pdp11

FORTRAN IV-PLUS
Object Time System
Reference Manual

Order No. DEC-11-LFPOA-A-D

digital

FORTRAN IV-PLUS
Object Time System
Reference Manual

Order No. DEC-11-LFPOA-A-D

digital equipment corporation · maynard, massachusetts

First Printing, December 1975

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-10
DECCOMM		TYPESET-11

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual contains detailed information about the FORTRAN IV-PLUS Object Time System (OTS) not contained in the FORTRAN IV-PLUS User's Guide. The material in this manual is not necessary for normal use of FORTRAN IV-PLUS. However, many users do require a more detailed knowledge of the OTS for specialized applications. This manual should be especially useful for programmers interfacing MACRO-11 and FORTRAN routines to the OTS.

INTENDED AUDIENCE

The reader is assumed to be proficient in MACRO and FORTRAN, and to be familiar with all the information presented in the FORTRAN IV-PLUS User's Guide, as well as the operating system's Executive Reference manual and I/O Operations Reference manual.

The material presented here is for information purposes. Internal OTS interfaces are NOT guaranteed to remain constant across releases of FORTRAN IV-PLUS. The use of calls to the OTS identical to the compiled code calls, and the use of the named offsets of the OTS will provide as much release-to-release compatibility as is possible.

DOCUMENTATION CONVENTIONS

Unless otherwise noted, all numeric values are represented in decimal notation. Values in MACRO-11 examples are in octal notation.

Unless otherwise specified, all commands terminate in a carriage return.

Variable information is indicated by lowercase characters, whereas literal information (i.e., must be entered exactly as shown) is indicated by all uppercase characters.

CHAPTER 1

INTRODUCTION

There are two versions of the FORTRAN IV-PLUS (F4P) Object Time System (OTS): one for RSX-11M, and one for RSX-11D and IAS. Either version can be built from the standard distribution kit. The hardware environment required is a PDP-11/45, 11/50 or 11/70 with Floating Point Processor (FP11). Optional hardware supported by the operating system is supported by the OTS.

1.1 OBJECT TIME SYSTEM SUMMARY

The FORTRAN IV-PLUS OTS is a library of assembly language modules that complements the compiled code. The OTS is composed of five principal parts:

1. I/O processing routines.
2. Mathematical functions and system subroutines.
3. Compiled-code support routines.
4. Error and exception-condition processing routines.
5. Tables, buffers and impure storage required by the routines.

1.1.1 I/O Processing Routines

The I/O processing routines are designed as a collection of small modules so that only those modules required by a given FORTRAN source program are actually loaded in the user's task.

An I/O statement produces three types of subroutine calls:

1. An initialization call

   This call sets up the I/O system for the specific I/O requested, opens the specified logical unit if necessary, and declares the I/O system to be active.

2. Element transmission calls (if any)

   Each entity in the I/O list generates a call to the OTS. Each call transmits a single value except in the case of arrays, which are transmitted through a single call.

3. A termination call

   This call completes the I/O operation and declares the I/O system inactive.

For example, the FORTRAN statements:

```
DIMENSION   A(10)
READ (2)    I,A,B
```

are compiled into the code:

```
MOV  #2,-(SP)        ;unit number
JSR  PC,ISU$         ;initialize READ
MOV  #I,-(SP)        ;address of I
JSR  PC,IOAI$        ;transmit integer
MOV  #A$ADB,-(SP)    ;address of ADB for A
JSR  PC,IOAA$        ;transmit array A
MOV  #B,-(SP)        ;address of B
JSR  PC,IOAR$        ;transmit real
JSR  PC,$EOLST       ;end-of-list
```

The various routines are described in Chapter 3.

1.1.2 Mathematical Functions and System Subroutines

The mathematical routines, called processor-defined functions (PDF's), are called from the compiled code using special names as described in the FORTRAN IV-PLUS User's Guide. The algorithms used for the mathematical library routines are described in Appendix B of the FORTRAN IV-PLUS User's Guide.

The system subroutines are described in Appendix D of the FORTRAN IV-PLUS User's Guide.

1.1.3 Compiled-Code Support Routines

These routines complement the compiled code by performing operations too complicated or cumbersome to perform with in-line code. Examples are array subscript checking, exponentiation and complex arithmetic.

1.1.4 Error Processing Routines

Errors detected by the OTS are signaled by executing a TRAP instruction with the error number in the low byte of the instruction. Floating point processor asynchronous traps are processed by a service routine within the error processing modules.

There are two methods of error recovery: an 'ERR=' transfer within an I/O statement, or a return to the error site for appropriate action. The actual action taken is determined by a byte within the OTS impure storage. Each defined error number has a corresponding error control byte which may be accessed using the FORTRAN-callable subroutines, ERRSET, ERRTST and ERRSNS.

