Paper SAS086-2014

Integrating Your Corporate Scheduler with Platform Suite for SAS® or SAS® Grid Manager

Paul Northrop, SAS Australia

ABSTRACT

SAS solutions are tightly integrated with the scheduling capabilities provided by SAS Grid Manager and Platform Suite for SAS. Many organizations require that their corporate scheduler be used to control SAS processing within the enterprise. Historically this has been a laborious process, requiring duplication of job and flow information using manual forms and cumbersome change management.

This paper provides proven techniques and methods that enable tight integration between the corporate scheduler and SAS without the administrative overhead. Platform Suite for SAS can be used to create flows, which are then executed by the corporate scheduler. The business unit can tweak the flow without reference to the enterprise scheduling team.

The approach discussed is to use the corporate scheduler to:

- Trigger SAS flows and to respond to flow return codes.
- Restart a SAS flow that has exited due to error conditions.
- Enable and disable LSF queues, allowing jobs that have been queued up to run within a time window that is managed on external dependencies rather than time.

This paper describes how to configure your SAS environment to leverage the provided capabilities and provides real-world use cases to highlight the features and benefits of this approach.

The contents of this paper are of interest to SAS administrators and IT personnel responsible for enterprise scheduling.

Full code and deployment instructions are available.

INTRODUCTION

Scheduling of jobs and flows created by SAS is typically managed using either the operating system scheduling services (AT on UNIX and Windows) or using Platform Suite for SAS and SAS Grid Manager. SAS Business Intelligence (BI) reports can be scheduled using the SAS in-process scheduling services.

Many customers already have a corporate scheduling package that is required to be used for SAS processes. A very common scenario is where customers require SAS to leverage an existing BMC Control-M scheduler. The benefit to the customer is that they maintain a single point of control for all automated processing, regardless of whether it is based on SAS.

Customers effectively have two options regarding how the integration of SAS processing can be incorporated into their enterprise scheduling platform. These options are:

- Modify the corporate schedule to specify every SAS process and associated dependencies; or
- Treat SAS as a “black box” and call SAS developed flows from the corporate scheduler

While it is possible for traditional jobs developed using SAS® Data Integration Studio to be incorporated at the job level into the corporate scheduler, this approach can present issues regarding change management. Customers often have very strict change management procedures regarding alterations to their corporate schedules, which can affect the ability of the processes developed in SAS to rapidly adjust to changing business requirements. Furthermore, because SAS solutions such as SAS® Marketing Automation and SAS® Marketing Optimization are written to leverage the capabilities of Platform Suite for SAS, the automation of such processes are dependent on the use of Platform Suite for SAS at run time. Therefore, it is beneficial to take the second approach listed above. In this scenario, the external scheduler will trigger a SAS flow. The flow could contain a single job or thousands of jobs with complex dependencies between them. From the corporate scheduler perspective, it is seen as a single job – for example, Load Customer Data Mart, or Run Household Campaigns. The complexity of the jobs and their dependencies are hidden from the corporate scheduler, but remain controlled and managed through a combination of software (Platform Suite for SAS) and process (SAS administrators and schedule administrators).

In order to get the most from this paper, readers should have a good working knowledge of administering IBM Platform LSF and an understanding of UNIX shell scripting concepts.

PURPOSE

The purpose of this paper is to present a solution for integrating SAS batch processing controlled by Platform Suite for SAS with an external corporate scheduler. While this document describes and mentions BMC Control-M, there is no reason why the same approach cannot be used for other third-party scheduling software.

The two focus areas of this document are:

- How to integrate the execution of scheduling flows defined in SAS with the corporate scheduler (“flow level” integration); and
- How to integrate jobs that execute on a common LSF queue from applications such as SAS® Marketing Automation with the corporate scheduler (“queue level” integration).

Flow level integration provides the ability for the external scheduler to execute a set of related SAS jobs in the order defined in the flow definition (this is created using SAS® Management Console). The complexity of the contents of the flow is hidden from the corporate scheduler. Changes to the flow can be managed by the SAS schedule administrator, and changes to SAS flows do not require a change to the corporate schedule. This provides a much faster “time to market” for new or changed SAS jobs.

Queue level integration enables the execution of related or unrelated SAS jobs within an LSF queue to be managed by the corporate scheduler. This method is particularly useful for running SAS Marketing Automation campaigns, because each campaign creates its own flow when it is scheduled. Because there is a one-to-one mapping of a campaign to a flow, flow level integration for marketing campaigns is not appropriate, as there would have to be an entry in the corporate scheduler for each campaign flow. Queue level integration provides a mechanism to work around this constraint.

A key driver in providing the solution to this challenge is to ensure that any customization is restricted and that standard functionality is used wherever possible. This approach ensures that any associated risks of customization are minimized.

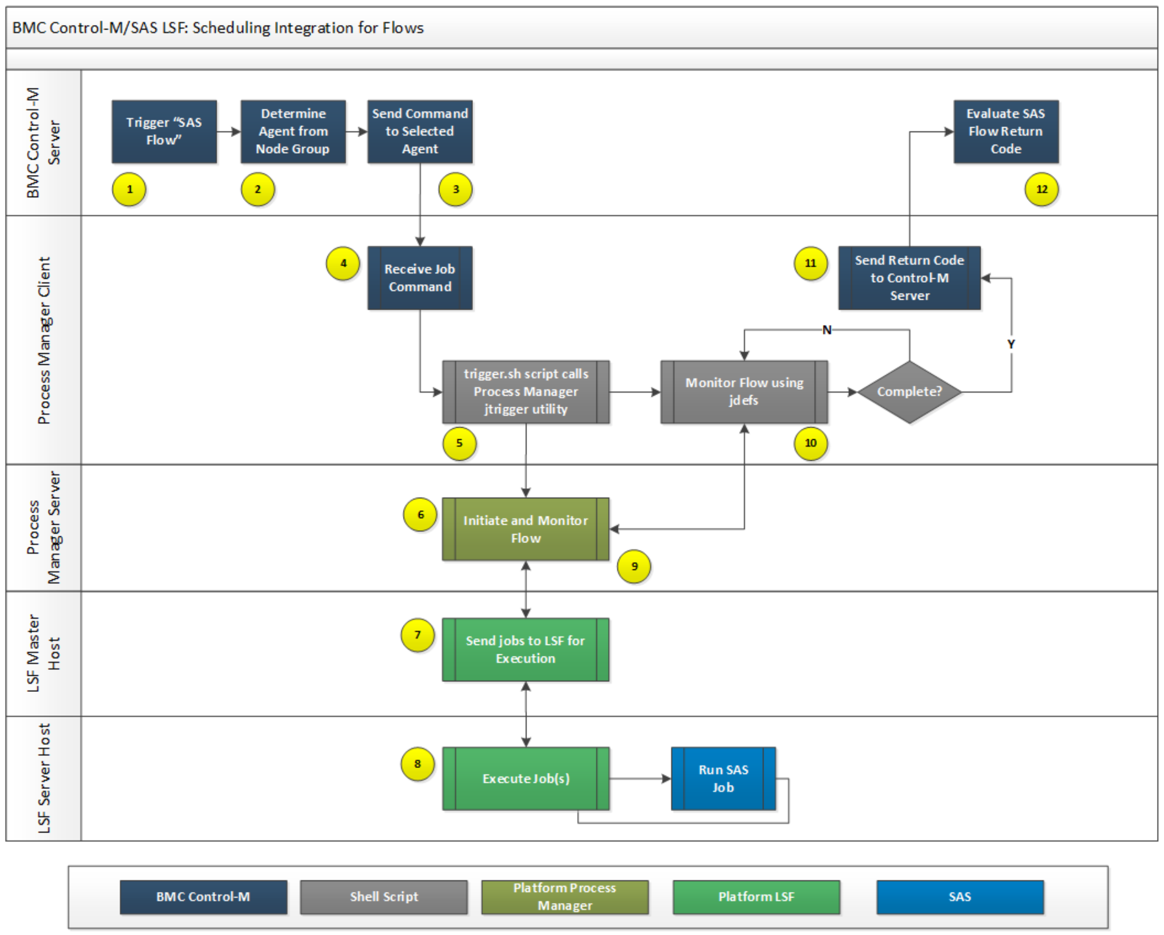
Both methods of integration are achieved through the deployment of a single shell script and some server-side configuration. The methods are described in the following sections.

