# CORE BANKING APPLICATION CUSTOMIZATION

TCB HUB SCHOOLS

# CORE BANKING SCRIPTING SYNTAX

# LEARNING OBJECTIVES

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

- Understand the concepts of Customisation through scripting logic.
- Understand the terms Scripting event & Userhooks
- Understand the naming conventions of scripting language & default directory structure
- Understand the terminology and syntax of scripting language
- Write small scripts using the syntax described.

# 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. Scripting is also used to solve the usability issues in the user interface and for interfacing with other delivery channels.

# REPOSITORY 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-274) for list of repositories, the classes and the variables available.

# REPOSITORY 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-274) for list of repositories, the classes and the variables available.

# SCRIPTING RESTRICTIONS

There is a size limit of 128 KB for any Finacle script file. If a script is greater than this size, then a fatal error will be thrown.

Example:

A script file ‘LoanDmdSatisfy.scr’ of size more than the maximum limit was created and this was the error thrown during execution: *“File [./LoanDmdSatisfy.scr] greater than 131072”*

# 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.

# SCRIPTING SYNTAX

***E.g.:***

```
sv_a = 12.45

.....

....

sv_a = "Hello"
```

Here, it 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.

- *Ensure that the first line of the script is the start tag. Though this is not essential but helps in debugging scripts when errors are encountered as script counts line numbers from the start tag. It is recommended to have all comments after the start tag only.Start tag is mandatory but end tag is not but its recommended to have end tag as well.*

# SCRIPTING 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 sequences of lines are to be treated as comments then each line must be preceded with '#'.

# OPERATORS – ARITHMETIC OPERATORS

‘+’, ‘-‘ , ‘*’, ‘/’.

Operator ‘+’ works both for STRING/CHAR type also.

‘-‘ , ‘*’ , ‘/’ work only for numeric type variables.

# OPERATORS – ARITHMETIC OPERATORS

```
<--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-->
```

# OPERATORS – 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.

# OPERATORS – COMPARISON OPERATORS

```
<--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-->
```

# OPERATORS – 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.

For Example:

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
```

# OPERATORS – 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-->
```

# EXPRESSIONS AND CONTROL STRUCTURE

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***

# EXPRESSIONS AND CONTROL STRUCTURE

* ***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.

# EXPRESSIONS AND CONTROL STRUCTURE

```
<--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.

# GOTO AND GOSUB

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 cannot 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.

# GOTO AND GOSUB

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**.

# GOTO AND GOSUB

```
<--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-->
```

# CALLING AND STARTING ANOTHER SCRIPT - SPAWNING

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.

# CALLING AND STARTING ANOTHER SCRIPT - SPAWNING

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.

# CALLING AND STARTING ANOTHER SCRIPT - SPAWNING

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.

# 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:<br>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. |

# BUILT-IN UTILITY FUNCTIONS

| | |
|---|---|
| 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 ‘ ‘. |

# BUILT-IN UTILITY FUNCTIONS

| | |
|---|---|
| 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<br>1 for INTEGER<br>2 for DOUBLE<br>3 for FLOAT<br>4 for CHAR<br>5 for STRING |
| DELETEREP (Var1) | Delete the Temporary Repository Var1 |
| DELETECLASS (Var1, Var2) | Delete the Temporary Class Var2 in Repository Var1 |

# USERHOOKS

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.

# USERHOOKS

***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.

# DEBUG 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_PID*.trc.**

The trace file generation also depends on another condition .i.e. the existence of

**".usertraceon"** file in the user home directory. If this file is present then irrespective of whether trace is on or off, the trace file is generated. If the file "**.usertraceon**" is not present and TRACE is OFF, then no trace file is generated.

# DEBUG UTILITIES

Within 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
.....
```

