CONTROL-SA®

Asynchronous Batch-Processing System

Developers Guide

[ for Enterprise SecurityStation® 3.2 ]

Version 1.2

October 30, 2008
Contents

1. Types of Async-Procedures .................................................................1
   INVOKING ASYNC-BATCHES .....................................................1
   INVOKING ASYNC-TRIGGERS ..................................................2
     Selecting the proper User Exit ....................................................2
     Adding Event-Processing Code to User Exits ............................4
     Invoking Async-Triggers from Event-Processing Code .............6

2. Coding Async-Procedures (batches or triggers) .................................9

3. Troubleshooting Async-Procedures ................................................11
   MONITORING JOB-EXECUTION .............................................13

4. Async TCL-Routines ........................................................................15

5. Coding Conventions .........................................................................22

6. Source-code Control System ..........................................................25
1. Types of Async-Procedures

The Async Batch Processing System is used primarily for two purposes...

- to execute batch procedures = async-batches
- to respond (asynchronously) to events = async-triggers

Both batches and triggers ultimately result in the execution of a program written in TCL. These TCL programs are referred to as async-procedures. The difference between async-batches and async-triggers is the way in which they are invoked. Async-batches are invoked from the UNIX command-line, or from a shell-script or a Job-Scheduler. Async-triggers are launched from ESS User-Exits in response to events.

Internally, there is no difference between the two. Both async-batches and async-triggers are coded in the same way. Both async-batches and async-triggers run fully within the ESS system - as if they were ordinary User-Exits written in TCL.

Invoking an async-procedure is accomplished by writing a job-file, requesting the procedure’s execution, to the async job-queue directory. Jobs can be submitted with one of four priorities: low, medium, high and immediate.

Immediate-priority jobs are processed immediately (synchronously) by a dedicated server process. All other jobs are processed asynchronously by the async job-processors. The job-processors will process jobs from oldest to most recent [first-in-first-out]. The job-processors will first process any high-level jobs in the queue. If there are no high-level jobs, then they will process medium-level jobs. Only if there are no high- or medium-level jobs, will low-level jobs be processed. It is also possible to tell the job-processors to only process high-level jobs, and to leave all other jobs in the queue. Alternatively, the processors can be told to ignore all low-level jobs.

Invoking Async-Batches

Async-batch jobs are invoked from the Unix command-line, or from ordinary programs or shell-scripts. The command "send.async.sh" is used to request the execution of an async-batch job. The syntax of this command is simply...

```
send.async.sh [options] command_name "description" "parm1" "parm2" .... \
"-PARM1" "val1" "-PARM2" "val2" ... \
```

- **command_name** is the name of the TCL program, without the ".tcl" extension
- **description** is a short description [which will identify the job in the ESS GUI]
- **parm1 parm2 ...** are parameters which are passed to the TCL program
- **-PARM1 val1 ...** override global parameters inside the TCL program

These [options] control the execution...

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-priority high</td>
<td>Submit high-priority job</td>
</tr>
<tr>
<td>-priority low</td>
<td>Submit low-priority job</td>
</tr>
<tr>
<td>-foreground -f -quiet</td>
<td>Run job immediately via dedicated job-processor</td>
</tr>
<tr>
<td></td>
<td>Return when execution has completed</td>
</tr>
<tr>
<td></td>
<td>-quiet &amp; -no_trace = avoid sending output to the terminal</td>
</tr>
<tr>
<td>-trace -foreground -interactive -tr -f -int</td>
<td>Used for testing async's</td>
</tr>
<tr>
<td></td>
<td>- job runs in foreground</td>
</tr>
<tr>
<td></td>
<td>- all output goes to terminal</td>
</tr>
</tbody>
</table>
Invoking Async-Triggers

Async-triggers are invoked from User Exits. At TeliaSonera, the code for all User Exits is to be found in the "~/TS/tcl" directory. For each User Exit, there is a file named "u_TS###.tcl", where ### is the exit-number.

Selecting the proper User Exit

The first step in constructing an event trigger, is to select the proper User Exit. The following table lists the various exits present in the "~/TS/tcl" directory, and discusses their usage.

<table>
<thead>
<tr>
<th>Exit Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_TS002.tcl</td>
<td>Invoked whenever an RSS-User is downloaded or is sent to the ESS by an interceptor. This exit should determine the corresponding Enterprise User, and set the &quot;user_id&quot; field accordingly. Typically async-triggers invoked from this exit will run with priority &quot;immediate&quot;, to avoid an inordinate number of async jobs from being generated. Selected user_id and user_name should be returned in the global variables EssEntUserID &amp; EssEntUserName. Alternatively, jobs could purposely be run asynchronously, to take advantage of the parallel processing offered by the async-processing system. In this case, the user_id would be set with batchrun.</td>
</tr>
<tr>
<td>u_TS003.tcl</td>
<td>Used to authorize operations based on entity details, or ESS Admin properties. Async-triggers run from this exit must run with priority &quot;immediate&quot;. They should attempt to valid the operation as fast as possible to avoid causing problems with the GUI. Async should return authorization in the global variable EssReturn.</td>
</tr>
<tr>
<td>u_TS004.tcl</td>
<td>Called during password sync operations. Exit can alter password before it is sent to an RSS. Async-triggers run from this exit must run with priority &quot;immediate&quot;, to be able to return a password. New password should be returned in the global variable EssPassword. The password-sync can be blocked for the indicated RSS by setting the global variable EssReturn to &quot;false&quot;.</td>
</tr>
<tr>
<td>u_TS005.tcl</td>
<td>Called whenever an alert is generated. Care must be taken when launching async-jobs from this exit, as async-jobs typically generate alerts themselves. This could result in 'infinite-loops'.</td>
</tr>
<tr>
<td>u_TS006.tcl</td>
<td>Called whenever an audit record is written. Used to respond to entity creation, modification and (sometimes) deletion. Most Async-jobs are launched from this exit (or exit 9). These async jobs are best run asynchronously, so as not to monopolize exit 6 (ie: AppServer/gateway). Note that certain operations may cause this exit to be called multiple times for a single operation. Care must be taken not to repeat work already done. Async-jobs have plenty of time to do their work. Use this time to perform the proper validation before responding to events.</td>
</tr>
<tr>
<td>u_TS007.tcl</td>
<td>Used to validate entity details. Based on keyword-values, the operation can be allowed or disallowed. Async-triggers run from this exit must run with priority &quot;immediate&quot;. They should attempt to valid the data as fast as possible to avoid causing problems with the GUI. Async should return validation in the global variable EssReturn.</td>
</tr>
<tr>
<td>u_TS008.tcl</td>
<td>Called just before records are written to the ESS Database. This exit can modify data, before the record is written to the database and sent to a target RSS. Async-jobs are often launched from this exit, as opposed to exit 6, if they need to access record contents before modification. Especially delete operations are typically processed from this exit. Code running in this exit receives new keyword-values via global variables, and can access old keyword values by querying the database. Please note, that this is only possible from within this exit. If an async is launched asynchronously, then by the time it is executed, the old record will have been overwritten by the new record [or it may have been deleted !]. Typically, code in &quot;u_TS009.tcl&quot;, will use EssGetDbData or EssEntityMultiValue to read the previous keyword-values from the database, write these to a file, and pass the name of the file to the async-procedure.</td>
</tr>
<tr>
<td>u_TS009.tcl</td>
<td>Called just before records are written to the ESS Database. This exit can modify data, before the record is written to the database and sent to a target RSS. Async-jobs are often launched from this exit, as opposed to exit 6, if they need to access record contents before modification. Especially delete operations are typically processed from this exit. Code running in this exit receives new keyword-values via global variables, and can access old keyword values by querying the database. Please note, that this is only possible from within this exit. If an async is launched asynchronously, then by the time it is executed, the old record will have been overwritten by the new record [or it may have been deleted !]. Typically, code in &quot;u_TS009.tcl&quot;, will use EssGetDbData or EssEntityMultiValue to read the previous keyword-values from the database, write these to a file, and pass the name of the file to the async-procedure.</td>
</tr>
</tbody>
</table>
Types of Async-Procedures

**u_TS010.tcl**
Called at the beginning and end of a password synchronization process - if the password for Enterprise User was modified. This exit can prevent the password change from being propagated to any of the RSS's.
To be able to prevent a password-sync operation from occurring, async's launched from this exit must be run with priority "immediate". The must return approval or rejection via the global variable EssReturn.
When responding to the end of a password-synchronization process, it is best to launch the async-procedure asynchronously with low-priority.

