize, @list1, @list2, @list1only, @identical, @largerlist1,  
@smallerlist1;
```

---

# 4

## File transfer commands

4.1. Connecting to remote hosts .....	23
4.2. Local and remote paths .....	25
4.3. Obtaining folder listings .....	26
4.4. Transferring files and folders .....	27
4.5. Modification time for remote file .....	28
4.6. Dealing with duplicate files .....	29
4.7. Passive mode transfers .....	29
4.8. ASCII and binary transfer types .....	30
4.9. Non RFC959 compliant FTP servers .....	31

## 4.1. Connecting to remote hosts

Connections to a remote FTP server are established using the **ftpconnect** command. The IP address of the remote server must be specified as the address string. The port number indicates the port where FTP connections are accepted by the remote server. The default port number for regular FTP connections is 21. The username and password strings are required to log into a specific user account. If the FTP server allows anonymous access, the username should be *ftp* and the password should be the user's email address. Some legacy FTP servers may also require an optional account string. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

Passwords and other strings can be protected using the **setprotectedvar** command. The contents of a protected variable will be automatically decrypted when the variable is passed to the `ftpconnect*` group of commands, the `certload` command, or the `pkeyload` command. In all other cases, including printing of the variable, only the encrypted value is made available. The encrypted string for a protected variable should be generated from the command line using the `-protectstring` option.

### Exhibit 4.1. Syntax of connect command for ftp

```
ftpconnect <address string>, <port number>, <username>, <password> [,
optional: <account>];
```

Secure connections using the SSL protocol (also known as FTPS) are established using the **ftpconnectssl** command. The **ftpconnectssli** command should be used if an implicit FTPS connection is required. The **ftpconnectsslc** command can be used if only the control connection needs to be encrypted using SSL.

### Exhibit 4.2. Syntax of connect command for ftps

```
ftpconnectssl <address string>, <port number>, <username>, <password> [,
optional: <account>];

ftpconnectssli <address string>, <port number>, <username>, <password> [,
optional: <account>];

ftpconnectsslc <address string>, <port number>, <username>, <password> [,
optional: <account>];
```

Client side SSL certificates may be used for authentication of a client to the remote server. The **certload** command enables a client side certificate to be loaded. The certificate must be loaded before establishing a connection using **ftpconnectssl** and must be in the *pem* format.

### Exhibit 4.3. Syntax of certificate loading command

```
certload <certificate file>, <private key file> [, optional:  
<passphrase>];
```

Secure connections using the SSH protocol (also known as SFTP) are established using the **ftpconnectssh** command. The default port number for ssh connections is 22.

### Exhibit 4.4. Syntax of connect command for sftp

```
ftpconnectssh <address string>, <port number>, <username>, <password> [,  
optional: <account>];
```

Public key authentication for SSH connections allows clients to use a private key for authentication instead of a password. The **pkeyload** command enables a private key to be loaded to perform SSH public key authentication. The key must be loaded before establishing a connection using **ftpconnectssh** and must be in the *pem* format.

### Exhibit 4.5. Syntax of privatekey loading command

```
pkeyload <private key file> [, optional: <passphrase>];
```

If it is necessary to connect using a http proxy server, the **proxyhttp** command should be called before establishing a connection.

### Exhibit 4.6. Syntax of command for using http proxy server to connect

```
proxyhttp <proxy address>, <proxy port> [, optional: <proxy username>,  
<proxy password>];
```

The **ftpdconnect** command is used to close a connection that was established using any of the above commands.

### Exhibit 4.7. Syntax of disconnect command for ftp

```
ftpdconnect;
```

**Exhibit 4.8. Example for commands for connecting to remote hosts**

```
ftpconnect "ftp.mysite.com", 21, "username", "password"; #connect to a
non-secure ftp server
if success eq ftpresult begin
end

certload "c:\\certs\\mycert.pem", "c:\\keys\\myprivkey.pem", "keypass";
ftpconnectssl "ftp.securesite.com", 21, "user", "pass"; # connect to an
SSL (FTPS) based secure server using client side certificate

pkeyload "c:\\keys\\privkey.pem", "keypass";
ftpconnectssh "ftp.securesite.com", 22, "", ""; #connect to an SSH (SFTP)
based secure server using public key auth

setprotectedvar ~mypassword,
":#!FEC016d09ab332ff7edfdbe90dd212c8b0e37dd033bc6cd7ad3d31f5a4075e94dlf1c0ef8
ZGCM8g5Z08ASGrrdPLDDux2QA/6wttec7/yUg1mmVUAdLeWE=";
ftpconnect "ftp.mysite.com", 21, "username", ~mypassword; #connect to
server - here, a protected variable is passed in as the password

ftpdconnect; #disconnect from the remote ftp server
```

## 4.2. Local and remote paths

The **ftpsetpath** command is used to set the current working path for the local and remote systems. The predefined keywords **local** or **remote** are used to specify either the local or remote system. The path string contains the new path to be set. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

**Exhibit 4.9. Syntax of command for setting local and remote paths**

```
ftpsetpath <keywords: local, remote>, <path string>;
```

**Exhibit 4.10. Example for using the command for setting path**

```
ftpsetpath local, "c:\\ftptemp"; #set the current local working path  
ftpsetpath remote, "/home/ftpin"; "set the current remote working path
```

### 4.3. Obtaining folder listings

The contents of a folder can be listed and stored into a user specified list name. Each item in the list can then be individually accessed. The **ftpgetlist** command can be used to obtain a listing of the current working path. The predefined keywords **local** or **remote** are used to specify either the local or remote system. The recurse level determines the number of levels of subfolders that need to be listed. If no recurse level is specified, it is set to 1 by default and lists only the top level files and folders in the current working path. Setting the recurse level to 0 will create a recursive listing of all subfolders in the current working path. A user specified list name is used to store the listing result. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

**Exhibit 4.11. Syntax of command for obtaining folder listings**

```
ftpgetlist <keywords: local, remote>, <list name>, <recurse level>;
```

**Exhibit 4.12. Example for using the command for folder listing**

```
ftpgetlist local, @local_list; #get the listing of the current local
working path

ftpgetlist remote, @remote_list; #get the listing of the current remote
working path

ftpgetlist remote, @remote_list, 0; #get recursive listing of the current
remote working path

ftpgetlist remote, @remote_list, 3; #get up to 3 levels of listings of the
current remote working path
```

## 4.4. Transferring files and folders

The **ftpdownload** command can be used to download either an individual file or an entire folder tree. The files and folders are downloaded to the current local working path that was set using the **ftpsetpath** command. The predefined keywords **file** and **folder** are used to specify either a file or a folder. An optional local name string can be provided to save the downloaded item to a new name. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

**Exhibit 4.13. Syntax of command for download files and folders**

```
ftpdownload <keywords: file, folder>, <remote name string> [, optional:
<local name string>];
```

**Exhibit 4.14. Examples for using the command for download files and folders**