FLOW LEVEL INTEGRATION

A flow is a grouping of one or more related jobs that are executed together. In a data warehouse or data mart, an example of a flow could be a group of jobs that extract data from a source, transform that data, and load it into a target database structure. From a SAS Marketing Automation perspective, every scheduled marketing campaign or communication resides in its own flow.

Flow level integration describes the scenario where one or more related SAS jobs are linked inside a scheduling flow, and this flow is controlled as a single entity from the corporate scheduler.

Figure 1 shows a flow diagram that describes how the integration occurs.



BMC Control-M/SAS LSF: Scheduling Integration for Flows
BMC Control-M Server
Process Manager Client
Process Manager Server
LSF Master Host
LSF Server Host
Trigger "SAS Flow"
Determine Agent from Node Group
Send Command to Selected Agent
Evaluate SAS Flow Return Code
Receive Job Command
Send Return Code to Control-M Server
trigger.sh script calls Process Manager jtrigger utility
Monitor Flow using jdefs
Complete?
N
Y
Initiate and Monitor Flow
Send jobs to LSF for Execution
Execute Job(s)
Run SAS Job
1
2
3
4
5
6
7
8
9
10
11
12
BMC Control-M
Shell Script
Platform Process Manager
Platform LSF
SAS

Figure 1: Flow Level Integration Process

There are five groups of computing resources in the diagram, where each relates to the logical component in a typical deployment. It is possible that all logical components can reside on a single server, but normally some of the components would exist on separate servers.

The diagram assumes that a SAS flow has been created and deployed from the SAS environment and that this flow has been added as a job within the Control-M master schedule.

Step 1

At the appropriate point in the running of a corporate schedule, the prerequisites for the execution of a SAS flow have been met. At this stage, Control-M has determined that the SAS flow must execute.

Step 2

Control-M reads its metadata to determine which Agent from the Node Groups should be told to execute the SAS flow.

Step 3

The command to run the flow is then sent by Control-M to the respective agent. This command consists of the full path to the custom trigger.sh script as well as the name of the flow to trigger. Internally, the script makes a call to the Process Manager client utility "jtrigger", which is a standard utility for triggering flows. Optionally, a wait period can be passed that determines how often the script will poll the Process Manager for status updates. See section 6.3 for example calls.

Step 4

The Control-M agent on the targeted server receives the command above to trigger the SAS flow, and submits it on the operating system using the specified account. In order for the flow to be successfully triggered, the correct authentication is needed for the account that spawns the script. This can be done in any of the following ways

- The current user account is defined as a Process Manager administrator account (JS_ADMINS).
- The current user account is defined as a Process Manager control administrator account (JS_CONTROL_ADMINS).
- The flow is owned by the current user account (this is the preferred option).

Step 5

The trigger shell script performs all the necessary functions to initiate a flow within the Platform Process Manager. If the flow cannot be initiated, the script exits with a return code of 10 and the process jumps to step 11.

The flow is triggered using the Platform Process Manager jtrigger command. If the script is called with the options required for re-running a previously exited flow, the Platform Process Manager jrerun command is used.

Step 6

The jtrigger (or jrerun) command sends the necessary signals to the Platform Process Manager server to start (or rerun) a given flow. The Process Manager server sets the state of the flow to Running and then sends all jobs without any dependencies to the LSF master batch daemon for execution.

Step 7 (the process in steps 7-9 are the standard process for executing a job flow with Platform Suite for SAS)

The LSF master batch daemon adds the new job to the run queue and will dispatch the job for execution by the most suitable candidate LSF host.

Step 8

The LSF slave batch daemon receives the job from the LSF master and executes the individual SAS job. The slave batch daemon monitors the execution of the job and sends information pertaining to the job back to the LSF master batch daemon on a regular basis.

Step 9

The LSF master batch daemon and the Process Manager server communicate on a regular basis. When a job ends, the Process Manager server will send any dependent jobs to the LSF master batch daemon if dependencies are met, or stop the submission of downstream jobs if an exit condition occurs. This communication proceeds until all jobs within a flow are completed (at which stage the flow status is updated to a “Done” state), or until jobs fail (at which stage the flow status is updated to an “Exit” state).

Step 10

While jobs and their dependencies are executed, the trigger.sh script communicates with the Process Manager server every 60 seconds (or as specified with the WAIT= option), using the Platform Process Manager jdefs command to determine the flow status.

If the flow status is determined to be “Done”, a zero return code is passed back to the Control-M agent on the Process Manager client. If the flow status is determined to be “Exit” or “Killed”, a return code of 12 is passed back to the Control-M agent on the Process Manager client.

Step 11

Once the Control-M agent receives the exit return code, it passes the value back to the Control-M server.

Step 12

The return code is interpreted by Control-M and downstream scheduling decisions and/or notifications take place.

