Chapter Twenty Four

THE NIGHTLY PROCESS

Introduction

The regular execution of the SuRPAS application at the client site or as a service to the client is referred to as the Client Nightly Process. The majority of the activities that make up this nightly process are executed in the same manner each night. The sequence changes when special events need to be run and these events may be specific to the calendar (e.g. month-end) or to the client. When a problem occurs, a support group is responsible for troubleshooting the error and completing the nightly processing.

In this chapter we will learn about:

- The SuRPAS Nightly Process
- Standard and Non-standard Abends
- Problem Research and Troubleshooting
- The Trace Facility

SuRPAS Nightly Process

Overview

The SuRPAS nightly process, also referred to as Client Nightly Processing, is a series of command procedures, each representing an event or series of events, that are submitted (in OpenVMS Batch Mode) and executed in a specific order. Additional, calendar-specific or task-specific events can be added to the processing manually on any given day.

Client Nightly Process

The client nightly processing activity is supported with one of three types of service. The client can run all nightly processing themselves, SuRPAS employees can run all night processing on behalf of the client, or a customized hybrid of the first two is accomplished. In any o fthe above choices a set of jobs is run at the end of the day that is called the Nightly Process. The main job is Jobstream and this event prices and posts the trades for the day.

Jobstream is the event that is run at the end of each day after the fund prices are received by the client and entered into SuRPAS. It is also known as Jobprocess. It is run under the control of a command procedure named JOBDAILY.COM found in FAL$PRGLIB. This is a large command procedure, but a hardcopy of this file (almost 70 pages when printed) should be in hand when examining the nightly process. A number of tasks are accomplished during the execution of the Jobstream. Although not all of these tasks are discussed in the following sections these tasks include:

- A backup of core SuRPAS files - uses OpenVMS BACKUP utility
- **Batch control** **- tags the transaction batches**
- Trade pricing - figures out how much you can redeem & how much your purchases will cost
- **Trade posting**
- Fee processing - all fees are calculated
- Opening/Closing balance processing
- **Ledger updates** **- sub-account (fund-specific) processing**
- Data aging - back-end loads, etc.
- **Data extracts** **- files sent to other systems**
- Reporting and statements
- **Cycling of entry (or online) dates** **- production dates**
- Update of Journal record pointers - all transactions are set up as journal records

File Backup

The code snippet on the next page (Example 24-1) shows the use of the OpenVMS BACKUP utility to backup each of the SuRPAS database directories. Notice that these backups are written to tape.

```
$       BACKUP/LOG/BLOCK=32256/NOREWI/'IGNORE_LOCK -
                'tape_compac'   FAL$MANAGER:*.*;*  FAL_TAPE:MANAGER.BCK
$ !
$ DATA_BASE:
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATACT:*.*;*   FAL_TAPE:DATACT.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATARC:*.*;*   FAL_TAPE:DATARC.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATBAT:*.*;*   FAL_TAPE:DATBAT.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATEXP:*.*;*   FAL_TAPE:DATEXP.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATINS:*.*;*   FAL_TAPE:DATINS.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATJNL:*.*;*   FAL_TAPE:DATJNL.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATLBL:*.*;*   FAL_TAPE:DATLBL.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATLED:*.*;*   FAL_TAPE:DATLED.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATMAS:*.*;*   FAL_TAPE:DATMAS.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATMIS:*.*;*   FAL_TAPE:DATMIS.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATNET:*.*;*   FAL_TAPE:DATNET.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATSAL:*.*;*   FAL_TAPE:DATSAL.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATSCH:*.*;*   FAL_TAPE:DATSCH.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATSFB:*.*;*   FAL_TAPE:DATSFB.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATSIB:*.*;*   FAL_TAPE:DATSIB.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATSUB:*.*;*   FAL_TAPE:DATSUB.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATTMP:*.*;*   FAL_TAPE:DATTMP.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATTRN:*.*;*   FAL_TAPE:DATTRN.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI -
        'tape_compac' FAL$DATWIR:*.*;*   FAL_TAPE:DATWIR.BCK
$       BACKUP/LOG/BLOCK=32256/NOREWI/'IGNORE_LOCK -
                'tape_compac' FAL$DSKDAT:*.*;*   FAL_TAPE:DSKDAT.BCK
$ !
$       TIMESTAMP JOBDAILY,,...END BACKUP
$ !
$       DISMOUNT FAL_TAPE
$       DEALLOCATE FAL_TAPE
```

Example 24-1 - JOBDAILY Database File Backup

Ledger Updates and Opening Balances

This snippet of code shows the sub-account (fund-specific) processing in JOBDAILY.COM. Notice that these tasks are built as parallel executing threads. Additonal threads support Ledger updates later in JOBDAILY as seen in the snippet of code on the next page (Example 24-3).

```
$ JOBQMAKE
$ DECK
T01$ !=============================================================================
T01$ ! Thread 01 : Ledger Updates & AM Div Stuff
T01$ !=============================================================================
T01$ !  Post Ledger & CDSC Lot Updates from Wire-orders
T01$ !
T01$    IF F$SEARCH("FAL$DATWIR:LTWFAL.DAT") .EQS. "" THEN GOTO SKIP_LTW
T01$ !
T01$       TIMESTAMP JOBDAILY,JXPOSLTW,...POST LTWS FROM WIRE ORDERS
T01$       RUN FAL$PRGLIB:JXPOSLTW.EXE
T01$       PURGE FAL$DATWIR:LTWFAL.DAT
T01$       RENAME FAL$DATWIR:LTWFAL.DAT LTWFAL.OLD
T01$       PURGE FAL$DATWIR:LTWFAL.OLD
T01$ SKIP_LTW:
T01$ !=============================================================================
T01$ !  Perform Daily AM Dividend Accruals/Payments
T01$ !
T01$    TIMESTAMP JOBDAILY,JXAMDIVS,...DAILY AM DIV ACC
T01$    RUN FAL$PRGLIB:JXAMDIVS.EXE
T01$ !=============================================================================
T01$ !  Opening Balance the Unaged Dividend Share File
T01$ !
T01$    TIMESTAMP JOBDAILY,JXBALDIVO,...OPN BAL DIV FIL
T01$    RUN FAL$PRGLIB:JXBALDIVO.EXE
T01$ !=============================================================================
T01$ !  Remove unaged dividend share items (must be after JXBALDIVO & JXAMDIVS;
T01$ !                                                      before JXPASS3)
T01$    TIMESTAMP JOBDAILY,JXRMVDIV,...UNAGED DIV SHRS
T01$    RUN FAL$PRGLIB:JXRMVDIV.EXE
T01$ !
$ EOD
$ JOBQSUBM 1
$ ! ===========================================================================
$ JOBQMAKE
$ DECK
T02$ !=============================================================================
T02$ ! Thread 02 : Opening Balances - Opening Balance the Unsettled Share File
T02$ !=============================================================================
T02$ !
T02$    TIMESTAMP JOBDAILY,JXBALSETO,...OPN BAL SET FIL
T02$    RUN FAL$PRGLIB:JXBALSETO.EXE
$ EOD
$ JOBQSUBM 2
```