```
ftpdownload file, "text.dat"; #download file

ftpdownload file, "text.dat", "text_0503.dat"; #download file and save as
text_0503.dat

ftpdownload folder, "www"; #download entire folder tree
```

The **ftpupload** command can be used to upload either an individual file or an entire folder tree. The files and folders are uploaded to the current remote working path that was set using the **ftpsetpath** command. The predefined

keywords **file** and **folder** are used to specify either a file or a folder. An optional remote name string can be provided to save the uploaded item to a new name. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

#### Exhibit 4.15. Syntax of command for upload files and folders

```
ftpupload <keywords: file, folder>, <local name string> [, optional:  
<remote name string>];
```

#### Exhibit 4.16. Examples for using the command for upload files and folders

```
ftpupload file, "out_text.dat"; #upload file  
  
ftpupload file, "out_text.dat", "out_text_0503.dat"; #upload file, saving  
it as out_text_0503.dat  
  
ftpupload folder, "www"; #upload entire folder tree
```

## 4.5. Modification time for remote file

The **ftpmodtime** command can be used to get the modification time for remote file.

#### Exhibit 4.17. Syntax of command for Modification time for remote file

```
ftpmodtime <filename>, <user variable>;
```

#### Exhibit 4.18. Examples for using the command for Modification time for remote file

```
ftpmodtime "remotefile.txt", ~remotefiletime; #get the modification time  
for remote file remotefile.txt
```

## 4.6. Dealing with duplicate files

During a download or an upload, if a file with the same name already exists in the destination path, the **setduperules** command can be used to specify a set of rules to compare the files by size or date and then choose to skip, overwrite, resume, or rename the file. The predefined keywords **bysize** and **bydate** are used to compare the files either by size or by date. The predefined keywords **skip**, **overwrite**, **resume**, and **rename** are used to specify the type of action to perform. An action is specified for each of the possible three cases - smaller/older, same size/date, and larger/newer. If the rename option is chosen for any of the three cases, an extension string to be used to rename the file can be optionally provided. The default rename extension is *.fbak*.

### Exhibit 4.19. Syntax of command for dealing with duplicate files

```
setduperules <keywords: bysize, bydate>, <action if smaller/older>,  
<action if same size/date>, <action if larger/newer> [, optional: <rename  
ext string>];
```

### Exhibit 4.20. Examples for using the command for dealing with duplicate files

```
setduperules bysize, resume, skip, skip; #resume if smaller, skip if same  
size or larger  
  
setduperules bydate, overwrite, skip, skip; #overwrite if older, skip if  
same date or newer  
  
setduperules bysize, rename, skip, rename, ".0503"; #skip if same size,  
rename with ext ".0503" if size is different
```

## 4.7. Passive mode transfers

Files may be transferred using server initiated (PORT) or client initiated (PASV) data connections. The **enablepasv** command enables PASV data connections while the **disablepasv** command enables PORT data connections. These commands do not apply to SSH based SFTP file transfers.

**Exhibit 4.21. Syntax of commands for enabling PASV and PORT data connections**

```
enablepasv;  
disablepasv;
```

**Exhibit 4.22. Example for using the commands for enabling PASV and PORT data connections**

```
enablepasv; #enable PASV data connections  
disablepasv; #enable PORT data connections
```

## 4.8. ASCII and binary transfer types

The **settransfertype** command is used to set the appropriate transfer type for the file being transferred. The *ASCII* transfer type is used to transfer text while the *binary* transfer type is used to transfer images. The *auto* transfer type tries to detect the file type based on its extension and performs either ASCII or binary transfer. The predefined keywords **ascii**, **binary**, and **auto** are used to select the desired transfer type.

**Exhibit 4.23. Syntax of command for setting ASCII and binary transfer types**

```
settransfertype <keywords: ascii, binary, auto>;
```

**Exhibit 4.24. Example for using the command for setting ASCII and binary transfer types**

```
settransfertype binary; #set transfer type to binary  
settransfertype ascii; #set transfer type to ascii
```

## 4.9. Non RFC959 compliant FTP servers

Some legacy FTP servers produce folder content listings in non-standard formats. In these cases, the **setnlst** command can be used to provide a name-only listing. When a listing is obtained in this manner, only the *.name* property of each list item is available for use in a foreach loop. Name-only listings may be turned off using the **unsetnlst** command.

### Exhibit 4.25. Syntax of commands setnlst and unsetnlst

```
setnlst;  
  
unsetnlst;
```

### Exhibit 4.26. Example for using the setnlst and unsetnlst commands

```
setnlst; #enable name-only listing  
  
unsetnlst; #disable name-only listing
```

---

# 5

## File system commands

5.1. Renaming files and folders .....	33
5.2. Deleting files and folders .....	33
5.3. Creating folders .....	34
5.4. Executing custom remote FTP commands .....	34
5.5. Executing local programs .....	35
5.6. Compressing and uncompressing local files and folders .....	36
5.7. Copying and moving local files and folders .....	37

## 5.1. Renaming files and folders

Files and folders can be renamed using the **ftprename** command. The predefined keywords **local** and **remote** are used to apply this command on either the local or remote computer. Both the original name string and the new name string should be specified. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 5.1. Syntax of command for renaming files and folders

```
ftprename <keywords: local, remote>, <original name string>, <new name string>;
```

### Exhibit 5.2. Examples for using the command for renaming files and folders

```
ftprename local, "file_1.txt", "file_1_0305.txt"; #rename local file
ftprename remote, "file.log", "file_0305.log"; #rename remote file
```

## 5.2. Deleting files and folders

The **ftpdelete** command is used to delete files and folders. The predefined keywords **local** and **remote** are used to apply this command on either the local or remote computer. The predefined keywords **file** and **folder** are used to specify whether the item is a file or folder. Folders will be recursively deleted. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 5.3. Syntax of command for deleting files and folders

```
ftpdelete <keywords: local, remote>, <keywords: file, folder>, <name string>;
```

### Exhibit 5.4. Examples for using the command for deleting files and folders

```
ftpdelete local, file, "file_1.txt"; #delete local file

ftpdelete remote, folder, "tmp_folder"; #delete remote folder and all
sub-folders recursively
```

## 5.3. Creating folders

The **ftpmakefolder** command is used to create a new folder. The predefined keywords **local** and **remote** are used to apply this command on either the local or remote computer. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 5.5. Syntax of command for creating files and folders

```
ftpmakefolder <keywords: local, remote>, <name string>;
```

### Exhibit 5.6. Examples for using the command for creating files and folders

```
ftpmakefolder local, "tempfiles"; #create local folder

ftpmakefolder remote, "tmp_folder"; #create remote folder
```

## 5.4. Executing custom remote FTP commands

The **ftpcustomcmd** command is used to execute a literal RFC959 based FTP command string on the remote server when connected to a non secure or SSL (FTPS) based secure FTP server. When connected to an SSH (SFTP) based secure server, a shell command string can be executed if permitted by the server. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 5.7. Syntax of command for executing custom remote FTP commands

```
ftpcustomcmd <command string>;
```

### Exhibit 5.8. Example for using the command for executing custom remote FTP commands