1.2 CONVENTIONS AND STANDARDS

The following sections describe procedural and naming conventions used in the FORTRAN IV-PLUS system.

1.2.1 Registers

The processor general registers are referred to as:

```
R0 - R5  =  Register 0-5
SP       =  Register 6
PC       =  Register 7
```

The Floating Point Processor accumulators 0-5 are referred to as F0-F5.

1.2.2 Calling Sequences

Four different calling sequence conventions are used by the FORTRAN IV-PLUS compiled code to call various components of the OTS. The calling conventions, and their usage, are described below:

1. R5 Calls - The standard PDP-11 FORTRAN Calling Sequence Convention, used for all system subroutines, most processor-defined functions, and all user routine calls.

2. PC Calls - Used for I/O operations and system dependent routines.

3. R4 Calls - Used for out-of-line, stack-oriented arithmetic routines and certain miscellaneous compiled-code support routines.

4. F0 Calls - Used for faster calls to certain processor-defined functions.

1.2.2.1 R5 Calls - This calling sequence convention is the standard PDP-11 FORTRAN Calling Sequence Convention. It is described in detail in the FORTRAN IV-PLUS User's Guide and is summarized below.

The basic form of the call is

```
; IN INSTRUCTION-SPACE

          MOV #LIST,R5    ;ADDRESS OF ARGUMENT LIST TO
                          ; REGISTER 5
          JSR PC,SUB      ;CALL SUBROUTINE
          ...

;IN DATA-SPACE

LIST:     .BYTE N,0       ;NUMBER OF ARGUMENTS
          .WORD ADR1      ;FIRST ARGUMENT ADDRESS
          ...
          .WORD ADRN      ;N'TH ARGUMENT ADDRESS
```

Note that the argument list must reside in Data-space and that except for label type arguments, all addresses in the list must also refer to Data-space.

Also note that the byte at address LIST+1 should be considered undefined and not referenced. This byte is reserved for use as defined by DIGITAL.

Control is returned to the calling program by restoring (if necessary) the stack pointer (SP) to its value on entry and executing

```
RTS PC
```

Function subprograms return a single result in the processor general registers. The register assignments for returning the different variable types are listed below:

Type	Result in
INTEGER*2 LOGICAL*1 LOGICAL*2	R0
INTEGER*4 LOGICAL*4	R0 -- low order result R1 -- high order result
REAL	R0 -- high order result R1 -- low order result
DOUBLE PRECISION	R0 -- highest order result R1 -- R2 -- R3 -- lowest order result
COMPLEX	R0 -- high order real result R1 -- low order real result R2 -- high order imaginary result R3 -- low order imaginary result

A calling program must save any values in general purpose registers R0 through R5 which it requires after a return from a subprogram. The argument list pointer value in register R5 can not be assumed to be valid after return. Any floating point registers in use by a calling program must also be saved and restored by the calling program. The calling program can not assume that the floating point status bits I/L (integer/long integer) or F/D (floating/double precision) are restored by the called routine.

Null arguments are represented in an argument list by using an address of -1 (177777 octal). This address is chosen to assure that the use of null arguments, in subprograms that are not prepared to handle them, will result in an error when the routine is called. The errors most likely to occur are illegal memory reference and/or word reference to odd byte address.

1.2.2.2 PC Calls - These calls receive all arguments on the stack, are called with a JSR PC,xxx, and return with the arguments deleted from the stack. Registers R0-R5, F0-F5 and the FPP status register are unmodified.

All I/O statements except OPEN and CLOSE are implemented by one or more calls using this convention. The STOP, PAUSE, Computed GO TO and Assigned GO TO statements use this convention. Array subscript checking, if enabled, and task initialization use this convention.

Example:

The FORTRAN statement:

```
REWIND   3
```

is compiled into the code:

```
MOV  #3,-(SP)        ;unit number
JSR  PC, REWI$       ;REWIND processor
```

1.2.2.3 R4 Calls - This convention is used for out-of-line, stack-oriented arithmetic routines and other compiled-code support. These routines receive argument values on the stack, or a pointer to an argument value as an in-line argument immediately following the call. The stack arguments are deleted and a result value is returned on the stack. The routine is called with a JSR R4,xxx instruction. Registers R0-R4, F0-F5 and the FPP status register are assumed to be modified by R4 calls. R5 is preserved. The modules that use this convention are described in Chapter 7.

Example:

The FORTRAN statement:

```
x=A**I
```

is compiled into the code:

```
MOV  A+2,-(SP)       ;push A
MOV  A,-(SP)
JSR  R4, PWRIC$      ;compute A**I
.WORD   I            ;address of I
MOV  (SP)+,X         ;store at X
MOV  (SP)+,X+2
```

1.2.2.4 F0 Calls - This convention is used for some commonly used processor-defined functions. The FPP F/D status bit is set to the type of argument and the argument is loaded into F0. The routine is called with a JSR PC,xxx instruction. The result value is returned in F0 with the FPP F/D status bit preserved. Registers R0-R5, F1-F5 and the FPP I/L status bit are assumed to be modified. The functions that use this convention are named $$xxxx and are described in Section 6.1.