Example 24-2 - JOBDAILY Ledger and Balance Updates

```
$ JOBQMAKE
$ DECK
T11$ ! ============================================================================
T11$ ! Thread 11 : Ledger updates & balances
T11$ ! ============================================================================
T11$ !  Perform Daily PM Dividend Accruals/Payments
T11$ !
T11$    TIMESTAMP JOBDAILY,JXPMDIVS,...DAILY PM DIV ACC!
T11$    RUN FAL$PRGLIB:JXPMDIVS.EXE
T11$ !
T11$ ! ============================================================================
T11$ !  Closing trial-balance the funds
T11$ !
T11$    TIMESTAMP JOBDAILY,JXBALFND,...BAL FUND (F)
T11$    RUN FAL$PRGLIB:JXBALFND.EXE
T11$ !
T11$ ! ============================================================================
T11$ !  Closing trial-balance the funds - version 2 (no register)
T11$ !
T11$    TIMESTAMP JOBDAILY,JXBLNFND,...BAL FUND (S)
T11$    FAL_CMDLINE = " "
T11$    RUN FAL$PRGLIB:JXBLNFND.EXE
T11$ !
$ EOD
$ JOBQSUBM 11
```

Example 24-3 - JOBDAILY Additional Ledger Update Thread (#11)

Closing Balance Processing

This code snippet (Example 24-4, below) shows an example of one of the many processes (executable images or programs) executed to handle the processing of daily closing balances.

```
$ !============================================================================
$ !  Closing Balance the Transaction Capture System
$ !
$       TIMESTAMP JOBDAILY,JXBCSCTR,...BATCH CONTROL
$ !
$       RUN FAL$PRGLIB:JXBCSCTR.EXE
$ !
$ !============================================================================
$ !  Report the Transaction Capture System Closing Control Numbers by Batch
$ !
$       TIMESTAMP JOBDAILY,JXBATCTR,...BATCH CONTROL
$ !
$       RUN FAL$PRGLIB:JXBATCTR.EXE
```

Example 24-4 - JOBDAILY Closing Balance Processing

Reporting

Two samples of report processing are shown in the following code snippets.

```
$ !
$ ! ============================================================================
$ !  Produce Daily Activity Detail Report  (PR3 sorted by ACT/FUND/SER)
$ !
$       TIMESTAMP JOBDAILY,JXREPDAR,...REPORT DAR
$ !
$       RUN FAL$PRGLIB:JXREPDAR.EXE
$ ! ============================================================================
$ !  Produce the Redemptions Warning Report if required
$ !
$       TIMESTAMP JOBDAILY,RXREDWRN,...REDEMPTION WARNING
$ !
$       RUN FAL$PRGLIB:RXREDWRN.EXE
$ !
$ ! ============================================================================
```

Example 24-5 - JOBDAILY Activity and Redemption Report Processing

```
$ ! ============================================================================
$ !  GENERATE 1% AND 5% SHAREHOLDERS REPORT  (needs LED balances in SUM)
$ !
$       JOBQSYNC 10
$       JOBQSYNC 11
$       JOBQSYNC 12
$
$       OPEN/READ/SHARE PRMFIL FAL$DATINS:PARAMS.DAT
$       READ/KEY="INSHEADERA03" PRMFIL RECORD
$       CLOSE PRMFIL
$       T14_FLG = F$EXTRACT(238,1,RECORD)    ! BYTE 239
$ !
$       IF (T14_FLG .NES. "Y") THEN GOTO SKIP_T14
$ !
$ JOBQMAKE
$ DECK
T14$ ! ============================================================================
T14$ ! Thread 14 : 1 and 5% reports
T14$ ! ============================================================================
T14$ !
T14$    TIMESTAMP JOBDAILY,JXSHHLD15,...1% AND 5% REP
T14$ !
T14$    RUN FAL$PRGLIB:JXSHHLD15.EXE
T14$ !
T14$    PU FAL$EXTRACT:SHHLD15.DAT
T14$ !
T14$ ! ============================================================================
$ EOD
$ JOBQSUBM 14
```

Example 24-6 - JOBDAILY 1% and 5% Shareholder Report Processing - Thread #14

Statement Processing

Statement processing is shown in the following code snippet (Example 24-7).

```
$ BEGSTMB:
$ IF (FAL_SITE .NES. “GTG” .AND. FAL_SITE .NES. “PSQ”) THEN $ GOTO ENDSTMB
$       TIMESTAMP JOBDAILY,SXSRFDSF,...MOVE STM REQ
$       RUN FAL$PRGLIB:SXSRFDSF.EXE
$ !
$ !  Check to see if prior year requests are set.  If not, jump over it.
$       OPEN/READ/SHARE PRMFIL FAL$DATINS:PARAMS.DAT
$       READ/KEY=”STMTFEEDERA1"/ERROR=AFTERPY3 PRMFIL RECORD
$        CLOSE PRMFIL
$       PRIYREQ = F$EXTRACT(51,1,RECORD)
$        IF PRIYREQ .NES. “Y” THEN $ GOTO AFTERPY3
$ !
$ !  Sort prior year requests by account and fund and run report
$        FILESIZE = 0
$       IF F$SEARCH(“FAL$SCRATCH:PRIORYREQ.DAT”) .NES. “” THEN -
       FILESIZE = F$FILE_ATTRIBUTES(“FAL$SCRATCH:PRIORYREQ.DAT”,”EOF”)
$       TIMESTAMP JOBDAILY,SORT,...SORT PRIOR YR REQUESTS
$        IF FILESIZE .GT. 0 THEN -
$       SORT/KEY = (CHARACTER, SIZE:20, POSITION:1,  ASCENDING) -
              FAL$SCRATCH:PRIORYREQ.DAT FAL$SCRATCH:PRIORYREQ.SRT
$       IF FILESIZE .GT. 0 THEN RUN FAL$PRGLIB:PRIORYREQ.EXE
$ AFTERPY3:
$       TIMESTAMP JOBDAILY,SXSRFDSF,...MOVE PR3 TO DSF
$       RUN FAL$PRGLIB:JXEXTSTM.EXE
$       CREATE/FDL=FAL$FILES:SRFFAL.FDL
$ ENDSTMB:
$ ! ============================================================================
$ !  Produce Daily Transaction Advices - Single Trade Variety
$       TIMESTAMP JOBDAILY,JXADVFMT,...ADVICES 1
$       RUN FAL$PRGLIB:JXADVFMT.EXE
$ ! ============================================================================
$ !  Produce Daily Transaction Advices - Multi-Trade Variety
$ !  Create PRW file to receive items
$       CREATE/FDL=FAL$FILES:PRWFAL.FDL
$ !
$ !  Extract Advice-Reportable transactions
$       TIMESTAMP JOBDAILY,JXEXTADV,...EXT ADVICES
$       RUN FAL$PRGLIB:JXEXTADV.EXE
$ !
$ !  Sort by Account, Fund, Sequecne
$        FILESIZE = 0
$       IF F$SEARCH(“FAL$DATTMP:PRWFAL.DAT”) .NES. “” THEN -
       FILESIZE = F$FILE_ATTRIBUTES(“FAL$DATTMP:PRWFAL.DAT”,”EOF”)
$       TIMESTAMP JOBDAILY,SORT,...SORT PRW FOR ADV
$        IF FILESIZE .GT. 0 THEN SORT -
               /KEY=(CHARACTER, SIZE:14,  POSITION:6,    ASCENDING) -
               /KEY=(CHARACTER, SIZE:6,   POSITION:33,   ASCENDING) -
               /KEY=(CHARACTER, SIZE:6,   POSITION:416,  ASCENDING) -
               FAL$DATTMP:PRWFAL.DAT FAL$DATTMP:PRWFAL.DAT
$ !
$ !  Produce the trade advices
$       TIMESTAMP JOBDAILY,JXADMFMT,...ADVICES 2
$       RUN FAL$PRGLIB:JXADMFMT.EXE
```