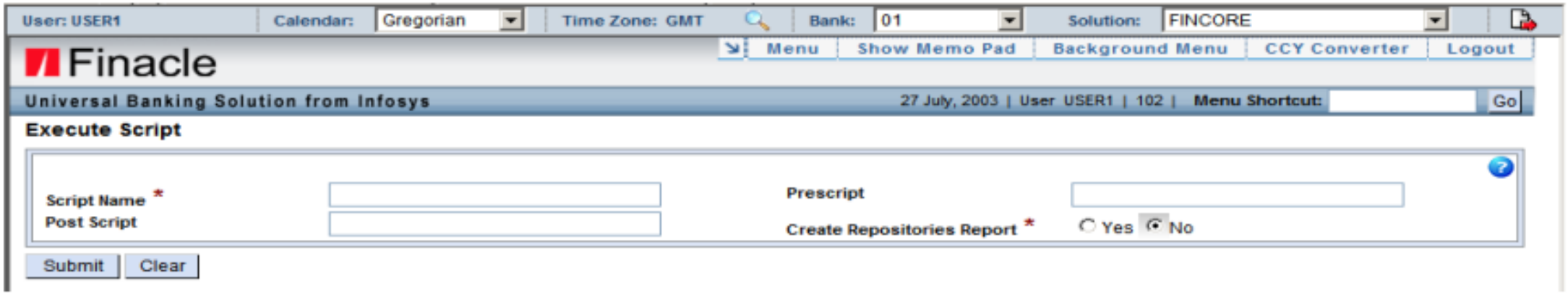
Similar to **".usertraceon"** file, there exists a file "**.userprinton"**. Now three possible cases for this are

a. The output of PRINT tokens is diverted to a file named *"<Filename>-<PID>.stdout"* – if *.userprinton* file is existing in the user's home directory.

b. All output of PRINT tokens diverted only to trace file - if TRACE is ON (or *.usertraceon* file exists)

c. All output of PRINT tokens diverted to console (stdout) - If TRACE is OFF

# HSCRIPT MENU – TESTING YOUR SCRIPT




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.

# 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 called from another script, *Script Engine* stops the whole execution and returns to the upper layer.

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

# BANCS REPO AND STDIN CLASS - FIELDS

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 |

# BANCS REPO AND STDIN CLASS - FIELDS

| | | |
|---|---|---|
| 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. |

# BANCS REPO AND STDIN CLASS - FIELDS

| | | |
|---|---|---|
| 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 |
| 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 |

# USERHOOKS AND EVENTS

# USERHOOK

Scripts have built-in functions known as Userhooks. Using these Userhooks, one is allowed to perform the required tasks .This document describes all such functions that are provided in Finacle, which can be used while writing a Finacle script. Some of these functions are generic and can be used in the context of any of the Script Events described in scripting events document

The general format for describing each function is as follows:

- Brief description of the function and context in which it can be used.
- Function syntax Detailed functionality
- Description of the Input fields and Output fields
- Example of the usage of the function

# REPOSITORY AND CLASSES

All Finacle userhooks (except for MTT related functions) interface with the scripts using a standard repository called "BANCS". There are two classes that have been predefined in this repository, INPARAM and OUTPARAM each of which holds 'String' type of fields. INPARAM is the class that is used to populate specified fields (depending upon the function) in the script so that the function can access those as parameters. All the fields (depending upon the function), that are the output of a function are populated in the OUTPARAM class.

In addition to INPARAM class, the input to the functions can also be provided as arguments to the function.

**Note**:

*The INPARAM class is cleared out once the userhook is executed and must be repopulated by the script when calling another userhook or the same userhook in a loop.*

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT VALIDATION

*urhk_valAcctNumber* validates a given account number in the FINACLE database and returns a value depending on whether the account number is valid or not. It can be used in the context of any Script Event.

```
<--start
sv_s = BANCS.INPUT.uAccount
sv_i = urhk_valAcctNumber(sv_s)
if (sv_i = 0) then                                  /* Account exists */
   BANCS.OUTPUT.fAccount = sv_s
   EXITSCRIPT
else                                        /* Account does not exist */
   /* apply new account number derivation logic here */
   BANCS.OUTPUT.fAccount = Result of above logic
endif
exitscript
end-->
```

The input to this function is an account number (FORACID).

The return value will be 0 in case a valid account number with the input string is found. Otherwise the return value is 1.

