Chapter Twenty

INPUT AND OUTPUT

Introduction

Fortran input and output is a process whereby information is transmitted between the memory of the Alpha and various external devices. In this chapter we will examine the techniques for transmitting this data using both the Fortran input and output statements and using the SuRPAS Input and Output System (FALIOS). This chapter also allows us to further review the FORMAT statement and its capabilities, as well as additional SuRPAS file and record attributes and characteristics

Objectives

To implement input and output activities in Fortran, a programmer should be able to:

- Design code that implements Fortran's intrinsic I/O calls
- Understand what OpenVMS I/O system services are available
- Design code that implements the SuRPAS FALIOS I/O command
- Be able to format data that is to be input and output
- Understand the organization, attributes, and characteristics of SuRPAS database files that are to be manipulated

Fortran I/O

Overview

Fortran, unlike a number of other high level languages, supports input and output within the language statement set. The Fortran compiler translates these I/O statements to appropriate OpenVMS Record Management System (RMS) calls. This section examines the various areas of Open VMS and Fortran necessary to designing code that implements Fortran I/O.

This section discusses:

- Logical Unit Numbers and the LUNASSIGN command
- The Record Management System (RMS) I/O calls
- Fortran Unformatted and Formatted I/O
- Formatting Input and Output
- The FORMAT statement
- Error and end-of-file conditions

The OpenVMS Record Management System

Logical Unit Numbers

The Logical Unit Number (LUN) or Logical Unit Specifier is an integer expression with a value that refers to a specific file or I/O device. On Alpha processors, the integer is in the range of 0 through ((2**31) -1).

A logical unit number is assigned to a file or device in one of two ways:

- Explicitly through an OPEN statement
- Implicitly by the Operating System

If you use the READ statement to implicitly open a file, the file is opened as STATUS='OLD'. If you use the WRITE statement to implicitly open a file, the file is opened as STATUS='NEW'.

RMS Data Structures

RMS system services have many capabilities and options that are stored in several large data structures. These data structures provide arguments to the RMS I/O routines and are used as the basis for many of the FALIOS data structures. The RMS data structures are:

- **File Access Block (FAB)** - used to describe and store general file information.
- **Record Access Block (RAB)** - used to describe and store information about records in a file.
- **Name Block (NAM)** - used to give supplementary information about the name of files beyond that provided in the FAB.
- **Extended Attributes Blocks (XABs)** - a family of related blocks that are linked to the FAB or RAB to communicate to RMS any file attributes beyond those expressed in the FAB.

The RMS data structures are used both to pass arguments to RMS services and to return information from RMS services to the calling program.

FORTRAN-77 I/O Calls

This section discusses the basic Fortran input and output (I/O) statements supported by Compaq Alpha Fortran; those that open and close files; and those that initiate data transfers. I/O statements have three basic components: the statement keyword, the control list, and the I/O list.

Although most SuRPAS I/O is accomplished through the use of FALIOS (SuRPAS file I/O system) calls discussed later in this chapter, some Fortran I/O calls are still used. The READ, WRITE, and FORMAT statements are currently used and supported for some SuRPAS I/O activities and the I/O error status discussion will be helpful when debugging your code. The OPEN and CLOSE statements are included for completeness, but are rarely used in SuRPAS code. FALIOS calls are used to open and close all files.

The OPEN Statement

The OPEN statement connects an existing file to a logical unit number or creates a new file and connects it to a logical unit. It can also specify file attributes that control file creation and subsequent processing. The OPEN statement has the following syntax:

```
OPEN (par [, par] ...)
```

The ***par*** arguments specify keyword phrases (**keyword= value**) that provide information about how a file is to be opened. The following is a sampling of functionality available:

Unit and file indentifications:

•	FILE =	file-spec	- File specification (file name)
•	STATUS =	‘NEW’ ‘OLD’ ‘UNKNOWN’ ‘SCRATCH’	- File existence status at OPEN
•	UNIT =	lun	- Logical Unit Number to be used (INTEGER*4)

File processing:

•	ACCESS =	‘SEQUENTIAL’ ‘DIRECT’ ’KEYED’ ‘APPEND’	- Fortran access method to be used
•	ORGANIZATION		- Logical file structure

Records in a file:

• FORM =	‘FORMATTED’ ‘UNFORMATTED’	- Type of Fortran record formatting
• RECL =		- Logical record length - a number (expressed in bytes or longwords)

- RECORDTYPE - Logical record format (‘FIXED’, ‘VARIABLE’, etc.)