QUEUE LEVEL INTEGRATION

Queue level integration describes the scenario where SAS flows are triggered by the Process Manager but the LSF queue or queues that the jobs are associated with are inactive. An inactive LSF queue will hold all jobs in that queue until the queue is activated. Activation of the queue in this scenario is triggered by the corporate scheduler. Once all jobs in the activated queue (and any dependent jobs) have completed, the queue is automatically inactivated again until the corporate scheduler sends another activation command.

There are two methods of invocation with the queue level integration. These are:

- Purge Queue: All jobs in the queue are removed, thus clearing out the queue in preparation for the next scheduled run.
- Activate Queue: All jobs that are in the queue are released by LSF to execute.

Figure 2 shows a flow diagram that describes how the queue level integration occurs.

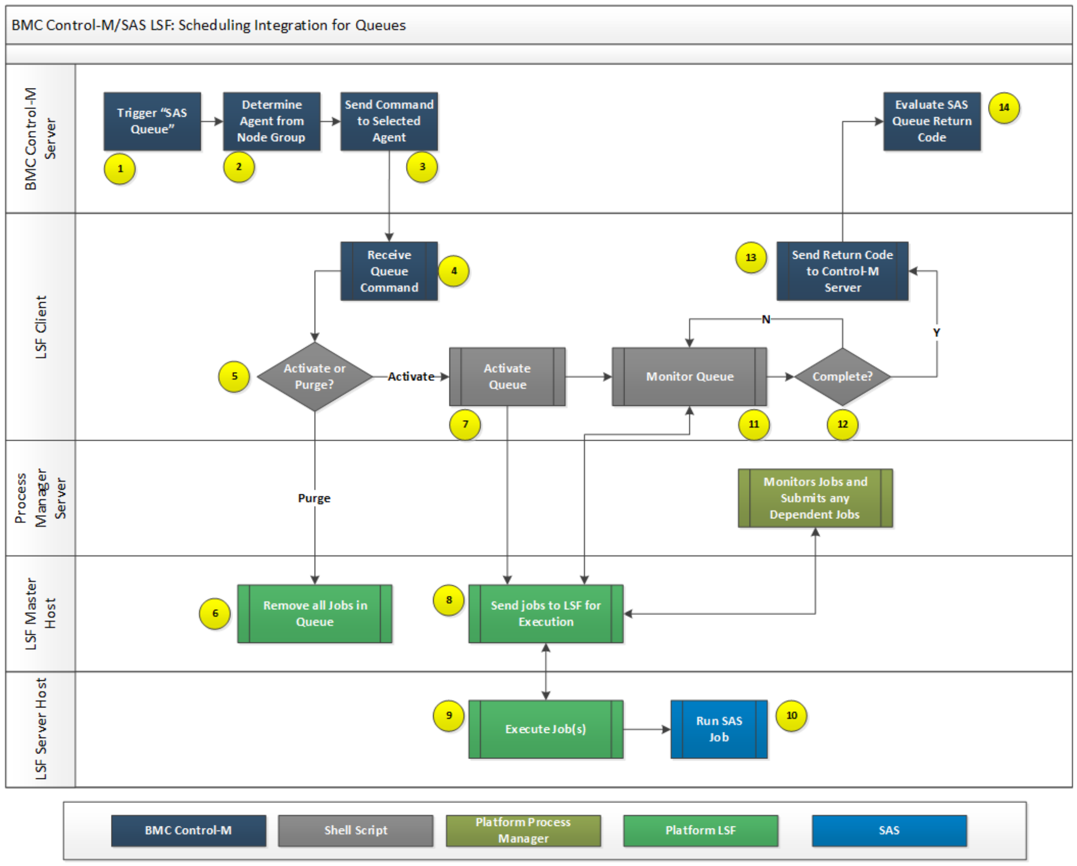


Figure 2: Queue Level Integration Process

Once again, there are five groups of computing resources in the diagram, where each relates to the logical component in a typical deployment. It is possible that all logical components can reside on a single server, but normally some of the components would exist on separate servers.

The diagram assumes that one or more SAS flows have been created and deployed from the SAS environment and that these flows have been triggered by Process Manager. All jobs in these flows execute on a single LSF queue. A control job within the Control-M master schedule has been created that initiates the activate queue procedure.

In a SAS Marketing Automation scenario, campaign analysts would schedule campaigns to run at a particular time on, for example, a daily basis. The selected time must be earlier than the anticipated time that the corporate scheduler will send the "go" command. This will ensure that the campaign jobs are waiting on the LSF queue prior to the "go" command being received.

Step 1

At the appropriate point in the running of a corporate schedule, the prerequisites for the activation of an LSF queue have been met. At this stage, Control-M has determined that the LSF queue should be activated so that the SAS jobs can execute.

Step 2

Control-M reads its metadata to determine which Agent from the Node Groups should be told to execute the queue control command.

Step 3

The queue control command is then sent by Control-M to the respective agent. This command consists of the full path to the trigger.sh script as well as the name of the queue to control. The command also includes the detail of whether the queue will be purged (in other words, all jobs will be removed from the queue) or activated (all jobs in the queue will be executed). Optionally, a wait period can be passed that determines how often the script will poll the Process Manager for status updates. See section 6.3 for example calls.

Step 4

The Control-M agent on the targeted server receives the command to activate (or purge) the LSF queue, and submits it on the operating system using the specified account. In order for the queue to be successfully managed, at least one of the following must be true regarding the account that is used to spawn the script:

- The current user account is defined as a Process Manager administrator account (JS_ADMINS).
- The user is an administrator of the LSF queue (this is the preferred option).

Step 5

The shell script performs all the necessary functions to initiate controlling the LSF queue.

If the command is to remove all jobs from the queue (PURGE option), then Step 6 is invoked.

If the command is to activate the queue (ACTIVATE option), then Step 7 is invoked.

Step 6

Using the LSF command line interface (CLI), all jobs are removed from the queue by using the command:

```
bkill –q <NAME OF QUEUE> 0
```

Once the queue has been purged, the return code (0 for success and 12 for failure) is set and Step 13 is invoked.

Step 7

Using the LSF command line interface (CLI), the queue is activated by using the command:

```
badmin qact <NAME OF QUEUE>
```

Step 8

LSF activates the queue and the LSF master batch daemon is able to dispatch the jobs for execution by the most suitable candidate LSF host.

Step 9

The LSF slave batch daemon receives the job from the LSF master and executes the provided command for the SAS job. The slave batch daemon monitors the execution of the job and sends information pertaining to the job back to the LSF master batch daemon on a regular basis.

Step 10

The SAS job is executed by the operating system.

