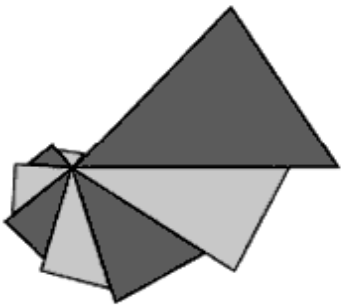


FINACLE *7.0*

DBA Training Manual

SCRIPTING SYNTAX

ExpertEdge Software

Info.training@cwlgroup.com

Document number:		VersionRev:	1.1
Authorized by:	Olawuwo Shade	Signature/Date:	

Document revision list

Ver.Rev	Date	Author	Description
1.00	01-02-2011	Arije Abayomi	*Original version*

TABLE OF CONTENTS

1 INTRODUCTION

This document discusses about the scripting language provided as a customisation tool and extensibility toolkit in Finacle™ and it's syntax.

Scripting is a programming language that Finacle™ supports for various events. Scripting also allows for formatting accounts numbers between Finacle™ and allied products like ATM, IBR[1] etc… The scripts for formatting account number across products and scripts for preload forms are pre-defined. All the scripts have to exist at predefined locations. The user can only alter the scripts as per their requirements.

Using scripting, the users can set the default values for various fields in different forms. Using scripting the user can also make a non-mandatory field as mandatory, protect certain fields after setting a default value and also hide them in the application. These are known as Form-Load Script events.

As any other programming language scripting has it's own syntax.

Scripting can use certain variables, which are available from certain memory areas called as repositories. The user can within the script access the variables from the repositories.

Scripts also have built-in functions known as user hooks. Using the userhooks only the user performs the required tasks.

Scripting also forms the underlying power of Workflow. The concept of Workflow is to automate a unit of work which may span across menu options (thread several menu options together, populating certain default values, passing the output of one menu option to the next etc.) which leads to reduction of data entry by the operator. For e.g. A TERM deposit account opening involves starting from CUMM, OAAC, TM, VCHR and finally DRP with verification needed for CUMM, OAAC & TM. Workflow scripts are created using the scripting language. The concept of workflow drastically reduces the end-user training and results in higher productivity once implemented.