Example:

The FORTRAN statement:

```
Y = SIN(X)
```

is compiled into the code:

```
SETF       ;set FPP mode
LDF  X,F0
JSR  PC,$$SIN
STF  F0,Y
```

1.2.2.5 Special Call Conventions - In addition to the four general calling conventions, special variants are used in the following cases:

OPEN (OPEN$) and CLOSE (CLOSE$) statements use the R5 convention with a special argument list encoding.

Object-time format compilation (FMTCV$) uses a PC call but returns a stack result, which is used in a subsequent I/O initialization call.

Adjustable array initialization (MAK1$, MAK2$, and MAKN$) uses a PC call but preserves only R5.

Traceback name initialization (NAM$) uses a co-routine call.

These special usages are described in detail in the corresponding module descriptions.

1.2.3 Data Formats

The data formats are described in the FORTRAN IV-PLUS User's Guide.

1.2.4 Labeling Conventions

All OTS routines have a title that begins with '$' followed by the name or a contraction of the name. All external entry point names contain a '$' as either the first or last character.

1.2.5 Context Save and Restore

The OTS register context conventions are determined by the calling sequence used, as described in Section 1.2.2.

Internal to the OTS, various conventions are used. In general the calling routine saves those registers it requires.

1.3 COMPATIBILITY WITH FORTRAN IV (FOR)

The OTS's for F4P and FOR are similar but NOT identical and cannot be used together in the same task. They do however have identical FFDB (see Section 2.3) definitions and some common modules. The impure areas and errors are defined to have identical values for identical use (i.e., offset VARAD is the list element address in both systems, and error ICERR is the input conversion error code in both systems.) Some FORTRAN IV-PLUS OTS modules contain conditional assembly code for FORTRAN IV. This code is conditionalized on the definition of symbol F4.

1.4 ASSEMBLY LANGUAGE INTERFACING TO THE OTS

The following short sections provide a brief summary of the methods of interfacing a MACRO-11 program to the FORTRAN IV-PLUS OTS.

1.4.1 Writing a FORTRAN Main Program in Assembly Language

The following MACRO-11 code represents a "canonical" FORTRAN main program:

```
START::

        JSR     PC, OTI$        ; initialize the OTS and FCS
          .
          .
          .
        MOV #^R<.MA>,-(SP)      ; first 3 letters of name in RADIX-50
        MOV #^R<IN.>, R4        ; last 3 letters of name in RADIX-50
        JSR     R4, NAM$        ; initialize traceback chain if desired
          .
          .
          .

        JSR     PC, EXIT$       ; close files and exit
        .GLOBL    $OTSVA        ; link in the impure area
        .GLOBL    RCI$          ; floating point format conversions
        .GLOBL    LCI$          ; logical format conversions
        .GLOBL    ICI$          ; integer format conversions
        .END    START
```

The call to OTI$ initializes the FPP (SFPA$S), the SST vector (SVTK$S) and FCS (FINIT$). The reference to $OTSVA loads the FORTRAN impure area. The references to the format conversion routines are required only if the desired conversion routine is required. Note that a FORTRAN subprogram that contains a FORMAT statement will contain the required format conversion references.

1.4.2 Linkage to the FORTRAN Impure Storage Area

The FORTRAN impure area defines a global symbol ($OTSVA) that is referenced by the compiled code in FORTRAN main programs (but not subprograms!). When the Task Builder processes a reference to this symbol, it loads the FORTRAN impure area and causes an additional global symbol ($OTSV) to be defined in the task that contains the address of the symbol $OTSVA. All of the FORTRAN OTS routines obtain the address of the impure area by referencing the contents of the location $OTSV; see the discussion of $AOTS macro in Section 9.2.4.

CHAPTER 2

DATA BASE

The information used and manipulated by the OTS is maintained in two major areas of impure storage: the work area and the device table.

The work area contains two kinds of information: task-specific data, such as address pointers, and information on the currently active operation, such as a direct access record number.

The device table contains a block of storage for each logical unit allocated to the FORTRAN OTS. This block contains all the information the OTS requires to perform I/O to the unit.

2.1 PROGRAM SECTION DESCRIPTIONS

This section describes the program sections (PSECTs) used by the OTS. PSECTs are named segments of code or data. The attributes associated with each PSECT direct the Task Builder when constructing an executable task image.

2.1.1 $$OTSI -- OTS Instructions

This PSECT contains all of the executable code in the OTS except the formatted and list-directed I/O processors. This PSECT has the attributes: RW,I,CON,LCL.