**u_TS011.tcl**
Called whenever transactions terminate. Usually invoked by the Gateway when transactions arrive from a remote Agent.
This exit is typically used - instead of exit 6 or 9 - when it is necessary to respond to actual changes to RSS entities. For example, if an RSS-User is modified, then exit 9 will be called with the new keyword-values. Then exit 6 will be called to indicate that the RSS-User has been modified. However, the modification can subsequently be sent to the RSS, and the RSS can reject the changes. If an async-job were to be launched from exit 6, then it could be responding to modifications that will never actually be made.
Code in u_TS011.tcl, can check the transaction status to determine if the operation was successful. Depending on the success or failure, it can launch an async to respond accordingly.

Please note, that this exit does not provide all data needed to process the event in global variables. Only the ID of the transaction is provided in $EssTransID. This must be used to fetch the identity of the entity involved from the "trans" table (via "EssEntityMultiValue" - see example 1)

**u_TS012.tcl**
Called whenever the password for an Enterprise User is changed, or an Agent intercepts a local change to an RSS-User's password.
If the async wishes to process the password change, and set the EssReturn variable, it must be invoked with priority "immediate".

**u_TS013.tcl**
Called when an Enterprise User is connected to a Job-Code. This exit selects an account (or accounts) to be used for the EU – JC connection.
This exit must select an account and return to the caller, thus async's here will typically be run with priority "immediate". However, often this exit is only used to detect the start of an EU – JC connect operation, and in this case any priority is acceptable.

**u_T000#.msg**
Error messages for the above User-Exits.
Return codes from each routine is either "0" or a message number in these files.
Message declarations are in the format...
```
set errmsg(6001) "Failure processing Insert Person Event"
```
To refer to this message, do the following in the routine...
```
set ReturnCode 6002
return $ReturnCode
```

**u_T000.tcl**
Contains global variables shared by all the above User Exits. New variables that need to be accessible in all User-Exits can be added to this file.

**u_T000.DEVE.tcl**
Variables for DEVE environment.

**u_T000.TEST.tcl**
Variables for TEST environment.

**u_T000.PROD.tcl**
Variables for PROD environment.

**u_T000.cfg**
Contains settings for the User-Exit processing code itself.

---

1. set fields [list rss_user_name act_msg]
   set parms [list serial $EssTransID]
   set results [EssEntityMultiValue (trans) $fields $parms]
   set resrow [lindex $results 0]
   set rss_user_name [lindex $resrow 0]
   set act_message [lindex $resrow 1]
Adding Event-Processing Code to User Exits

The second step in constructing an event trigger, is to add event-processing code to the User Exit. Open the proper "u_TS0##.tcl" file, and add a new procedure with the name "essue### Xxxx Yyyy", where "###" is the exit-number, "Xxxx" is the event name, and "Yyyy" is the entity-name (name of the table in the ESS database).

The following tables show the required procedure-names for several selected exits & events...

| - EXIT 006 - |
|-----------------|---------------------------------------------------------------|
| essue006_Insert_<entity> | Add/create new records in the RSS database (e.g., add an RSS user or user group). |
| essue006_Update_<entity> | Change existing records in the RSS database (e.g., change an RSS user or user group). |
| essue006_Delete_<entity> | Delete/remove existing records from the RSS database (e.g., delete an RSS user). |
| essue006_Sync_<entity> | Data synchronization actions, which copy records from the RSS database (e.g., perform data synchronization of a user group). |
| essue006_LoadACL_<entity> | Load resource ACL records from the RSS database. |
| essue006_LoadResACL_<entity> | Load resource and resource ACL records from the RSS database (e.g., load resource connections to user groups). |
| essue006_PasswordSync_<entity> | Password synchronization (originating either in Enterprise SecurityStation or in an RSS). |
| essue006_Download_<entity> | Data download actions, which download records from the RSS database (e.g., perform an initial download for a newly-added RSS). |
| essue006_GlobalSync_<entity> | Global data synchronization actions, which synchronize current Enterprise SecurityStation data with current RSS data (e.g., synchronize data after a communication failure). |
| essue006_RstDownload_<entity> | Restart an interrupted download of the entire RSS database. |
| essue006_HandshakeEnded_<entity> | Successful connection. |
| essue006_LoginLogout_<entity> | Login and Logout activity in the GUI. |
| essue006_GatewayUpDown_<entity> | Start and completion of ESS gateway access. |
| essue006_RouterU/Down_<entity> | Start and completion of Router access. |
| essue006_Transaction_<entity> | Activity in the Transaction window. |
| essue006_Alert_<entity> | Change the status of an alert in the Alert window. |
| essue006_Batchrun_<entity> | Actions by utility batchrun. |

| - EXIT 009 - |
|-----------------|---------------------------------------------------------------|
| essue009_Insert_<entity> | Insert a new entity record. |
| essue009_Update_<entity> | Update an existing entity record. |
| essue009_Delete_<entity> | Delete an existing entity record. |

| - EXIT 011 - |
|-----------------|---------------------------------------------------------------|
| essue009_Insert_<entity> | The current operation is an insert. |
| essue009_Update_<entity> | The current operation is an update. |
| essue009_Delete_<entity> | The current operation is a delete. |
| essue009_Sync_<entity> | The current operation is a sync. |
| essue009_Password_<entity> | The current operation is a password change. |
| essue009_Revoke_<entity> | The current operation is a user revoke. |
| essue009_Restore_<entity> | The current operation is a user restore. |
| essue009_Load_<entity> | The current operation is a request to perform an RSS download. |
The following code is an example of a typical event-processing procedure:

```tcl
proc essue006_Insert_rss_user {} {
    set procedure [lindex [info level [info level]] 0]
    set ReturnCode 0

    set ru_name [EssGetFieldValue rss_user_name] ; # fetch properties current entity

    ...respond to event...

    if { ...processing was successful... } {
        set ReturnCode 0
    } else {
        set ReturnCode 6989
    }

    return $ReturnCode
}
```

6989 is a unique failure number.

For each "u_TS###.tcl" file, there is a corresponding "u_TS###.msg" file with the User Exit error messages.

For this 6989 failure, a message needs to be added to the "u_TS006.msg" file with the following format:

```
set errmsg(6989) "Explanation...."
```

return 0 on success
Invoking Async-Triggers from Event-Processing Code

The third step in constructing an event trigger, is to invoke the Async-Trigger procedure from the event-processing code. The event-processing code in the user-exits should do nothing more than gathering enough information to identify the entity being processed. This info is then passed on to the Async-Trigger where the real processing of the event can occur.

The TCL procedure "$async_send_cmd" is used to request the execution of an async-trigger. The syntax of this command is simply...

```
async_send_cmd $cmdName $descr $parmList $message $priority
```

- $cmdName is set to the name of the TCL program, without the ".tcl" extension
- $descr contains a short description [which will identify the job in the ESS GUI]
- $parmList is a list-variable containing the parameters which are passed to the TCL program
- $message is a message to write to the log-file before launching the async
- $priority indicates the priority ("high", "normal", "low", "immediate")

The following code is an example of a typical async-trigger invocation....

```
set cmdName "new_account"
set descr   "New Account Created: $ru_name on $EssRSSName / $EssRSSType"
set parmList [list $ru_name $EssRSSName $EssRSSType] ; # input-parameters
set res [ async_send_cmd $cmdName $descr $parmList ]
if {$res == 1} {
    set ReturnCode 0 ; # procedure submitted ok
}
```

This will run the procedure "~/TS/tcl/new_account.tcl", and pass it the parameters $ru_name, $EssRSSName and $EssRSSType.