```
ftpcustomcmd "SITE CHMOD 444 myfile.txt"; #run a site specific FTP command string
```

## 5.5. Executing local programs

The **ftprunprogram** command runs a program on the local computer. The path to the program executable and the parameters to be passed to the program should be specified. The program can also be executed as a specific user if an optional username and password are provided. The command will fail to run the program if a username and password are specified but the user process running the ftp script does not have sufficient permissions to login as the specified user. It is therefore possible that the command successfully invokes the program when running under the scheduler service or as part of a server script (both of which run with systemwide permissions) but fails when run from the command line, logged in as a specific user.

A timeout value in seconds can be optionally specified to terminate the program if it has not completed in the specified time. If a timeout value is not specified or if a value of 0 is specified, the script will wait until the program completes execution. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully. An expected exit code value can also be optionally specified to compare against the actual program exit code. If the exit codes do not match, The **ftpresult** predefined status flag will not be set to the predefined constant **success**.

### Exhibit 5.9. Sytax of command for executing local programs

```
ftprunprogram <program path string>, <parameter string> [, optional:  
<username>, <password>, <timeout value>, <expected error code>];
```

**Exhibit 5.10. Example for using the command for executing local programs**

```
ftprunprogram "c:\\Program Files\\notepad.exe", "C:\\myfile.txt"; #open a
file in notepad

ftprunprogram "c:\\Program Files\\myprog.exe", "", "administrator",
"admin"; #run a program as the administrator

ftprunprogram "c:\\Program Files\\myprog.exe", "", "", "", 10; #run a
program but terminate it if it takes more than 10 seconds to run

ftprunprogram "c:\\Program Files\\myprog.exe", "", "", "", 0, 1; #run a
program to completion and verify that the exit code is 1
```

**5.6. Compressing and uncompressing local files and folders**

Files and folders can be compressed using the **ftpcompress** command. The name string specifies the file or folder to compress. If compression is successful, the output name variable contains the name of the compressed file and the **ftpresult** predefined status flag is set to the predefined constant **success**.

**Exhibit 5.11. Syntax of command for compressing local files and folders**

```
ftpcompress <name string>, <output name variable>;
```

**Exhibit 5.12. Examples for using the command for compressing local files and folders**

```
ftpcompress "www_folder", ~zip_file_name; #compress a folder and all its
contents

ftpcompress "index.txt"; #compress a file
```

A compressed file can be uncompressed using the **ftpuncompress** command. The name string specifies the file to be uncompressed. The *zip*, *gz*, *tar.gz*, and *tgz* compressed file formats are supported. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

**Exhibit 5.13. Syntax of command for uncompressing local files and folders**

```
ftpuncompress <name string>;
```

**Exhibit 5.14. Examples for using the command for uncompressing local files and folders**

```
ftpuncompress "in22.tar.gz"; #uncompress a tar.gz file  
ftpuncompress "dw441.zip"; #uncompress a zip file
```

**5.7. Copying and moving local files and folders**

Files and folders can be copied using the **ftpcopylocal** command. The item identified by the source name is copied to the specified destination path. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

**Exhibit 5.15. Syntax of command for copying local files and folders**

```
ftpcopylocal <source name>, <destination path>;
```

**Exhibit 5.16. Examples for using the command for copying local files and folders**

```
ftpcopylocal "filelist.txt", "C:\\\\backup"; #copy a file  
ftpcopylocal "dw_folder", "C:\\\\backup"; #copy a folder
```

Files and folders can be moved using the **ftpmovelocal** command. The item identified by the source name is moved to the specified destination path. The **ftpresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### **Exhibit 5.17. Syntax of command for moving local files and folders**

```
ftpmovelocal <source name>, <destination path>
```

### **Exhibit 5.18. Examples for using the command for moving local files and folders**

```
ftpmovelocal "dirlist.txt", "C:\\\\backup"; #move a file  
ftpmovelocal "ac_folder", "C:\\\\backup"; #move a folder;
```

---

# 6

## **Recursive File Transfers**

A **recursive** list of files can be obtained using the **ftpgetlist** command by setting the recurse level to 0. The names of files within a subfolder are stored in the `.name` property as `subfolder/filename.txt` for a remote listing or `subfolder\filename.txt` for a local listing. A `foreach` statement can then be used to select each file in the list and perform the file transfer using the `ftpupload` or `ftpdownload` commands. The following examples will upload or download files recursively by creating subfolders as needed.

### Exhibit 6.1. Example for recursively uploading files that match the pattern \*.txt

```
ftpsetpath local, "c:\\ftpout";
ftpgetlist local, @mylocallist, 0;
foreach $ftpitem in @mylocallist begin
  if $ftpitem.name eq "*.txt" begin
    ftpupload file, $ftpitem.name;
  end
end
```

### Exhibit 6.2. Example for recursively downloading files that match the pattern \*.txt

```
ftpsetpath remote, "/ftpac";
ftpgetlist remote, @myremotelist, 0;
foreach $ftpitem in @myremotelist begin
  if $ftpitem.name eq "*.txt" begin
    ftpdownload file, $ftpitem.name;
  end
end
```

If all the files in a folder tree need to be **uploaded** or **downloaded** to a single destination folder, the `/` or `\` characters in the `.name` property that denote a path can be replaced by an underscore or another character using the `regexreplace` command. The following example replaces the `/` character with a `-` character and causes files from an entire folder tree in the remote folder to be saved the current local folder as `subfolder-filename.txt`.

**Exhibit 6.3. Example for recursively downloading files  
from a remote folder tree into a single local folder**

```
ftpsetpath remote, "/ftppacc";
ftpgetlist remote, @myremotelist, 0;
foreach $ftpitem in @myremotelist begin
  if $ftpitem.name eq "*.txt" begin
    strprint ~myfile, $ftpitem.name;
    regexreplace ~myfile, "/", "-", "ig";
    ftpdownload file, ~myfile;
  end
end
end
```

---

# 7

## Comparing Files and Folders

A Folders can be **compared** between a local and remote computer, two remote computers, or even two local folders by first obtaining a listing using the **ftpgetlist** command and then using the **listcompare** command to compare the two lists. **Comparisons** can be performed based on size or date. The four output lists contain items in the first list items that are not present in the second list, identical to those in the second list, newer or larger compared to the second list, and older or smaller compared to the second list.

### **Exhibit 7.1. Syntax of command for comparing folder listings**

```
listcompare <keywords: bydate, bysize>, <list 1>, <list 2>, <items only in  
list 1>, <identical items>, <newer or larger items>, <older or smaller  
items>;
```

### **Exhibit 7.2. Example of command for comparing folder listings**