Optional keywords:

• ERR =	statement label	-control is transferred to if an error occurs
• IOSTAT =	status value	- indicates an error condition

OPEN Statement Examples

The following statement creates a new sequential formatted file on unit #1 with the default file name FOR001.DAT.

```
OPEN (Unit=1, STATUS=’NEW’, ERR=100)
```

The following statement creates a fixed-length, sequential file on unit #3 with the name MYDATA.DAT.

```
OPEN (Unit=3, STATUS=’NEW’,
1 ORGANIZATION=’SEQUENTIAL’, RECORDTYPE= ‘FIXED’,
2 FILE=’MYDATA.DAT’, ERR=3000)
```

The READ Statement

The READ statement transfers input data to internal memory from records contained in external files or devices, or to internal memory from internal files. Compaq Fortran supports sequential, direct access, indexed or internal files access. There are two forms of the READ statement. The long and short form syntax for the formatted READ statement is:

```
READ (ext-unit, format[, iostatus] [,err=]
1 [,end=]) [iolist]

READ format[, iolist]
```

The long and short form syntax of the list-directed READ statement is:

```
READ (ext-unit, *[, iostatus] [,err=] [,end=])
1 [iolist]

READ *[, iolist]
```

The optional arguments available to the READ statement are:

- **UNIT = ext-unit** - the logical unit number or external unit specifier
- **FMT=format** - the FORMAT statement label that applies to this READ statement
- ***** - is a list-directed format specifier.
- **IOSTAT=iostatus** - is an I/O status variable that will receive the completion status
- **ERR=, END=** - these are label specifiers, that specify what label to jump to when an error or end-of-file event occurs.
- **iolist** - the list of data variables that will receive the data transferred on the READ

The WRITE Statement

The WRITE statement transfers output data from internal storage to user-specified external logical units (such as disks, printers, display devices) or internal files. Sequential WRITE statements can be formatted, list-directed, or unformatted. The syntax for the formatted WRITE statement is:

```
WRITE (ext-unit, format[, iostatus] [,err=])
1  [iolist]
```

The syntax of the list-directed WRITE statement is:

```
WRITE (ext-unit, *[, iostatus] [,err=]) [iolist]
```

The arguments available to the WRITE statement are:

- **UNIT=ext-unit** - the logical unit number or external unit specifier
- **FMT=format** - FORMAT statement label that applies to WRITE statement
- ***** - is a list-directed format specifier.
- **IOSTAT=iostatus** - an I/O status variable that receives completion status
- **ERR=** - label specifier that specifies label to jump to when error event occurs.
- **iolist** - list of data variables that receives data transferred on the WRITE

The PRINT Statement

The PRINT statement transfers output data from internal storage to external records that are sequentially accessed. There are two forms for the PRINT statement. The syntax for the two formats is:

```
PRINT format [, iolist]
PRINT *[, iolist]
```

The arguments available to the PRINT statement are
:

- **format** - the FORMAT statement label that applies to this WRITE statement
- **iolist** - the list of data variables that will receive the data transferred on the WRITE

In the following example, the PRINT statement writes one record to the default output device. The record has four fields of character data.

```
    CHARACTER*16  Name,  Job
    PRINT  400,  Name,Job
400 FORMAT  (‘NAME=’,A16,’JOB=’,A16)
```

The CLOSE Statement

The CLOSE statement disconnects a file from a logical unit. It has the following syntax:

```
CLOSE ([UNIT=] lun [, ERR=label] [,
1  IOSTAT=iostatus])
```

The arguments available to the CLOSE statement are:

- **UNIT=lun** - the logical unit number or external unit specifier
- **IOSTAT=iostatus** - is an I/O status variable that will receive the completion status
- **ERR=label** - this is a label specifier, that specifies what label to jump to when an error event occurs.

```
     PROGRAM  IO_EXAMPLE
C
C Description: This example program provides an example of the
C   OPEN, READ, PRINT, and CLOSE statements.  The progam opens
C   a file, then in a DO loop, reads and displays each record,
C   until it reaches the end of file (END=)
C
     INTEGER*4   Infile_1 /1/  ! LUN
     INTEGER*4   Infile_2 /2/  ! LUN
     INTEGER*4   Outfile  /3/  ! LUN
     INTEGER*4   Open_Status   ! Status of OPEN
     INTEGER*4   Open_Status_Out
     INTEGER*4   Read_Status   ! Status of READ
     INTEGER*4   Write_Status  ! Status of WRITE
     INTEGER*4   Close_Status  ! Status of CLOSE
     INTEGER*4   Close_status_Out
C
     CHARACTER*50  Input_Record   ! Contents of the Read Record
     INTEGER*4   Record_Counter /0/ ! Counts records read
C
1000 CONTINUE
C
     OPEN (UNIT=1,                ! OPEN the WISHLIST2.TXT file
 1        STATUS='OLD',           ! as an existing file
 2        IOSTAT=Open_Status,     ! store the OPEN status
 3        ERR=1100,               ! jump to 1100 on an error
 4        FILE='WISHLIST2.TXT')   ! input file name

     OPEN (UNIT=3,                ! OPEN WISHLIST_NEW.TXT file
 1        STATUS='NEW',           ! as a new file
 2        IOSTAT=Open_Status_Out, ! store the OPEN status
 3        ERR=1105,               ! jump to 1105 on an error
 4        FILE='WISHLIST_NEW.TXT') ! output file name
```