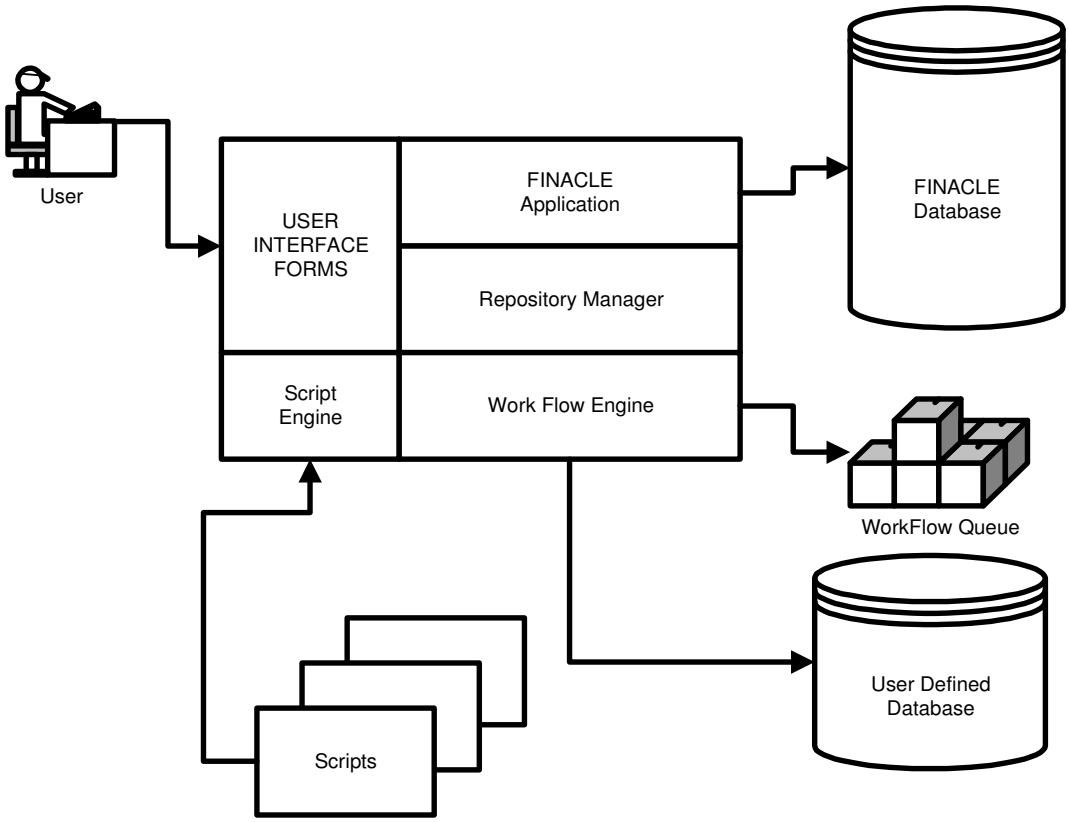
[1] IBR - Inter Branch Reconcillation

2 SECTION OBJECTIVES

At the end of the session, the user should be able to

- Understand the concepts of Customisation through scripting logic.
- Understand the terminology and syntax of scripting language.
- Write small scripts using the syntax described.

3 PROCESS DIAGRAM



4 PROCESSES INVOLVED

4.1 TECHNICAL TERMS

TBAForms - This module controls all online interactions made with Finacle™ through a Web browser. The Menu Manager in this module recognises threaded menu options and executes the associated Workflow script. It transfers data between the repository and the forms, on startup and execution of the workflow.

Repository Manager - This module provides the capability of storing variable values and identifying these variables by a name. This module is used by the Workflow script engine to store values and transfer them between menu options. The Forms module is used to pass values between fields in a screen. The structure of a repository field reference is as follows:

[Repository Name].[Class Name].[Field Name]

Workflow Engine - The script engine allows you to thread a series of menu options together to form a workflow. The Workflow engine module executes this script. It also provides the capability of suspending a thread, forwarding a thread to a specified user or workgroup and restarting threads from the point where it was previously suspended or forwarded.

Queue Table - This is the area where uncompleted workflow is stored temporarily, until retrieved for further processing. On completion of the workflow, the stored data is deleted and an audit record is created optionally.

TBA_SCRIPTS – TBA_SCRIPTS is an environment variable set in the commonenv file. This is the directory where all the scripts should exist. The application will look for this directory $TBA_PROD_ROOT/cust/scripts. If the script is not available in this directory, it will pick the script from the directory $TBA_PROD_ROOT/cust/INFENG/scripts. If the script is also not available in this directory also, the default logic (if available) will be applied as specified in each case.

4.2 WHAT IS SCRIPTING?

Scripting is a programming language supported by Finacle™ for manipulating a set of defined input data in a script and gives a defined output data for further processing. This will provide more flexibility to the end user in manipulating the data entered in the front end before the back end to suit the needs processes it. Scripting is also used to solve the usability issues in the user interface and for interfacing with other delivery channels.

For example:

The bank decides the account number format. But this format may contain some format that is common for all the accounts in the Service Outlet and the user wants the same to be appended to the account number entered in some particular format by default. It might be the Bank code, Branch code, Currency code, currency alias, Data center alias etc. which are common to all accounts.

The account number in the database might be 01-02-03-04-0001

Where 01 - corresponds to Bank code, 02 - corresponds to Branch code, 03 - corresponds to the currency alias, 04 - corresponds to the scheme code and 0001- is the actual account number.

The format of the account is such that the user has to enter '-' in between the account number which is very cumbersome. By making use of script, this can be done by default and the user needs to enter only the actual account number '010203040001' and it will be converted to the format by the script written by the user before it is further processed.

2. The Branch may be connected to an ATM that supports only 10 character account number and the actual account number size in FINACLE™ may be more. In this case, the FINACLE™ account number format is to be converted to ATM account number format using some logic. This can be done in scripting. The bank will decide on the logic to derive the unique account number for ATM from FINACLE™ format and will code the same in the Script.

The account number in FINACLE™ may be 01-02-03-04-0001.

Where 01 - corresponds to Bank code 02 - corresponds to Branch code 03 - corresponds to the currency alias 04 - corresponds to the scheme code 0001 is the actual account number.

To convert to 10 character ATM account number format, the bank may decide to leave out the bank code portion as the ATM is attached to the branches of the same BANK and leave out the formatting of account number with '-'.

The ATM account number that is derived can be 0203040001 that should be unique in the data center.