```
listcompare bydate, @list1, @list2, @list1only, @identical, @newerlist1,  
@olderlist1;  
  
listcompare bysize, @list1, @list2, @list1only, @identical, @largerlist1,  
@smallerlist1;
```

---

# 8

## **Synchronizing Files and Folders**

Files can be **synchronized** between source and destination folders. Synchronization can be performed between local and remote computers or between two drives or folders on a local computer.

The **setsource** command is used to specify the source folder that needs to be synchronized and the rules for including or excluding files and folders contained in the source. The local or remote keywords indicate if the source is located on a local or remote computer. The **include** or **exclude** keyword is followed by a list of files or folders to be excluded or included in the source folder. Each of the file or folder pattern is prefixed by a "file=" or "folder=" string to indicate if the pattern should match a file or folder.

### **Exhibit 8.1. Syntax of command to set source path for synchronization**

```
setsource <keywords: local, remote>, <path of source folder>, [<keywords:
include, exclude>, "<file, folder>=<pattern>" ...];
```

### **Exhibit 8.2. Example of command to set source path for synchronization**

```
setsource local, "c:\\ftpfolder", include, "file=*.txt", "file=*.doc",
"folder=*data*", "folder=*info*";
```

The **setdestination** command is used to specify the destination folder that needs to be synchronized and the rules for preserving or removing extra files and folders contained in the destination (that are not present in the source). The local or remote keyword indicates if the destination is located on a local or remote computer. The preserve or remove keyword is followed by a list of files or folders that are present only in the destination and need to be preserved or removed. Each of the file or folder pattern is prefixed by a "file=" or "folder=" string to indicate if the pattern should match a file or folder.

### **Exhibit 8.3. Syntax of command to set destination path for synchronization**

```
setdestination <keywords: local, remote>, <path of destination folder>,
[<keywords: preserve, remove>, "<file, folder>=<pattern>" ...];
```

### **Exhibit 8.4. Example of command to set destination path for synchronization**

```
setdestination remote, "/ftpdata", remove, "file=*.obj", "file=*.bin",
"folder=*temp*", "folder=*tmp*";
```

The **syncrun** command is used to start the synchronization operation. The **bysize**, **bydate**, **bylargersize**, **bynewerdate**, **bycrc**, **bymd5**, and **bysha1** keywords are used to indicate the type of comparison to be performed between the source and destination. The main difference between synchronization and backup is that during synchronization, files and folders that exist only in the destination, that are selected to be preserved, are copied to the source folder. Also, if the **bylargersize**, and **bynewerdate** comparison types are selected, the source files will be replaced by the destination files if they are found to be newer or larger than corresponding files in the source folder.

### **Exhibit 8.5. Syntax of syncrun command**

```
syncrun <keywords: bysize, bydate, bylargersize, bynewerdate, bycrc,  
bymd5, bysha1>;
```

### **Exhibit 8.6. Example of syncrun command**

```
syncrun bysize; #synchronize using size based comparison
```

---

# 9

## **Backup Files and Folders**

Files can be **backed up** from a source folder to a destination storage folder. The destination folder can be either a local drive (local backup) or a remote server (online backup). Similarly the source can be a folder on a local computer or a remote server.

The **setsource** command is used to specify the source folder that needs to be backed up and the rules for including or excluding files and folders contained in the source. The local or remote keywords indicate if the source is located on a local or remote computer. The **include** or **exclude** keyword is followed by a list of files or folders to be excluded or included in the source folder. Each of the file or folder pattern is prefixed by a "file=" or "folder=" string to indicate if the pattern should match a file or folder.

### Exhibit 9.1. Syntax of command to set source path for backup

```
setsource <keywords: local, remote>, <path of source folder>, [<keywords:
include, exclude>, "<file, folder>=<pattern>" ...];
```

### Exhibit 9.2. Example of command to set source path for backup

```
setsource local, "c:\\ftpfolder", include, "file=*.txt", "file=*.doc",
"folder=*data*", "folder=*info*";
```

The **setdestination** command is used to specify the destination backup folder and the rules for preserving or removing extra files and folders contained in the destination (files not present in the source). The local or remote keyword indicates if the backup destination is located on a local or remote computer. The preserve or remove keyword is followed by a list of files or folders that are present only in the destination and need to be preserved or removed. Each of the file or folder pattern is prefixed by a "file=" or "folder=" string to indicate if the pattern should match a file or folder.

### Exhibit 9.3. Syntax of command to set destination path for backup

```
setdestination <keywords: local, remote>, <path of destination folder>,
[<keywords: preserve, remove>, "<file, folder>=<pattern>" ...];
```

### Exhibit 9.4. Example of command to set destination path for backup

```
setdestination remote, "/ftpdata", remove, "file=*.obj", "file=*.bin",
"folder=*temp*", "folder=*tmp*";
```

The **backuprun** command is used to start the backup operation. The **bysize**, **bydate**, **bylargersize**, **bynewerdate**, **bycrc**, **bymd5**, and **bysha1** keywords are used to indicate the type of comparison to be performed between the source and destination to determine which files and folders need to be backed up. The main difference between backup and synchronization is that during backup, files and folders that exist only in the destination, that are selected to be preserved, will never be **copied** to the source folder. Also, if the **bylargersize**, and **bynewerdate** comparison types are selected, if the destination files are found to be newer or larger than corresponding files in the source folder, the source files will simply be skipped. The source files will never be replaced by destination files.

### **Exhibit 9.5. Syntax of backuprun command**

```
backuprun <keywords: bysize, bydate, bylargersize, bynewerdate, bycrc,  
bymd5, bysha1>;
```

### **Exhibit 9.6. Example of backuprun command**

```
backuprun bysize; #backup using size based comparison
```

---

# 10

## Reading and writing local files

10.1. Opening and closing local files .....	51
10.2. Reading lines .....	52
10.3. Writing lines .....	52

## 10.1. Opening and closing local files

A file can be opened for reading or writing using the **fileopen** command. The predefined keywords **read**, **write**, and **append** are used to open the file for reading, writing, or appending. If the file is opened for writing, any existing file with the same name is deleted and a new file will be created. The name string specifies the name of the file to open. The **fileresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 10.1. Syntax of command for opening local files

```
fileopen <keywords: read, write, append>, <name string>;
```

### Exhibit 10.2. Examples for using the command for opening local files

```
fileopen write, "error.txt"; #create or open a new file for writing  
fileopen append, "output.txt"; #open a file to append data to the end of  
the file
```

An opened file can be closed with the **fileclose** command. Since only one file can be opened at a time, no arguments are required.

### Exhibit 10.3. Syntax of command for closing local files

```
fileclose;
```

### Exhibit 10.4. Example for using the command for closing local files

```
fileclose; #close the file
```

## 10.2. Reading lines

The **filereadline** command can be used to read lines from a file that has been opened with the **fileopen** command in read mode. A line of text is read from the file and placed in the user specified variable. The **fileresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 10.5. Syntax of command for reading lines

```
filereadline <user specified variable>;
```

### Exhibit 10.6. Examples for using the command for reading lines

```
fileopen read, "eg_12.txt";  
filereadline ~one_line; #read a line from the file  
  
filereadline ~one_line; #read the next line in the file
```

## 10.3. Writing lines

The **filewriteline** command can be used to write lines from a file that has been opened with the **fileopen** command in write or append mode. The line specified in the text line string is written to the file. The **fileresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 10.7. Syntax of command for writing lines