Code Example 20-1 - Program IO_EXAMPLE.FOR

```
      DO WHILE (Read_Status .EQ. 0)

        READ (UNIT  =1,FMT=1010,      ! READ a record
   1      IOSTAT = Read_Status,       ! store the READ status
   2      ERR    = 1125,              ! jump to 1125 on error
   3      END    = 1175) Input_Record ! jump to 1175 end of file
 1010     FORMAT (A50)

        Record_Counter = Record_Counter + 1

        PRINT 1015, Record_Counter, Input_Record
 1015   FORMAT (10X,'Record #',I2,' = ', A50)

        WRITE (UNIT = 3,FMT=1020,     ! WRITE a record
   1      IOSTAT = Write_Status,      ! store the READ status
   2      ERR    = 1125) Input_Record ! jump to 1125 on error
 1020     FORMAT (A50)

      END DO  ! Read_Status .EQ. 0

      CLOSE (1, IOSTAT = Close_Status,    ERR = 1150)
      CLOSE (3, IOSTAT = Close_Status_Out, ERR = 1160)
      GO TO 8000

 1100 CONTINUE
 1105 CONTINUE
      PRINT 1120
 1120 FORMAT (' Error on OPEN Statement')
      GO TO 8000
 1125 CONTINUE
      CLOSE (UNIT=1)
      PRINT 1130
 1130 FORMAT (' Error on the READ statement')
      GO TO 8000
 1150 CONTINUE
 1160 CONTINUE
      PRINT 1170
 1170 FORMAT (' Error on Close Statement')
      GO TO 8000
 1175 CONTINUE
      CLOSE (UNIT=1)
      PRINT 1180
 1180 FORMAT (' End of File Reached...File has been CLOSED!')

 8000 CONTINUE

     END  ! of Program IO_EXAMPLE
```

Code Example 20-1 - Program IO_EXAMPLE.FOR (continued)

Formatting Input and Output

Formatting I/O statements specify the form of data being transferred. They also specify the data conversion (editing) required to achieve that form. The primary formatting statement is the FORMAT statement: a nonexecutable statement used in conjunction with formatted I/O statements and ASSIGN statements.

The syntax of the FORMAT statement is:

```
FORMAT (rt f1, f2, ...fn, rt)
```

The **rt** is a record terminator; zero or more slashes (/). The **fn** is a field descriptor, an edit descriptor, or a group of field or edit descriptors enclosed in parentheses. Each field or edit descriptor is separated by a comma delimiter. The entire list of field descriptors, edit descriptors, and record terminators is called the format specification and is surrounded in a pair of parentheses.

FORMAT Codes

The following is a partial list of formatting codes and their effect:

Code	Form	Effect
A	A[int]	Transfers character values.
F	**Fw.d**	**Transfers REAL values. The *.d* value indicates the minimum number of fractional positions**
I	I<int>[.min]	Transfers decimal (base-10), integer values. An optional *.min* value can be use to indicate the minimum integer positions to be used (zero-filled)
L	**L<int>**	**Transfers logical data: on input, this transfers characters; on output, this transfers T or F.**
P	nP	Alters the locations of decimal points.
T	**Tn**	**Specifies positional tabulation.**
X	nX	Specifies that n characters are to be skipped (spaces).
$	**$**	**Suppresses trailing carriage return during interactive I/O**

General Rules for Writing FORMAT Statements

This section summarizes the rules for constructing and using format specifications and their components in FORMAT statements.

- A FORMAT statement must always be labeled.
- In a field descriptor, the integer value must be an unsigned integer constant.
- In a field descriptor, the integer value representing the number of decimal positions must be an unsigned integer. The decimal point is also required.
- If the associated I/O statement contains an I/O list, the format specification must contain at least one field descriptor (I, O, F, L, or A).

Fortran Run-time Errors

Fortran status errors, also known as the condition status code, defines the type of error that occurred during run-time processing. The following is a partial list of those errors.