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

*urhk_getAcctDetailsInRepository* returns information about a given account number in the FINACLE database. It can be used in the context of any Script Event.

```
sv_a = BANCS.INPUT.AcctNum
    sv_b = urhk_getAcctDetailsInRepository(sv_a)
    if (sv_b == 1) then
        BANCS.OUTPUT.OutString = "Account number is invalid"
        BANCS.OUTPUT.successOrFailure="F"
        exitscript
    endif
    sv_c = cdouble(BANCS.OUTPARAM.AvailableAmt)
    sv_d = cdouble(BANCS.OUTPARAM.FullEffAvailableAmt)
```

The input to this function is an account number (FORACID).

Return value is 0, if account found, else 1.

This function will return the following Account details into OUTPARAM class of BANCS repository if the account exists in the datacenter database.

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| Field Name in GAM table | Field name in BANCS repository |
|---|---|
| sol_id | BANCS.OUTPARAM.acctSolId |
| Bacid | BANCS.OUTPARAM.acctPlaceHolder |
| acct_name | BANCS.OUTPARAM.acctName |
| cust_id | BANCS.OUTPARAM.custId |
| emp_id | BANCS.OUTPARAM.empId |
| gl_sub_head_code | BANCS.OUTPARAM.glSubHeadCode |
| acct_ownership | BANCS.OUTPARAM.acctOwnerShip (C'ustomer, E'mployee, O'ffice account) |
| schm_code | BANCS.OUTPARAM.schmCode |
| schm_type | BANCS.OUTPARAM.schmType |
| acct_opn_date | BANCS.OUTPARAM.acctOpenDate |
| acct_cls_date | BANCS.OUTPARAM.acctCloseDate |
| acct_cls_flg | BANCS.OUTPARAM.acctCloseflg (Y/N) |
| mode_of_oper_code | BANCS.OUTPARAM.modeOprnCode |
| acct_locn_code | BANCS.OUTPARAM.acctLocnCode |
| acct_crncy_code | BANCS.OUTPARAM.acctCrncyCode |
| system_only_acct_flg | BANCS. OUTPARAM.systemOnlyAcctFlg (Y/N) |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| | |
|---|---|
| Foracid | BANCS.OUTPARAM.AcctId |
| dr_bal_lim | BANCS.OUTPARAM.debitBalanceLimit |
| frez_code | BANCS.OUTPARAM. freezeCode |
| frez_reason_code | BANCS.OUTPARAM. freezeReasonCode |
| clr_bal_amt | BANCS.OUTPARAM. clearBalance |
| un_clr_bal_amt | BANCS.OUTPARAM. unclearBalance |
| ledg_num | BANCS.OUTPARAM. ledgerNumber |
| drwng_power | BANCS.OUTPARAM. drawingPower |
| sanct_lim | BANCS.OUTPARAM. sanctionLimit |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| Field Name in GAM table | Field name in BANCS repository |
|---|---|
| adhoc_lim | BANCS.OUTPARAM. adhocLimit |
| emer_advn | BANCS.OUTPARAM. emergencyAdvance |
| dacc_lim | BANCS.OUTPARAM. DACCLimit |
| system_reserved_amt | BANCS.OUTPARAM. systemReservedAmt |
| single_tran_lim | BANCS.OUTPARAM. singleTranLimit |
| clean_adhoc_lim | BANCS.OUTPARAM. cleanAdhocLimit |
| clean_emer_advn | BANCS.OUTPARAM. cleanEmergencyAdvance |
| clean_single_tran_lim | BANCS.OUTPARAM. cleanSingleTranLimit |
| system_gen_lim | BANCS.OUTPARAM. systemGeneratedLimit |
| chq_alwd_flg | BANCS.OUTPARAM. chequeAllowed |
| cash_excp_amt_lim | BANCS.OUTPARAM. cashExceptionAmtLimit |
| clg_excp_amt_lim | BANCS.OUTPARAM. clearingExceptionAmtLimit |
| xfer_excp_amt_lim | BANCS.OUTPARAM. transferExceptionAmtLimit |
| cash_cr_excp_amt_lim | BANCS.OUTPARAM. cashCrExceptionAmtLimit |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| | |
|---|---|
| clg_cr_excp_amt_lim | BANCS.OUTPARAM. clearingCrExceptionAmtLimit |
| xfer_cr_excp_amt_lim | BANCS.OUTPARAM. transferCrExceptionAmtLimit |
| cash_abnrml_amt_lim | BANCS.OUTPARAM. cashAbnormalAmtLimit |
| clg_abnrml_amt_lim | BANCS.OUTPARAM. clearingAbnormalAmtLimit |
| xfer_abnrml_amt_lim | BANCS.OUTPARAM. TransferAbnormalAmtLimit |