```
filewriteline <text line string>;
```

### Exhibit 10.8. Examples for using the command for writing lines

```
fileopen append, "out_log.txt";  
filewriteline "File 1 was uploaded"; #write a line to the file  
  
filewriteline "File 2 was uploaded"; #write the next line to the file
```

---

# 11

## Email notification

11.1. Creating email messages .....	54
11.2. Multi line messages and attachments .....	54
11.3. Sending email .....	55

## 11.1. Creating email messages

The **mailcreate** command is used to create an email message that can be sent to recipients using the **mailsend** command. The sender's name and email address should be specified in addition to the email's subject field and the email message that needs to be sent. The **mailresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 11.1. Syntax of command for creating email messages

```
mailcreate <sender name>, <sender email address>, <email subject>, <email message>;
```

### Exhibit 11.2. Example for using the command for creating email messages

```
mailcreate "John Doe", "john@jdinc.com", "Script status", "Upload successful!"; #create email message
```

## 11.2. Multi line messages and attachments

The **mailaddline** command is used to add an additional line to the message body that was created using the **mailcreate** command. The **mailresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 11.3. Syntax of command for adding multi lines to messages

```
mailaddline <additonal message line>;
```

### Exhibit 11.4. Example for using the command for adding multi lines to messages

```
mailcreate "John Doe", "john@jdinc.com", "Script status", "Upload
successful!"; #create email message

mailaddline "additional message line"; #add an additional line to the
email message
```

The **mailaddfile** command is used to attach a file to the message body that was created using the **mailcreate** command. The **mailresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

### Exhibit 11.5. Syntax of command for attaching a file to messages

```
mailaddfile <path to file to be attached>;
```

### Exhibit 11.6. Example for using the command for attaching a file to messages

```
mailcreate "John Doe", "john@jdinc.com", "Script status", "Upload
successful!"; #create email message

mailaddfile "c:\\temp\\status.txt"; #attach a file to the email message
```

## 11.3. Sending email

The **mailsend** command is used to send an email message that has been created using the **mailcreate** command. The same email message can be sent to multiple recipients by calling this command multiple times with the email address of each recipient. The internet address of the mail server and the corresponding port number must be specified in addition to the recipient's name and email address. In most cases, these values can be obtained from your default email program. The **mailresult** predefined status flag is set to the predefined constant **success** if the command completed successfully.

An optional username and password may also be supplied if the mail server requires it. Many mail servers now implicitly require users to authenticate before sending mail. In these cases, the username and password used to read email from this server should be supplied.

### Exhibit 11.7. Syntax of command for sending email

```
mailsend <mail server address>, <server port>, <recipient name>,  
<recipient email> [, optional: <username>, <password>];
```

### Exhibit 11.8. Examples for using the command for sending email

```
mailsend "mail.jdinc.com", 25, "Jim Jones", "jj@jj.com", "username",  
"pass"; #send email message  
  
mailsend "mail.jdinc.com", 25, "Joe Smith", "js@js.com", "username",  
"pass"; #send same message to another recipient
```

In addition, the **mailsendssl** command can be used to send an email message if the mail server accepts only **ssl** encrypted connections.

### Exhibit 11.9. Syntax of command for sending email for ssl

```
mailsendssl <mail server address>, <server port>, <recipient name>,  
<recipient email> [, optional: <username>, <password>];
```

### Exhibit 11.10. Examples for using the command for sending email for ssl

```
mailsendssl "mail.jdinc.com", 25, "Jim Jones", "jj@jj.com", "username",  
"pass"; #send email message  
  
mailsendssl "mail.jdinc.com", 25, "Joe Smith", "js@js.com", "username",  
"pass"; #send same message to another recipient
```

And the **mailsendtls** command can be used to send an email message if the mail server requires the use of **STARTTLS** to initiate secure **SSL/TLS** encrypted connections.

### Exhibit 11.11. Syntax of command for sending email for tls

```
mailsendtls <mail server address>, <server port>, <recipient name>,  
<recipient email> [, optional: <username>, <password>];
```

### **Exhibit 11.12. Examples for using the command for sending email for ssl**

```
mailsendtls "mail.jdinc.com", 25, "Jim Jones", "jj@jj.com", "username",  
"pass"; #send email message
```

```
mailsendtls "mail.jdinc.com", 25, "Joe Smith", "js@js.com", "username",  
"pass"; #send same message to another recipient
```

---

# 12

## OpenPGP Encryption and Decryption

12.1. OpenPGP automation concepts .....	59
12.2. Generating OpenPGP key pairs .....	59
12.3. Keyrings and key management .....	60
12.4. Encrypting files with OpenPGP .....	63
12.5. Decrypting OpenPGP encrypted files .....	65

## 12.1. OpenPGP automation concepts

**PGP (Pretty Good Privacy)** is an encryption framework based on public key cryptography, originally developed by Philip Zimmermann. It is widely used to protect access to sensitive files and messages. Public key cryptography is based on the concept of a pair of cryptographically strong keys known as public and private keys. Users can generate a key pair and then publish the public key to any one that wants to send them files or messages. The sender will encrypt the file or message using this public key. Once encrypted, the file or message can only be decrypted by the person who has the corresponding private key.

This scripting language provides commands to store and retrieve multiple public and private keys using keyrings, import and export public and private key files, and encrypt and decrypt files using OpenPGP. The `pgpresult` predefined status flag is set to the predefined constant `success` if the corresponding command completed successfully.

## 12.2. Generating OpenPGP key pairs

The `sysaxftp.exe` console program can be used to generate a OpenPGP key pair using the `-pgpkeygen` option. An optional size value for the encryption key can also be specified. Valid key sizes are 1024, 2048, 4096, or 8192. If no `keysize` is specified, a default key size of 1024 is used. The new key pair is generated and stored in the default keyring. The username is specified using the `-username` option and the corresponding passphrase is specified using the `-passphrase` option. The username is a text string or email address that is linked with the generated key pair and is used to select the private or public key belonging to the key pair.

### Exhibit 12.1. Syntax of command for generating a openPGP key pair

```
sysaxftp.exe -pgpkeygen [1024(default), 2048, 4096, or 8192]  
-username <username or email> -passphrase <passprase value> [-keytype  
<rsa(default), or dsaelg>]
```

## Exhibit 12.2. Examples for using the command for generating a openPGP key pair

```
sysaxftp.exe -pgpkeygen -username john.doe -passphrase johnspassword
#generate a RSA key pair with default size of 1024 and link it to a
username

sysaxftp.exe -pgpkeygen -username john.doe -passphrase johnspassword
-keytype dsaelg #generate a DSA/Elgamal key pair with default size of
1024 and link it to a username

sysaxftp.exe -pgpkeygen 2048 -username john.doe@email.com -passphrase
keypassword #generate a RSA key pair with size of 2048 and link it to an
email address
```

## 12.3. Keyrings and key management

Public and private key files used for **OpenPGP** encryption and decryption are stored in keyring files. The **Sysax FTP Automation** program maintains a default keyring. When a keyring file name is not explicitly specified, the default keyring is used. The `-pgexportpublickey` option is used to export a public key and the `-pgexportprivatekey` option is used to export a private key.

### Exhibit 12.3. Syntax of commands for exporting public and private keys