2.1.2 $$OTSD -- OTS Pure Data

This PSECT contains all of the read-only pure data in the OTS except the formatted and list-directed I/O data. This PSECT contains constants and dispatch tables used by the code in $$OTSI. It has the attributes: RW,D,CON,LCL.

2.1.3 $$AOTS -- OTS Impure Storage

$$AOTS contains the FORTRAN work area impure storage associated with each task. It must be contained in the task's root segment and is pointed to by the contents of global symbol $OTSV. A detailed description is contained in Appendix A. All references in this manual to "the work area" or "the FORTRAN work area" apply to this PSECT, which has the attributes: RW,D,CON.

2.1.4 $$DEVT -- Logical Unit Device Table

$$DEVT defines the FORTRAN logical unit device table. The entries in this table are fixed-length FORTRAN File Descriptor Blocks (FFDBs). An FFDB is composed of a File Control Services (FCS) FDB and a 6-word header for FORTRAN usage. At task startup, the actual number of FFDBs available to the FORTRAN task is determined from the size of $$DEVT. This area is pointed to by the value of offset W.DEV in the work area. This PSECT has the attributes: RW,D,OVR.

2.1.5 $$IOB1 -- User Record Buffer

$$IOB1 defines the FORTRAN user record buffer. The length is determined at task build time by the MAXBUF keyword; the default value is 132 (decimal) bytes. This area is pointed to by offsets W.BFAD (start address) and W.BEND (end address+1) in the work area and its length is computed at task initialization and stored at offset W.BLEN in the work area. This PSECT has the attributes: RW,D,OVR.

2.1.6 $$OBF1 -- Object-Time Format Buffer

$$OBF1 defines the FORTRAN object time format buffer. The length is determined at task build time by the FMTBUF keyword; the default value is 64 (decimal) bytes. This area is pointed to by offsets W.OBFL (start address) and W.OBFH (end address+1) in the work area. This PSECT has the attributes: RW,D,OVR.

2.1.7 Format Conversion PSECTs

The formatted and list-directed I/O processors minimize task size by loading only those format conversion modules referenced by the user's format specifications. Each module is in an independent PSECT and places a pointer to itself in a special PSECT used as a dispatch table. These PSECTs have the global (GBL) attribute to ensure that this collection of modules will be placed in the lowest common seqment of an overlaid task.

The PSECTs are named as follows:

$$FIOC - Contains the format processor code and the list-directed processor code;

$$FIOD - Contains the format and list-directed processor pure data;

$$FIOI - Contains the integer and octal conversions;

$$FIOL - Contains the logical conversions;

$$FIOR - Contains the floating point conversions; and

$$FIO2 - Contains the conversion dispatch table.

2.2 IMPURE STORAGE DESCRIPTION

This section describes the contents of the FORTRAN impure storage PSECT $$AOTS.

2.2.1 Named Offsets

The named offsets comprise the first 167 words of the work area. They have names of the form W.xxxx or xxxxxx. There are both word and byte offsets as well as a 16-word pushdown list for format processing. Several words in the work area are used in more than one context. Consult Appendix A for the list of offsets and their use.

2.2.2 QIO Directive Parameter Block

The work area contains a 12-word DPB for performing error message QIO's to the user's terminal using event flag 30. This DPB is used in RSX-11M for all message output and in RSX-11D and IAS if the message output task (MO....) is not loaded or not present.

2.2.3 Error Message Text Buffer

A buffer for the error text message line is allocated. This buffer is 57 words in RSX-11D and IAS, and 32 words in RSX-11M. This buffer is pointed to by offsets W.ERLN (start address) and W.ERLE (end address+1).

2.2.4 Error Control Table

A 56-word area for the error control table is allocated with one byte for each error. These are impure data and are used and manipulated by the error handling routines. A prototype version of the table is copied into this area by the task initialization routine OTI$.

2.2.5 Synchronous System Trap Vector Table

The Synchronous System Trap (SST) vector address table, $SST, contains an entry for each defined SST. User level modification or interception of trap vectors is described in Section 5.7.

2.3 LOGICAL UNIT DEVICE TABLE

All FORTRAN input/output is done through logical units. Each unit has a FORTRAN File Descriptor Block (FFDB) allocated in the PSECT $$DEVT. There is one FFDB allocated for each unit declared in the Task Builder UNITS= statement or the default value of 6 units. Each FFDB is a fixed length block consisting of an FCS FDB and a 6-word header for FORTRAN usage. Each FFDB is initialized to 0 at task initialization and is also zeroed by a close operation.

2.3.1 FFDB Offsets

The FORTRAN header portion of the FFDB is described by offsets of the form D.xxxx as follows:

D.STAT - status word 1 (bits defined below)
D.STA2 - status word 2 (bits defined below)

D.RCNM - number of direct access records (low order)
D.RCN2 - number of direct access records (high order)
D.RCCT - record count for BACKSPACE (low order)
D.RCC2 - record count for BACKSPACE (high order)
D.AVAD - direct access associated variable address (0 if not present)

Several of the words have different uses depending upon the kind of I/O operations.