Example 24-7 - JOBDAILY Statement Processing

Monthend

The Monthend event is run on a monthly basis when all the processing for the month is complete. This event will close out information for the month and create a copy of the files in a month end environment for reporting needs throughout the month. The Monthend event is run under the control of the JOBMONEND.COM command procedure found in FAL$PRGLIB. There are a number of activities included in the Monthend event. Some of these tasks are discussed in the following sections. These tasks include:

- A backup of SuRPAS data files - all directories are PURGEd and then backed up using the OpenVMS BACKUP utility
- Initialization of Ledger fields and the setup of period begin dates
- 12B1 processing - fees & commissions for brokers, reps, etc.
- Date aging (12B1) / rolling of buckets
- Data extracts
- Period reporting and statement preparation
- Updating of journal record pointers

The Monthend event is run after all the processing for the month has been completed. It is run after the last Jobprocess for the month. The Monthend event can be physically run on the first business day of the month, with the month end dates set to the prior month. The Monthend event must be run before trades can be entered for the next month (buckets are zeroed).

Monthend Cleanup and New Month Initialization

These two code snippets show the purging of database file directories and a small portion of the new month initialization activities. Although tape backup is not shown, code similar to that found in the JOBDAILY code snippets is also found in JOBMONEND.

```
$ !  Cleanup the directories
$       TIMESTAMP MONTHEND,PURGE,...CLEANUP DIR
$ !
$       PURGE/LOG  FAL$DATACT:*.*
$       PURGE/LOG  FAL$DATARC:*.*
$       PURGE/LOG  FAL$DATBAT:*.*
$       PURGE/LOG  FAL$DATEXP:*.*
$       PURGE/LOG  FAL$DATINS:*.*
$       PURGE/LOG  FAL$DATJNL:*.*
$       PURGE/LOG  FAL$DATLBL:*.*
$       PURGE/LOG  FAL$DATLED:*.*
$       PURGE/LOG  FAL$DATMAS:*.*
$       PURGE/LOG  FAL$DATMIS:*.*
$       PURGE/LOG  FAL$DATNET:*.*
$       PURGE/LOG  FAL$DATSAL:*.*
$       PURGE/LOG  FAL$DATSCH:*.*
$       PURGE/LOG  FAL$DATSFB:*.*
$       PURGE/LOG  FAL$DATSIB:*.*
$       PURGE/LOG  FAL$DATSUB:*.*
$       PURGE/LOG  FAL$DATTMP:*.*
$       PURGE/LOG  FAL$DATTRN:*.*
$       PURGE/LOG  FAL$DATWIR:*.*
$       PURGE/LOG  FAL$DSKDAT:*.*
```

Example 24-8 - JOBMONEND Database Directory PURGing

```
$ BEGPROCESS:
$ !
$ ! ============================================================================
$ !  Produce month-end dealer fund participation report
$ !
$       TIMESTAMP MONTHEND,RXDLRFND,...DLR RPT
$       RUN FAL$PRGLIB:RXDLRFND
$ ! ============================================================================
$ !  Tag the journal record pointers
$ !
$       TIMESTAMP MONTHEND,MXSETJNL,...SET JNL PTRS
$       RUN FAL$PRGLIB:MXSETJNL
$ ! ============================================================================
$ !  Set the date for the annual purge event
$ !
$       TIMESTAMP MONTHEND,MXSETPUR,...SET PURGE DATE
$       RUN FAL$PRGLIB:MXSETPUR
```

Example 24-9 - JOBMONEND New Month Initialization

12B1 Processing

12B1 processing is spread throughout the month-end processing activities. The code snippet in Example 24-10 below, is a major portion, but only the beginning.

```
$ ! 12B1 Processing:
$ !   - Catch-up 12b1 aggregate balances in B12FAL.DAT through calendar month-end
$ !   - Sort B12FAL.DAT copy in account/fund order socan be processed by MXINIMON
$
$       TIMESTAMP MONTHEND,JX12BDDB,...12B AGG BAL THRU EOM
$       COPY FAL$DATLED:B12FAL.DAT FAL$SCRATCH:*.* /LOG
$       FAL_CMDLINE = "/ME"
$       RUN FAL$PRGLIB:JX12BDDB
$       DELETE FAL$SCRATCH:B12FAL.DAT;*
$       IF F$SEARCH("FAL$SCRATCH:B12FAL.OUT;*") THEN -
                DELETE FAL$SCRATCH:B12FAL.OUT;*
$       SORT/STAT -
           /KEY=(CHARACTER, SIZE:14, POSITION:7, ASCENDING) -
           /KEY=(CHARACTER, SIZE:6,  POSITION:1, ASCENDING) -
              FAL$DATLED:B12FAL.DAT FAL$HLDFIL:B12FAL.DAT
$       PURGE FAL$HLDFIL:B12FAL.INP
$ !
$       TIMESTAMP MONTHEND,JXB12TOMIDP,SPLIT B12 into TWO FILES
$ !     FAL$HLDFIL:B12FAL.DAT will ve split into two files
$ !
$ !     1. FAL$HLDFIL:B12FAL.INP that contains NON Mid-Period funds and Mid-
$ !             Period NOT active funds for the current processing date
$ !             i.e. only funds that Ending date is not current processing date
$ !             or Ending dateis not falling into next holiday or weekend.
$ !             -FAL$HLDFIL:B12FAL.INP is input file for ME12BUPD.for in MXINIMON
$ !     2. FAL$SCRATCH:B12FAL.OTH has all other
$ !             records from B12FAL i.e.
$ !             FAL$SCRATCH:B12FAL.OTH = FAL$HLDFIL:B12FAL.DAT
$ !             minus  FAL$HLDFIL:B12FAL.INP.
$ !     3. FAL$HLDFIL:B12FAL.OTH  will be appended
$ !             to FAL$SCRATCH:B12FAL.OUT - Created by ME12BUPD.for
$ !             and copied to replace current B12FAL.dat in FAL$DATLED
$ !             later, producing final B12FAL.dat output.
$ !             - FAL$DATLED:B12FAL.DAT is intact in this step
$ !
$       PURGE FAL$SCRATCH:B12FAL.*
$       PURGE FAL$HLDFIL:B12FAL.*
$       FAL_CMDLINE = "/ME"
$       RUN FAL$PRGLIB:JXB12TOMIDP.EXE
$       PURGE FAL$SCRATCH:B12FAL.*
$       PURGE FAL$HLDFIL:B12FAL.*
```