```
sysaxftp.exe -pgexportpubkey <username or email> -pgpkeyout <output
filename> [-pgpkeyring <keyring file name>]

sysaxftp.exe -pgexportprivkey <username or email> -pgpkeyout <output
filename> [-pgpkeyring <keyring file name>]
```

### Exhibit 12.4. Examples for using the commands for exporting public and private keys

```
sysaxftp.exe -pgexportpubkey john.doe -pgpkeyout keyout.pub -pgpkeyring
mykeyring.pgp #export public key for user john.doe from mykeyring.pgp

sysaxftp.exe -pgexportprivkey john.doe -pgpkeyout keyout.priv #export
private key for user john.doe from default keyring
```

The **-pgpimportpubkey** option is used to import a previously exported public key or a public key from a user to whom a file or message needs to be sent. The **-pgpimportprivatekey** option is used to import a previously exported private key.

### Exhibit 12.5. Syntax of commands for importing public and private keys

```
sysaxftp.exe -pgpimportpubkey <public key filename> [-pgpkeyring <keyring  
file name>]  
  
sysaxftp.exe -pgpimportprivkey <private key filename> [-pgpkeyring  
<keyring file name>]
```

### Exhibit 12.6. Examples for using the commands for importing public and private keys

```
sysaxftp.exe -pgpimportpubkey key.pub -pgpkeyring mykeyring.pgp #import  
public key from key.pub to mykeyring.pgp  
  
sysaxftp.exe -pgpimportprivkey key.priv #import private key from  
key.priv to default keyring
```

The **-pgplistkeys** option is used to list the contents of the default public and private keyring or a specific keyring file that is specified.

### Exhibit 12.7. Syntax of command for listing keyring contents

```
sysaxftp.exe -pgplistkeys [-pgpkeyring <keyring file name>]
```

### Exhibit 12.8. Examples for using the command for listing keyring contents

```
sysaxftp.exe -pgplistkeys #list contents of the default public and  
private keyring  
  
sysaxftp.exe -pgplistkeys -pgpkeyring mykeyring.pgp #list contents of  
mykeyring.pgp keyring
```

The **pgpexportpubkey** and **pgpexportprivkey** commands can be used to export public and private keys from within a script.

The **pgpresult** predefined status flag is set to the predefined constant success if the corresponding command completed successfully.

### Exhibit 12.9. Syntax of commands for keyrings and key management

```
pgpexportpubkey <username or email>, <output filename>, [<optional keyring
file>];

pgpexportprivkey <username or email>, <output filename>, [<optional
keyring file>];
```

### Exhibit 12.10. Examples for using the commands for keyrings and key management

```
pgpexportpubkey "john.doe", "keyout.pub", "mykeyring.pgp";    #export
public key for user john.doe from mykeyring.pgp

pgpexportprivkey "john.doe", "keyout.priv";    #export private key for
user john.doe from default keyring
```

The **pgpimportpubkey** and **pgpimportprivkey** commands can be used to import public and private keys from within a script.

The **pgpresult** predefined status flag is set to the predefined constant success if the corresponding command completed successfully.

### Exhibit 12.11. Syntax of commands for keyrings and key management

```
pgpimportpubkey <key file to import>, [<optional keyring file>];

pgpimportprivkey <key file to import>, [<optional keyring file>];
```

### Exhibit 12.12. Examples for using the commands for keyrings and key management

```
pgpimportpubkey "key.pub", "mykeyring.pgp";    #import public key from
key.pub to mykeyring.pgp

pgpimportprivkey "key.priv";    #import private key from key.priv to
default keyring
```

## 12.4. Encrypting files with OpenPGP

The **sysaxftp.exe** program can be used to encrypt files using the **-pgpencrypt** option. If a keyring file name is not explicitly specified, the default keyring is used to obtain the public key used for encryption. The **-armor** option is used to convert the encrypted binary file into an ascii text format. The **-signusername** option also can be used to sign the encrypted file using the private key of the sender to establish the source of the encrypted file.

### Exhibit 12.13. Syntax of command for encrypting files with openPGP

```
sysaxftp.exe -pgpencrypt <file to encrypt> -username <username or email>
[-armor] [-pgpout <output filename>] [-pgpkeyring <keyring file name>]
[-signusername <username or email>] [-signpassphrase <private key
passphrase>] [-signkeyring <keyring file name>]
```

### Exhibit 12.14. Examples for using the command for encrypting files with openPGP

```
sysaxftp.exe -pgpencrypt myfile.txt -username john.doe #encrypt myfile.txt
to myfile.pgp using the public key for john.doe from the default keyring

sysaxftp.exe -pgpencrypt myfile.txt -username john.doe -pgpout myfile.enc
-pgpkeyring mykeyring.pgp #encrypt myfile.txt to myfile.enc using the
public key from mykeyring.pgp
```

The **pgpencrypt** command can be used to encrypt files from within a script. The **pgpresult** predefined status flag is set to the predefined constant success if the corresponding command completed successfully. If an empty string is passed in for the output filename, it will be derived from the input filename.

### Exhibit 12.15. Syntax of command for encrypting files with openPGP

```
pgpencrypt <file to encrypt>, <username or email>, <output filename>,
[<keyring file name>];
```

**Exhibit 12.16. Examples for using the command for encrypting files with openPGP**

```
pgpencrypt "myfile.txt", "john.doe", "myfile.enc"; #encrypt myfile.txt to
myfile.enc using the public key for user john.doe

pgpencrypt "myfile.txt", "john.doe", ""; #encrypt myfile.txt to
myfile.pgp using the public key for user john.doe

pgpencrypt "myfile.txt", "john.doe", "myfile.enc", "mykeyring.pgp";
#encrypt myfile.txt using the public key from mykeyring.pgp
```

The **pgparmoron** or **pgparmoroff** commands can be called before the **pgpencrypt** command to enable or disable the conversion of the encrypted binary file into an ascii text format.

**Exhibit 12.17. Syntax of commands for encrypting files with openPGP**

```
pgparmoron;

pgparmoroff;
```

**Exhibit 12.18. Examples for using the commands for encrypting files with openPGP**

```
pgparmoron; #turn on ascii text armoring

pgparmoroff; #turn off ascii text armoring
```

The **pgpsign** command can be called before the **pgpencrypt** command to sign the encrypted file using the private key of the sender to establish the source of the encrypted file.

**Exhibit 12.19. Syntax of command for encrypting files with openPGP**

```
pgpsign <username or email>, <private key passphrase>, [<keyring file
name>];
```

**Exhibit 12.20. Examples for using the command for encrypting files with openPGP**