Step 11

While jobs and their dependencies are executed, the trigger.sh script communicates with the LSF server every 60 seconds (or as specified with the WAIT= option), using the LSF bjobs command to determine whether all jobs on the queue have completed.

Step 12

If all jobs have completed, the trigger script inactivates the LSF queue, thus preventing any more jobs from running until another activation command is sent. Queue inactivation is performed using the command:

```
badmin qinact <NAME OF QUEUE>
```

The overall status of the jobs is used to generate a return code that is passed back to the Control-M agent.

A zero return code is generated if all jobs complete successfully.

A return code of 12 is passed back to the Control-M agent if one or more jobs fail.

Step 13

Once the Control-M agent receives the exit return code, it passes the value back to the Control-M server.

Step 14

The return code is interpreted by Control-M, and downstream scheduling decisions and notifications take place.

INSTALLATION AND CONFIGURATION

In order to use the techniques in this paper, you will need to perform some installation and configuration work.

Error! Reference source not found. shows which configuration steps are required depending on the type of integration you require:

Step to Follow	Required for Flow Level Integration?	Required for Queue Level Integration?
Process Manager Client Configuration	Y	N
LSF Configuration	N	Y
Install and configure trigger.sh script	Y	Y

Table 1. Configuration Steps by Integration Type

PROCESS MANAGER CLIENT CONFIGURATION

When the Platform Process Manager server and client software is installed, the parameter JS_LOGIN_REQUIRED is set to true by default. In this scenario, users who connect to the Process Manager server by using the standard user interfaces (Platform Flow Manager, Platform Calendar Editor, SAS Management Console Schedule Manager plug-in, and Process Manager Command Line Interfaces (CLI)) are required to enter valid credentials before a connection is made to the Process Manager server.

The successful use of a script to call Platform Process Manager CLI commands requires that no credentials are requested. While the Process Manager server configuration could be altered to not require credentials, this is not recommended from a security perspective. Therefore, the solution to the problem is to create a special batch mode copy of the Process Manager configuration files. It is recommended that a specific user account (termed a service account) be used for running flows. In order to enable Process Manager CLI commands to work without prompting for credentials, the following should be done:

- Copy the Process Manager conf directory to the service account's (svc_acct) home directory (for example, /home/<svc_acct>/pm_conf)
- Edit the js.conf file in the /home/<svc_acct>/pm_conf directory and change the value of JS_LOGIN_REQUIRED to false.

The actual location where the modified js.conf file is stored is used to set the JS_ENVDIR variable in the trigger.sh file.

The service account's home directory should be secured to prevent unauthorized access.

LSF CONFIGURATION

To support queue level integration, any queue that is controlled by the trigger.sh script must be modified to include the following directives in the definition of the queue in the lsb.queues file:

```
ADMINSTRATORS=<account(s) that run the trigger.sh script>
PRE_EXEC=custom_pre_exec.sh
POST_EXEC=custom_post_exec.sh
```

Once changes to this file have been made, the LSF administrator must force the LSF batch daemons to re-read the configuration files by using the command:

```
badmin reconfig
```

INSTALL AND CONFIGURE TRIGGER.SH SCRIPT

The supplied script (trigger.sh) is designed to work on any UNIX or Linux platform supported by SAS. It has been tested using Korn (ksh), Bourne (sh) and Bourne Again (bash) default shells, as well as on Windows platforms with the UNIX utilities installed.

The script should be installed into a directory that is accessible on the server that run the Process Manager daemon, and it must be executable by the service accounts under which LSF jobs will run.

Table 2 lists the environment variables used in the trigger.sh script that need to be verified or updated for your installation environment.

Environment Variable	Description	Default Value
PM_DIR	Installation directory for Platform Process Manager software	/var/opt/sas/sw/platform/pm
LSF_DIR	Installation directory for Platform LSF software	/var/opt/sas/sw/platform/lsf
JS_ENVDIR	Location of Platform Process Manager conf directory that has had js.conf modified with JS_LOGIN_REQUIRED=false	~/pm/conf
LOG_DIR	Directory where the script will write its log files. This location should be writable for all users calling this script.	${C4LOG}/ctm

Table 2: Environment Variables to Update in trigger.sh Script

SCRIPT SYNTAX AND EXAMPLE CALLS

The trigger.sh script has different parameters depending on whether it is being used for flow or queue level integration.

The syntax and example calls for each method are discussed in this section.

FLOW LEVEL INTEGRATION

Syntax for Flow Level Integration

The syntax for the trigger.sh script when using flow level integration is:

```
trigger.sh FLOW=<Flow Name>[ MODE=RUN|RERUN DEBUG=0|1 WAIT=60|<Number of Seconds> ]
```

The values in bold text indicate default values that apply if the parameter is not explicitly specified.

Note that each parameter is specified as a name-value pair separated with an equal sign (=), but without spaces. Additional parameters must be separated by at least one space.

Table 3 describes the use and valid values of each parameter.

Parameter Name	Description	Notes
FLOW	Indicates the name of the flow to trigger or re-run.	Mandatory for flow level integration. No default.
MODE	Either RUN or RERUN. If RUN is specified, the script will initiate a new instance of the flow. If RERUN is specified, the latest existing flow instance is restarted from where it had previously failed.	Optional for flow level integration. Default is RUN.
WAIT	Number of seconds that the script will wait between checking the status of the flow	Optional. Default is 60 (1 minute).

Parameter Name	Description	Notes
DEBUG	If set to 1, additional entries are written to the log file indicating that the flow was seen as running at each check. Using DEBUG=1 can result in large log files, especially if the WAIT period is short and the flow runs for a long time.	Optional. Default is 0 – Running statuses are not logged.

Table 3: Flow Level Parameters for the trigger.sh Script

Sample Calls for Flow Level Integration

The examples below are valid calls (assuming there is a flow called MyFlow that is either owned or controllable by the current user).

```
trigger.sh FLOW=MyFlow
```

This initializes a new instance of the flow called MyFlow. The status is checked every 60 seconds, and only start and stop messages are logged.

```
trigger.sh WAIT=10 FLOW=MyFlow
```

This initializes a new instance of the flow called MyFlow. The status is checked every 10 seconds, and only start and stop messages are logged.

```
trigger.sh  FLOW=MyFlow MODE=RERUN
```

This re-runs the flow called MyFlow if the last occurrence of the flow exited. If the latest occurrence of MyFlow did not exit, then the script fails validation and exits.

The following are examples of invalid calls:

```
trigger.sh MODE=RUN
```

No flow has been specified and the script exits with return code 10.

```
trigger.sh FLOW= MyFlow
```

There is a space after the = character. This is invalid and the script exits with return code 10.