Example 24-10 - JOBMONEND 12B1 Processing

Initialization of the New Month

This code snippet is a portion of processing to initialize activities for the new month. This snippet includes both leger and 12B1 initialization activites.

```
$ !  Initialize the ledger and set the period beginning dates
$ !  New version includes production of ledger status report (invoice)
$ !  previously done in mngeninv
$ !
$       TIMESTAMP MONTHEND,MXINIMON,...INIT LEDGER
$       FAL_CMDLINE = "/ME"
$       RUN FAL$PRGLIB:MXINIMON
$       PURGE FAL$HLDFIL:B12FAL.*
$       PURGE FAL$SCRATCH:B12FAL.*
$ !
$       TIMESTAMP MONTHEND,APPEND B12FAL.OTH to B12FAL.OUT
$       APPEND/LOG FAL$SCRATCH:B12FAL.OTH FAL$SCRATCH:B12FAL.OUT
$ ! ===========================================================================
$ !  Put initialized B12FAL.DAT back into database sorted by fund//account
$
$       IF F$SEARCH("FAL$SCRATCH:B12FAL.OUT") .NES. ""
$       THEN
$          TIMESTAMP MONTHEND,SORT,...INIT B12FAL.DAT
$          DELETE FAL$DATLED:B12FAL.DAT;*$
$          SORT/STAT -
              /KEY=(CHARACTER, SIZE:6,  POSITION:1, ASCENDING) -
              /KEY=(CHARACTER, SIZE:14, POSITION:7, ASCENDING) -
                FAL$SCRATCH:B12FAL.OUT  FAL$DATLED:B12FAL.DAT
$          DELETE FAL$SCRATCH:B12FAL.OUT;*
$       ENDIF
$ !
$ ! ===========================================================================
$ !  Initialize the B12EXTSHR indexed file for the next month distributions.
$
$       IF F$SEARCH("FAL$DATTMP:B12EXTSHR.DAT") .NES. ""
$       THEN
$          TIMESTAMP MONTHEND,SORT,...INIT B12EXTSHR.DAT
$          COPY FAL$DATTMP:B12EXTSHR.DAT FAL$HLDFIL:B12EXTSHR.OLD
$          PURGE/KEEP=3 FAL$HLDFIL:B12EXTSHR.OLD
$       ENDIF
$ !
$       CREATE/FDL=FAL$FILES:B12EXTSHR FAL$DATTMP:B12EXTSHR.DAT
$       PURGE FAL$DATTMP:B12EXTSHR.DAT
```

Example 24-11 - JOBMONEND New Month Ledger and 12B1 Processing

Additional Month-end Processing and Report Generation

The code snippets in Examples 23-12 and 24-13 show month-end report generation and additional month-end processing,

```
$ !  Perform bluesky end-of-month processing
$ !
$       TIMESTAMP MONTHEND,MXINISKY,...INIT BLUESKY
$       RUN FAL$PRGLIB:MXINISKY
$ !
$ ! ============================================================================
$ ! Status check the ledgers
$ !
$ !  9/90 (EKW) -  Report is now produced in MXINIMON
$ !
$ ! ============================================================================
$ !  Produce PrimeMix Report if PrimeMix in Use
$ !
$       TIMESTAMP MONTHEND,MXMIXINV,...PRIMEMIX REPORT
$       RUN FAL$PRGLIB:MXMIXINV.EXE
$ !
$ ! ============================================================================
$ !  Produce Report Generator Usage Report
$ !
$       TIMESTAMP MONTHEND,JXRPTUSG,...RPT GEN STS
$       RUN FAL$PRGLIB:JXRPTUSG.EXE
$       COPY FAL$REPORT:RPTGENUSG.RPT FAL$HOME:*.*
```

Example 24-12 - JOBMONEND Blue-Sky & PrimeMix Processing

```
$ !  Print reports automatically
$ !
$       TIMESTAMP MONTHEND,JXAUTPRI,...BEG AUTPRI
$       RUN FAL$PRGLIB:JXAUTPRI.EXE
$       @FAL$REPORT:AUTREPORT.COM
$       TIMESTAMP MONTHEND,JXAUTPRI,...END AUTPRI
$ !
$ ! ============================================================================
$ ENDTRP:
$ !
$ !  This is the normal exit block.
$       TIMESTAMP MONTHEND,,*** END EVENT ***
```

Example 24-13 - JOBMONEND Monthend Report Processing

Yearend

The Yearend event sets up the core files for the new year. The command procedure is JOBYEREND.COM and is found in FAL$PRGLIB. The activities accomplished during the Yearend event include:

- Copying the Year-to-date file to tape (YTDFAL)

- Copying the current Journal file to the Year-to-date file (JNLFAL to YTDFAL)
  - The YTDFAL file now contains the prior year's transactions.
- Deleting the current Journal file (JNLFAL)

- Creating an empty Journal file (JNLFAL)
  - The JNLFAL is now ready for the current year's transactions

- Incrementing the Tax Year for Tax processing

The Yearend event is run after Jobstream and Monthend events have run and before processing begins for the new year.

Abends and Troubleshooting

Overview

Handling the aborted ending to an event can be accomplished in a number of ways. When troubleshooting an aborted event an understanding of threads and how to read log reports can be critical. In addition, the use of SuRPAS utilities such as the Trace Facility can be helpful.

This section discusses:

- Standard and non-standard abends
- Problem research
- An overview of threads in OpenVMS
- Checkpointing
- The trace facility

Abends

The term abend has its origin in the IBM world. In both IBM and SuRPAS environments ***abend*** stands for ***abnormal end***. It refers to the abnormal ending of a program or routine and requires a series of steps to solve the problem that caused the abend.

Standard Abends

Standard abends end in a controlled manner. These errors create a message in an error file found in the FAL$JE directory called JOBERR.LOG. This is the primary source of information when troubleshooting an abend, assuming that no other processing has occurred since the error.

Most common files (e.g. JOBDAILY) will output a time stamp and description to indicate progression through the command file. The coding templates (TEMPLATE.FOR) also contain time stamps to indicate the beginning and end of each code module. Additional time stamps may be added to indicate that a significant event has occurred. Time log stamps appear in a file called TIMELOG.DAT. This file also resides in the FAL$JE directory. This may assist in determining where a command file actually failed (which modules of code have executed completely and which modules have not). In the case of synchronous executables, it will indicate whether a synchronous executable completed properly.

Nonstandard Abends

Nonstandard abends cause the program to terminate unexpectedly. In this case no message is written to the JOBERR.LOG file. The program may fail with any one of the following errors:

- Subscript Out of Range
- Access Violation
- Record Not Found
- End of File
- Array Out of Bounds
- Subroutine Missing Argument

The program may also fail for reasons not currently known. That is why they are non-standard. The actual screen information is captured in the FAL.USER.uuuu directory in a file named FALUSR.ERR. The information includes the routine name and the line number (corresponding to the source code listing line number). This provides a starting point for researching the error.

Abend Research

The JOBERR Log File

SuRPAS standard abend messages are written to the job error log file, The error log file, named JOBERR.LOG is located in FAL$JE. The purpose of the job error log is to list all standard abend messages. This is the primary source of information in researching the cause of a problem or abend while running SuRPAS events. A block of information can be found in the log that will provide you with an error code, an error message, the trail of routines with the appropriate line number. (Be wary of exact line numbers- the Fortran optimizer may render the line number useless - making it a pointer to a general location rather than an exact line.) New messages are appended to the error log every time you run an event.

The actual routine to review and the line number within the routine is often a function of the type of error that has occurred. For example, if the program was expecting a record and it was not there, the program may terminate with a “record not found”. The last two routines listed in the error output will be the error processing routines MSGERR and MSGERRM). These are of little interest, but the routine that called these routines may be the culprit, and it is the routine name and line number that are of interest.