```
pgpsign "jane.doe", "mypass"; #sign using the private key for user
jane.doe

pgpsign "jane.doe", "mypass", "mykeyring.pgp"; #sign using the private key
for user jane.doe from mykeyring.pgp
```

**12.5. Decrypting OpenPGP encrypted files**

The **sysaxftp.exe** program can be used to decrypt files using the `-pgpdecrypt` option. If a keyring file name is not explicitly specified, the default keyring is used to obtain the private key used for decryption.

**Exhibit 12.21. Syntax of command for decrypting openPGP encrypted files**

```
sysaxftp.exe -pgpdecrypt <encrypted file> -passphrase <private key
passphrase> [-pgpout <output filename>] [-pgpkeyring <keyring file name>]
```

**Exhibit 12.22. Examples for using the command for decrypting openPGP encrypted files**

```
sysaxftp.exe -pgpdecrypt encrypted.pgp -passphrase mypass; #decrypt
encrypted.pgp using private key of user jane.doe

sysaxftp.exe -pgpdecrypt encrypted.pgp -passphrase mypass -pgpout
myfile.txt -pgpkeyring mykeyring.pgp ; #decrypt encrypted.pgp to
myfile.txt using private key from mykeyring.pgp
```

The **pgpdecrypt** command can be used to decrypt files from within a script. The `pgpresult` predefined status flag is set to the predefined constant `success` if the corresponding command completed successfully. If an empty string is passed in for the output filename, it will be derived from the encrypted input file.

**Exhibit 12.23. Syntax of command for decrypting openPGP encrypted files**

```
pgpdecrypt <file to decrypt>, <private key passphrase>, <output filename>,
[<keyring file name>];
```

**Exhibit 12.24. Examples for using the command  
for decrypting openPGP encrypted files**

```
pgpdecrypt "encrypted.pgp", "mypass", "myfile.txt"; #decrypt
encrypted.pgp to myfile.txt using the private key for user jane.doe

pgpdecrypt "encrypted.pgp", "mypass", ""; #decrypt encrypted.pgp to
encrypted.txt using the private key for user jane.doe

pgpdecrypt "encrypted.pgp", "mypass", "myfile.txt", "mykeyring.pgp";
#decrypt encrypted.pgp using the private key from mykeyring.pgp
```

---

# 13

## Error reporting

13.1. Setting program exit codes .....	68
13.2. Writing out error files .....	68
13.3. Email notifications .....	68

## 13.1. Setting program exit codes

The **setexitcode** command sets an exit code for the program that is passed back to the command shell on program termination. This is used to report error codes when running FTP scripts from the command line.

The syntax is

### Exhibit 13.1. Syntax of command for setting program exit codes

```
setexitcode <integer number>;
```

### Exhibit 13.2. Example for using the command for setting program exit codes

```
setexitcode 5;
```

## 13.2. Writing out error files

The section on reading and writing local files contains detailed information on opening and writing to a file.

### Exhibit 13.3. Example for using some commands for writing out error files

```
fileopen write, "c:\\errorlog.txt";  
filewriteline "An error has occurred";  
fileclose;
```

## 13.3. Email notifications

The section on email notification contains detailed information on creating and sending email.

### **Exhibit 13.4. Example for using some commands for email notifications**

```
mailcreate "John Doe", "support@jdoe.com", "Script status regd.", "Failed  
to upload file.";  
mailsend "mail.jdoe.com", 25, "Joe Smith", "joe@js.com", "username",  
"password";
```

---

# 14

## Working with timestamps

14.1. Generating timestamps .....	71
14.2. Formatted Timestamps .....	71
14.3. Using timestamps .....	72

## 14.1. Generating timestamps

The **gettimestamp** command is used to generate the current timestamp in YYYYMMDDhhmmss format (Y=year, M=month, D=day, h=hour, m=minute, s=second). The generated timestamp is copied to the user variable. The predefined keywords **incr** and **decr** can be optionally used to generate timestamps for previous or future dates. The predefined keywords **second**, **minute**, **hour**, **day**, **week**, **month**, and **year** are used to specify the time interval to adjust, while the interval count specifies the number of such intervals.

### Exhibit 14.1. Syntax of command for generating timestamps

```
gettimestamp <user variable> [, optional: <keywords: incr, decr>,  
<keywords: second, minute, hour, day, week, month, year>, <interval  
count>];
```

### Exhibit 14.2. Examples for using the command for generating timestamps

```
gettimestamp ~my_timestamp; #get current timestamp  
  
gettimestamp ~my_timestamp, decr, month, 2; #get a timestamp that is 2  
months old  
  
gettimestamp ~my_timestamp, incr, day, 5; #get a timestamp that is 5 days  
in the future
```

## 14.2. Formatted Timestamps

The **formattimestamp** command enables timestamps to be represented in multiple formats. The same year, month, date, hour, minute, or second values are still used, but they are arranged in different ways like **MM/DD/YYYY** or **MM-DD-YY**. The command takes a timestamp format string as input and generates the corresponding timestamp. An existing timestamp in the default **YYYYMMDDhhyyss** format that was previously generated using the **gettimestamp** command can also be used passed as an input to the **formattimestamp** command. If no timestamp is passed in, the current timestamp value is automatically generated and used.

The following values in the format string are replaced with the corresponding **timestamp** values:

%YYYY% - replaced by the 4 digit year  
%YY% - replaced by the 2 digit year  
%MM% - replaced by the 2 digit month  
%DD% - replaced by the 2 digit day  
%hh% - replaced by the 2 digit hour  
%mm% - replaced by the 2 digit minute  
%ss% - replaced by the 2 digit second

### Exhibit 14.3. Syntax of the formattimestamp command

```
formattimestamp <user variable>, <format string> [, optional: <timestamp  
in YYYYMMDDhhmmss format>];
```

### Exhibit 14.4. Example of using the formattimestamp command

```
formattimestamp ~mytimestamp, "%MM%-%DD%-%YY%", "20090201092130"; #output  
generated is 02-01-09  
  
formattimestamp ~mytimestamp, "%MM%-%DD%-%YY%"; #the current timestamp is  
generated in MM-DD-YY format
```

## 14.3. Using timestamps

Timestamps or portions of timestamps are frequently added to file names. The following examples show how time stamps can be appended to files and how file names can be searched for specific timestamps.

**Exhibit 14.5. Example for using timestamps**

```
gettimestamp ~my_timestamp; # generate current timestamp

foreach $local_item in @local_list begin
  strprint ~new_name, ~my_timestamp, "_", $item.name; #prepend timestamp
  to file name
end

gettimestamp ~my_timestamp, decr, day, 1; #generate timestamp that is a
day old

foreach $local_item in @local_list begin
  strprint ~search_name, "*", ~my_timestamp, "*"; #generate search string
  if ~search_name eq $local_item.name begin
    #search for files with this timestamp as part of their filename
  end
end
```

---

# 15

## Accessing system information

15.1. Reading environmental variables .....	75
15.2. Getting the local computer name .....	75
15.3. Getting username running the script .....	76
15.4. Reading windows registry values .....	76

## 15.1. Reading environmental variables

The **getenvvar** command is used to access the values of environmental variables. The user variable receives the value of the specified environmental variable string. Refer to the section on command line script parameters for a method to create and set program specific environmental variables from the command line.

### Exhibit 15.1. Syntax of command for reading environmental variables