To run the async-trigger with priority "immediate", do the following....

```
set res [ async_send_cmd $cmdName $descr $parmList [-] immediate ]
if {$res == 1} {
    set ReturnCode 0 ; # procedure executed ok
}
```

This will run the procedure "~/TS/tcl/new_account.tcl" immediately, and will return once execution has completed. Please note: only use immediate-priority if absolutely necessary, as it can cause timing and resource problems.

When is "immediate priority" necessary ???

- If the async-trigger needs to return information to the event-processing code.
- If the order in which events are processed is essential.
  - NOTE: job-chaining [see below] can be used to determine the order in which jobs submitted from the same event-procedure are executed, but there is no way to guarantee the order of execution of jobs submitted from different event-procedures.
**Job-Chaining**

Sometimes it is important to execute triggers one after another in a particular order. This can be accomplished by using the job-chaining feature...

```plaintext
global ASYNC_AFTER_JOB ASYNC_JOB_NUM

set res1 [ Async_send_cmd "cmd1" "desc1" $parms1 ]

set ASYNC_AFTER_JOB $ASYNC_JOB_NUM
set res2 [ Async_send_cmd "cmd2" "desc2" $parms2 ]

set ASYNC_AFTER_JOB $ASYNC_JOB_NUM
set res3 [ Async_send_cmd "cmd3" "desc3" $parms3 ]
```

Async-trigger "cmd1" will be executed first, followed by "cmd2", and as last "cmd3".

After each job is launched, $ASYNC_JOB_NUM will contain the job-id of the job just launched. Setting SASYNC_AFTER_JOB to this id, will cause the following job to pause until the specified job has completed. As long as the specified job is in the queue, the dependent job will not be executed.

Note, that it is also possible to specify that several jobs wait until a single job has been executed, allowing slightly more sophisticated dependencies to be specified.

Please also note that this only works for jobs launched from the same piece of code.

**Execute at "end"**

Sometimes it is necessary to execute an async-trigger AFTER all other triggers have finished. This can be accomplished by calling the special async-trigger "wait_no_jobs"...

```plaintext
global ASYNC_AFTER_JOB ASYNC_JOB_NUM

set max_wait 60 ; # Wait for no more than one hour. (Set to zero to wait forever)
set extra_wait 2 ; # After all jobs have finished, wait for an extra 2 minutes
# to make certain no new jobs have been launched. (Set to zero to skip this test)
set end_time "20:34" ; # To wait until all jobs launched before a particular time
# have completed, specify this time as a third parameter
# Do not provide a third parameter if you wish to wait until all jobs have completed, that have been launched before
# "wait_no_jobs" was launched.
set parms [list $max_wait $extra_wait $end_time]
set dummy [ Async_send_cmd "wait_no_jobs" "Wait until all else completes" $parms ]

set ASYNC_AFTER_JOB $ASYNC_JOB_NUM
set res2 [ Async_send_cmd "mycmd" "Do this at end" $myparms ]
```

The async-job "wait_no_jobs" waits until all jobs have terminated, that were put into the Queue before it. Optionally, it can be asked to block until after a particular time.

Once the wait is done, "wait_no_jobs" will terminate, and the following (dependent) job or jobs will run.
One of the more difficult parts of constructing triggers - which respond to CONTROL_SA events - is determining which exits to use, what the entity and event names are, and which global-variables are available. At TeliaSonera, the user-exits can be configured to write this type of information automatically to a log file, while processing events.

On the test system, make sure no one is using the GUI, and no Agent interceptors are producing events
[ie: all is quiet]

Shutdown ORBIX and the gateways...

async stop orbix gateways

Open the "~/TS/tcl/u_TS000.cfg" file, and set the "MIN_LEVEL_LOG" parameter to 75 or less.

Now, open the various ‘STCL’ exits (~/appl/ess.version/exits/Tcl/src/essue0*.tcl), and search for the following lines....

if 0 { set X "Variables:" 
    foreach _ [info vars Ess*] {set X "$X\n\t$_=[eval set $_ "$_"]" } 
    ess_p "$X" }

if 0 { ess_p [ess_display_fields] }

Change the "if 0" to "if 1".

Restart ORBIX and the gateways...

async start orbix gateways

Now initiate the events you wish to trigger on. Trace information will be written to the "${ESS_LOGDIR}/TCL.log" and "${ESS_LOGDIR}/essue___.log" files. This should provide enough data to construct the triggers.
2. Coding Async-Procedures
(batches or triggers)

An async-procedure resembles any other TCL program. There are only a few lines of code that are required to make the program function as an async-procedure.

The following code is an example of a typical async-procedure...

```tcl
# PROGRAM:       new_account.tcl (called as async-trigger)
# DESCRIPTION:   Processes new accounts
# Version 001: (major)
# 25 apr 2004 - Initial Version
# Copyright: 2004 - BMC Software - Richard Haaf

#global SIMULATE
set SIMULATE 0 ; # 0 = execute statements, don't display them
# ---------------- no user editable parts beyond this point ---
# -------------------------------------------------------------
# Copywrite: 2004 - BMC Software - Richard Haaf
# "main" procedure
#-------------------------------------------------------------
# Copywrite: 2004 - BMC Software - Richard Haaf
#program is created in the "~/vw/tcl" directory.

# ---------------- no user editable parts beyond this point ---
# -------------------------------------------------------------
# Copywrite: 2004 - BMC Software - Richard Haaf
#program is created in the "~/vw/tcl" directory.

global SIMULATE
set SIMULATE 0 ; # 0 = execute statements, don't display them
# -------------------------------------------------------------
# ---------------- no user editable parts beyond this point ---
# -------------------------------------------------------------
# Copywrite: 2004 - BMC Software - Richard Haaf
# "main" procedure
#-------------------------------------------------------------
# Copywrite: 2004 - BMC Software - Richard Haaf
#program is created in the "~/vw/tcl" directory.

# Copywrite: 2004 - BMC Software - Richard Haaf
# "main" procedure
#-------------------------------------------------------------
# Copywrite: 2004 - BMC Software - Richard Haaf
#program is created in the "~/vw/tcl" directory.

# main

# mandatory code (insert exactly as it appears here)
```
if {[llength $INPUT_PARMS] >= 3} {
    set RU_NAME [lindex $INPUT_PARMS 0]
    set RSS_NAME [lindex $INPUT_PARMS 1]
    set RSS_TYPE [lindex $INPUT_PARMS 2]
    if {[catch {set res [ Process_Account $RU_NAME $RSS_NAME $RSS_TYPE ]} errMsg]}
        logit $routine "XXX995: FATAL TCL ERROR..."
        logit $routine $errMsg
        set res 0
    }
    set RESULT_CODE $res
    switch -- $res {
        {-1}    { set RESULT_MESSAGE "PROCESSING NEW-ACCOUNT PRODUCED WARNINGS." }
        {1}       { set RESULT_MESSAGE "PROCESSING NEW-ACCOUNT SUCCEEDED." }
        {default} { set RESULT_MESSAGE "PROCESSING NEW-ACCOUNT FAILED."  }
    }
    append RESULT_MESSAGE $PROG_MESSAGE
} else {
    logit $routine "XXX999: Only received [llength $INPUT_PARMS] parameters."
    set RESULT_CODE 0
    set RESULT_MESSAGE "USAGE: set INPUT_PARMS \[list ru_name rss_name rss rss_type\];
                      source new_account.tcl"
} 

############################################################
new_account.tcl

It should be clear from the example above, that an Async-Procedure is the same as any ordinary TCL program. Only a few required elements need to be added at the end of the code...