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR$NOTFORSPE	1	F	not a FORTRAN-specific error
FOR$_BUG_CHECK	8	F	internal consistency check failure
FOR$_SYNERRNAM	17	F	syntax error in NAMELIST input
FOR$_TOOMANVAL	18	F	too many values for NAMELIST variable
FOR$_INVREFVAR	19	F	invalid ref. to var. in NAMELIST input
FOR$_REWERR	20	F	REWIND error
FOR$_DUPFILSPE	21	F	duplicate file specifications
FOR$_INPRECTOO	22	F	Input record too long
FOR$_BACERR	23	F	BACKSPACE error
FOR$_ENDDURREA	24	F	end-of-file during read
FOR$_RECNUMOUT	25	F	record number outside of range
FOR$_OPEDEFREQ	26	F	OPEN or DEFINE FILE required
FOR$_TOOMANREC	27	F	too many records in I/O statement
FOR$_CLOERR	28	F	CLOSE Error
FOR$_FILNOTFOU	29	F	File not found
FOR$_OPEFAI	30	F	open failure
FOR$_MIXFILACC	31	F	mixed file access modes
FOR$_INVLOGUNI	32	F	invalid logical unit number
FOR$_ENDFILERR	33	F	ENDFILE error
FOR$_UNIALROPE	34	F	unit already open
FOR$_SEGRECFOR	35	F	segmented record format error
FOR$_ATTACCNON	36	F	attempt to access non-existant record
FOR$_INCRECLEN	37	F	inconsistent record length
FOR$_ERRDURWRI	38	F	error during write
FOR$_ERRDURREA	39	F	error during read
FOR$_RECIO_OPE	40	F	recursive I/O operation
FOR$_INSVIRMEM	41	F	insufficient virtual memory
FOR$_NO_SUCDEV	42	F	no such device
FOR$_FILNAMSPE	43	F	file name specification error
FOR$_INCRECTYP	44	F	inconsistent record type
FOR$_KEYVALERR	45	F	keyword value error in OPEN statement
FOR$_INCOPECLO	46	F	inconsistent OPEN/CLOSE parameters
FOR$_WRIREAFIL	47	F	write to READONLY file
FOR$_INVARGFOR	48	F	invalid argument to FORTRAN R.T.L.
FOR$_INVKEYSPE	49	F	invalid key specification

FORTRAN Condition Symbol	Error Number	Severity	Message Text
FOR$_INCKEYCHG	50	F	inconsistent key change or duplicate key
FOR$_INCFILORG	51	F	inconsistent file organization
FOR$_SPERECLOC	52	F	specified record locked
FOR$_NO_CURREC	53	F	no current record
FOR$_REWRITERR	54	F	REWRITE error
FOR$_DELERR	55	F	DELETE error
FOR$_UNLERR	56	F	UNLOCK error
FOR$_FINERR	57	F	FIND error
FOR$_FMYSYN	58	I	format syntax error at or near xxx
FOR$_LISIO_SYN	59	F	list-directed I/O syntax error
FOR$_INFFORLOO	60	F	Infinite format loop
FOR$_FORVARMIS	61	F	format/variable-type mismatch
FOR$_SYNERRFOR	62	F	syntax error in format
FOR$_OUTCONERR	63	E	output conversion error
FOR$_INCONERR	64	F	input conversion error
FOR$_OUTSTAOVE	66	F	output statement overflows record
FOR$_INPSTAREQ	67	F	input statement requires too much data
FOR$_VFEVALERR	68	F	variable format expression value error
MTH$_WRONUMARG	80	F	wrong number of arguments
MTH$_INVARGMAT	81	F	invalid argument to math library
MTH$_UNDEXP	82	F	undefined exponentiation
MTH$_LOGZERNEG	83	F	logerithm of zero or negative value
MTH$_SQUROONEG	84	F	square root of negative number
MTH$_SIGLOSMAT	87	F	significance lost in math library
MTH$_FLOOVEMAT	88	F	floating overflow in math library
MTH$_FLOUNDMAT	89	F	floating underflow in math library
FOR$_ADJARRDIM	93	F	adjustable array dimension error
FOR$_INVMATKEY	94	F	invalid key match specifier for key dir.
FOR$_FLOCONFAI	95	E	floating point conversion failed
FOR$_UNFIO_FMT	256	F	unformatted I/O to unit open for formatted transfers
FOR$_FMTIO_UNF	257	F	formatted I/O to unit open for unformatted transfers
FOR$_DIRIO_KEY	258	F	direct-access I/O to unit open for keyed access
FOR$_SEQIO_DIR	259	F	sequential-access I/O to unit open for direct access
FOR$_KEYIO_DIR	260	F	keyed-access I/O to unit open for direct access
FOR$_OPEREQSEQ	265	F	operation requires sequential file organization and access

Accessing SuRPAS Database Files

Overview

SuRPAS database files are supported by a series of descriptions, definitions, and record layouts. This support information can be found in files with the same name as the database file it is supporting, but with a different extension and directory location

This section discusses:

- Common File Definition files (CFDs)
- Common Record Definition files (CRDs)
- File Definition Layout files (FDLs)

Database File Layouts

File attributes for each database file are described in a file definition layout file (.FDL). These files have the name of the database file with the "***.FDL***" filetype and are located in the SuRPAS directory FAL$FILES. The files are ASCII files and contain file organization information as well as the primary and alternate keys (if the file is indexed sequential),

Sequential Database File Definition Layouts (FDLs)

Some of the SuRPAS database files are built as fixed-length sequential files. The file definition layouts for these database files include:

- Database file name
- File organization and format (fixed-length, sequential)
- Record size

The example below (example 20-2) shows the **`JNLFAL.FDL`**, the file definition layout for the journal database file.

```
TITLE    "FDL for Current Year Transaction History file
(FAL$DATJNL:JNLFAL.DAT)"

IDENT    "20-DEC-1993 20:54:24    VAX-11 FDL Editor"

SYSTEM
        SOURCE                   VAX/VMS

FILE
        ALLOCATION               4096
        BEST_TRY_CONTIGUOUS      yes
        EXTENSION                4096
        NAME                     "FAL$DATJNL:JNLFAL.DAT"
        ORGANIZATION             sequential

RECORD
        BLOCK_SPAN               yes
        CARRIAGE_CONTROL         none
        FORMAT                   fixed
        SIZE                     876
```