QUEUE LEVEL INTEGRATION

Syntax for Queue Level Integration

The syntax for the trigger.sh script when using queue level integration is:

```
trigger.sh QUEUE=<LSF Queue Name>[ ACTION=RUN|RERUN DEBUG=0|1 WAIT=60|<Number of
Seconds> ]
```

The values in bold text indicate default values that apply if the parameter is not explicitly specified.

Note that each parameter is specified as a name-value pair separated with an equal sign (=), but without spaces. Additional parameters must be separated by at least one space.

Table 4 describes the use and valid values of each parameter.

Parameter Name	Description	Notes
QUEUE	Name of the queue to purge or activate.	Required for queue level integration. No default value. Must match an LSF queue name and is case sensitive.
ACTION	PURGE or ACTIVATE	Required for queue level integration. No default value and is case sensitive.
WAIT	Number of seconds that the script will wait between checking the status of the queue.	Optional. Default is 60 (1 minute).
DEBUG	If set to 1, additional entries are written to the log file indicating that the queue had active jobs at each check. Using DEBUG=1 can result in large log files, especially if the WAIT period is short and the flow runs for a long time.	Optional. Default is 0 – Running statuses are not logged.

Table 4: Queue Level Parameters for the trigger.sh Script

Sample Calls for Queue Level Integration

The examples below are valid calls (assuming there is a queue called CAMPAIGNS that is either owned or controllable by the current user).

```
   trigger.sh QUEUE=CAMPAIGNS ACTION=ACTIVATE
```
This activates the CAMPAIGNS LSF queue allowing jobs that are queued up to execute. The status is checked every 60 seconds, and only start and stop messages are logged.

```
   trigger.sh ACTION=PURGE QUEUE=normal
```
This removes all jobs from the normal LSF queue. Any running jobs are killed.

```
   trigger.sh QUEUE=CAMPAIGNS WAIT=30 DEBUG=1 ACTION=ACTIVATE
```
This activates the CAMPAIGNS LSF queue allowing jobs that are queued up to execute. The status is checked every 30 seconds, and all checks as well as start and stop messages are logged.

The following are examples of invalid calls:

```
   trigger.sh QUEUE=CAMPAIGNS
```
No ACTION parameter has been specified. The script exits with return code 10.

```
   trigger.sh QUEUE =normal ACTION=ACTIVATE
```
There is a space before the = character after the QUEUE keyword. This is invalid and the script exits with return code 10.

CONCLUSION

Integrating your corporate scheduler with Platform Suite for SAS or SAS Grid Manager allows you to leverage the standard capability in SAS applications to schedule jobs, and ensures that SAS jobs are run at the correct time based on enterprise dependencies. Furthermore, by using queue and flow control jobs in the corporate scheduler, users of the SAS applications are able to be more responsive to changes in business requirements to add and remove SAS jobs, without having adhere to potentially long operational lead times that normally apply when making changes to an enterprise-wide schedule.

The integration components offered in this paper can be deployed and tested in a matter of hours, by a person with a good understanding of Unix scripting and Platform Suite for SAS. This solution has been in use at a large customer in Australia for close on two years without issue and is therefore considered to be a tried and tested solution, ready for implementation at your site. However, neither SAS Institute Inc, nor the author provide any guarantee that this solution will work in your specific environment.

ACKNOWLEDGMENTS

The author would like to thank Bill Gibson for reviewing the contents of this paper and making suggestions to improve it.

RECOMMENDED READING

The following documentation is provided by SAS and is a valuable source of information for SAS administrators who need to understand or configure the Platform Suite for SAS.

- *Platform LSF Foundations* – provides an overview of the concepts and terminology of Platform LSF.
- Platform LSF Configuration Reference – provides detailed information about configuring Platform LSF.
- *Administering Platform Process Manager* – provides detailed information about configuring and managing Platform Process Manager.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Paul Northrop
SAS Institute Australia (Pty) Ltd
300 Burns Bay Road
Lane Cove
NSW 2066
Australia