1. Add a call to "parse_opts" before any processing occurs...
   set INPUT_PARMS [ parse_opts $INPUT_PARMS ]

2. Extract the input parameters from the INPUT_PARMS list-variable...
   set PARM1 [lindex $INPUT_PARMS 0]
   set PARM2 [lindex $INPUT_PARMS 1]
   set PARM3 [lindex $INPUT_PARMS 2]

3. Call the "main" procedure inside a "catch" clause...
   if {[catch {set res [ Main_Routine $PARM1 $PARM2 ..... ]} errMsg]}
       logit $routine "XXX995: FATAL TCL ERROR..."
       logit $routine $errMsg
       set res 0
   }

4. Return the result of processing in the RESULT_CODE global variable...

<table>
<thead>
<tr>
<th>value</th>
<th>meaning</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>all okay</td>
<td>Alert is sent with status &quot;R&quot; (YELLOW) All temporary files &amp; records are cleaned up</td>
</tr>
<tr>
<td>-1</td>
<td>warning</td>
<td>Alert is sent with status &quot;U&quot; (ORANGE) All temporary files &amp; records are cleaned up</td>
</tr>
<tr>
<td>0</td>
<td>fatal error</td>
<td>Alert is sent with status &quot;V&quot; (RED) All temporary files are left in the $ESS_BATCH_DIR Site table entry is NOT deleted</td>
</tr>
</tbody>
</table>

5. Return a message in the RESULT_MESSAGE global variable.

RESULT_MESSAGE appears in "Remarks" field of the Alert :
3. Troubleshooting Async-Procedures

To understand the mechanism used to execute async-procedures, the preceding sample-procedure will be invoked from the UNIX command-line in interactive-mode. The output produced by the "send.async.sh" command will be analyzed, line for line.

```
send.async.sh -f -int -tr new_account "Test of new_account" test1 e2e_tam TAM
```

- async-procedure "new_account" is launched in the foreground, in interactive mode with full-tracing
- the parameters being passed to "new_account" are: "test1", "e2e_tam" and "TAM"

```
```

- job-file "04201240.26434.immediate" is created in async-queue directory
- "04201240.26434" is the job ID number. This number will appear in all error-messages, the site-table record and the Alert.
- "immediate" indicates that this is an "immediate" priority job, so it will processed by a dedicated async job-processor

```
```

- There are 5 ESS processes currently running on the server. This is less than the maximum of 60, so processing can continue

```
04-04-20.12:40:02 T (ess:04201240.26434) [process.async.26582:chk_orbix]   ORBIX is Up - continuing...
```

- ORBIX is up and running, so processing can continue

```
04-04-20.12:40:02 T (ess:04201240.26434) [process.async.26582:send_async_cmd]  Sending command "new_account" (Test of new_account)...
```

- async-command "new_account" is 'sent' by inserting a record of the type "CMD_EXECUTER" in the "site-table"
- this record will appear in the ESS GUI as shown here...

```
async-procedure name
async-procedure description
```

```
async-procedure name
```

- the site-table insertion will be trapped by "essue006.tcl" [running in address-space of ess-batchrun].
- code in "essue006.tcl", will execute the TCL program "~/TS/tcl/new_account.tcl".
- the "parse_opts" routine strips off and processes any system parameters. [in this case, a parameter to override the global variable "QUIET" which prevents trace output from being written to the log-file]

- TCL programs are supposed to begin with a call to "registerit", which produces the above message. "registerit" output cannot be suppressed, and is used to mark the start of an async-program.

Note the upper-case "R" after the date & time. Searching for "<space>R<space>" can be used to find the start of async-programs in the "async_exec.log" file.

Note also, following the "R", the name of the ESS-Administrator, the job-id and the async-program name.

- This line is produced by the call to "traceit" in the procedure 'Process_Account'. (Note the message type "T")

Each procedure in an async-program should start with a call to "traceit". This makes following process-flow easy. When "send.async.sh" is invoked with the "-interactive" parameter, all "traceit" output will go to the terminal-window, in real-time.

"traceit" calls are also important in the event of failure. When a fatal error occurs, the async-mechanism will collect all "traceit" messages that were sent before the error occured, and write them to the Alert, which is sent to the ESS GUI.

- The TCL program now returns a zero in the RESULT_CODE variable, and an error-message in the RESULT_MESSAGE variable.

- Execution returns to "essue006", where the RESULT_CODE variable is examined. Since it is zero, a failure is reported.

- "essue006", reports the total execution time of the async-procedure in seconds (less than one second).

- An Alert is sent to the ESS-GUI with the failure.

note: "job-name" job-ID (description)

process.async.26582: - SUBMISSION OK
process.async.26582: finished processing 1 file(s).  DONE.

- "dedicated async job-processor number 26582 terminates. Since job-submission was ok, job-file will be removed.

---

Monitoring Job-Execution

There are a number of async-commands available to monitor the async-batch processing system...

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async running</td>
<td>display list of all currently running jobs.</td>
</tr>
<tr>
<td>async running -v</td>
<td>display details on each running job</td>
</tr>
<tr>
<td>async waiting</td>
<td>display jobs waiting to be executed</td>
</tr>
<tr>
<td>async failed</td>
<td>display jobs for which submission failed</td>
</tr>
<tr>
<td></td>
<td>(batchrun returned error for &quot;INSERT site_table&quot; statement)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async tree</td>
<td>display process tree for job-processors</td>
</tr>
<tr>
<td>async tree -ess -async</td>
<td>include ESS processes in process-tree</td>
</tr>
<tr>
<td>async tree -v</td>
<td>display &quot;ps&quot; output for each process in tree (verbose)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async lims</td>
<td>display current state of job-processing system</td>
</tr>
<tr>
<td></td>
<td>will indicate if system is overloaded, or if processing is paused</td>
</tr>
<tr>
<td>async set -l</td>
<td>display settings (async.ini)</td>
</tr>
<tr>
<td>async set MAX_PROCS 80</td>
<td>set MAX_PROCS parameter to 80</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async log</td>
<td>display the contents of the async_exec.log</td>
</tr>
<tr>
<td>async log -v</td>
<td>display all details (date-time, ess-admin, job-id &amp; procedure-name)</td>
</tr>
<tr>
<td>async log 200</td>
<td>display last 200 lines of log-file</td>
</tr>
<tr>
<td>async log +0</td>
<td>display entire log-file</td>
</tr>
<tr>
<td>async log -f</td>
<td>continuously display log-output (tail -f) [stop with &lt;control-c&gt;]</td>
</tr>
<tr>
<td>async err</td>
<td>display only errors from log-file</td>
</tr>
<tr>
<td>async err +0 -v</td>
<td>grep &quot;${job_id}&quot;</td>
</tr>
<tr>
<td>async log +0 -v</td>
<td>grep &quot; R &quot;</td>
</tr>
</tbody>
</table>

There are a number of async-commands available to control the async-batch processing system...

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>async pause</td>
<td>pause all job-processing [after current jobs have finished]</td>
</tr>
<tr>
<td>async resume</td>
<td>resume job-processing</td>
</tr>
<tr>
<td>async pause -nolow</td>
<td>do not process any &quot;low&quot; priority jobs [until &quot;async resume&quot;]</td>
</tr>
<tr>
<td>async pause -highonly</td>
<td>only process high-priority jobs</td>
</tr>
<tr>
<td>async increase 5</td>
<td>add 5 more job-processors</td>
</tr>
<tr>
<td>async decrease 3</td>
<td>stop 3 of the job-processors</td>
</tr>
<tr>
<td>async start</td>
<td>make sure at least MIN_SERVERS processors are running</td>
</tr>
<tr>
<td>async stop</td>
<td>stop all processing after current jobs have finished</td>
</tr>
<tr>
<td>async kill</td>
<td>perform immediate emergency stop - killing all running jobs</td>
</tr>
</tbody>
</table>
async running
> There are 8 Async-Jobs currently being processed.
> JOB 1: Server 5476: new_account "Test 1" (0201433.15633)...
> JOB 2: Server 5403: new_account "Test 4" (0201433.15789)...
> JOB 3: Server 5385: new_account "Test 2" (0201433.15667)...

async running -v
> There are 6 Async-Jobs currently being processed.