2.3.2 FFDB Status Bit Definitions

The FORTRAN header portion of the FFDB contains two status words. The bits contained in these status words have symbolic names of the form DV.xxx. Definitions of these bits follow:

status bits - word 1

DV.FAK - only a partial FFDB is allocated for ENCODE/DECODE
DV.FNB - File Name Block is initialized
DV.DFD - direct access unit
DV.FACC - file attributes byte of FDB (F.FACC) is defined
DV.OPN - unit is open
DV.FMP - formatted unit
DV.UFP - unformatted unit
DV.ASGN - filename is defined
DV.CLO - close in progress
DV.FRE - free format allowed (short field termination)
DV.RW - input or output operation (0 = read, 1 = write)

status bits - word 2

DV.AI4 - associated variable is INTEGER*4
DV.CC - explicit carriage control
DV.SPL - DISP = 'PRINT' specified
DV.DEL - DISP = 'DELETE' specified
DV.SAV - DISP = 'SAVE' specified
DV.RDO - READONLY specified
DV.UNK - TYPE = 'UNKNOWN' specified
DV.OLD - TYPE = 'OLD' specified
DV.NEW - TYPE = 'NEW' specified
DV.SCR - TYPE = 'SCRATCH' specified
DV.APD - ACCESS = 'APPEND' specified

2.4 ERROR PROCESSING DATA STRUCTURES

Error processing and reporting in FORTRAN IV-PLUS is done through TRAP instructions. The low byte of the TRAP contains the FORTRAN error number plus 128 (decimal). The error processing is controlled by an error control byte in impure storage.

The FORTRAN error numbers range from 1 to 110 (decimal). Not all numbers are defined. Appendix C of the FORTRAN IV-PLUS User's Guide describes the errors in detail.

The error control byte is bit-encoded. The bit descriptions are:

EC.CON - continue
EC.CNT - count
EC.UER - use ERR= exit if 1, return if 0

EC.LOG - log
EC.INU - this number defined for use
EC.RTS - return continuation permitted
EC.ERE - ERR= continuation permitted

The sign bit is unnamed, but if clear, this error has not occurred; if set, the error has occurred. This bit is tested and cleared by the ERRSET system subroutine.

The standard bit combinations used are:

Fatal
Errors: EC.FAT = EC.INU + EC.LOG

I/O
Errors: EC.IO = EC.INU + EC.CON + EC.CNT + EC.LOG + EC.UER + EC.ERE

Other
Errors: EC.NRM = EC.INU + EC.CON + EC.CNT + EC.LOG + EC.RTS

2.5 ARRAY DESCRIPTOR BLOCKS

Array descriptor blocks (ADBs) are used for dummy arrays, input/output statements that transmit an entire array, and array subscript checking (/CK compiler switch). The ADB is initialized at compile time for all constant items. Variable portions of ADBs for dummy arrays are initialized at subprogram entry. The ADB contains the array base address, the array zeroth-element address, the upper and lower bounds and the dimension spans for each dimension.

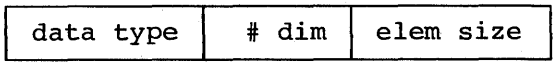
2.5.1 ADB Offsets

The offsets within the ADB are denoted as follows:

A.ASTR - actual base storage address (1st element)

A.A0 - zeroth-element address (address of A (0,0,0...0))

A.CWRD - codeword containing the number of dimensions, data type, and element size in bytes:



A.BPE - number of bytes per array element (BPE) (low byte of A.CWRD)

A.D1 - first dimension span. (Other dimensions follow A.D1 but are not named; i.e., A.D1+2 is the second dimension span.) Dimension spans are described in Section 2.5.2.

A.SIZB - total array size in bytes, A.SIZB = D1*D2*...Dn*BPE

A.PLYA - addressing polynomial evaluated for the first element, polyA (L1,L2,...Ln)

A.PWRD - (used for adjustable arrays) 2N 1-bit fields denoting an adjustable/non-adjustable bound. Encoding is left justified as follows:

Un	Ln	Un-1	.....	U1	L1	not used

A.UN - last upper bound. (Other bounds are stored in front of A.UN but are not named; i.e., A.UN-2 is the last lower bound, A.UN-4 is the next-to-last upper bound, etc.).

The data type codes contained in A.CWRD are:

A.LGC1 = LOGICAL*1 (BYTE)
A.LGC2 = LOGICAL*2
A.LGC4 = LOGICAL*4
A.INT2 = INTEGER*2
A.INT4 = INTEGER*4
A.REA4 = REAL*4
A.REA8 = REAL*8 (DOUBLE PRECISION)
A.CMP8 = COMPLEX
A.HOLL = Hollerith

These codes are used not only for ADB's, but also for I/O transmission to denote the list item data type.