| acrd_cr_amt | BANCS.OUTPARAM. accruedCreditAmt |
| pb_ps_code | BANCS.OUTPARAM. passbookOrPasssheet |
| serv_chrg_coll_flg | BANCS.OUTPARAM. serviceChrgCollectedFlg |
| int_paid_flg | BANCS.OUTPARAM. interestPaidFlag |
| int_coll_flg | BANCS.OUTPARAM. interestCollectedFlag |
| limit_prefix | BANCS.OUTPARAM. limitPrefix |
| limit_suffix | BANCS.OUTPARAM. limitSuffix |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| | |
|---|---|
| clg_cr_excp_amt_lim | BANCS.OUTPARAM. clearingCrExceptionAmtLimit |
| xfer_cr_excp_amt_lim | BANCS.OUTPARAM. transferCrExceptionAmtLimit |
| cash_abnrml_amt_lim | BANCS.OUTPARAM. cashAbnormalAmtLimit |
| clg_abnrml_amt_lim | BANCS.OUTPARAM. clearingAbnormalAmtLimit |
| xfer_abnrml_amt_lim | BANCS.OUTPARAM. TransferAbnormalAmtLimit |
| acrd_cr_amt | BANCS.OUTPARAM. accruedCreditAmt |
| pb_ps_code | BANCS.OUTPARAM. passbookOrPasssheet |
| serv_chrg_coll_flg | BANCS.OUTPARAM. serviceChrgCollectedFlg |
| int_paid_flg | BANCS.OUTPARAM. interestPaidFlag |
| int_coll_flg | BANCS.OUTPARAM. interestCollectedFlag |
| limit_prefix | BANCS.OUTPARAM. limitPrefix |
| limit_suffix | BANCS.OUTPARAM. limitSuffix |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| Field Name in GAM table | Field name in BANCS repository |
|---|---|
| drwng_power_ind | BANCS.OUTPARAM. drawingPowerIndicator |
| drwng_power_pcnt | BANCS.OUTPARAM. drawingPowerPercentage |
| notional_rate | BANCS.OUTPARAM. notionalRate |
| notional_rate_code | BANCS.OUTPARAM. notionalRateCode |
| fx_clr_bal_amt | BANCS.OUTPARAM. FXClearBalance |
| crncy_code | BANCS.OUTPARAM. FCNRCrncyCode |
| wtax_flg | BANCS.OUTPARAM. TaxFlg |
| wtax_amount_scope_flg | BANCS.OUTPARAM. TaxAmtScopeFlg |
| lien_amt | BANCS.OUTPARAM. lienAmt |
| acct_mgr_user_id | BANCS.OUTPARAM. accountManagerUserId |
| schm_type | BANCS.OUTPARAM. schemeType |
| Partitioned_flg | BANCS.OUTPARAM. partitionedFlg |
| Partitioned_type | BANCS.OUTPARAM. partitionedType |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM AvailableAmt |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| | |
|---|---|
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FxAvailableAmt |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FFDAvailableAmt |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FxFFDAvailableAmt |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM EffAvailableAmt |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FxEffAvailableAmt |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FullAvailableAmt |
| Not in GAM table (Sum of | BANCS.OUTPARAM FxFullAvailableAmt |

# FUNCTIONALITIES, INPUT AND OUTPUT – ACCOUNT INFORMATION

| Field Name in GAM table | Field name in BANCS repository |
|---|---|
| different limits) | |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FullEffAvailableAmt |
| Not in GAM table (Sum of different limits) | BANCS.OUTPARAM FxFullEffAvailableAmt |

# FUNCTIONALITIES, INPUT AND OUTPUT – GET FILE LOCATION

This function will return the full path of the file name specified as input. It can typically be used in scripts to get the location of a script file, which needs to be called from within another script. The search for the file will be carried out according to the site-customization directory search rules.

```
<--start
sv_a = <"filename">
sv_b = "SCRIPT" + "|" + sv_a
sv_c = urhk_getFileLocation(sv_b)
sv_d = BANCS.OUTPARAM.fileLocation
START(sv_d, sv_a)
exitscript
end-->
```

Input through INPARAM repository: NONE

Input String as function argument: Contains two parts separated by a '|' character.