E-mail: paul.northrop@sas.com

APPENDIX – SOURCE CODE

TRIGGER.SH SCRIPT SOURCE CODE

```
#! /bin/sh
# -------------------------------------------------------
# Purpose
# A) To allow Control-M to trigger SAS schedules via Platform Process Manager CLI
utilities
# OR
# B) to allow Control-M to activate a specific queue and to then run all pending jobs,
#    then inactivate the queue again
# -------------------------------------------------------

# -------------------------------------------------------
# Parameters for Purpose A:
# -------------------------------------------------------
# FLOW = Name of Flow to trigger/re-run.  This is CASE-SENSITIVE.
# WAIT = Wait Period (time to wait between checking status of flow, default = 60
seconds)
# MODE = RUN|RERUN.  RUN is default.
#        Use RERUN to restart a flow that has previously exited
# DEBUG = 0 or 1.  Default is 0.  If set to 1, messages are logged for running flows

# -------------------------------------------------------
# Parameters for Purpose B:
# -------------------------------------------------------
# QUEUE = Name of queue to activate.  This is CASE-SENSITIVE.
# ACTION = Either PURGE or ACTIVATE.
#          PURGE will remove all jobs in the queue.
#          ACTIVATE will allow pending jobs to start running.
# WAIT = Wait Period (time to wait between checking status of flow, default = 60
seconds)
# DEBUG = 0 or 1.  Default is 0.  If set to 1, messages are logged for running flows
# -------------------------------------------------------

# -------------------------------------------------------
# Initialise site specific environment
# -------------------------------------------------------
# Set PM_DIR to location of Process Manager
# -------------------------------------------------------
PM_DIR=/var/opt/sas/sw/platform/pm
. ${PM_DIR}/conf/profile.js

# -------------------------------------------------------
# Source LSF environment
# -------------------------------------------------------
LSF_DIR=/var/opt/sas/sw/platform/lsf
. ${LSF_DIR}/conf/profile.lsf

# -------------------------------------------------------
# Set JS_ENVDIR to point to location of js.conf that has
# JS_LOGIN_REQUIRED=false
# -------------------------------------------------------
JS_ENVDIR=~/pm/conf

# -------------------------------------------------------
# Set LOG_DIR to location where log files of this script will be created
# -------------------------------------------------------
#LOG_DIR=/var/opt/sas/conf/Lev3/SASCI/BatchServer/Logs
LOG_DIR=${C4LOG}/ctm
# -------------------------------------------------------
# Set defaults....
# -------------------------------------------------------
RC=99
FLOW=NULL
FLOWID=NULL
QUEUE=NULL
QUEUE_JOBS=0
DEP_JOB_WAIT=15
STATUS=INIT
WAIT=60
MODE=RUN
DEBUG=0
TMP_FILE=NULL
LOG_FILE=NULL
ACTION=NULL
LOG_DT=`date +%Y_%m_%d`
MAIN_LOG_FILE=$LOG_DIR/trigger_flow_${LOG_DT}.log


# -------------------------------------------------------
# Define script functions
# --------------------------------------------------------

function setLogHeader {
  LOG_HEADER=`date +%Y-%m-%d\ %H:%M:%S`
  LOG_HEADER="$LOG_HEADER : ${USER} : ${FLOW} : ${FLOWID} : ${STATUS} :"
  export LOG_HEADER
}


function get_flow_status {
  jdefs $FLOW | grep $FLOWID > $TMP_FILE
  STATUS=`awk -F '\\\\(|\\\\)' '// {print $2}' $TMP_FILE`
  setLogHeader
}


function wait_for_flow {
  IS_RUNNING=Y
  while [ "$IS_RUNNING" = "Y" ]
  do
    sleep $WAIT_PERIOD
    setLogHeader
    get_flow_status

    if [ "$STATUS" = "Killed" ]
      then
        RC=12
        IS_RUNNING=N
        echo ${LOG_HEADER} INFO: Flow has been killed. >> $MAIN_LOG_FILE
        return
    fi
    if [ "$STATUS" = "Done" ]
      then
        RC=0
        IS_RUNNING=N
        echo ${LOG_HEADER} INFO: Flow has completed successfully. >> $MAIN_LOG_FILE
        return
    fi

    if [ "$STATUS" = "Exit" ]
      then
        echo ${LOG_HEADER} ERROR: Flow has ended with WARNINGS OR ERRORS. Exiting
RC=12. >> $MAIN_LOG_FILE
        IS_RUNNING=N
        RC=12
        return
    else
      if [ $DEBUG -eq 1 ]
        then
          echo ${LOG_HEADER} INFO: Current status is $STATUS >> $MAIN_LOG_FILE
      fi
    fi
  done
}


function purge_queue {
  echo "${LOG_HEADER} INFO: Received request to PURGE the LSF queue called '$QUEUE'.
Purging queue." >> $MAIN_LOG_FILE

  bkill -q $QUEUE 0
  RC=$?
  if [ $RC -ne 0 -a $RC -ne 255 ]
    then
      echo "${LOG_HEADER} ERROR: Command 'bkill -q $QUEUE 0' exited with return code 
$RC." >> $MAIN_LOG_FILE
  else
      echo "${LOG_HEADER} INFO: LSF queue '$QUEUE' was purged successfully." >> 
$MAIN_LOG_FILE
  fi
  return
}


function get_qjob_count {

  bjobs -q ${QUEUE} -u all 2>/dev/null | grep -v JOBID > $QUEUE_JOB_FILE 2>/dev/null

  QUEUE_JOBS=`cat $QUEUE_JOB_FILE | wc -l`
  return
}

function remove_bjobs_file {
  if [ -e $QUEUE_JOB_FILE ]
    then
      rm $QUEUE_JOB_FILE
  fi
  return
}

function activate_queue {
  echo "${LOG_HEADER} INFO: Received activate queue ($QUEUE) command ..." >> 
$MAIN_LOG_FILE

  QUEUE_JOB_FILE=${LOG_DIR}/bjobs_$$.txt

  remove_bjobs_file

  get_qjob_count

  if [ $QUEUE_JOBS -eq 0 ]
    then
      echo "${LOG_HEADER} INFO: There are no jobs waiting to run in queue $QUEUE. 
Exiting - RC=12." >> $MAIN_LOG_FILE
      RC=12
      remove_bjobs_file
      exit $RC
  else
      echo "${LOG_HEADER} INFO: There are $QUEUE_JOBS jobs waiting to run in queue 
$QUEUE. Activating queue." >> $MAIN_LOG_FILE
      # Activate the queue and check return code...
      badmin qact $QUEUE
      RC=$?
      if [ $RC -ne 0 ]
        then
          echo "${LOG_HEADER} ERROR: Unable to activate queue ($QUEUE). Exiting - 
RC=12." >> $MAIN_LOG_FILE
          remove_bjobs_file
          RC=12
          exit $RC
      fi
  fi
  return
}
```

```
function monitor_queue {
  QUEUE_BUSY=Y
  while [ "$QUEUE_BUSY" = "Y" ]
    do
      sleep $WAIT_PERIOD
      setLogHeader
      get_qjob_count
      if [ $QUEUE_JOBS -eq 0 ]
        then
          # There are no jobs at the moment, but let's wait to allow Process Manager
to submit dependent jobs...
          sleep $DEP_JOB_WAIT
          get_qjob_count
      fi

      if [ $DEBUG -eq 1 ]
        then
          echo ${LOG_HEADER} INFO: There are now $QUEUE_JOBS jobs waiting to run in
queue $QUEUE >> $MAIN_LOG_FILE
      fi

      if [ $QUEUE_JOBS -eq 0 ]
        then
          QUEUE_BUSY=N
          echo "${LOG_HEADER} INFO: There are no more jobs waiting to run in queue
($QUEUE)." >> $MAIN_LOG_FILE
          RC=0
      else
        # monitor_queue
        QUEUE_BUSY=Y
      fi
  done

  return
}

function inactivate_queue {
  echo "${LOG_HEADER} INFO: Received InActivate $QUEUE queue command ..." >>
$MAIN_LOG_FILE

  # Activate the queue and check return code...
  badmin qinact $QUEUE
  RC=$?
  if [ $RC -ne 0 ]
    then
      echo "${LOG_HEADER} ERROR: Unable to InActivate $QUEUE queue. Exiting - RC=12."
>> $MAIN_LOG_FILE
      remove_bjobs_file
      RC=12
      exit $RC
  else
    echo "${LOG_HEADER} INFO: Successfully InActivated $QUEUE queue." >>
$MAIN_LOG_FILE
    remove_bjobs_file
  fi

  return
}

function verify_job_success {
  QUEUE_LOG_FILE=$LOG_DIR/${QUEUE}_${LOG_DT}.log

  EXIT_JOBS=0
  DONE_JOBS=0
  EXIT_JOBS=`cat ${QUEUE_LOG_FILE} | grep \|\ EXIT | wc -l`
  DONE_JOBS=`cat ${QUEUE_LOG_FILE} | grep \|\ DONE | wc -l`


  if [ $EXIT_JOBS -ne 0 ]
    then
      echo "${LOG_HEADER} INFO: $DONE_JOBS job(s) have run successfully during
execution." >> $MAIN_LOG_FILE
      echo "${LOG_HEADER} ERROR: $EXIT_JOBS job(s) have exited during execution." >>
$MAIN_LOG_FILE
      RC=12