Example 20-2 - JNLFAL File Definition Layout File (FDL)

Indexed Sequential Database File Definition Layouts (FDLs)

Most of the SuRPAS database files are built as indexed sequential files. The file definition layouts for these database files include:

- Database file name
- File organization and format (fixed-length, indexed)
- Record size
- Primary and Alternate Key Information
  - Changes (yes/no)
  - Data Area
  - Data Fill
  - Duplicates Allowed (yes/no)
  - Index Area
  - Index Fill (in bytes)
  - Level Index Area
  - Name of field(s) in key (e.g. “LEDACT//LEDFND”)
  - Prologue
  - Segment Information (a set for each named key in the index)
    - √ Segment Length (in bytes)
    - √ Segment Position (offset from beginning of record)
  - Key data value type (e.g. string)

The example on the next page (example 20-3) shows portions of **`JLEDFAL.FDL`**, the file definition layout for the journal database file.

```
TITLE   “FDL FOR FAL$DATLED:LEDFAL.DAT”

IDENT   “ 3-APR-1991 17:03:23   VAX-11 FDL Editor”

SYSTEM
        SOURCE                  VAX/VMS

FILE
        NAME                    “FAL$DATLED:LEDFAL.DAT”
        ORGANIZATION            indexed

RECORD
        CARRIAGE_CONTROL        none
        FORMAT                  fixed
        SIZE                    800

AREA 0
        ALLOCATION              1024
        BEST_TRY_CONTIGUOUS     yes
        BUCKET_SIZE             8
        EXTENSION               1024

KEY 0
        CHANGES                 no
        DATA_AREA               0
        DATA_FILL               80
        DUPLICATES              no
        INDEX_AREA              1
        INDEX_FILL              100
        LEVEL1_INDEX_AREA       1
        NAME                    “LEDACT//LEDFND”
        PROLOG                  3
        SEG0_LENGTH             14
        SEG0_POSITION           0
        SEG1_LENGTH             6
        SEG1_POSITION           14
        TYPE                    string

KEY 1
        CHANGES                 no
        DATA_AREA               2
        DATA_FILL               100
        DUPLICATES              no
        INDEX_AREA              3
        INDEX_FILL              80
        LEVEL1_INDEX_AREA       3
        NAME                    “FUND ID // ACCOUNT”
        SEG0_LENGTH             6
        SEG0_POSITION           14
        SEG1_LENGTH             14
        SEG1_POSITION           0
        TYPE                    string
```

Example 20-3 - LEDFAL File Definition Layout File (FDL)

SuRPAS Database File Keys

Primary and alternate keys are used when sorting and searching for information in SuRPAS database files. The following is a list of keys for some of the SuRPAS database files. Although keys are not often added or changed, please be careful when using this list. It may not represent the version of SuRPAS that you are currently associated with. To be certain, check the appropriate FDL files in the version of SuRPAS that you are working with.

MASFAL (Shareholder Master File)

• Primary Key	MASACT
• Key 1	MASSNA // MASACT
• Key 2	MASTAX // MASACT
• Key 3	DLR/BRN/REP/DLA/SNA/ACT
• Key 4	TIM/TMA/SNA/ACT
• Key 5	AFF/AFA/SNA/ACT
• Key 6	CON/ACT
• Key 7	ROA/ACT
• Key 8	MASSKY/MASACT
• Key 9	CSR/SNA/ACT
• Key 10	MASSTA/MASACT
• Key 11	MASDLR/MASDLA/MASACT

ACTFAL (Shareholder Account Information File)

• Primary Key	LEDACT//LEDFND
• Key 1	

LEDFAL (Fund Ledger File)

• Primary Key	LEDACT//LEDFND
• Key 1	FUND ID // ACCOUNT

Database File and Record Definition Files

Database files are supported by common file and record definitions. These definitions, which detail each field in the record and file are stored in the logical directory **FAL$LIBRARY**. Use the DCL commands LIBRARY/LIST to get a listing of all of the common files and LIBRARY/EXTRACT or WorkBench to get a copy of a specific definitions file.

Database File Definitions (CFDs and CRDs)

There is a common file definitions file (CFD) for each database file. This contains the attributes (file name, size, offset, and data type) for each field in the file. The files are written as a FORTRAN INCLUDE files and are stored in an OpenVMS library file named **FAL$LIBRARY:FILES.TLB.** Each field is defined as a FORTRAN data variable of size and type. Each field is an entry to a FORTRAN EQUIVALENCE statement that defines the field's offset in the record/file. A simple example of this is ACTFAL.CFD, shown below in Example 20-4.