------------------------------------------------------------------ [ running jobs ]
> JOB 1: Server with PID 5420 is processing COMMAND 04201433.16519...
> - JOB-FILE = /export/home/ess/TS/data/jobs/04201433.16519.low.5420.processing
> - CMD-NUMBER = 04201433.16519
> - ESS-USER = ess
> - DATE = Tue Apr 20 14:33:35 MEST 2004
> - COMMAND-NAME = "new_account"
> - DESCRIPTION = "Test 10"

------------------------------------------------------------------ [ job-details ]
> JOB 1: Server with PID 5420 is processing COMMAND 04201433.16519...
> - JOB-FILE = /export/home/ess/TS/data/jobs/04201433.16519.low.5420.processing
> - CMD-NUMBER = 04201433.16519
> - ESS-USER = ess
> - DATE = Tue Apr 20 14:33:35 MEST 2004
> - COMMAND-NAME = "new_account"
> - DESCRIPTION = "Test 10"

------------------------------------------------------------------ [ tree ]
> SERVER: 5367 ................/bin/ksh ./asyncsrv -quiet
>     `-28447 .................ess.batchrun -A -i /export/home/ess/TS/tmp/Alert_2126976225.cmd
>     `-29512 .................<defunct>
>     `-29517 ................../bin/sh /export/home/ess/appl/ess.version/etc/xshell_timeout_undo
>       `-29526 ................./bin/sh /export/home/ess/appl/bin/ess -exec nddes -d -k 2000.0ms
>         `-29527 ...............<defunct>
>         `-29865 ................./bin/sh /export/home/ess/appl/bin/ess -exec nddes -d -k 2000.0ms
>           `-29866 ..............<defunct>
>           `-29867 ................/bin/sh /export/home/ess/appl/bin/ess -exec nddes -d -k 2000.0ms
>             `-29868 ..............<defunct>
>             `-29870 ................/bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>               `-29871 .............<defunct>
>               `-29872 .............../bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>                 `-29880 ..............<defunct>
>                 `-29881 ................/bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>                   `-29882 ..............<defunct>
>                   `-29883 ................/bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>                     `-29884 ..............<defunct>
>                     `-29885 ................/bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>                       `-29886 ..............<defunct>
>                       `-29887 ................/bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>                         `-29888 ..............<defunct>
>                         `-29889 ................/bin/sh /export/home/ess/appl/bin/ess -exec dfl_get -T VALUE -I xshell_timeout
>                           `-29890 ..............<defunct>
>                           `-29891 ................/usr/bin/expr //export/home/ess/appl/bin : \{{{[^/]*|[^/]*}}/[^/]*}|[^/]*} ? \(/[^/]*\)
Async TCL-Routines

Async-procedures inherit a set of TCL-library routines that can be used to interact with CONTROL-SA. This small collection of routines provides 90% of all functionality needed within a typical CTSA batch-procedure or event-trigger. All these routines are available in all user-exits (u_TS###.tcl routines), as well as in the async-procedures themselves. There is no need to load a TCL library; it is always pre-loaded.

**Routines to write messages to log-file / terminal**

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>registerit</td>
<td>procedure_name message</td>
<td>mark start of major block of processing</td>
</tr>
<tr>
<td>notifyit</td>
<td></td>
<td>place a notification in the async-log</td>
</tr>
<tr>
<td>traceit</td>
<td></td>
<td>display program flow</td>
</tr>
<tr>
<td>dumpit</td>
<td></td>
<td>display variable values and other 'dump' info</td>
</tr>
<tr>
<td>logit</td>
<td></td>
<td>register an ERROR, should only be used if an error has occurred</td>
</tr>
</tbody>
</table>

**examples**

```tcl
set routine [my_procedure]
registerit "$routine "Start of large block of code..."
notifyit "$routine "IMPORTANT: something important happened."
traceit "$routine "Now $this is about to happen..."
dumpit "$routine "- myvariable = \"$myvariable\"
logit "$routine "MPG902: This fatal error occurred...\n\nNotes:
These routines write their output to "$EWS_LOG_DIR|async_exec.log", where $EWS_LOG_DIR is defined in "$/u_TS.cfg".
```

**EssBatchRun**
Routine to execute a file with batch-run statements

**EssExecBatch**
Routine to execute a single batch-statement

**EssGetDbData**
Routine to fetch data from the ESS Database - returns data in a two-dimensional list

**EssEntityMultiValue**
Routine to quickly fetch one row of data from the ESS Database - returns data in a (two-dimensional) list

**EssEntityValue**
Routine to quickly fetch a single field from one row of data from the ESS Database

**EssEntityExists**
Routine to quickly test for the existence of a given entity

**EssIndexFieldName**
Retrieve index-field name for user-defined keyword

**EssFullName**
Determine full-field name for non-structure keywords (name with "__##__" prefix)

**EssGetParm**
Fetch a Parameter Value from the Site-Table

**EssAlert**
Send an Alert to ESS GUI
Routine to execute a file with batch-run statements

routine EssBatchRun
parameters cmdFile resFile {dataFile} {useEssClient}

description
Executes file with batch-run statements. Detects non-reported errors. Has a retry mechanism to detect failures caused by overloading of Oracle database & retry operation. Reports - per failed batchrun-statement - the errors that occurred (to log-file).

description
...create input file with batchrun statements
...don't forget to terminate lines with a ";"
...add "execute;" on a separate line to force sending transactions
...pass file-name of input file, and a second file-name for results, to EssBatchRun...

set res [ EssBatchRun $myInputFile $myOutputFile {} 0 ]
# ^--- always do INSERT/UPDATE/DELETE via BatchRun not ess.client
if {$res == 1} {
# all statements executed without errors
catch [exec rm $myInputFile]
catch [exec rm $myOutputFile] ; # clean up both files
} else {
# failure(s) occurred ... errors are to be found in $resFile
}

notes
This routine can either use "ess-client" or "ess -batchrun" to execute the batch-statements. To force the use of "ess-client", add a "1" to the end of the parameter list. To force the use of "batchrun", add a "0". Ordinarily, all DML is done via batch-run. However, ess-client could prove to give better performance. The input file is assumed to be in batch-run format. It is converted to ess-client format, if ess-client is used to execute the file.

There is no need to enclose the call to EssBatchRun in a "catch" block, as all error trapping is performed by EssBatchRun.

Note, this routine has been tested with very large batch-files that take many hours to process. It has proven to be very reliable, and able to detect most every form of failure (including database connection problems).

Use the global variable ESS_BATCH_PARMS to pass special parameters to "ess batchrun"...

-c Check commands only
-t Do not create transactions
-S trans_priority Transaction priority
-S list_sep List separator character
-S sub_list_sep Sublist separator character
-S char end of line character
-S RUID Delimiter character
-v verbose mode Database verbosity level

If the batch-command resulted in one or more transactions, the transaction-numbers are to be found in the global list-variable ESS_TRANSID_LIST.

Please note that for Logical transactions (eg: Profile-Person Connects), the transaction-id is often the transaction-id of the last child transaction launched by the parent Logical transaction. If no child-transactions were launched, the transaction-id is often garbage. [this is a "feature" of ess-batchrun]

Routine to execute a single batch-statement

routine EssExecBatch
parameters batchcmd description useEssClient

description
Puts batchrun-statement in a temporary file, passes the file to EssBatchRun, and cleans up the files if no errors occurred.

description
...create input file with batchrun statements
...don't forget ';' at end of each command.
..."execute;" is added automatically at end of batch-file.

set cmd "INSERT ent.user WITH user_id = "000123" SET user_name = "Pieterson, Sam;""
- don't forget ";" at end of each command.
- "execute;" is added automatically at end of batch-file.

set res [ EssExecBatch $cmd "Create Pieterson" 0 ]
# ^--- always do INSERT/UPDATE/DELETE via BatchRun not ess.client
if {$res == 1} {
# statement executed without errors
} else {
# failure(s) occurred
}
input parameter can contain several batch-commands, separated by new-lines...
set cmd "\nINSERT ent1 ....;
INSERT ent2.......\n"

See EssBatchRun command for further information.