  else
      echo "${LOG_HEADER} INFO: All $DONE_JOBS job(s) have run successfully during
execution." >> $MAIN_LOG_FILE
      RC=0
  fi
  echo "${LOG_HEADER} INFO: $QUEUE Job details are listed below" >> $MAIN_LOG_FILE
  cat $QUEUE_LOG_FILE >> $MAIN_LOG_FILE
  rm $QUEUE_LOG_FILE
return
}

function clean_up {

  if [ -e $LOG_FILE  ]
    then
        rm $LOG_FILE
  fi

  if [ -e $TMP_FILE ]
    then
        rm $TMP_FILE
  fi

  return
}

function close_queue {

  echo "${LOG_HEADER} INFO: Received Close $QUEUE queue command ..." >> $MAIN_LOG_FILE

  # Close the queue and check return code...
  badmin qclose $QUEUE
  RC=$?
  if [ $RC -ne 0 ]
    then
      echo "${LOG_HEADER} ERROR: Unable to Close $QUEUE queue. Exiting - RC=12." >>
$MAIN_LOG_FILE
      # remove_bjobs_file
      RC=12
      exit $RC
  else
    echo "${LOG_HEADER} INFO: Successfully Closed $QUEUE queue." >> $MAIN_LOG_FILE
   # remove_bjobs_file
  fi

  return
}
```

```
function open_queue {

  echo "${LOG_HEADER} INFO: Received Open $QUEUE queue command ..." >> $MAIN_LOG_FILE

  # Open the queue and check return code...
  badmin qopen $QUEUE
  RC=$?
  if [ $RC -ne 0 ]
    then
      echo "${LOG_HEADER} ERROR: Unable to Open $QUEUE queue. Exiting - RC=12." >>
$MAIN_LOG_FILE
      # remove_bjobs_file
      RC=12
      exit $RC
  else
    echo "${LOG_HEADER} INFO: Successfully Opened $QUEUE queue." >> $MAIN_LOG_FILE
   # remove_bjobs_file
  fi

  return
}


function get_flowid {
  TMP_FILE=${LOG_DIR}/_getflowid_$$.txt
  TMP_FILE1=${LOG_DIR}/_getflowid_$$_1.txt
  TMP_FILE2=${LOG_DIR}/_getflowid_$$_2.txt


  jdefs -u $USER ${FLOW} > $TMP_FILE
  tail -1 $TMP_FILE > $TMP_FILE1
  FLOWID=`awk -F '\ ' '{print $4}' $TMP_FILE1`
  if [ "" = "$FLOWID" ]
    then
      FLOWID=`awk -F '' '{print $1}' $TMP_FILE1`
  fi
  echo $FLOWID > $TMP_FILE1
  export FLOWID=`awk -F '(' '{print $1}' $TMP_FILE1`
  FLOWSTATUS=`awk -F '(' '{print $2}' $TMP_FILE1`
  echo $FLOWSTATUS> $TMP_FILE2
  FLOWSTATUS=`awk -F ')' '{print $1}' $TMP_FILE2`
  if [ "Exit" != "$FLOWSTATUS" ]
    then
      STATUS=ERROR_CANNOTRERUN
      setLogHeader
      echo "${LOG_HEADER} ERROR: Cannot Restart flow ID($FLOWID) - current state is
$FLOWSTATUS" >> $MAIN_LOG_FILE
      exit 10
  fi

  rm $TMP_FILE $TMP_FILE1 $TMP_FILE2
  return
}

# ########################################
# MAIN
# ########################################

setLogHeader

# -------------------------------------------------------
# Check Log directory can be written to, exit if false
# -------------------------------------------------------
```

```
if [ ! -w $LOG_DIR ]
  then
    echo "ERROR: Cannot write to log directory (${LOG_DIR})"
    exit 10
fi

# ------------------------------------------------------
# Process Command Line Parameters
# ------------------------------------------------------
for arg in "$@"
  do
    export $arg
  done



# ------------------------------------------------------
# If MAIN_LOG_FILE does not exist, create it and write header records
# ------------------------------------------------------
if [ ! -e $MAIN_LOG_FILE ]
  then
    echo
"*************************************************************************" >>
$MAIN_LOG_FILE
    echo "* trigger_flow log file for flows triggered on `date +%Y-%m-%d`
" >> $MAIN_LOG_FILE
    echo "* Created on `date +%Y-%m-%d\ %H:%M:%S`
" >> $MAIN_LOG_FILE
    echo
"*************************************************************************" >>
$MAIN_LOG_FILE
    echo "*Timestamp          : User ID : Flow Name    : Flow ID : Status : Message
Type : Message" >> $MAIN_LOG_FILE
    echo
"*************************************************************************" >>
$MAIN_LOG_FILE
    chmod g+w $MAIN_LOG_FILE
fi

function validate_parameters {
# ------------------------------------------------------
# Validate parameters...
# ------------------------------------------------------

if [ "$QUEUE" != "NULL" ]
  then
   if [ "$ACTION" = "ACTIVATE" -o "$ACTION" = "PURGE" ]
      then
        return
   fi
fi


if [ "$QUEUE" != "NULL" ]
  then
    if [ "$ACTION" != "ACTIVATE" -a "$ACTION" != "PURGE" ]
      then
        # No valid action for the QUEUE
        STATUS=ERROR
        setLogHeader
        echo ${LOG_HEADER} ERROR: ACTION value of $ACTION is invalid.  Specify
ACTION=ACTIVATE or ACTION=PURGE. Exiting RC=10 >> $MAIN_LOG_FILE
        exit 10
    fi
fi

if [ "$FLOW" = "NULL" ]
  then
    # No flow name or queue provided, so exit
    STATUS=ERROR
    setLogHeader
    echo ${LOG_HEADER} ERROR: No FLOW= parameter has been passed. Exiting RC=10 >>
$MAIN_LOG_FILE
    exit 10
fi


if [ "$FLOW" = "NULL" -a "$MODE" = "RERUN" ]
  then
        echo ${LOG_HEADER} ERROR: No FLOW= parameter has been passed for MODE=${MODE}.
Exiting RC=10 >> $MAIN_LOG_FILE
        exit 10
  else
    setLogHeader
    echo ${LOG_HEADER} INFO: Script initialised with command $0 $* >> $MAIN_LOG_FILE
fi


LOG_FILE=$LOG_DIR/${FLOW}.log
if [ -e $LOG_FILE ]
  then
    # Delete existing log file
    rm $LOG_FILE
    RC=$?
    if [ $RC -ne 0 ]
      then
        STATUS=ACCESS_ERROR
        setLogHeader
        echo ${LOG_HEADER} ACCESS: User does not have write permission to ${LOG_FILE}.
Exiting RC=10 >> $MAIN_LOG_FILE
        exit 10
    fi
fi
}

validate_parameters

# -------------------------------------------------------
# Set the WAIT_PERIOD if not specified on command line...
# -------------------------------------------------------
if [ "x$WAIT" = "x" ]
  then
    WAIT_PERIOD=60
else
    WAIT_PERIOD=$WAIT
fi

# -------------------------------------------------------
# Handle queue activation...
# -------------------------------------------------------

if [ "$QUEUE" != "NULL" -a "$ACTION" = "ACTIVATE" ]
  then
   activate_queue
   monitor_queue
   inactivate_queue
   verify_job_success
   exit $RC
fi

# -------------------------------------------------------
# Handle queue purge...
# -------------------------------------------------------

if [ "$QUEUE" != "NULL" -a "$ACTION" = "PURGE" ]
  then
   purge_queue
   exit $RC
fi

# -------------------------------------------------------
# Trigger or rerun the flow...
# -------------------------------------------------------
if [ "$MODE" = "RUN" ]
  then
    jtrigger $FLOW > $LOG_FILE 2>/dev/null
    RC=$?

    if [ $RC -ne 0 ]
      then
        STATUS=ERROR_START
        setLogHeader
        echo ${LOG_HEADER} ERROR: Flow ${FLOW} cannot be triggered. jtrigger return
code is $RC. Exiting with RC=10 >> $MAIN_LOG_FILE
        clean_up
        exit 10
    fi
    FLOWID=`awk -F '<|>' '/:/ {print $4}' $LOG_FILE`

    if [ "x$FLOWID" = "x" ]
      then
        STATUS=ERROR_INIT
        setLogHeader
        echo ${LOG_HEADER} ERROR: Flow ID cannot be determined. Exiting with RC=10 >>
$MAIN_LOG_FILE
        clean_up
        exit 10
    fi

    STATUS=START
    setLogHeader
    echo ${LOG_HEADER} INFO: LSF Flow ID is $FLOWID... >> $MAIN_LOG_FILE

else
  if [ "$MODE" = "RERUN" ]
    then
      STATUS=GET_FLOWID
      setLogHeader
      get_flowid
      jrerun $FLOWID > $LOG_FILE 2>/dev/null
      RC=$?
      if [ $RC -ne 0 ]
        then
          STATUS=ERROR_START
          setLogHeader
          echo ${LOG_HEADER} ERROR: Flow ID ${FLOWID} cannot be rerun. jrerun return
code is $RC. Exiting with RC=10 >> $MAIN_LOG_FILE
          clean_up
          exit 10
      fi

  else
      RC=10
  fi
fi

export TMP_FILE=${LOG_DIR}/_tmp_${FLOWID}.txt

wait_for_flow

# ############################################################
# Clean Up...
# ############################################################
clean_up

exit $RC
```