2.5.2 Array Dimension Spans

The dimension spans (Di) for arrays are the sizes of each dimension:

Di = upper bound (Ui)-lower bound (Li) + 1.

Dimension spans are used to determine the subscript value by the compiled code. The upper bounds and lower bounds for arrays are retained in the ADB for determining the size and shape of arrays.

2.5.3 Notes on ADB Usage

The array addressing polynomial function, polyA, for a 3-dimensional array is defined by:

DIMENSION A(L1:U1,L2:U2,L3:U3)

polyA(I,J,K)=((K*D2+J)*D1+I)*BPE

A.A0 is defined as A.ASTR - polyA (L1,L2,L3)

The address of an array element is then calculated as:

address of A(i,j,k)=A.ASTR+polyA(i,j,k) -polyA (L1,L2,L3)

=A.A0+polyA(i,j,k)

Array bounds checking consists of verifying that the array element address is:

1. greater than or equal to the base address, A.ASTR
2. less than the high address + 1, A.ASTR+A.SIZB

Note that this means only that the complete subscript value is within the array; individual dimensions are NOT checked against their corresponding dimension bounds.

The FORTRAN statements:

```
      SUBROUTINE X(A,N)
      DIMENSION I(100), A(10:N-1,N)
```

create the following ADB's for I and A:

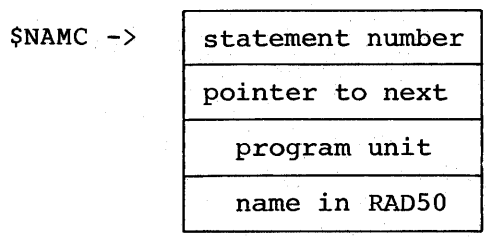
```
          .WORD     310     ; A.SIZB
I.ADB:    .WORD     I       ; A.ASTR
          .WORD     I-2     ; A.A0
          .WORD   20402     ; A.CWRD
                            ;no Di values since I is not
                            ;an adjustable array

          .WORD     12      ;   L 1  = 10
          .WORD      0      ;   U 1  = N-1
          .WORD      1      ;   L 2  = 1
          .WORD      0      ;   U 2  = N
          .WORD  120000     ; A.PWRD
          .WORD      0      ; A.SIZB
A.ADB:    .WORD      0      ; A.ASTR
          .WORD      0      ; A.A0
          .WORD  31004      ; A.CWRD
          .WORD      0      ;   D1
          .WORD      0      ;   D2
```

2.6 TRACEBACK CHAIN

The traceback chain for error processing is a linked list constructed dynamically on the run-time stack.

The list head and the current statement number are contained in the work area. The list head is at offset W.NAMC with global name $NAMC. The current statement number is at offset W.SEQC with global name $SEQC.

The list elements are 4-word blocks located on the stack in the following form:



The list head points to the currently active program unit entry. This block contains the currently active program unit name in Radix-50; the current statement number in the calling program at the time of the call; and a pointer to the calling program list block. Note that the statement number pertains to the program unit of the NEXT list block since the current program unit statement number is maintained at the fixed global location $SEQC.

NOTE

This list structure is "known" to the RSX-11D and IAS message output task (MO) and cannot be changed.

If traceback level NONE is used, no list block is created.

If traceback level NAMES is used, a list block is linked in but $SEQC is zero.

If traceback level BLOCKS is used, a list block is linked in and $SEQC is periodically updated with the negative of the current statement number, i.e., -21 for statement 21.

If traceback level LINES is used, a list block is created and $SEQC is incremented on every statement, i.e., a positive number is maintained.

CHAPTER 3

INPUT/OUTPUT PROCESSING

This chapter discusses the basic procedures, program flow and module interdependencies of the I/O processing portion of the OTS.

3.1 COMMON I/O SUPPORT

This section documents the processing common to all forms of input/output: opening files, closing files, accessing the device table, default values, I/O initialization and element transmission.

All OTS I/O calls preserve the user-level register state and generally take all arguments on the stack and return with stack arguments deleted. During I/O processing the calling program generally saves those registers it requires and passes arguments in registers. The actual conventions used are given in the module descriptions.

The general form of I/O involves three phases:

1. An initialization call to an operation-specific module as shown in Table 3-1.

   These modules set the correct mask word for the operation into R1 and call $INITIO to initialize the I/O system.

2. Element transmission calls to pass the list elements to the I/O system. This is done in a co-routine fashion between the user level and the I/O processor through work area offset W.EXJ. The user code calls the element transmission routine, which saves the registers, sets the work area offsets:

   ```
   VARAD = variable address
   ITEMSZ = variable length
   W.VTYP = variable data type
   ```

   and calls the I/O processor through W.EXJ. The I/O processor transfers the data and does a co-routine call back to the element processor, which saves this address at W.EXJ, restores the registers and returns to the user code.