Routine to fetch data from the ESS Database - returns data in a two-dimensional list

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssGetDbData</td>
<td>query colNames description {useEssClient}</td>
</tr>
</tbody>
</table>

Queries the ESS Database. Detects non-reported errors, and non-existent entities. Has a retry mechanism to detect failures caused by overloading of Oracle database & retry operation. Returns data in a two-dimensional TCL list-variable.

example

set ret 0
set query "READ_ALL rss_user_name rss_name rss_type __99__ru_manager FROM rss_user"
set query "$query WITH rss_name = SAP*"
set colNames [list rss_user_name rss_name rss_type __99__ru_manager]
- colNames is a repeat of columns specified in query [ColNames can be shorter & have keywords in a different order]
- order of keywords in colNames determines order of keywords returned by EssGetDbData

set results [ EssGetDbData $query $colNames "accounts & managers" ]
if { $results == <ERROR> } { # fatal error occurred, no valid data returned
  return $status
} elseif { $results == <EOF> } { # entity not found, no data returned
  return $status
} else {
  set cnt 0
  foreach rowinfo $results { # all ok
    dumpit $routine "- received [llength $results] row(s)."
    set ru_name [lindex $rowinfo 0]
    set rss_name [lindex $rowinfo 1]
    set rss_type [lindex $rowinfo 2]
    incr cnt
    if [ $cnt >= 200000 ] { # process account number $cnt....
      set ret 1
    } else {
      # all ok
      dumpit $routine "- received [llength $results] row(s)."
    }
  }
  return $ret
}

notes

This routine can either use "ess-client" or "ess -batchrun" to retrieve the data. To force the use of "ess-client", add a "1" to the end of the parameter list. To force the use of "batchrun", add a "0". Note, that using "batchrun" should be avoided, as it can cause deadlocks to occur.

There is no need to enclose the call to EssGetDbData in a "catch" block, as all error trapping is performed by EssGetDbData. Note, this routine has been tested with queries returning 200,000 or more records. It has also been tested with queries which return embedded carriage-returns, tabs or control-characters, strange international characters, empty fields, etc. There are currently no known bugs or problems.

The global parameter MAX_RETRY_CNT in "~TS/tcl/u_TS000.cfg" determines the maximum number of retries in the event of failures. However, if the failure is that the wrong ESS-Admin password is being used, then retrying a query three times will revoke the ESS-Admin's account ! Therefore, as a precaution, an "ess-client" command will never be issued more two times. Batch-run does not have this limitation.

In order to use ess.client to fetch data, this routine needs a valid ess-administrator's name and password. The name of the ess-administrator is taken from the UNIX environment variable $ESS_ADMIN. The password for this user is stored encoded in a data file. This file is ordinarily written by code in exit 6 that runs when the administrator named $ESS_ADMIN changes his/her password in the ESS GUI. However, this code is not present at TeliaSonera. At TeliaSonera, this data-file has been filled manually. The password in this file can be over-riden, by setting the GLOBAL parameter "ESS_GUI_P" in "~TS/tcl/u_TS000.tcl" to the proper password (only if this parameter is empty, is the data-file used).

This routine needs to know if the connection between ess.client and the AppServer is secure. This is determined by the global parameter "CLNT_ENCR" in "~TS/tcl/u_TS000.tcl". Set this variable to "1" if the connection is secure, "0" if it is not.
### Routine to quickly fetch one row of data from the ESS Database - returns data in a (two-dimensional) list

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssEntityMultiValue</td>
<td>TableName FieldList parmList</td>
</tr>
</tbody>
</table>

**description**

This routine uses EssReadDBTable to fetch the contents of a single record from the ESS database. If it is known that a query will return a single row of data, then this routine can typically retrieve the data in a small fraction of the time EssGetDbData would need to retrieve the same data. The limitation is that only structure-keywords can be used in the filter condition. It is possible to return user-defined keywords. [see EssIndexFieldName for work-around]

This routine also has a retry-mechanism that will detect failures and retry the query. The data returned by this function is identical to that returned by EssGetDbData. These two routines are designed to be interchangeable.

**example**

```tcl
set fields [list __99__Department user_name]
- fields are returned in this order

set parms [list user_id $UserId]
- parms is: 
  [[list keyword1 value1 keyword2 value2 ...]
- specify enough fields to uniquely identify record (ie: primary key)

set results [ EssEntityMultiValue {ent_user} $fields $parms ]
if {$results == {<ERROR>}} {
    # fatal error occurred, no valid data returned
} elseif {$results == {<EOF>}} {
    # entity not found, no data returned
} else {
    # all ok
    dumpit $routine "- received [llength $results] row(s)." ; # will always be one
    foreach rowinfo $results {
        set dept [lindex $rowinfo 0]
        set uname [lindex $rowinfo 1]
        ...process user....
    }
    set ret 1
}
return $ret
```

**notes**

Although this routine always returns only one row of data, the function returns a two-dimensional list. This is to make this routine interchangeable with EssGetDbData. A call to EssGetDbData can be replaced with a call to EssEntityMultiValue with very little code change.

This routine should be used instead of EssGet.DbData where-ever possible. The performance difference between the two routines is quite significant.

---

### Routine to quickly fetch a single field from one row of data from the ESS Database

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssEntityValue</td>
<td>TableName FieldName parmList</td>
</tr>
</tbody>
</table>

**description**

This routine uses EssReadDBTable to fetch the contents of a single field from a record in the ESS database. This routine is capable of returning any field from a given record. However, only structure-keywords can be used to locate and identify the record. [see EssIndexFieldName for work-around]

This routine also has a retry-mechanism that will detect failures and retry the query.

**example**

```tcl
set parms [list rss_user_name $ru_name rss_name $rss rss_type $rss_type]
set result [ EssEntity rss_user uid $parms ]
if {$result == {<ERROR>}} {
    # fatal error occurred
} elseif {$result == {<EOF>}} {
    # entity not found
} else {
    # ok, result contains field value
    set uid $result
}
```

**notes**

The advantage of using this routine, instead of EssReadDBTable directly, is that this routine handles errors and detects an "entity not found" condition. Also, the retry mechanism gives your code more resilience.
Routine to quickly test for the existence of a given entity

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssEntityExists</td>
<td>TableName, FieldName, parmList</td>
</tr>
</tbody>
</table>

**Description**

This routine uses EssReadDBTable to check for the existence of a given entity. Note that only structure-keywords can be used to locate and identify this entity. It is not possible to use user-defined keywords to identify the entity. [see EssIndexFieldName for workaround]

This routine also has a retry-mechanism that will detect failures and retry the query.

**Example**

```tcl
set parms [list rss_user_name $ru_name rss_name $rss rss_type $rss_type]
set exists [EssEntityExists rss_user_name parms]
- FieldName (rss_user_name here) can be any valid field, as long as it is not empty (mandatory / not null)
if {$exists == {-1}} {
    # error occurred
} elseif {$exists == 0} {
    # entity does NOT exist
} else {
    # entity DOES exist
}
```

**Notes**

This routine provides an easy way to test if an entity exists. It typically performs the test in a small fraction of a second.

Retrieve index-field name for user-defined keyword

<table>
<thead>
<tr>
<th>Routine</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssIndexFieldName</td>
<td>Entity, FieldName</td>
</tr>
</tbody>
</table>

**Description**

The routines EssEntityMultiValue, EssEntityValue and EssEntityExists provide very fast access to the ESS Database. Unfortunately, they can only use structure-keywords to locate records. Sometimes there is a user-defined keyword that also uniquely defines an entity. If there is an index on this keyword, then the index-field can be used to locate the entity. This routine will provide the name of this index-field, allowing EssEntityMultiValue, EssEntityValue and EssEntityExists to search using the user-defined keyword.