```
getenvvar <user variable>, <environment variable string>;
```

### Exhibit 15.2. Example for using the command for reading environmental variables

```
getenvvar ~os_variable, "OS"; #get the value of the environmental variable  
OS
```

## 15.2. Getting the local computer name

The **getcompname** command gets the name of the computer that the program is running on. The computer name is copied to the specified user variable.

### Exhibit 15.3. Syntax of command for getting the local computer name

```
getcompname <user variable>;
```

An example is

### Exhibit 15.4. Example for using the command for getting the local computer name

```
getcompname ~comp_name; #get the name of this computer
```

## 15.3. Getting username running the script

The **getusername** command can be used to get the username currently running the script.

### Exhibit 15.5. Syntax of command for getting Getting username running the script

```
getusername <user variable>;
```

An example is

### Exhibit 15.6. Example for using the command for Getting username running the script

```
getusername ~currentuser; #get the username currently running the script
```

## 15.4. Reading windows registry values

The **readwinregistry** command can be used to get the windows registry value corresponding to parameters (registry key, registry subkey, registry entry name) of the command.

### Exhibit 15.7. Syntax of command for Reading windows registry values

```
readwinregistry <registry key><registry subkey><registry entry name><user  
variable>;
```

An example is

### Exhibit 15.8. Example for using the command for Reading windows registry values

```
readwinregistry "HKEY_LOCAL_MACHINE", "SOFTWARE\Microsoft\DirectX",  
"Version", ~dxversion; #get the windows registry value corresponding to  
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\DirectX\Version
```

---

# 16

## Command line script parameters

16.1. Running a script from the command line .....	78
16.2. Specifying log files .....	78
16.3. Passing Environmental variables .....	78
16.4. Protecting passwords and other strings .....	79

## 16.1. Running a script from the command line

An FTP script can be run from the command line using the sysaxftp.exe console program with the -script switch.

### Exhibit 16.1. Syntax of command for running a script from the command line

```
sysaxftp.exe -script <script file name>>
```

## 16.2. Specifying log files

The following switches are related to log file generation.

-logfile	generate an output log file
-tslogfile	generate a time stamped output log file
-append (default is overwrite)	append to log file generated during previous run
-append <max file size>	roll over the log file after the maximum file size specified in bytes

### Exhibit 16.2. Examples for using the commands for specifying logfiles

```
sysaxftp -script "test.fscr" -tslogfile "test.log" #generate a time  
stamped log file  
  
sysaxftp -script "test.fscr" -logfile "test.log" -append 200000 #rollover  
the appended log file every 200000 bytes
```

## 16.3. Passing Environmental variables

The following switch enables environmental variables to be created and passed to the program.

`-set <environment variable>=<value>`      create and set an environmental variable

### **Exhibit 16.3. Examples for using the switch for passing environmental variables**

```
sysaxftp -script "run.fscr" -set PASSWORD="mypass" #create env variable
PASSWORD and set it to mypass

sysaxftp -script "run.fscr" -set SERV_IP="121.122.12.1" #pass env
variable. Value can be extracted from the script using getenvvar
```

## **16.4. Protecting passwords and other strings**

The following switch encrypts the string that is passed in. It is part of a mechanism used to hide passwords and other important strings. The generated encrypted string can be used with the `setprotectedvar` command within a script. The contents of a protected variable will be automatically decrypted when the variable is passed to the `ftpconnect*` group of commands, the `certload` command, or the `pkeyload` command. In all other cases, including printing of the variable, only the encrypted value is made available.

`-protectstring <string to be protected>`      encrypt a string for use with the `setprotectedvar` command

### **Exhibit 16.4. Examples for using the switch for creating protected strings**

```
sysaxftp -protectstring mypassword #protect the "mypassword" string by
encrypting it

sysaxftp -protectstring mypassword > passfile.txt #encrypt "mypassword"
and save it to passfile.txt where it can be copied and pasted inside a
script for use with the setprotectedvar command.
```

---

# Index

## A

append, 51  
ascii, 30  
auto, 30

## B

begin, 6  
binary, 30  
bydate, 29  
bysize, 29

## C

certload, 23

## D

day, 71  
decr, 71  
decrement, 12  
disablepasv, 29

## E

else, 4  
enablepasv, 29  
end, 6  
endscript, 9  
exitloop, 8

## F

file, 27, 28, 33  
fileclose, 51  
fileopen, 51, 52, 52  
filereadline, 52  
fileresult, 3, 7  
filewriteline, 52  
folder, 27, 28, 33  
foreach, 3, 6, 19  
ftpcompress, 36  
ftpconnect, 23  
ftpconnectssh, 24, 24  
ftpconnectssl, 23, 23  
ftpconnectsslc, 23

ftpconnectssli, 23  
ftpcopylocal, 37  
ftpcustomcmd, 34  
ftpdelete, 33  
ftpdconnect, 24  
ftpdownload, 27  
ftpgetlist, 3, 7, 19, 20, 26  
ftpmakefolder, 34  
ftpmodtime, 28  
ftpregexmatch, 16  
ftpremovefile, 37  
ftprename, 33  
ftpresult, 3, 7, 16, 16, 19, 23, 25, 26, 27, 28, 33, 33, 34,  
34, 35, 35, 36, 36, 37, 37, 51, 52, 52  
ftprunprogram, 35  
ftpsetpath, 25, 27, 27  
ftpuncompress, 36  
ftpupload, 27  
ftpwildcardmatch, 16

## G

getcompname, 75  
getenvvar, 75  
gettimestamp, 71  
getusername, 76

## H

hour, 71

## I

if, 4  
in, 6, 19  
incr, 71  
increment, 12

## L

left, 13  
listappend, 3  
listcompare, 3  
local, 19, 25, 26, 33, 33, 34  
loop, 6

## M

mailaddfile, 55

mailaddline, 54  
mailcreate, 54, 54, 55, 55  
mailresult, 3, 7, 54, 54, 55, 55  
mailsend, 54, 55  
minute, 71  
month, 71

## O

overwrite, 29

## P

pkeyload, 24  
print, 11  
proxyhttp, 24

## R

read, 51  
readwinregistry, 76  
regexreplace, 14  
remote, 19, 25, 26, 33, 33, 34  
rename, 29  
resume, 29  
right, 13

## S

second, 71  
setduperules, 29  
setexitcode, 68  
setnlst, 31  
setprotectedvar, 11  
settransfertype, 30  
setvar, 11  
skip, 29  
strlower, 12  
strprint, 11  
strslice, 13  
strupper, 12

## U

unsetnlst, 31

## W

waitsecs, 8

week, 71  
write, 51

## Y

year, 71