Common Record and Report Definitions (CRDs) files are also built as FORTRAN INCLUDE files and are stored in a library file named **FAL$LIBRARY:CARDS.TLB**. A simple example of this is SRFREFFEE.CRD, shown on the next page in Example 20-5..

```
C               CFD: ACTFAL.CFD              - OLD-NEW Account Xref
C
C         Generated: 11-JUN-96  13:24:53        BY RXF
C
        CHARACTER*20    ACTOLD          ! OLD SYSTEM ACCOUNT NUMBER
        CHARACTER*7     ACTOFD          ! OLD SYSTEM FUND ID
        CHARACTER*4     ACTMGT          ! OLD SYSTEM MANAGEMENT CODE
        CHARACTER*14    ACTNEW          ! NEW SYSTEM ACCOUNT NUMBER
        CHARACTER*3     FILLA           ! FILLER
C
        CHARACTER*48   ACTIOB              ! BUFFER FOR IOS CALLS
C
        INTEGER*4 ACTIOB_LEN
        PARAMETER (ACTIOB_LEN = 48   )
C
C
        EQUIVALENCE (ACTIOB(    1:   20), ACTOLD )
        EQUIVALENCE (ACTIOB(   21:   27), ACTOFD )
        EQUIVALENCE (ACTIOB(   28:   31), ACTMGT )
        EQUIVALENCE (ACTIOB(   32:   45), ACTNEW )
        EQUIVALENCE (ACTIOB(   46:   48), FILLA )
C
```

Example 20-4 - ACTFAL Common File Definition File (CFD)

```
C       Common File Description :              SRFREFFEE.CRD
C
C       TEMP FILE FOR SRF REFERRAL FEE REPORT - FAL$SCRATCH:REFFEE.DAT
C       Data from YTDFAL and JNLFAL selected by Trade Date, Timer, Dealer, and
C       Cycle Code.  Used to produce Report #????.
C
C       -------------------------MODIFICATION  HISTORY-------------------------
C       SWS  05/20/96  V4.5off  5969   013092
C            Inception
C       SWS  06/19/96  V4.5off  6269   013092
C            ADDED NEW FIELD FND
C       -------------------------------------------------------------------<EOM>
C
      INTEGER         REFFEE_LEN
      PARAMETER       (REFFEE_LEN = 150)
C
      STRUCTURE /REFFEE/
      UNION
          MAP
            CHARACTER*1     PRT             ! Indicates whether trade is to
C                                                be printed ('Y'=print)
            CHARACTER*5     BRK             ! Dealer (from MASDLR)
            CHARACTER*15    REP             ! Rep (from MASREP)
            CHARACTER*5     BRN             ! Branch (from MASBRN)
            CHARACTER*5     TIM             ! Allocation Table (from MASTIM)
            CHARACTER*8     POS             ! Post Date
C
            CHARACTER*14    ACT             ! Account #
            CHARACTER*6     FND             ! Fund Name
            CHARACTER*8     TRD             ! Trade Date
C
            CHARACTER*3     CYC             ! Cycle Code
            CHARACTER*3     COD             ! Tran. Code
            CHARACTER*2     BIN             ! Tran. Code BIN (first 2 chars)
            CHARACTER*10    RVS             ! Original Serial # Reversed
            REAL*8          SHR             ! Share Amount
            REAL*8          BAL             ! Balance
            REAL*8          RCV             ! Amount Received
            REAL*8          GRS             ! Gross Amount
            REAL*8          NAV             ! Price
            REAL*8          OLD_SHR         ! Old Shares (held > 1 year)
            REAL*8          NEW_SHR         ! New Shares (held < 1 year)
            REAL*8          NEW_DLR         ! New Dollars ($ value of New
C                                                Shares on the day purchased)
            CHARACTER*1     PAD
          END MAP
          MAP
            CHARACTER*(REFFEE_LEN)  IOB
          END MAP
      END UNION
      END STRUCTURE
C
      RECORD /REFFEE/ REFFEE
```

Example 20-5 - SRFREFFEE.CRD Common Record/Report Definition File (CRD)

SuRPAS I/O in Fortran

Overview

SuRPAS supports Fortran input and output through a series of subroutines that make up the FAL Input and Output System (FALIOS). These routines were developed within PFPC to handle all file processing for standard SuRPAS files as well as other supporting files. FALIOS is the I/O layer of the SuRPAS system.

FALIOS is made up of six Fortran-callable subroutines. These routines initialize I/O (FALINI), open files (FALEXTOPN, INIFOPBLK), handle I/O (FALIOS), and close files (FALCLS, FALCLS1).

This section discusses:

- The FAL Input and Output System (FALIOS)
- Initializing and Opening SuRPAS and generic files
- Issuing calls for file I/O
- Closing SuRPAS and generic files

An Introduction to the SuRPAS I/O System

The purpose of the FAL Input and Output System (FALIOS) is to bury the Fortran I/O statements in a set of I/O routines that address the specific needs of SuRPAS I/O. It is the I/O layer of the SuRPAS system. It is made up of six subroutines that initialize and carry out all I/O.