**Example**

```tcl
set ixfield [EssIndexFieldName {ent_user} __99__PERSON_ID]
if {$ixfield != {<EOF>} && $ixfield != {<ERROR>}} {
    # got index-field, try EssEntityMultiValue...
    set fields [list {user_id} {user_name}]
    set parms [list $ixfield $PersonID]
    set results [EssEntityMultiValue {ent_user} $fields $parms]
} else {
    # Index field-name unknown, must use EssGetDbData...
    set results {<ERROR>}
}
if {$results == {<ERROR>}} {
    # fatal error, no data retrieved
    set ret 0
} elseif {$results == {<EOF>}} {
    # entity was not found
    set ret (-1)
} else {
    set resrow [lindex $results 0]
    set user_id [lindex $resrow 0]
    set user_name [lindex $resrow 1]
    .....process record....
    set ret 1
}
```

**Notes**

If user-defined keyword is indexed, this routine can help to measurably speed up queries.
Determine full-field name for non-structure keywords (name with "__##__" prefix)

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssFullFieldName</td>
<td>Entity FieldName RSS_TYPE</td>
</tr>
</tbody>
</table>

**Description**

Some non-structure-keywords have a different full-field name for each RSS-Type. A good example is the UNIX GID field. This field has a different name for Solaris, AIX, HP, etc. This routine returns the full-field name when provided the name without the "__##__", and the RSS-Type.

**Example**

```tcl
set gid_list {}
set fldbname [ EssFullFieldName {user_group} {GID} $RSS_TYPE ]
if {$fldbname == {<ERROR>} || $fldbname == {<EOF>}} { # Can't determine name of GID field for $RSS_TYPE
    set gid_list {<ERROR>}
} else {
    set query "READ_ALL $fldbname FROM user_group WITH rss_type = \"$RSS_TYPE\";";
    set colNames [list $fldbname]
    set results [ EssGetDbData $query $colNames "GID List" ]
    if {$results == {<ERROR>}} { # Can't fetch list of GID numbers for $RSS_TYPE
        set gid_list {<ERROR>}
    } elseif {$results != {<EOF>}} {
        foreach resrow $results {
            set gid [lindex $resrow 0]
            lappend gid_list $gid
        }
    }
}
return $gid_list
```

**Notes**

This routine allows code to be built without hard-coded full-field names. This eliminates the need to maintain code as new RSS-Types are added to a CONTROL-SA system.

---

Fetch a Parameter Value from the Site-Table

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssGetParm</td>
<td>async_procedure_name parameter_name default_value</td>
</tr>
</tbody>
</table>

**Description**

Fetches a parameter-value from the site-table (type = IDM_ENV name = "${proc}_${parm_name}").

**Example**

```tcl
set parm [ EssParmValue "my_proc" "my_parm" "DEFAULT" ]
```

**Notes**

If the parameter does not exist, then this routine will try to add it to the site-table with type="IDM_ENV" name="${proc_name}_${parm_name}" and description="$default". If a failure occurs, it will return the specified default-value.
# Send an Alert to ESS GUI

<table>
<thead>
<tr>
<th>routine</th>
<th>parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>EssAlert</td>
<td><code>message {remarks} {severity} {ess_user} {rss_type} {rss_name} {trans_id} {platform} {rss_user_name} {ug_name} {res_name} {oe_full_name} {SerialNum}</code></td>
</tr>
</tbody>
</table>

**description**

Sends an Alert to the ESS GUI (inserts record in alarm table). The value of the "severity" parameter determines the color of the alert...

- severity: "R" 💼 (Regular - YELLOW)
- "U" 🔴 (Urgent - ORANGE)
- "V" 💥 (Very Urgent - RED)

**example**

```tcl
global ESS_ADMIN
set today [clock format [clock seconds] -format %y-%m-%d]
set now   [clock format [clock seconds] -format %H:%M:%S]
set msg "Failure processing User "$user_name" on $today at $now"
set Rem "The following errors occurred...
$n
$ERROR_MESSAGES"
set sev {V}
set res [ EssAlert $msg $Rem $sev $ESS_ADMIN ]
```

**notes**

Please note that this is a costly operation. It typically takes 5 to 10 seconds for this routine to execute.

This routine has filters for the message & remarks fields. It will remove any strange characters not accepted by the ESS. It will also truncate messages which are too long for the fields. Text which exceeds field length is replaced with a "+" sign.
5. Coding Conventions

As with any development framework, when coding async-procedures, it is best to follow the general coding conventions used by most programmers. This makes exchanging code simpler, and makes it easier for consultants to read each other's code. There are no hard-rules here, simply guidelines and suggestions.

To explain the conventions, the previous example from section "Coding Async-Procedures (batches or triggers)" will be reused...

```tcl
# PROGRAM:       upd_dept.tcl (called as async-trigger)
# DESCRIPTION:   Update rss_user's manager field.
# INVOKATION:    Call when department-field RU is changed, 
or new RU is added to system
#
# Version 001: (major)
# 25 apr 2004 - Initial Version
# the version here refers to the number generated by the "cout" utility (see "Source-code Control System" - p.25)
#
# Copywrite: 2004 - BMC Software - Richard Haaf
#
# don't forget a copywrite statement

global SIMULATE
set SIMULATE [ EssGetParm "upd_dept" "SIMULATE" \ 0 ] ; # 1 = print batchrun statements instead of executing them 

- most async-procedures are written to support a simulation mode. In simulation mode, EssExecBatch statements should be skipped by the code. This will allow logic to be tested, without effecting the ESS DB.
- global parameters are usually stored in the site-table, and retrieved with EssGetParm
- global parameter names should be in all upper-case letters (numbers and underscore ok)
- global parm settings can be overwriten by passing them as parameters to the async-procedure:
  example: [-SIMULATE 1] (extra parm to "send.async.sh" or Async_send_cmd)

# -------------------------------------------------------------
# ---------------- no user editable parts beyond this point ---
# -------------------------------------------------------------
# put user-configurable code above this point
# tell customer to leave the rest of the code alone

global PROG_MESSAGE
set PROG_MESSAGE {}

proc Process_Account { ru_name rss_name rss_type } {
  global PROG_MESSAGE SIMULATE
  - start procedures with global declarations
  set routine "Process_Account"
  - declare a variable with the name of the procedure
  set ret 1
  - initialize return code
  traceit $routine "Processing "$ru_name" on $rss_name/$rss_type..."
  - start each procedures with message about what the procedure will do
  - display input variables
  set fields [list __99__Department __99__Manager]
  set parms  [list rss_user_name $ru_name rss_name $rss_name rss_type $rss_type]
  set results [ EssEntityMultiValue {ent_user} $fields $parms ]
```
if {$results == {<ERROR>}} {
  - "<ERROR>" always indicates fatal error - should probably terminate processing
  set msg "Error accessing account details for "
  logit $routine "NAT001: $msg - cannot continue."
  - call logit to record failure.
  - NAT is code name for this async-procedure
  - 001 is error number
  append PROG_MESSAGE "\n> $msg"
  - add failure to PROG_MESSAGE.
  return 0
  - zero is 'failure' in TCL
}

if {$results == {<EOF>}} {
  - "<EOF>" indicates data not found
  set msg "Account "$ru_name" on $rss_name/$rss_type was not found."
  traceit $routine "$msg - cannot continue."
  - call "traceit" to record missing entity (only use "logit" for real failures).
  append PROG_MESSAGE "\n* $msg"
  - add warning to PROG_MESSAGE.
  return {-1}
  - minus-one is 'warning' in ASYNC
}

set rowinfo [lindex $results 0]
- extract first row
set dept [lindex $rowinfo 0]
set mangr [lindex $rowinfo 1]
- fetch field values
dumpit $routine "- dept \[${dept}\]
- mangr \[${mangr}\]"
- display field values
set org_manager [ Get_Org_Manager $dept ]
dumpit $routine "- orgman \[${org_manager}\]"

if {$org_manager == {<ERROR>}} {
  traceit $routine "Can't determine manager - cannot continue."
  - routine just called should set PROG_MESSAGE & generate errors
  return 0
}

if {$org_manager == {<EOF>}} {
  set org_manager {<unknown>}
  set msg "Department is unknown - manager set to "$org_manager"." 
  traceit $routine $msg
  append PROG_MESSAGE "\n* $msg"
  - indicates warning message
  set ret {-1}
  - return warning
}

if {$org_manager == $mangr} {
  - AVOID ANY UNNECESSARY TRANSACTIONS!
  set msg "Manager is already set correctly. Nothing to do."
  traceit $routine $msg
  append PROG_MESSAGE "\n- $msg"
  - indicates information message
  return $ret
  - no error - just return
}

set cmd "UPDATE rss_user WITH rss_name="\${ru_name}""
set cmd "$cmd rss_name="\${rss_name}" rss_type="\${rss_type}""
set cmd "$cmd SET __99__Manager="\${org_manager}";"

if {$SIMULATE} {
  traceit $routine "Would have set manager to "\${org_manager}"."
  set res 1
  - SIMULATE: do nothing
} else {
  set res [ EssExecBatch $cmd "Update Manager" 0 ]
  - always do UPDATE via BatchRun
}
if {$res == 1} {
    set msg "Manager changed from "$mangr" to "$org_manager"."
    traceit $routine $msg
    append PROG_MESSAGE "$msg"
} else {
    set msg "Could not change Manager from "$mangr" to "$org_manager"."
    logitit $routine "NAT002: $msg"
    - NAT is code name for this async-procedure
    - 002 is error number
    append PROG_MESSAGE "$msg"
    set ret 0
    - zero is error in TCL
}
return $ret