4.3 REPOSITORIES AND CLASSES

Scripting implements a way of passing data between the Finacle™ programs, User Hooks, User routines, User Executables and the script using *Repositories*.

A Repository is a named entity, which holds a set of named *Classes.*

A Class is a set of name, value pairs. The name, value pair is called a *fieldname* and *fieldvalue.* A Class can hold only one type of fieldvalue*,* which is specified during Class creation.

Integer, Float, Double, Character and String Classes are currently supported. A fieldvalue is referenced using the following syntax:

Repository-name.Class-name.Field-name

Some standard repositories and classes are managed by Finacle™ and are available for use. Refer appendix (page-33) for list of repositories, the classes and the variables available.

The bank can also create their own repositories, classes and fields while implementing their scripting logic. However, it is their responsibility to cleanup these repositories after use since otherwise, they may occupy memory unnecessarily.

The INPUT CLASS contains fields' specific to the events. The list of fields available to individual events is described in the section below.

The OUTPUT CLASS contains fields' specific to events. The details mentioned for each scripting event.

☞ *APPENDIX (on page 33) covers the details of the default values available in repository BANCS and class STDIN.*

4.3.1 MANIPULATING REPOSITORY VARIABLES

The state of information is stored in repository variables only. To give more flexibility for the application programmer, Scripting allows the programmers to access these repository fields just like any other built-in variables.

The programmer can create new fields in the Repository dynamically by giving new field names and values to them, though the type of field is fixed depending on the class type.

All the repository fields must be accessed by specifying

REPOSITORY-NAME.CLASS-NAME.FIELD-NAME

Repository variables **REPOSITORY-NAME.CLASS-NAME.FIELD-NAME** can be given as

(Expression1).(Expression2).(Expression3), where Expression1, Expression2 and Expression3 reduce to strings

e.g. : if sv_a = "hello"

then ("t" + "e" + "s" + "t").("str" + "ing").(sv_a)

will reduce to test.string.hello

REP.CLASS.FIELD, where Rep, Class and Field are the Repository Name, Class Name and Field Name respectively.

All the repository fields are globally accessible across the scripts.

```
<--start
#ACCESS repository fields in one of the following ways
#assign it to a scratchpad variable
sv_a = MYREP.integerCLASS.FIELD1
#sv_a will become an INTEGER type variable

#access it directly in statements
IF (MYREP.integerCLASS.FIELD1 == 1234)
# do something
ENDIF

#MODIFY repository fields by directly assigning a literal/expression
sv_a = "Hello World"
MYREP.stringCLASS.FIELD2 = sv_a
MYREP.stringCLASS.FIELD2 = "Hello World"
MYREP.stringCLASS.FIELD2 = "Hello" + "World"

#all the above statements modify or create FIELD2
#Create a new field in a repository in this way
MYREP.stringCLASS.NEWFIELD1 = "Hello World"
MYREP.integerCLASS.NEWFIELD2 = 123
end-->
```

Built in repository functions are provided for checking if the Repository / Class / Field exists. See section (on page 21) on in-built utilities for ***REPEXISTS***, ***CLASSEXISTS***, ***FIELDEXISTS***, ***CREATEREP***, ***CREATECLASS***, ***DELETEREP*** and ***DELETECLASS***.

4.4 RETURN STATUS OF THE SCRIPT.

One of the standard output fields from the script is successOrFailure. In case a FAILURE is encountered in the script then the OUTPUT CLASS variable successOrFailure can be set to FAILURE as shown

BANCS.OUTPUT.successOrFailure = "F"

The valid values are "S" for success and "F" Failure.

The default return status from the script is SUCCESS.

4.5 SCRIPTING SYNTAX

Every script must begin with a '←start' and end with a 'end→' tags.

Script has 26 built-in scratch pad variables. These are sv_a, sv_b, … sv_z.

All these variables are globally accessible across scripts. The user need not define the scratch pad variables to be of a particular type. The assignment to a scratchpad variable will decide the type of the variable based on the type of the right-hand side. Scripting allows the dynamic type change.

This type is maintained until a new assignment is made. So **Scripting** allows the dynamic type change.

E.g.: sv_a = 12.45

.....

....

sv_a = "Hello"

makes the type of 'sv_a' to be set to DOUBLE type at the first line. Until the last line sv_a carries its type as DOUBLE. In the last statement its type is set to STRING.