- File Type – The type of the file whose path is required. The possible values for this are: "FORM", "EXE", "SCRIPT", "MRT", "SQL", "COM" and "MENU".
- File Name – The name of the file whose path is required.

Both the input parameters are mandatory.

Output through OUTPARAM repository: The userhook returns the path of the file in the variable "fileLocation". This can be accessed as follows:

sv_d = BANCS.OUTPARAM.fileLocation

Function return value: Returns 1 if path is successfully got else returns 0.

# FUNCTIONALITIES, INPUT AND OUTPUT – ROUND OFF AMOUNT

This function rounds off the amount to a nearby integer value depending upon the input value provided.

```
<--start
trace on
sv_a = 10986792.2358
#Input Amount
sv_b = 100
# Round off to 100 Rupees
sv_c = "L"
# Lower amount

BANCS.INPARAM.InputAmount = sv_a
BANCS.INPARAM.RoundOffAmt = sv_b
BANCS.INPARAM.RoundOffFlag = sv_c
sv_r = urhk_B2k_RoundOff ("")
if (sv_r == 1) then
# Error processing
endif
sv_q = BANCS.OUTPARAM.OutputAmount
# sv_q contains the rounded amount. This should be 10986700.0000
end-->
```

# FUNCTIONALITIES, INPUT AND OUTPUT – ROUND OFF AMOUNT

All the input variables should be populated in the INPARAM class of the BANCS repository. The variables are (all STRINGS):

InputAmount - The amount that is to be rounded off

RoundOffAmt - The numeric digit by which the amount to be rounded off

---

Example: If the amount is in Rupees

100 – 100 Rupees

.01 – 1 Paise

---

RoundOffFlag - Valid values are:

H - Roundoff to the next higher amount

L - Roundoff to the previous lower amount

N - Roundoff to the nearest amount

# FUNCTIONALITIES, INPUT AND OUTPUT – ROUND OFF AMOUNT

All the input variables should be populated in the INPARAM class of the BANCS repository. The variables are (all STRINGS):

InputAmount - The amount that is to be rounded off

RoundOffAmt - The numeric digit by which the amount to be rounded off

---

Example: If the amount is in Rupees

100 – 100 Rupees

.01 – 1 Paise

---

RoundOffFlag - Valid values are:

H - Roundoff to the next higher amount

L - Roundoff to the previous lower amount

N - Roundoff to the nearest amount

# FUNCTIONALITIES, INPUT AND OUTPUT – ROUND OFF AMOUNT

All the input variables should be populated in the INPARAM class of the BANCS repository. The variables are (all STRINGS):

InputAmount - The amount that is to be rounded off

RoundOffAmt - The numeric digit by which the amount to be rounded off

---

Example: If the amount is in Rupees

100 – 100 Rupees

.01 – 1 Paise

---

RoundOffFlag - Valid values are:

H - Roundoff to the next higher amount

L - Roundoff to the previous lower amount

N - Roundoff to the nearest amount

# FUNCTIONALITIES, INPUT AND OUTPUT – ROUND OFF AMOUNT

Function returns 0 or 1. If 1, then there was failure in roundingoff. If 0,

All output variables will be populated in the OUTPARAM class of the BANCS repository. The variables are (all STRINGS):

OutputAmount - The roundedoff amount

# FUNCTIONALITIES, INPUT AND OUTPUT – DATE ADD

This userhook is used to arrive at a date based upon the year and number of days supplied as an input.

```
< - -start

BANCS.INPARAM.calBase=”01”

sv_e=BANCS.OUTPARAM.acct_opn_date

sv_q = BANCS.OUTPARAM.days

sv_v = LEFT$(sv_e,10) + "|" + FORMAT$(CINT(sv_q), "%d")

sv_m = urhk_B2k_date_add(sv_v)

sv_n = LEFT$(BANCS.OUTPARAM.dateadd,10)

sv_o = LEFT$(BANCS.OUTPARAM.locdateadd,10)

print(sv_n)

print(sv_o)

end- ->
```