# MAIN PROGRAM
- mark start of main-routine

set routine "main"
global INPUT_PARMS RESULT_CODE RESULT_MESSAGE
set INPUT_PARMS [ parse_opts $INPUT_PARMS ]
set RESULT_CODE 1
set RESULT_MESSAGE ""

if {$SIMULATE} {
    registerit $routine "*SIMULATION* of UPDATING ACCOUNT MANAGER..."
} else {
    registerit $routine "UPDATING ACCOUNT MANAGER..."
}
- start with "registerit", to mark begin of async-procedure

set possible_number_of_parms { 3 }
if {[lsearch -exact $possible_number_of_parms [llength $INPUT_PARMS]] >= 0} {
    - check number of input-parms
    set RU_NAME [lindex $INPUT_PARMS 0]
    set RSS [lindex $INPUT_PARMS 1]
    set RSS_TYPE [lindex $INPUT_PARMS 2]
    - fetch input parms

    if {[catch {set res [ Process_Account $RU_NAME $RSS $RSS_TYPE ]} errMsg]} {
        - execute main procedure with "catch" clause to trap TCL syntax errors
        logit $routine "NAT995: FATAL TCL ERROR..."
        - NAT is code name for this async-procedure
        - 995 is error number
        logit $routine $errMsg
        - display TCL error on a separate line
        set res 0
    }

    set RESULT_CODE $res
    - set return-code
    switch -- $res {
        { -1 } { set RESULT_MESSAGE "UPDATE MANAGER PRODUCED WARNINGS." }
        { 1 } { set RESULT_MESSAGE "UPDATING MANAGER SUCCEEDED." }
        default { set RESULT_MESSAGE "UPDATING MANAGER FAILED." }
    }
    - set return-message
    append RESULT_MESSAGE $PROG_MESSAGE
    - add program-messages to return-message
}
else {
    logit $routine "NAT999: Received [llength $INPUT_PARMS] parameters."
    set RESULT_CODE 0
    set RESULT_MESSAGE "USAGE: set INPUT_PARMS "
    set RESULT_MESSAGE "$[RESULT_MESSAGE] \[list ru_name rss_name rss rss_type\]"
    set RESULT_MESSAGE "$[RESULT_MESSAGE] source new_account.tcl"
    - display usage
}
- mark end of code

set possible_number_of_parms { 3 }
if {[llength $INPUT_PARMS] >= 0} {
    - check number of input-parms
    set RU_NAME [lindex $INPUT_PARMS 0]
    set RSS [lindex $INPUT_PARMS 1]
    set RSS_TYPE [lindex $INPUT_PARMS 2]
    - fetch input parms

    if {[catch {set res [ Process_Account $RU_NAME $RSS $RSS_TYPE ]} errMsg]} {
        - execute main procedure with "catch" clause to trap TCL syntax errors
        logit $routine "NAT995: FATAL TCL ERROR..."
        - NAT is code name for this async-procedure
        - 995 is error number
        logit $routine $errMsg
        - display TCL error on a separate line
        set res 0
    }

    set RESULT_CODE $res
    - set return-code
    switch -- $res {
        { -1 } { set RESULT_MESSAGE "UPDATE MANAGER PRODUCED WARNINGS." }
        { 1 } { set RESULT_MESSAGE "UPDATING MANAGER SUCCEEDED." }
        default { set RESULT_MESSAGE "UPDATING MANAGER FAILED." }
    }
    - set return-message
    append RESULT_MESSAGE $PROG_MESSAGE
    - add program-messages to return-message
}
else {
    logit $routine "NAT999: Received [llength $INPUT_PARMS] parameters."
    set RESULT_CODE 0
    set RESULT_MESSAGE "USAGE: set INPUT_PARMS "
    set RESULT_MESSAGE "$[RESULT_MESSAGE] \[list ru_name rss_name rss rss_type\]"
    set RESULT_MESSAGE "$[RESULT_MESSAGE] source new_account.tcl"
    - display usage
}
6. Source-code Control System

The async-exec batch processing system includes a source-code control system for managing TCL programs and shell-scripts. Actually, 'source-code control' is a big word. It is a simple version-management tool that allows multiple versions of a source-file to be 'checked-in' and 'checked-out' by various programmers on a team.

The SCCS consists of two simple commands...

<table>
<thead>
<tr>
<th>cout</th>
<th>check out source for modification</th>
</tr>
</thead>
<tbody>
<tr>
<td>cin</td>
<td>check modified source back in for testing</td>
</tr>
</tbody>
</table>

The way it works is simple:

Login with your own account (not ess-owner - same user-group as ess-owner) Go to the directory containing the source that needs to be modified. Type: "cout source_name", where "source_name" is name of source without extensions (.tcl, .sh .pl .001 etc).

"cout" will find the latest source (highest version number), and make a copy with your UID & GID and append a new version-number. If the original file has no version-number, a second copy is made with version "000", and you get version "001".

Now modify your version. [ "cout" will tell you which version it created. ]

When you are ready to test, type: "cin source_name".

"cin" will find source with the highest version number, and copy it to a file with no version number. Alternatively, you can type: "cin -l source_file". This will make a symbolic link to your source, instead of copying it.

After testing, you will want to check the previous version back in. Type: "cin source_name version_number", where "version_number" is the previous version.

Here is a sample editing session...

tcl
- go to TCI directory

su $your_account
- change your UID

cout my_source
- check-out source for modification

CHECKING OUT NEW VERSION OF my_source.tcl...
- PREV : my_source.tcl
- NEW : my_source.tcl.001
/export/home/ess/TS/tcl/my_source.tcl.001

vi my_source.tcl.001
- make changes to source

exit
- back to ESS owner

cin my_source
- check-in modified version

ACTIVATING LAST VERSION OF u_TS0011.tcl...

mv "u_TS0011.tcl" u_TS0011.tcl.000

mv "u_TS0011.tcl.001" u_TS0011.tcl.000

mv /export/home/ess/TS/tcl/u_TS0011.tcl.001 u_TS0011.tcl.000

mv /export/home/ess/TS/tcl/u_TS0011.tcl.000 u_TS0011.tcl

cin: activated version 001 of "u_TS0011.tcl".

......TEST......

cin my_source 0
- check previous version back in

ACTIVATING VERSION 000 OF u_TS0011.tcl...

mv "u_TS0011.tcl" "u_TS0011.tcl.bck"

mv /export/home/ess/TS/tcl/u_TS0011.tcl.000 u_TS0011.tcl.000 BCK

mv /export/home/ess/TS/tcl/u_TS0011.tcl.000 u_TS0011.tcl.001

mv /export/home/ess/TS/tcl/u_TS0011.tcl.001 u_TS0011.tcl

cin: activated version 000 of "u_TS0011.tcl".

Advantage of this is that "ls -l" will show which versions are currently checked in. Disadvantage is that if someone copies a file with no version number to this directory, your source will be trashed !! Your choice.