3. An end-of-list call sets offset VARAD to zero to indicate the end of the I/O list and calls the I/O processor. The I/O processor completes the I/O operation and returns to the end-of-list routine, which restores the registers, releases the I/O system and returns to the user code.

The work area offset FILPTR contains zero if the I/O system is inactive, and the address of the active FFDB if I/O is in progress. This location is loaded by $INITIO at I/O initialization and cleared by the end-of-list processor ($EOLST).

Within the I/O portion of the OTS the following register assignments are normally made:

R0 = address of the FCS FDB

R1 = address of the FFDB

R3 = address of the work area

R2 and R4 are scratch registers

All routines, except the co-routine calls, are called by a JSR PC,xxx instruction. R5 is generally preserved.

Example:

The FORTRAN statements:

```
DIMENSION  A(10)
WRITE  (3,100,ERR=99)  I,A,B+1.
```

are compiled into the code:

```
MOV  #3,-(SP)        ; unit number
MOV  #.100,-(SP)     ; FORMAT statement address
CLR  -(SP)           ; NO END=address
MOV  #.99,-(SP)      ; ERR=address
JSR  PC, OSFE$       ; initialize I/O
MOV  #I,-(SP)        ; address of I
JSR  PC, IOAI$       ; transmit integer
MOV  #A.ADB,-(SP)    ; address of ADB for A
JSR  PC,IOAA$        ; transmit array A
LDF  B,F0            ;
ADDF #1.0,F0         ; compute B+1.0
STF  F0,-(SP)        ; value
JSR  PC, IOVR$       ; transmit real value
JSR  PC, EOLST$      ; end-of-I/O-list
```

3.1.1 $FCHNL, $GETFILE and $IOEXIT

These routines serve as the common entrance and exit to the I/O system.

$FCHNL locates the FFDB for a given logical unit number, and issues an error for invalid units. This routine is called with the unit number in R2 and returns the address of the associated FFDB in R0. The PSW C-bit is used as an error flag on return: set if error, clear if no error.

$GETFILE executes a $FCHNL, sets the FILPTR offset and checks the status of the unit. This routine is called identically to $FCHNL. It does not return the C-bit error flag. The PSW N-bit is returned as a status flag (N-bit set if the unit is open, N-bit clear if closed).

$IOEXIT restores the user level status and executes the 'ERR=' transfer. This routine is called with the ERR=transfer address in R4 and the work area pointer in R3.

3.1.2 Default OPEN -- $OPEN

A default OPEN is the implicit opening of a unit due to execution of a READ or WRITE statement to a closed unit. The default file access is FO.WRT if a WRITE statement is being executed and FO.UPD if a READ statement is being executed.

3.1.3 File Close -- $CLOSE

When a close operation is specified by a CLOSE statement, a CALL CLOSE, or by task termination, the close processor handles the file disposition: 'SAVE', 'PRINT' or 'DELETE'. A specification in a CLOSE statement overrides an existing specification. The following rules are enforced:

1. A 'SCRATCH' file cannot be saved or printed.

2. A 'READONLY' file cannot be deleted or printed.

The close processor is called with the unit number in R2.

3.1.4 I/O Initialization -- $INITIO

The call to $INITIO to initialize the I/O system places a bit-encoded value into R1 to denote the arguments present in the I/O statement and the processing forms required. These encodings are:

```
Bit 15 = 1  -->  END=/ERR= addresses present
Bit 14 = 1  -->  ENCODE/DECODE array address present
Bit 13 = 1  -->  FORMAT address present
Bit 12 = 1  -->  INTEGER*4 record number present
Bit 11 = 1  -->  set up ENCODE/DECODE
       = 0  -->  set up normal I/O
Bit 10 = 1  -->  Direct access operation
Bit 9 = 1   -->  Formatted operation
      = 0   -->  Unformatted operation
Bit 8 = 1   -->  Write operation
      = 0   -->  Read operation
Bit 7 = 1   -->  Open file if not open
```

Common operations and combinations are defined as follows:

```
FL.FMP  =  formatted operation
FL.ERR  =  END=/ERR= present
FL.ENC  =  ENCODE/DECODE operation
FL.FMT  =  FORMAT present and formatted operation
FL.REC  =  direct access operation/record number present
FL.WRT  =  write operation with open implied
FL.RD   =  read operation with open implied
```

$INITIO initializes the I/O system, performs a default OPEN if needed and sets up the element transmission co-routine.

The FORTRAN statement:

```
READ (1,5,END=99) A
```

has the mask value FL.ERR+FL.FMT+FL.RD.

The FORTRAN statement:

```
WRITE (3'I) Z
```

has the mask value FL.REC+FL.WRT.