The FALIOS routine is a single Fortran subroutine that can be used to access both standard SuRPAS routines as well as generic files.

- Variable names are defined in named common areas.
- File Identification numbers (FID's) created in the File maintenance section of SuRPAS Workbench, are used to describe standard SuRPAS data files. These records are stored in FAL$LIBRARY:FILDIC.CFD, the file dictionary. Each record contains the FID, the file name, the file location, and the file length.
- CFD's that contain information.

Initializing the SuRPAS I/O System

The FALINI Call

The FALINI routine initializes the SuRPAS I/O system in each program. It is called at the beginning of each program and is only called once, from the main program unit. The FALINI routine has the following syntax:

```
CALL FALINI ()
```

FALINI initializes the LUNTBL table structures.

- FILDIC.CFD is loaded into a common array.
  ```
  COMMON / LUNAMS_COM / LUNAMS
  ```
- COMMAREA.CFD is initialized.
  ```
  COMMON / COMMAREA / ...
  ```
- FALCOM.CFD is initialized.
  ```
  COMMON / FALCOM / ...
  ```

Opening Non-standard SuRPAS Files

The FALEXTOPN Call

FALEXTOPN is a subroutine used to explicitly open files for the I/O system. It defines external (non-standard) SuRPAS files. The run-time parameter needed for the file open is not available implicitly via FALIOS.

FALEXTOPN is normally used when you want to open a user-defined file for temporary usage. There are times when you want to open a standard file with a different access, such as:

- Bucket read size change
- Read using a different index (alternate)
- VMS sort
- Open sequential for performance improvement

An example of the FALEXTOPN call and its associated error message call is found below:

```
CALL FALEXTOPN (TMP,MASLUN,LUNTEL,FOPBLK,RETIOS)
IF (RETIOS .NE. 0) CALL MFSERR (RETIOS)
```

The File ID is assigned on the fly. It is deleted when the file is closed. The file ID is created in the File maintenance section of SuRPAS Workbench. The file information is stored in FILDIC.CFD.

The INIFOPBLK routine is called once for each file opened by FALEXTOPN.

The INIFOPBLK Routine

INIFOPBLK initializes the FOPBLK. It is called once each time FALEXTOPN is called. The format of the INIFOPBLK routine is:

```
CALL  INIFOPBLK  (<fopblk>)
```

where the ***fopblk*** argument refers to the file open block of the file to be opened. The format of this structure is defined in the FOPBLK.CFD module found in FAL$LIBRARY:FILES.TLB. A copy of this module is found in an example that follows.

Managing SuRPAS File I/O

The FALIOS Routine

The FALIOS subroutine call supports the majority of input and output necessities within your SuRPAS code development activities.

When FALIOS is called there is no information on the file, where it is, or how to find it. The known file ID (stored in FILDIC.CFD) for each SuRPAS file points to data structures that describe the file. This FID is created in the SuRPAS Workbench toolkit, using the FIL option. The FIL option supports the creation and maintenance of file IDs in FILDIC.CFD. The information collected includes the file name, directory location, size in bytes, and of course the file ID (FID). This information is found in FAL$STDFIL:FILSRC.DAT

The FIDBLK structure maintains a field indicating whether the file is already open. If it is, the I/O continues based on the logical unit number (LUN) stored in the FOPBLK. If not, a call to FALOPEN is made to open the file. The file is opened and the FOPBLK and FIDBLKs are updated with the LUN, the file location, the record length, and any other information that RMS needs to know.

The following is the syntax of the call to the FALIOS subroutine:

```
Call Falios ( fid,  luntbl, ‘op code’, ‘hon’,
             ‘prc’, keyrec,  keyval,    ios,
              pariod)
```

The Arguments

FID - is the file ID. It is created within Workbench and contains a Logical Unit Number (LUN).

LUNTBL - is a table of FIDBLK. There is one for each process. It is indexed by the FID. LUNTBL is described in COMMAREA.CFD as:

```
CHARACTER*16 LUNTBL  (600)
```

and is incremented by the FID.

OPC is the operation code

Access Options:

- KEY - key equal
- KEQ - key equal, same as KEY
- KGT - key greater than
- KGE - key greater than or equal to
- GET - direct read of a sequential file
- NEX - read the next record - sequential reads
- PUT - write a record
- UPD - rewrite a modified record
- UNL - unlock a record on a stream
- DEL - delete the current record
- FRE - sys$free - ??????
- RFA - record file address access

HON - record lock honor code - what do I do when I find that the record is locked? Options are:

- RRL - read regardless of the current lock state (read only)
- REA - honor the current lock state
- WAT - wait for a record to be unlocked - loop until it is unlocked - for update operations
- <space> - use for other than a read

PRC - Lock process codes (for read operations only) - do I want to lock the record? Am I just reading or am I planning on updating the record? Options are:

- LCK - lock a record on read
- NLK - do not lock a record on read

KEYREC - for a keyed read, this is the key of reference

0 = primary key
1 = 1st alternate key
2 = 2nd alternate key
... and so on.