The JELOGIN Command Procedure

The routine JELOGIN.COM writes out the following information to JOBERR.LOG:

- which events are executed
- the command file used
- the job begin time
- the user’s initials
- the database and software version

If the event does not abend, JELOGIN.COM also displays a “Normal End of Processing” message and the job end time in the job error log. Event command file DCL statement confirmation messages are also written to the error log. For example, if you purge a file using a DCL command in an event command file, a confirmation message will appear in the job error log detailing which files were deleted. There are two methods available to the programmer for writing messages to the job error log.

The TIMELOG.DAT File

When command files are executed, most command files will output a time stamp with a description indicating its progression through the command file. This time stamp appears in a file named TIMELOG.DAT, in the FAL$JE directory. The information in this file may assist in determining where a command file actually failed. In the case of synchronous executables, it will indicate whether the program completed properly. This is particularly true in the JOBSTREAM event (the command file name is FAL$PRGLIB:JOBDAILY.COM). This is only minimally helpful with standard abends, but it can be very helpful with non-standard abends because it will focus your troubleshooting. The following was extracted from TIMELOG.DAT to demonstrate the kind of information you will find.

```
!  JOBPROCESS   >>>START   19-AUG-1997  16:56:41.83     1121   501   116   151  @FAL$PRGLIB:JOBDAILY.COM
!  JOBDAILY                  19-AUG-1997  16:56:42.28     1204   520   128   160 *** BEGIN EVENT ***
!  JOBDAILY                  19-AUG-1997  16:56:49.97     2302   831   491   360 ...DELETE PR3SSR.DAT
!  JOBDAILY                  19-AUG-1997  16:56:50.17     2400   858   498   370 ...MAKE SCRATCH FILES
!  JOBDAILY                  19-AUG-1997  16:56:57.66     4023  1348   807   616 ...PURGE DIRECTORIES
!  JOBDAILY                  19-AUG-1997  16:57:02.97     5162  1888  1028   799 ...CLEANUP NBR FILE
!  JOBDAILY      JXTAGCSM    19-AUG-1997  16:57:04.32     5455  2004  1077   838 ...TAG OPN BALS
!  JOBDAILY      1:252       19-AUG-1997  16:57:07.20     6071  2180  1252   930 @T$JOBQT1.COM
!  JOBDAILY      2:253       19-AUG-1997  16:57:08.22     6542  2317  1272   978 @T$JOBQT2.COM
```

Example 24-14 - TIMELOG.DAT example

PUTERR Subroutine

PUTERR.FOR is a Fortran subroutine primarily used to write informational and/or warning messages to the job error log. These may include additional parameters or missing/incorrect record warnings. Abends are not directly associated with a call to PUTERR. The syntax of the PUTERR subroutine call is:

```
CALL  PUTERR  ()
```

MFSERRM Subroutine

MFSERRM.FOR is used to supply additional information when an abend does occur. This information is used as an aid in researching and fixing the problem. MFSERRM.FOR commonly supplies a pertinent one line error message/error code and may also include information such as fund number, fund name, account number, etc. The syntax of the MFSERRM subroutine call is:

```
CALL  MFSERRM  ()
```

Standard Abend Research

Assuming that no other processing has occurred since the abend, open the error file JOBERR.LOG, and go to the end. The information at the end of the file will generously contain an error code, an error message, and the trail of routines with the appropriate line number. (The specific routine and line number is a function of the type of error.)

As an example of researching a standard abend, if a program was expecting a record and it was not there, the program might terminate with a "record not found" error. In this case, the last two routines will contain the error processing routines. As discussed earlier, you are not interested in the those routines, but you are interested in the routine that calls the error routine. It is this routine name and line number that you want to make note of for your research.

To determine the location in the code where the problem occurred compile the code with the /LIST option and modify the declaration portion of your code so the /LIST option is also attached to the appropriate INCLUDE files (be sure to place the /LIST inside of the single quotes). Open the listing file (.LIS file) in EVE and look up the line number that was given in the error message. The line numbers and the CALL to the error routine may not exactly line up but they will be within a few lines of code, often one or two previous to the line number given. By understanding this portion of the code you can determine what file was being read and what specific record the routine was expecting to read.

If other users continue to process while you are researching a problem, it maybe necessary to make a copy of a file or files to avoid an access conflict and capture the file (s) before they are modified or deleted. If you need to make a copy of JOBERR.LOG use the ERRLOG utility. At the dollar-sign prompt enter:

```
$  ERRLOG  JEnn
```

where ***nn*** is the database number. This will issue a BACKUP command with the qualifier /IGNORE=INTERLOCK switch. A copy of the file will be created in the same directory as the original file, but it will have the name JOBERR.JE*nn.*

You may also want to copy the TIMELOG.DAT file to a TIMELOG.JEnn file for the same reasons. However, you will have to use the BACKUP utility directly as there is no symbol or associated command file. To accomplish this, at the dollar-sign prompt type:

```
$     BACKUP  /NEW  /IGNORE=INTERLOCK  /LOG  -
_$     FAL$USER_DISK:[FAL.USER.JEnn]TIMELOG.DAT;0  -
_$    *.JEnn
```

Files and Reports for Troubleshooting

If an event has been run and you need to research what options or parameters were selected for the event, this information can be provided by the JBXLIST report. This report is created by running the JBXLIST routine found in FAL$PRGLIB (run it from that directory). The report will be created in the standard report directory FAL$REPORT with a file name of JBXLIST.RPT. Other reports found in FAL$REPORT include:

Report Name	**Report Purpose**
JSUMRPT.RPT	Job processing completion summary
JBQLIST.RPT	List of current jobs in the Job Processing Queue
JBXINIT.RPT	Initialization list of the completed jobs file (JBX can be initialized, this report lists the items that are being deleted)

JOB QUEUE Lists

When a job is queued, it is added to the JBQFAL.DAT file. As jobs are run, they are appended to the JBXFAL.DAT file. If a program abends (with clean-up), JBQFAL is copied to JBXFAL. Both of these files are located in the FAL$DATSCH directory. The directory structure for this logical named directory is:

DISK*##*:[FALDAT*db*.DATSCH]

where ***##*** is the disk number and ***db*** is the database number.

Standard Abends - An Indexed File Example

The printout which follows on the next page shows an example of a FORTRAN error (error #36) with a description of “attempt to access non-existent record” (i.e. the program is looking for a record that does not exist in the file).

The Detail

The first block is the information from FAL$JE:JOBERR.LOG. The file name is the PARAMS file and the key value for the read is “IRSFD” on key 0 (the primary key). The calling tree is listed in reverse order. That is, the first routine listed, MFSERRM, is the last routine that was called when the error occurred. This routine was called by the next routine, MFSERR, (the familiar MFS error routine that is the normal result of a conditional status check following every FALIOS call). The routine that we are interested is the one that calls the error routines, TXSUMMNT in this example. The line number is 2904.

The second block of code is a portion of the listing from the TXSUMMNT routine. You can see the call to MFSERR immediately prior to line 2904. The error is a result of the read on the PARAMS file. The key value searched was IRSFD + the form number + the fund. Note that the key value listed in the error message does not have a form number or a fund ID. This is an indication that these fields are blank. (The arguments of the FALIOS call are discussed in detail in Chapter 20.)