# FUNCTIONALITIES, INPUT AND OUTPUT – DATE ADD

1) 4 digit year string

2) Number of days (“-“indicates that it has to be subtracted)

3) Calendar Base – Determines the calendar type (Hijri/Gregorian/Buddha)

The output is available in the repository variables “dateadd” and locdateadd, which can be accessed as

BANCS.OUTPARAM.dateadd

BANCS.OUTPARAM.locdateadd

# FUNCTIONALITIES, INPUT AND OUTPUT – DATABASE SELECT

This userhook allows script writer to select data from database tables. The user specifies the title of each selected field separated by comma before pipe symbol and specifies the complete query after that. It stores the output error code of the query in BANCS.OUTPARAM.DB_ERRCODE and message in BANCS.OUTPARAM.DB_ERRMSG.

sv_r = *urhk_dbSelect*(variable)

The variable can be a scratch pad variable (sv_a) or a string (“user_id | select user_id from upr where session_id = 'xxxx'”).

Here note that everything before '|' (pipe symbol) is taken as the title value where as post pipe symbol is taken as query.

# FUNCTIONALITIES, INPUT AND OUTPUT – DATABASE SELECT

```
sv_a = getenv("B2K_SESSION_ID")
sv_b = "user_id | select user_id from lgi where session_id = '" + sv_a +
"'"
sv_r = usrhk_dbSelect(sv_b)
```

Comma separated Field titles followed by the Query, separated by pipe symbol. The field symbols should not have spaces.

The output fields are stored in BANCS.OUTPARAM.<title field>. An error free condition is represented by 0 value of BANC.OUTPARAM.DB_ERRCODE.

- In case the query returns more than single row, it gives a FATAL error hence one need to be very meticulous during query writing.
- There is a size limitation of the query string size. In case where query size is very large make use of scratch pad variables.

# FUNCTIONALITIES, INPUT AND OUTPUT – DATABASE DML

This userhook allows script writer to perform DML Statements on the database tables. Any updation in the database with this userhook will make an entry in SDA table.

sv_r = *urhk_dbSQL*(variable)

The variable can be a scratch pad variable (sv_a) or a string (“update Upr Set User_logged_on_flg = ‘N’ where session_id = 'xxxx'”).

```
<--start
trace on

sv_a="TRG2"
sv_b=urhk_dbSQL("update upr set user_logged_on_flg='N' where
user_id='"+sv_a+"'")
print(sv_b)


trace off
end→
```

# FUNCTIONALITIES, INPUT AND OUTPUT – DATABASE DML

This userhook allows script writer to perform DML Statements on the database tables. Any updation in the database with this userhook will make an entry in SDA table.

sv_r = *urhk_dbSQL*(variable)

The variable can be a scratch pad variable (sv_a) or a string (“update Upr Set User_logged_on_flg = ‘N’ where session_id = 'xxxx'”).

```
<--start
trace on

sv_a="TRG2"
sv_b=urhk_dbSQL("update upr set user_logged_on_flg='N' where
user_id='"+sv_a+"'")
print(sv_b)


trace off
end→
```

# FUNCTIONALITIES, INPUT AND OUTPUT – DATABASE DML

The output fields are stored in BANCS.OUTPARAM.<title field>. An error free condition is represented by 0 value of BANC.OUTPARAM.DB_ERRCODE.

**Note:**

*This userhook must be used with utmost caution since none of the Finacle Application logic will apply here.*

# FUNCTIONALITIES, INPUT AND OUTPUT – PRINT REPO FIELDS

This userhook is used to print the structure of a repository. The input to this userhook is a scratch pad variable or a string specifying repository variable.

```
<--start

trace on

sv_r=urhk_B2k_PrintRepos("BANCS")

end-->
```

Input would be the name of the repository.

The output is available in the trace file under CDCI_LOGS.