4.5.1 COMMENT

Scripting allows comments in scripts.

All statements with '#' as the first non-white space character are treated as comments. This is the only way to make a line as comment.

Even if a sequence of lines are to be treated as comments then each line must be preceded with a '#'.

4.5.2 OPERATORS

Scripting language supports the following operators:

ARTHEMATIC OPERATORS

'+', '-' , '*', '/'.

Operator '+' works both for STRING/CHAR type also.

'-' , '*' , '/' work only for numeric type variables.

```
<--start
sv_a = "FIRST"
sv_b = sv_a + " AND "
#sv_b is "FIRST AND "
sv_c = "SECOND"
sv_a = sv_b + sv_c
#now sv_a is "FIRST AND SECOND"
sv_a = "FIRST" + " AND " + "SECOND"
# sv_a is "FIRST AND SECOND"

sv_a = 10
sv_b = 5
sv_c = sv_a + sv_b
# sv_c is 15
sv_d = sv_a / sv_b
# sv_d is 2
sv_e = sv_a - sv_b
# sv_e is 5
sv_f = sv_a * sv_b
# sv_f is 50
end-->
```

COMPARISON OPERATORS

'==', '<=', '>=', '<', '>', '!='

The outcome all these operations are **TRUE (1)** or **FALSE (0)**.

In all these operations, if both arguments are STRINGs the string comparisons are made else, the STRINGs are converted to INTEGERs and the comparisons are done. If CHAR types are available they are accessed as INTEGER types and comparisons are made.

```
<--start
sv_a = 10
sv_b = 12

if (sv_a == sv_b) then
    print ( "sv_a is equal to sv_b")
else
    print ( "sv_a is not equal to sv_b")
endif

if (sv_a != sv_b) then
    print ( "sv_a is not equal to sv_b")
else
    print ( "sv_a is equal to sv_b")
endif

if (sv_a < sv_b) then
    print ( "sv_a is less than sv_b")
else
    print ( "sv_a is not less than sv_b")
endif
end-->
```

LOGICAL OPERATORS:

'AND', 'OR'

The outcome all these operations are TRUE (1) or FALSE (0).

In all these operations, if both arguments are STRINGs, the string logical operations are made. If CHAR types are available they are accessed as INTEGER types and logical operations are made.

 *Caution : In Scripts whenever there is a Logical AND/OR separating two conditions, then these conditions should be separated with parenthesis from the AND/OR keyword. Only then the proper evaluation of the conditions happens.*

For ex.:

To do a processing only when the values in variable sv_a and sv_b are not null the condition should be written as

```
if ( (sv_a != "") AND (sv_b != "") ) then
  -----do processing -----
endif
```

it should not be written as

```
 if(sv_a != "" AND sv_b != "")
```

This does not evaluate properly.

```
<--start
sv_a = "Hello"
sv_b = "Hi"

if (sv_a != "" AND sv_b != "") then
    print ( "Does not evaluate properly")
endif

if ( (sv_a != "") AND (sv_b != "") ) then
    print ( "Both are not null")
endif

sv_a = 10
sv_b = 0

if (sv_a AND sv_b != “”) then
    print ("Doesn't work")
endif
end-->
```

UNARY OPERATORS:

'-'

The same '-' operator can be used as binary as well as unary operator.

UNARY operator can only be used with numeric types.

```
<--start
sv_a = 10
sv_b = -sv_a
# Now sv_b is -10
end-->
```

4.5.3 EXPRESSIONS AND CONTROL STRUCTURES

In scripting, Expressions are valid combinations of variables, literals and operators.

Scripting provides the following control structures.

Checking a condition:

* ***IF (condition) THEN***

  statements

  ENDIF

* ***IF (condition) THEN***

  statements

  ELSE

  statements

  ENDIF

These two statements can be used depending on the necessity. Nested IF conditions are allowed and the execution of these statements depends on the outer loop condition.

```
<--start
sv_a = 10
sv_b = 12

if (sv_a > sv_b) then
    print ( "sv_a is greater than sv_b")
else
    print ( "sv_a is less than or equal to sv_b")
endif
end-->
```

Each statement should be in a new line.

Looping :

* ***WHILE (condition)***
  statements
  DO

The above statement executes the ***statements*** until the condition in the ***WHILE*** becomes **FALSE (0)**.