Once you have found the error in the log and the code error location is determined, the next step in researching an abend is to trace through the code and find out why the fields are blank.

```
**************  DATE:30-JUL-97 17:48:02  **********  ERR#    36  *************
 Job:            $5$DRB0:[FALSYSD_AXP.PRGLIB]TXFUNDCOR.EX    User: SWS
     RMS I/O STATUS:       0    FORTRAN ERROR :       36
     LOGICAL UNIT  :       6    DISK14:[FALDAT53.DATINS]PARAMS.DAT;13
 %RMS-E-RNF, record not found
 %FOR-F-ATTACCNON, attempt to access non-existent record:/unit !SL file !AS!
               MFSERR :        36
      FID  21  LUNPAR :         6   DISK14:[FALDAT53.DATINS]PARAMS.DAT;13
       OP,LCK HON,LCK :   KEY   RRL   NCK
         KEY/REC#,VAL :         0   IRSFD
 %SYSTEM-F-ABORT, abort
 %TRACE-F-TRACEBACK, symbolic stack dump follows
  Image Name  Module Name      Routine Name    Line Number   rel PC      abs PC
  TXFUNDCOR   MFSERR           MFSERRM                3647  00001058   0004E568
  TXFUNDCOR   MFSERR           MFSERR                   79  00000034   0004D544
  TXFUNDCOR   TXSUMMNT         TXSUMMNT               2904  000002AC   000655CC
  TXFUNDCOR   TECORDIV         TECORDIV               4636  000029EC   0005B11C
  TXFUNDCOR   TXFUNDCOR        TXFUNDCOR              3768  00000A60   00040A60
  TXFUNDCOR                                              0  00089064   00099064
  DEC$FORRTL                                             0  00072818   00966818
  TXFUNDCOR                                              0  00089064   00099064
                                                         0  9AB36170   9AB36170



 2896
 2897       KEYPAR         = 'IRSFDt      '
 2898       KEYPAR(6:6)    = FORM
 2899       KEYPAR(7:12)   = FUND
 2900
 2901      CALL FALIOS(LUNPAR,LUNTBL,'KEY','RRL','NLK',0,KEYPAR,IOS,PARIOB)
 2902       IF (IOS .NE. 0) CALL MFSERR(IOS)
 2903
 2904       SEQUENCE = PARIOD(13:15)
 2905
```

Example 24-15 - Index File Standard Abend Example

Standard Abends - A Direct Access File Example

The next printout shows an example of an attempt to access a record that does not exist. This is a direct access file so the FALIOS call is issuing a “GET” operation code. This time the FORTRAN error (error #36) has a description of “record not found” (again, the program is looking for a record that does not exist in the file).

The Detail

Notice that the call to FALIOS has the ‘GET’ operation code (positional argument #3) and the KEY / Record # parameter (positional argument #7) has a value of 538976288. When you are looking at index sequential files the argument #7 is the key value; when looking at fixed sequential or direct access files this argument is the record number being examined. It is not likely that there would be over 500 million check records in the CKN file. Consequently, this routine is terminated because the record number value is not being set properly.

The next step would be to find line #2340 in the SXSCHKREC routine, find the variable holding the record number, and then trace back the loading of that variable. Then continue this process until the problem is found.

```
************** DATE: 3-JUN-97 09:57:39  **********  ERR#    36  *************
 Job:            $1$DKA303:[FALSYSD.PRGLIB]SXSCHKREC.EXE    User: SFB
     RMS I/O STATUS:       0    FORTRAN ERROR :       36
     LOGICAL UNIT  :      12    DISK12:[FALDAT10.DATINS]CKNFAL.DAT;3438
 %RMS-E-RNF, end of file detected
 %FOR-F-ATTACCNON, attempt to access non-existent record:/unit !SL file !AS!
               MFSERR :        36
      FID  52  LUNPAR :        12   DISK12:[FALDAT10.DATINS]CKNFAL.DAT;3438
       OP,LCK HON,LCK :   GET
         KEY/REC#,VAL : 538976288
 %SYSTEM-F-ABORT, abort
 %TRACE-F-TRACEBACK, symbolic stack dump follows
  Image Name  Module Name      Routine Name    Line Number   rel PC      abs PC
  SKSCHKREC   MFSERR           MFSERRM               3604  00001058   0003EFF8
  SKSCHKREC   MFSERR           MFSERR                  79  00000034   0034DFD4
  SKSCHKREC   SKSCHKREC        SKSCHKREC             2340  00000A84   000564A4
  SKSCHKREC                                             0  000464A4   000564A4
  DEC$FORRTL                                            0  00072818   001C6818
  SKSCHKREC                                             0  000464A4   000564A4
                                                        0  921AC170   921AC170

 2336
 2337       CALL FALIOS(LUNPAR,LUNTBL,'GET','RRL','NLK',0,538976288,IOS,PARIOB)
 2338       IF (IOS .NE. 0) CALL MFSERR(IOS)
 2339
 2340       SEQUENCE = PARIOD(13:15)
 2341
```

Example 24-16 - Direct Access File Standard Abend Example

Threads

The OpenVMS operating system supports the ability to execute one or more computational entities on a processor. Each of these entities is referred to as a ***thread of execution*** or simply a ***thread***. A thread can be a program unit, a task, a procedure, a DO loop, a command file or some other unit of computation.

All threads executing within a single process share the same address space and other process contexts, but have their own unique per-thread hardware context. This allows a multi-threaded process to execute different threads on different processors if the Alpha is a multiprocessor computer.

Command Files As Threads

There are two basic types of command files within the OpenVMS operating system. The simplest command file executes the DCL commands from top to bottom in a single stream of execution, or a single thread.

The second and more complicated type of command file creates secondary process threads that are executed simultaneously. This type of command file is called a multi-threaded command file.

In a multi-threaded command file each thread executes independently of the other threads on the same or different processors. The command file is not considered to have completed until all threads within the command file have completed. Under certain circumstances it is necessary to coordinate activities of two or more executing threads. This may be necessary when one thread requires information calculated by another thread before the first thread can continue. This synchronization requirement is also supported by OpenVMS.

An understanding of threads is critical when supporting the SuRPAS nightly process. Without this understanding error troubleshooting is very difficult. When troubleshooting, be aware that certain command procedures/jobs have been selected for multithreading because they have several processing steps that can be run independently of the main control procedure. Examples of these command procedures (and their processes/jobs) are:

- JOBDAILY.COM (JOBSTREAM, JOBPROCESS)
- JOBCKDSK.COM (BACKUPDSK)
- JOBSAVMON.COM (SAVEMONEND)

Levels of Threads

In the non-backup types of jobs that support multithreading, thread 0 (zero) is always the main command file out of which all other threads are submitted. Other threads are jobs that are independent steps within the event that are submitted to a job batch queue where they may be executed in parallel to the main thread, as well as any other threads. When a job thread is submitted to the queue it is assigned a unique thread number (sequentially, beginning with 1).

In the mean time, the main job may continue its execution. If a later step in the event requires the completion of one or more of the previously submitted threads, the main job will SYNCH (synchronize) on the prerequisite thread(s) completion prior to proceeding. In general, all threads are still considered a part of the main job in that they must finish before the main job can be considered successfully completed.

If the job is the BACKUPDSK event, all of the threads must be SYNCHed before the next job can be run.

Creating a Thread Batch Job

The batch queue that runs the threads in SuRPAS is called **FAL$JOBQUE_##**, where **##** is the database slot number. This queue is created whenever a new event that creates a set of threads is run. The queue is initialized with the ***maxjobs*** parameter set to the maximum number of threads. Each database has its own thread queue.