KEYVAL - for keyed reads, this is the key value to search with
Generic key matching is implicit based on the length of KEYVAL
For random reads, this is the six character Record File Address (RFA).

IOS - the returned I/O status. This is FOR$IOSTATUS
0 = success
-1 = end of file

PARIOB - the returned data buffer.

Closing Files

The FALCLS Routine

FALCLS is a subroutine to close all files used by FALIOS in a given process. It usually issued when the process is about to exit. When FALCLS is issued in the process, the OpenVMS Record Management Services (RMS) is called to close each of the files. There are no arguments in the FALCLS call. The syntax of the FALCLS routine is:

```
     CALL FALCLS ()
```

The FALCLS1 Routine

FALCLS1 is a subroutine to close an individual file. It is usually used when FALEXTOPN has been used to open a second channel to a file. FALCLS1 is used to close the second channel. It has the syntax:

```
     CALL FALCLS1 (fid, luntbl)
```

It has two arguments:

FID - the file ID
LUNTBL - the table of FIDBLK's

MASLEDEXC - A FALIOS Example

```
C PROGRAM    : MASLEDEXC.FOR
C
C       COPYRIGHT 2001  :       PFPC Global Fund Services
C                               Berwyn, PA  19312
C
C AUTHOR  :  Gary W. Vicoli
C
C DESCRIPTION  :   Training Exercise - display account, fund id,
C                         and share balance for accounts in PA.
C
C INPUT FILES  :        MASFAL, LEDFAL
C
C OUTPUT FILES :        none.
C
C REPORT FILES :       none.
C
C       ------------MODIFICATION  HISTORY-----
C       WHO    DATE    RELEASE  SEQ#    FTK#  NOTE#
C       ===   ========  ======= =====  ======= =====
C       ----------------------------

 PROGRAM MASLEDEXC

 IMPLICIT NONE

C -------------------------------
C COMMON FILE DEFINITIONS:

 INCLUDE '($FORIOSDEF)'
 INCLUDE 'FAL$LIBRARY:FILES(COMMAREA.CFD)'
 INCLUDE 'FAL$LIBRARY:FILES(FILDIC.CFD)'
 INCLUDE 'FAL$LIBRARY:FILES(MASFAL.CFD)'
 INCLUDE 'FAL$LIBRARY:FILES(LEDFAL.CFD)'

C -------------------------------
C LOCAL VARIABLES:

 CHARACTER*2 STATE

C -------------------------------
```

Example 20-6 - MASLEDEXC - A FALIOS Example

```
C VARIABLE ASSIGNMENTS:

 STATE = ‘PA’

C ------------------------------
C FILE CABINET:

 CALL FALINI

C =============================================================
1000 CONTINUE

C /***********************/
C /* Initialization area */
C /***********************/

C ------------------------------
2000 CONTINUE

C /***********************************************/
C /* outer loop to pass master file on state key */
C /***********************************************/

  CALL FALIOS (LUNMAS,LUNTBL,'KEQ','RRL','NLK',10,STATE,IOS,MASIOB)

 IF ((IOS .NE. 0) .AND.
     +      (IOS .NE. FOR$IOS_ATTACCNON)) CALL MFSERR(IOS)
```

Example 20-6 - MASLEDEXC - A FALIOS Example (continued)

```
 DO WHILE (IOS .EQ. 0 .AND. MASSTA .EQ. STATE)

C       /*****************************************************/
C      /* inner loop to pass ledger file for given account */
C       /*****************************************************/

      CALL FALIOS (LUNLED,LUNTBL,'KEQ','RRL','NLK',0,MASACT,IOS,LEDIOB)

      IF ((IOS .NE. 0) .AND.
      +    (IOS .NE. FOR$IOS_ATTACCNON)) CALL MFSERR(IOS)

      DO WHILE (IOS .EQ. 0 .AND. LEDACT .EQ. MASACT)

C   /***************************/
C  /* display required fields */
C   /***************************/

   IF (LEDBAL .GT. 0.0D0) THEN ! suppress zero balance ledgers
       TYPE *, ' ACCOUNT = ', LEDACT,
     +         ' FUND ID = ', LEDFND,
     +         ' SHR BAL = ', LEDBAL
   ENDIF

C   /******************/
C  /* get next ledger */
C   /******************/

   CALL FALIOS (LUNLED,LUNTBL,'NEX',' ',' ',0,MASACT,IOS,LEDIOB)
   IF (IOS .GT. 0) CALL MFSERR(IOS)

      ENDDO   ! end of ledger loop

C      /******************/
C      /* get next master */
C      /******************/

      CALL FALIOS (LUNMAS,LUNTBL,'NEX',' ',' ',0,MASACT,IOS,MASIOB)
      IF (IOS .GT. 0) CALL MFSERR(IOS)

 ENDDO   ! end of master loop

C -------------------------------
9000 CONTINUE

C /***************/
C /* end of MAIN */
C /***************/

 END
```