Each statement should be in a new line.

```
<--start
sv_s = 1
print (sv_s)
while (sv_s <= 1000)
    sv_s = ((sv_s + (10*(2+3))) + 23)
    print (sv_s)
do
end-->
```

4.5.4 GOTOS AND GOSUBS

Transfer of control from **one part of the script to another in the same script file** is facilitated by two constructs provided in the Scripting. The two constructs are GOTO and GOSUB.

The syntax is as follows:

GOTO LabelName

GOSUB LabelName

The LABEL referred by ***LabelName*** in GOTO statement can only be forward referenced i.e., GOTO can not reference a label that exists before this statement in the script.

In case of GOTO the execution of the script starts from the statement where the LABEL is declared.

The LABEL referred by ***LabelName*** in GOSUB statement can be anywhere in the script, i.e., the Label can be either before or after this GOSUB statement

In case of GOSUB the execution of the script starts from the statement where the LABEL is declared and returns back to the next statement after the GOSUB as soon as it finds **RETURN** statement in the script.

☞ **Note about GOSUB** *: GOSUB cannot be used inside the* **Control structure** *to point to a SUBROUTINE that is outside the innermost* **Control structure***.*

Following SCRIPT will give an error.

```
<--start
sv_a = 1
# if condition starts here
If(sv_a == 1) THEN
    GOSUB subRoutine1
# if condition ends here
ENDIF

EXITSCRIPT

# sub routine is outside the if-endif condition
subRoutine1:
    print (sv_a)
    RETURN
end-->
```

Following SCRIPT will work fine.

```
<--start
sv_a = 1
# if condition starts here
if(sv_a == 1) THEN
    GOSUB subRoutine1
# we added a GOTO statement to jump
# beyond the subroutine after the sub
# routine exits
    GOTO jmp1

# sub routine is inside the if-endif condition
subRoutine1:
    print (sv_a)
    RETURN

jmp1:
# if condition ends here
endif

EXITSCRIPT
end-->
```

4.5.5 CALLING AND STARTING ANOTHER SCRIPT

Transfer of control from **one script to another** is facilitated by two constructs provided in the Scripting. The two constructs are CALL and START.

The syntax is as follows:

CALL (ScriptName)

START (ScriptName)

where ***ScriptName*** is either a string, e.g., "script1.htt", or Repository variable/Scratchpad variables of STRING type.

In both the above statements, if the script file doesn't exist in the PATH, an error is reported.

CALL (Path, ScriptName)

START (Path, ScriptName)

where ***Path*** is either a string or a Repository variable/scratchpad variable of STRING type,

ScriptName is either a string or a REP variable/scratchpad variable of STRING type

Path and ***ScriptName*** are appended to get the full path.

If the script file exists and is readable, the execution of the file starts from the beginning.

In case of CALL the execution of the script returns back to the same point in the old script as soon as it finds **EXITSCRIPT** statement in the new script.

All the scratchpad variables have a global context. That is, all the scratchpad variables are carried over from the caller script to the called script and vice-versa.

4.5.6 BUILT-IN UTILITY FUNCTIONS

Script can use the following built-in functions at the appropriate places in expressions.	The evaluation of these functions results as follows: An error will result in all the cases if data-type mismatch occurs in any fields.
MID$ (Var, StartPosition, Length)	Returns the substring from a given position upto a defined length in a given variable.
LEFT$ (Var, Length)	Returns the leftmost Length number of bytes from Var.
RIGHT$ (Var, Length)	Return the rightmost Length number of bytes from Var.
CINT (Var)	Convert Var to an integer.
CDOUBLE (Var)	Convert Var to a double.
TOLOWER (Var)	Convert all character(s) of STRING/CHAR to