When you want the main command file to wait for the completion of a thread, used the SYNCH command. The SYNCH command has the syntax:

SYNCH (*thread #*)

When the threads SYNCH, status is checked. If the thread was successful, the job continues, or else the entire job abends.

In most SuRPAS command procedures the SYNCH command is issued within the JOBQSYNC command procedure. This command procedure handles SYNCH management and reporting in addition to the actual issuing of the SYNCH command. The following lines, taken from JOBDAILY.COM demonstrate the use of this command procedure to synchronize threads 10, 11, and 12:

```
$ JOBQSYNC 10
$ JOBQSYNC 11
$ JOBQSYNC 12
```

Thread Composition

The DCL code for thread 1 in a thread batch job has the format shown in the following example:

```
$ JBQMAKE     !  Makes the job queue for the thread, has OPEN privileges
$ DECK        ! Indicates the beginning of the thread deck of instructions
   T01$       ! The following are commands to run the job being submitted
   T01$
     •
     •
     •
   T01$
$ EOD         ! Indicates the End Of Deck of instructions
$ JBQSUBM 1   ! This is the Job Queue Submit command for thread #1
```

The following files are created from the above example. They are all stored in FAL$JE:

- T$JOBQT1.COM - the command file (the 1 indicates the thread number)
- T$JOBQT1.LOG - the log file
- T$JOBQT1.ERR - the error log appended to JOBERR.CUR at the point of SYNCH (synchronization) or CLR (clear)
- T$JOBQT1.TIM - the timestamp appended to the timelog at SYNCH or CLR

In JOBDAILY.COM, the threads before JXPASS3 are numbered 1 through 3; the jobs after this step are numbered 10 through 15. Examples of some of these thread streams can be found in examples 24-2, 24-3, and 24-6.

Multi-threading Troubleshooting

The Job Executive tool supports the troubleshooting of job activities, especially those jobs run in client nightly processing. You can execute this tool by selecting the Job Control Executive or JE4 option in the Operator Main Menu. The Operator Main Menu is available by pressing the PF3 key at the SuRPAS Main Menu.

The ***MON*** screen option is the MONITOR function. It allows you to monitor a job and its threads to acquire a task status. The ***CLR*** screen option is available when there is a problem with a job stream or one of its threads.

Selecting the CLR option will pick up the error logs for all of the threads and append them to the main JOBERR.CUR file. JOBERR.CUR appends JOBERR.LOG unless a fatal error condition has occurred. When studying the CLR screen in the Job Executive tool note that the **'f'** under the thread number indicates that the thread has finished and an **"e"** indicates that an error occurred. It is important to check this prior to clearing the job.

You can examine the errors in JOBERR.CUR by looking in the FAL$JE directory for the file. Note that time stamping is defined by the thread code, but the SYNCH is accomplished in the main job; therefore the last entry in the JOBERR.CUR file may not be the one you are looking for. You may have to look at previous entries in JOBERR.CUR or you may have to look in the specific thread error log. Error logs for the threads that are running or have not been cleaned up will be in FAL$JE:T$JOBQT#.ERR where the **#** represents the specific thread number.

If the main job abends and threads are still running, you may have to stop these threads before restoring the database and rerunning the job. There are a number of options available for stopping these executing threads (killing the process).

- From DCL run the JE tool and press the PF1 key from the CLR option screen to delete each job

- If this doesn't work, from DCL, issue the STOP/QUEUE/RESET command on the file FAL$JOBQUE this will stop the queue. An example of this command is:

```
$ STOP/QUEUE/RESET FAL$JOBQUE
```

- An alternative to stopping the queue is deleting it, as shown in the following example:

```
$ DELETE/QUEUE FAL$JOBQUE
```

- If you want to abort a specific job, enter the following DCL command (where **##** indicates the thread number)

```
$ DELETE/ENTRY=##
```

If you are not going to restore the database, let the threads finish, clear the ABEND and write a ***checkpoint*** command procedure.

Checkpointing

Checkpointing is the term used to describe a method of restarting SuRPAS events after they have failed. Typically, this involves researching and fixing the problem that caused the failure first, then restarting the event after the point of failure. This is accomplished by making a copy of the event's command file and modifying it to skip the steps of the event that have already completed and to execute of those that remain.

Checkpointing JOBSTREAM

The following paragraphs discuss how to checkpoint JobStream after you've resolved the problem that caused the abend. While these steps provide specific information from JobStream, checkpointing other multithreaded events is handled in a similar manner.

First, copy FAL$PRGLIB:JOBDAILY.COM to FAL$JE:CHKPOINT.COM using the sample DCL command shown below. When you are ready to restart the SuRPAS events you will queue the CHKPOINT event which, in turn, executes FAL$JE:CHKPOINT.COM. The DCL command is as follows:

```
$     COPY/LOG  FAL$PRGLIB:JOBDAILY.COM  -
_$                FAL$JE:CHKPOINT.COM
```

Once the file has been copied you can view the time log to determine which steps have actually completed (and don't have to be restarted) and which events will require new execution. The time log is located in the file FAL$JE:TIMELOG.DAT.

You can now edit FAL$JE:CHKPOINT.COM to remove unnecessary DCL command execution lines and modify any required execution lines. Follow these steps to properly modify the CHKPOINT command file:

- Find the phrase **"Begin Event".** This will get you past all of the comments at the beginning of the command file.

- Immediately following the line containing the **"Begin Event"** phrase Insert the following two DCL commands:

```
$ FAL_EVENT_NAME :== JOBDAILY
$ GOTO CHECKPOINT
```

- Determine where you want to continue processing events so you can properly place the CHECKPOINT label.

  - If the abend occurred inside of a thread, make sure that the thread finished.
    - √ Put the **$ CHECKPOINT:** after the Jobsync for that thread
    - √ DO NOT put it after the thread was created and submitted or you will run the thread software twice

  - If the abend occurred outside of a thread, put **$ CHECKPOINT:** statement at the location where you need to continue processing.

Clear the abend, delete Jobprocess from the schedule and the queue/execute the CHKPOINT event.

When writing your checkpoint command procedure, remember to comment out or delete the threads that have already completed successfully. Also, make sure that you have commented out the SYNCHs for the threads that have already run (and are being skipped). The main program will abend when it gets to the SYNCH, if there are no threads to synchronize. Be careful not to comment out or delete symbol definitions that have been defined at the top of the command procedure. These symbols may be used throughout the COM file. It is a SuRPAS standard to place all symbol definitions after the end of comments marker (EOM).

If you find that a small job thread has abended, you may decide to fix the problem and submit the job from the DCL command prompt ($). If you do, be sure to bypass the entire thread in the checkpoint procedure that you write. Don't forget to remove the SYNCH for the thread also. You can also modify the procedure to run the remainder of the thread as a part of the main thread (THREAD 0). This can be done by removing the 'T##" thread ID in front of the DCL prompt ($) in your procedure code (see the DECK example on page 17).

After all of the threads are cleaned up (the term wrap up is common) and rerun, the user must clear the abend before any other jobs can be run through the job executor. If the threads have not been wrapped, the thread status flags will be ***Submitted, Running*** and ***Abend***. The wrap option in the JE monitor/clear screen changes the flags to ***Finished*** and ***Error***.