CUSTOM_PRE_EXEC SCRIPT SOURCE CODE

```
!# /bin/sh
# -------------------------------------------------------
# Purpose
# This script is called as a PRE_EXEC script for all jobs in the  LSF queue
# It writes out a log file that contains the start times of jobs
#
# The log produced by this script is used by the POST_EXEC script
(campaigns_post_exec.sh)
# The LSF queue must be modified to call this as a POST_EXEC script
# -------------------------------------------------------
# Set LOG_DIR to an appropriate location on your server
LOG_DIR=${C4LOG}/ctm
LOG_FILE=${LOG_DIR}/${LSB_JOBID}_pre.log
NOW=`date +%Y-%m-%d\ %H:%M:%S`
echo $NOW > $LOG_FILE
exit 0
```

CUSTOM_POST_EXEC SCRIPT SOURCE CODE

```
!# /bin/sh
# -------------------------------------------------------
# Purpose
# This script is called as a POST_EXEC script for all jobs in the CAMPAIGNS LSF queue
# It writes out a log file that describes the start and end times of jobs as well as
return codes and status
#
# This script depends on the log file (<JOBID>_pre.log) created by the PRE_EXEC script
(campaigns_pre_exec.sh)
# The LSF queue must be modified to call this as a POST_EXEC script
# -------------------------------------------------------


QUEUE=$LSB_QUEUE
# Set LOG_DIR to an appropriate location on your server
LOG_DIR=${C4LOG}/ctm
LOG_DT=`date +%Y_%m_%d`


QUEUE_LOG_FILE=${LOG_DIR}/${QUEUE}_${LOG_DT}.log
```

```
# Read Job's start time from the pre log file, then remove the pre log file as we
don't want them lying around
START=`cat ${LOG_DIR}/${LSB_JOBID}_pre.log`
rm ${LOG_DIR}/${LSB_JOBID}_pre.log


NOW=`date +%Y-%m-%d\ %H:%M:%S`

if [ ! -e $QUEUE_LOG_FILE ]
  then
    # QUEUE_LOG file does not exist, so create it.
    echo
"******************************************************************************" >>
$QUEUE_LOG_FILE
    echo "* Execution log file for jobs triggered on `date +%Y-%m-%d`
" >> $QUEUE_LOG_FILE
    echo "* Created on `date +%Y-%m-%d\ %H:%M:%S`
" >> $QUEUE_LOG_FILE
    echo
"******************************************************************************" >>
$QUEUE_LOG_FILE
    echo "*Job ID  | Job Name       | Start Time | End Time | Return Code | Status
" >> $QUEUE_LOG_FILE
    echo
"******************************************************************************" >>
$QUEUE_LOG_FILE
    chmod g+w $QUEUE_LOG_FILE
fi


function check_rc {
  # Return codes should be between 0 and 255.
  # sometimes the return code is bigger than 255, so needs to be "fixed"
  if [ $LSB_JOBEXIT_STAT -gt 255 ]
    then
      LSB_JOBEXIT_STAT=`expr $LSB_JOBEXIT_STAT - 255`
      if [ $LSB_JOBEXIT_STAT -gt 255 ]
        then
          check_rc
      fi
  fi
  return
}
check_rc
if [ $LSB_JOBEXIT_STAT -eq 0 ]
  then
    STATUS=DONE
else
    STATUS=EXIT
fi
echo "${LSB_JOBID} | $LSB_JOBNAME | $START | $NOW | $LSB_JOBEXIT_STAT | $STATUS" >>
$QUEUE_LOG_FILE
exit 0
```