	lowercase.
TOUPPER (Var)	Convert all character(s) of STRING/CHAR to uppercase.
FORMAT$ (Var, FormatString)	Formats the contents of Var according to the FormatString specified in the C printf style.
SET$ (Var1, From, Length, Var2)	Sets the contents of Var1 from From position till Length bytes to the content of Var2.
STRLEN (Var)	Returns the length of Var.
CHARAT (Var, Position)	Returns the character at Position in Var.
LTRIM (Var1 [, Var2])	Left trim Var1. Var2 is the character to be trimmed. Default value of Var2 is ' '.
RTRIM (Var1 [, Var2])	Right trim Var1. Var2 is the character to be trimmed. Default value of Var2 is ' '.
TRIM (Var1 [, Var2])	Trim Var1. Var2 is the character to be trimmed. Default value of Var2 is ' '.
LPAD (Var1, Var2 [, Var3])	Left pad Var1 with Var3 upto the length Var2. Var3 is the character to be used for padding. Default value of Var3 is ' '
RPAD (Var1, Var2 [, Var3])	Right pad Var1 with Var3 to make the length Var2. Var3 is the character to be used for padding. Default value of Var3 is ' '.
REPEXISTS (Var1)	Checks if Repository Var1 exists.
CLASSEXISTS (Var1, Var2)	Checks if Class Var2 exists in Rep Var1.
FIELDEXISTS(REP.CLA.FLD)	Checks if Field FLD exists in Class FLD which exists in Rep REP.
GETPOSITION (Var1, Var2)	Returns the first position of Var2 in Var1. Var1 is String and Var2 is String/Char. Case Sensitive.
GETIPOSITION (Var1, Var2)	Returns the first position of Var2 in Var1. Var1 is String and Var2 is String/Char. Case Insensitive.
STRICMP (Var1, Var2)	Does String Comparison of two Strings/ Characters without regard to case.
GETSTRING (Var1)	Converts a Char type to String Type.

CREATEREP (Var1)	Create a Temporary Repository Var1.
CREATECLASS (Var1, Var2, Var3)	Create a Temporary Class Var2 in Repository Var1 of the type Var3. Var3 has the following value 1 for INTEGER 2 for DOUBLE 3 for FLOAT 4 for CHAR 5 for STRING
DELETEREP (Var1)	Delete the Temporary Repository Var1
DELETECLASS (Var1, Var2)	Delete the Temporary Class Var2 in Repository Var1

Examples of the above utility functions

```
<--start
# Checking MID$ operation
sv_a = Mid$("1234567890",3,5)
print (sv_a)
# Prints "45678"

# Checking LEFT$ operation
sv_a = Left$("1234567890",3)
print (sv_a)
# Prints "123"

# Checking RIGHT$ operation
sv_a = Right$("1234567890",4)
print (sv_a)
# Prints "7890"

# Checking CINT operation
sv_a = CINT("123456")
print (sv_a)
# Prints "123456"
```

```
# Checking CDOUBLE operation
sv_a = CDOUBLE("1234.5678")
print (sv_a)
# Prints "1234.567800"

# Checking TOLOWER operation
sv_a = TOLOWER("kfsjlLKJLZVXLkljsl")
print (sv_a)
# Prints "kfsjllkjlzvxlkljsl"

# Checking TOUPPER operation
sv_a = TOUPPER("werQWIUsd5678")
print (sv_a)
# Prints "WERQWIUSD5678"

# Checking the FORMAT$ operations
sv_s = Format$(12, "%d")
print (sv_s)
# Prints "12"
sv_s = Format$(1454, "%ld")
print (sv_s)
# Prints "1454"
sv_s = Format$(12.3, "%2.3f")
print (sv_s)
# Prints "12.300"

# Checking the SET$ operations
sv_s = Set$("1234567890", 3, 5, "ASDFG")
print (sv_s)
# Prints "123ASDFG90"
sv_s = Set$("1234567890", 3, 3, "ASDFG")
print (sv_s)
# Prints "123ASD7890"
sv_s = Set$("1234567890", 7, 5, "ASDFG")
print (sv_s)
```

```
# Prints "1234567ASD"

# Checking STRLEN operation
sv_n = "Some Useless Text"
sv_u = STRLEN (sv_n)
print (sv_u)
# Prints "17"

# Checking CHARAT operation
sv_n = "Some Text"
sv_u = CHARAT (sv_n, 5)
print (sv_u)
# Prints "T"

# Checking RTRIM operation
sv_a = "AS  "
sv_b = rtrim(sv_a)
print (sv_b)
# Prints "AS"

sv_a = "AS****"
sv_b = rtrim(sv_a, '*')
print (sv_b)
# Prints "AS"

# Checking LTRIM operation
sv_a = "   AS"
sv_b = ltrim(sv_a)
print (sv_b)
# Prints "AS"

sv_a = "****AS"
sv_b = ltrim(sv_a, '*')
print (sv_b)
# Prints "AS"
```

```
# Checking TRIM operation
sv_a = "  AS  "
sv_b = trim(sv_a)
print (sv_b)
# Prints "AS"

sv_a = "****AS****"
sv_b = trim(sv_a, '*')
print (sv_b)
# Prints "AS"

# Checking LPAD operation
sv_a = "AS"
sv_b = lpad(sv_a, 10, '*')
sv_c = sv_b + "END"
print (sv_c)
# Prints "********ASEND"

# Checking RPAD operation
sv_a = "AS"
sv_b = rpad(sv_a, 10, '*')
sv_c = sv_b + "END"
print (sv_c)
# Prints "AS********END"

# Checking the basic Repository / Class create / delete operations
# Repository "myrep" is created
if (REPEXISTS("myrep") == 0) then
   CREATEREP ("myrep")
endif

# Class "myclass" is created
if (CLASSEXISTS("myrep", "myclass") == 0) then
   CREATECLASS ("myrep", "myclass", 1)
endif
```

```
# Class "myclass" is created
if (FIELDEXISTS(myrep.myclass.myfield) == 0) then
   print ("myrep.myclass.myfield doesn't exist")
   myrep.myclass.myfield = 102
   print (myrep.myclass.myfield)
endif

# Class "myclass" is deleted
DELETECLASS ("myrep", "myclass")

# Repository "myrep" is deleted
DELETEREP ("myrep")

# Checking GETPOSITION operation
sv_a = "Some very long absurd text"
sv_b = "long"
sv_c = getposition (sv_a, sv_b)
print (sv_c)
#prints "11"

sv_b = "LONG"
sv_c = getiposition (sv_a, sv_b)
print (sv_c)
#prints "11"

# Checking STRICMP operation
sv_a = "Hello"
sv_b = "hello"
if(STRICMP(sv_a, sv_b) == 1) then
   print ("The strings are equal")
else
   print ("The strings are not equal")
endif

# Checking GETSTRING operation
sv_a = 'A'
```

```
sv_b = GETSTRING (sv_a)
print (sv_b)
#prints "A"
end-->
```