The following options are available on the monitor/clear screen after an abend:

Option	Option Description
Wrap Up	Wraps up any orphan threads that remain and puts their job error IOB output and time stamp output with Thread 0 output
Clear	Available after all threads have been wrapped up
Exit	Will exit the user out of the clear abend screen
Refresh	Repaints the screen. Does not update the screen.
Delete Other	Used if control information does not seem to know if there is an event in the queue. Allows deletion of the completed event from the queue.
Update	Updates the screen with the current status information.

Checkpointing Non-Multithreaded Events

The following paragraphs discuss how to checkpoint an event that does not contain threads other than the main thread (Thread 0). As in earlier discussions it is fair to assume that checkpointing is done after you've resolved the problem that caused the abend.

First, copy the event's command file to FAL$JE:CHKPOINT.COM using the sample DCL command shown below. When you are ready to restart the SuRPAS events you will queue the CHKPOINT event which, in turn, executes FAL$JE:CHKPOINT.COM. The DCL command is as follows:

```
$     COPY/LOG  FAL$PRGLIB:<your  event  file>.COM  -
_$                FAL$JE:CHKPOINT.COM
```

Once the file has been copied you can view the time log to determine which steps have actually completed (and don't have to be restarted) and which events will require new execution. The time log is located in the file FAL$JE:TIMELOG.DAT.

You can now edit FAL$JE:CHKPOINT.COM to remove unnecessary DCL command execution lines and modify any required execution lines. Follow these steps to properly modify the CHKPOINT command file:

- Get past all of the comments at the beginning of the command file.
- Insert any logical definitions that may have been removed from the beginning of the command procedure and then inset the following DCL command:

  ```
  $  GOTO  CHECKPOINT
  ```

- Determine where you want to continue processing your event so you can properly place the CHECKPOINT label.
- Put the **`$ CHECKPOINT:`** label where you want to continue processing your events.

Have a customer service (CSR) and/or operator clear the abend, delete Jobprocess from the schedule and the queue/execute the CHKPOINT event.

The Trace Facility

The Trace Facility can be useful when you have to read through a lot of data for calculations, decision making, or tracking performance. It helps in tracking data within your program. The Trace Facility should not be used for or in place of general debugging. Use the OpenVMS Debugger (DEBUG64) for general debugging. You can find examples of the Trace in the run time balance calculator (RTBCLC.FOR), average cost calculation (AVGCLC.FOR), and FALIOS.FOR.

Trace Facility Authorization

You can authorize the Trace facility for the database that you are working in by using the Parameters (PAR option in SuRPAS). In the Insert screen (INS) change INS 096 to Yes ("Y"). The screen should look similar to the lines that follow:

```
ENTER INS # : 096
ALLOW/HONOR TRACE PARAMETER RECORDS Y

  Option Descriptions: BLANK = DO NOT ALLOW TRACE
                       "Y" = ALLOW TRACE
                       "N" = DO NOT ALLOW TRACE
```

Add Parameter Records

Add a parameter record for each user who will use the Trace Facility. Typically you would not want data collected for all users running the program in question. You would only add a parameter record for the user having the problem or the programmer checking the data.

This can be handled using the PAR (parameter maintenance) option in SuRPAS WorkBench (see Chapter 22 for more information on the PAR option). At the main PAR Menu (a portion shown below), select the PARAMS Maintenance Byte-editor option (PAR).

PAR - PARAMS Maintenance Byte-editor

LIS - List of PARAMS Keys

DUM - Dump of PARAMS File

EXT - Extract PARAMS records to file

INC - Include PARAMS records from file

MEN - Return to Main Menu

This will bring you to the Byte editor Maintenance screen and you will be prompted for the action and the key. The Action prompt provides five options: **A**dd, **C**hange, **D**elete, **V**iew and **Q**uit. Select A to add a new trace key. Then enter the name of the key. In this case the Trace key is usually named TRACE followed by your TLA (three-letter-acronym) or the three initials of your name. An example of this is shown below:

```
ENTER  ACTION  (A,C,D,V,Q)  :  A
ENTER  KEY                  :  TRACEEKW
```

The Byte-editor Maintenance screen will then be updated to look similar to the following:

```
ENTER  ACTION  (A,C,D,V,Q)  :  A
ENTER  KEY                  :  TRACEEKW
ENTER  BEG  BYTE  POSITION  :  013      [ enter routine in bytes 13-20]
ENTER  END  BYTE  POSITION  :  020
ENTER  BUFFER               :  /RTBCLC/
```

Notice that the buffer defined begins in byte position 13 and ends in byte position 20. This allows you eight characters (six characters surrounded by forward slashes) to enter the Fortran module name for which you want Trace Facility access.

Generate an Audit Trail

Add the code necessary for generating the audit trail to the routine in question. Be sure to add the Trace Facility Include file at the beginning of any program using Trace. An example of the appropriate INCLUDE statement follows:

```
INCLUDE  'FAL$LIBRARY:FILES(TRACECOM.CFD)'
```

Code can now be included in your Fortran routine to initialize the trace, open the audit trail, and write the data you want tracked to the audit file. Once the trace is complete and you have resolved any data problems. Be sure to turn Trace off, either by setting PAR-INS-INS #096 to "N", or by deleting the TRACEXXX parameter record for the defined Trace user.

Trace Audit Trail Example Code

The following are snippets of code taken from RTBCLC.FOR to demonstrate how an Audit Trail can be created and used to track data that caused an abend. The complete routine code can be found in FAL$SUB:RTBCLC.FOR.

Include the following file:

```
INCLUDE 'FAL$LIBRARY:FILES(TRACECOM.CFD)'
```

Initialize the TRACE:

```
IF (TRACEINIT .NE. FSY.TRACEINIT) THEN
    TRACEINIT = FSY.TRACEINIT
    TRACEON = FSYTCGET('/RTBCLC/',' ') [use routine name from parameter record]
ENDIF
```

Open the audit trail (SYS$LOGIN:RTBCLC.AUD)

```
IF (FSTPAS) THEN
    IF (TRACEON) THEN
        CALL LUNASSIGN('G',' ',LOGLUN)
        LOGFIL = 'SYS$LOGIN:RTBLOG.AUD'
       OPEN (UNIT = LOGLUN,
     +       FILE              = LOGFIL,
     +         ORGANIZATION    = 'SEQUENTIAL',
     +       ACCESS            = 'APPEND',
     +       FORM              = 'FORMATTED',
     +         CARRIAGECONTROL = 'LIST',
     +       STATUS            = 'UNKNOWN',
     +       BLOCKSIZE         =  512,
     +       INITIALSIZE       =  100,
     +       EXTENDSIZE        =  100,
     +      RECL               =  132,
     +       IOSTAT            =  IOS,
     +       SHARED)

        IF (IOS .NE. 0) CALL MFSERR(IOS)
   ENDIF
    FSTPAS = .FALSE.
ENDIF
```

Decide what data you want to track and write it to the audit file. TRACEON is set in the initialization if the Trace Facility (in the INCLUDE CFD file).

```
IF (TRACEON) THEN
        WRITE (LOGLUN,200)
        WRITE (LOGLUN,201)
         WRITE (LOGLUN,202)LEDACT, LEDFND, INCERC,
       +     RTT_TYPE, FILE_NAME, XMODE, THRESHOLD,
      +     THOLD, RTS, PROC_DATE
        WRITE_HEADER = .TRUE.
ENDIF
```