4.5.7 EXECUTING USER HOOKS

Scripting provides for certain functions, which can be called within a script. These are known as User Hooks.

The following is the syntax for calling a User routine within a script:

sv_a = URHK_FunctionName (STRING)

where ***sv_a*** will have the value returned from the function ***FunctionName***, i.e tha value of sv_a will be 0(TRUE) if the userhook is successful or else it will be 1(FALSE) for failure.

FunctionName is the user hook and

STRING is either a string, e.g., "Data", or a Repository variable/scratchpad variable of STRING type.

There are default scripting user hook functions available for use in the script. These user hook functions are explained in a separate topic. These hook functions will provide the functionality as explained and the input and output parameters are defined.

4.5.8 DEBUGGING UTILITIES

Application programmer can debug the application by setting the **TRACE** option **ON**.

To set the trace off, **TRACE OFF** can be used.

By default, trace is **OFF**.

Setting **TRACE ON**, makes the *Script Engine* to log all the information about each statement it has executed in a log file with name ***ScriptFile*.trc.**

The trace file will be formed in the directory depending on the following

- The trace path is read from environment variable "SE_TRACE_PATH". If the environment variable is not set or there is no write permission for the path specified then,
- "..\log" is taken as the log path. If there is no write permission for the specified path then, current directory is taken as the log path. If there is no write permission for the specified path then an error is set.

With in the script, to log information about a part of the script we can use the combination of these two commands.

```
For e.g.:
   ......
   ......
   TRACE ON
   .....
   .....
   TRACE OFF
   .....
```

Scripting also provides another important command to print a particular Built-in variable value or REP/LL field value.

The syntax of PRINT command is

```
PRINT ( SV_S )

(or)

PRINT ( "My String" )

(or)

PRINT ( GLOBAL.SECLASS.OUTPUTDATA )
```

If the trace is on and the trace file has been successfully opened, then the output of PRINT command goes to the trace file, else the output is dumped to *stdout* [standard output].

```
<--start
#...statements...
#...statements...

TRACE ON
# trace logging will be done for all statements from now onwards

#...statements...
#...statements...

sv_a = 1234
PRINT ("value of sv_a = " + sv_a)
#.above will print "value of sv_a = 1234" on screen and in trace file

TRACE OFF
#trace logging will not be done now

#...statements...
#...statements...
end-->
```

Finacle™ provides a menu option under 'DB' application to test Finacle™ scripts. The menu option is "SCRIPT". It brings up a form, which accepts a script name and executes it with same context as the actual event except that the fields in INPUT class will not be provided and the fields in OUTPUT class will not be used by the application. This facility can be used to do some initial testing of a script by providing 'default INPUT values' in the script itself and printing out variables (including OUTPUT fields) in the trace file.

MENU OPTION: SCRIPT

```
+------------------------------------------------------------------------------+
|bafe3260                      Execute Script                    03-12-1998  |
|                                                                              |
|------------------------------------------------------------------------------|
|  Note: This utility can be used only to test out Event scripts and not       |
|        WorkFlow scripts. Pre-script  should perform event specific           |
|        initialisations and post-script should examine the event specific     |
|        output for correctness.                                               |
|                                                                              |
|  Script Name          : script.scr                                           |
|                                                                              |
|  Pre script           : pre_script.scr                                       |
|                                                                              |
|  Post script          : post_script.scr                                      |
|                                                                              |
|  Display Repositories?: n                                                    |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
|                                                                              |
+------------------------------------------------------------------------------+
```

the ' pre script " can be used to initialize , set values for variables and the post script to do processing based on the values got from script.

4.5.9 EXITING A SCRIPT

Scripting provides a safe way of exiting the execution of the script. Whenever the *Script Engine* finds **EXIT** statement in the execution path stops the execution and returns to the Calling routine.

Even when the **EXIT** statement is encountered in a new script that has been **CALL**ed from another script, *Script Engine* stops the whole execution and returns to the upper layer.

To return from **CALL**ed script to the calling script, use **EXITSCRIPT**.

4.5.10 LOCATION OF THE SCRIPT FILES

Scripting locates the script file to execute, using the same logic as **Finacle™** for its executables. It first checks under the language specific directory $TBA_PROD_ROOT/cust/language/scripts, if the user's language is not INFENG. If not found it checks in $TBA_PROD_ROOT/cust/scripts directory. If not found, it checks in $TBA_PROD_ROOT/cust/INFENG/scripts.

4.5.11 LOCATION OF SAMPLE SCRIPT FILES

The sample scripts can be located in $TBA_PROD_ROOT/sample/scripts directory. Please note that these scripts are for sample purpose only hence they have different file extension - **sscr** (stands for **sample scripts**) and not **scr** which is for usual scripts. Developers can use them as templates to start with. For this, the sample script must to be copied to $TBA_PROD_ROOT/cust/scripts directory followed by the relevant modifications. At the end, the extension should be changed from **sscr** to **scr**.

5 APPENDIX

Fields values available for any scripts in the repository **BANCS** and class **STDIN**

#	*FIELD NAME*	*VALUE CONTAINED*
1.	"languageCode"	This field contains the value of the language code of the user e.g. INFENG
2.	"userId"	The userid of the user who executed the script
3.	"onlineOrBatch"	Whether this script is being executed through a batch program or online ("O" or "B")
4.	"userWorkClass"	The workclass of the userid who executed the script from UPM
5.	"menuOption"	The menu option which called this script
6.	"homeCrncyCode"	The home currency of the data center from SCFM
7.	"homeCrncyAlias"	The home currency alias
8.	"CurrentBancsVersion"	The Finacle™ version
9.	"myBankCode"	The bank code of the database
10	"myBrCode"	The branch code of the SOL
11	"myExtCode"	The Extension counter
12	"mySolId"	The SOLID
13	"mySolAlias"	The SOLALIAS
14	" mySolDesc"	The description for the Sol as specified in SCFM
15	" homeSolId"	The Sol to which the User belongs
16	" homeSolAlias"	The Home SOL Alias
17	" homeSolDesc"	The Home Sol description as specified in SCFM.
18	"dcAlias"	The DC ALIAS
19	"SBString"	The value of custoption for SBSTING
20	"CAString"	The value of custoption for CASTING
21	"LLString"	The value of custoption for LLSTING
22	"CCString"	The value of custoption for CCSTING
23	"sysDate"	The system date of the machine
24	"BODDate"	The current BOD date of the SOL
25	"termClass"	The terminal class from TPM

#	FIELD NAME	VALUE CONTAINED
26	" moduleIdentity"	The module which is calling the Script
27	"TestFlg"	Whether the script is being invoked in test mode. E.g through menu option script
28	"WFflg"	Whether the script is a workflow script
29	"ScriptName"	The name of the script

******{XXXXXX}******