3.1.5 Element Transmission -- $IOELEM and $IOARY

Module $IOELEM contains 17 entry points to transmit individual list items to the I/O processor. The element transmission entry names are of the form:

IOat$

where

a designates whether the argument is an address or a value: A means address, V means value.

t designates the data type of the list element as follows:

B - Byte
L - Logical*2
M - Logical*4
I - Integer*2
J - Integer*4
R - Real
D - Double Precision
C - Complex

Complex list elements are passed to the I/O processor as a sequence of two Real values. Offset W.CPXF is 0 for a non-Complex value; +1 for the real part of a Complex value and -1 for the imaginary part.

For output, the entry IOAH$ transmits a Hollerith constant or alphanumeric literal where the argument is the address of the first byte of the constant.

Module $IOARY contains the entry point IOAA$ which is used for array input/output. Its argument is the address of the array descriptor block (ADB). For formatted I/O, each array element is transmitted individually to the I/O processor. For unformatted I/O, the entire array is transmitted as a single element.

3.2 RECORD PROCESSING SUPPORT

There are 24 entry points for initializing I/O operations. They are shown in Table 3-1 and described below. Each I/O operation has two entry points: XXX$ and XXXE$. The XXX$ entry is used for normal I/O statements. The XXXE$ entry is used if END= or ERR= elements are present in the I/O statement. The notation "[+FL.ERR]" is used in the description to distinquish the mask used for the two entry points.

Table 3-1
I/O Initialization Entries

Entry Name	Arguments	Function
ISF$	u,f	Input sequential formatted
ISFE$	u,f,e	Input sequential formatted with END=/ERR=
ISU$	u	Input sequential unformatted
ISUE$	u,e	Input sequential unformatted with END=/ERR=
IRF$	u,r,f	Input direct access formatted
IRFE$	u,r,f,e	Input direct access formatted with END=/ERR=
IRU$	u,r	Input direct access unformatted
IRUE$	u,r,e	Input direct access unformatted with END=/ERR=
OSF$	u,f	Output sequential formatted
OSFE$	u,f,e	Output sequential formatted with END=/ERR=
OSU$	u	Output sequential unformatted
OSUE$	u,e	Output sequential unformatted with END=/ERR=
ORF$	u,r,f	Output direct access formatted
ORFE$	u,r,f,e	Output direct access formatted with END=/ERR=
ORU$	u,r	Output direct access unformatted
ORUE$	u,r,e	Output direct access unformatted with END=/ERR=
ENF$	c,f,a	ENCODE
ENFE$	c,f,a,e	ENCODE with END=/ERR=
DEF$	c,f,a	DECODE
DEFE$	c,f,a,e	DECODE with END=/ERR=
ISL$	u	Input sequential list-directed
ISLE$	u,e	Input sequential list-directed with END=/ERR=
OSL$	u	Output sequential list-directed
OSLE$	u,e	Output sequential list-directed with END=/ERR=

u = logical unit number (INTEGER*2)
c = character count for ENCODE/DECODE
r = record number for direct access (INTEGER*4)
f = compiled format string address (see discussion of FMTCV$)
a = array address of data for ENCODE/DECODE
e = END= address followed by ERR= address. If only one address is present, then 0 is supplied for the other address.

3.2.1 Formatted I/O

The I/O routines for each type of formatted I/O contain two code segments: an initialization segment and a record processing segment.

The initialization segment performs the following operations:

1. Store the proper argument mask in R1 and call $INITIO to initialize the I/O system.

2. Store the record processing segment address into work area offset RECIO. The format processors perform record transfers by executing a JSR PC,@RECIO(R3) instruction.

3. Miscellaneous operation-specific initialization e.g., record buffer pointers.

4. Jump to the format processor: $FIO for formatted I/O, $LSTI for list-directed input, and $LSTO for list-directed output.

The record processing code segment calls the appropriate I/O transmission utility (Section 3.4) and updates miscellaneous information such as buffer pointers and counts as required.

3.2.1.1 Sequential Input -- ISF$, ISFE$ - These routines have FL.FMT+FL.RD[+FL.ERR] as the argument mask. The record processing code calls $GETS to read the record.

3.2.1.2 Sequential Output -- OSF$, OSFE$ - These routines have FL.FMT+FL.WRT[+FL.ERR] as the argument mask, and initialize the work area buffer pointers. The record processing code computes the actual record length and calls $PUTS to write the record.

3.2.1.3 Direct Access Input -- IRF$, IRFE$ - These routines have FL.RD+FL.REC+FL.FMT[+FL.ERR] as the argument mask. The record processing code checks for multiple record requests and calls $GETR to read the record.

3.2.1.4 Direct Access Output -- ORF$, ORFE$ - These routines have FL.WRT+FL.REC+FL.FMT[+FL.ERR] as the argument mask, call $PUTRI to initialize the record and initialize the work area buffer pointers. The record processing code checks for multiple record requests and calls $PUTR to write the record.

3.2.1.5 List-Directed Input -- ISL$, ISLE$ - These routines have FL.RD+FL.FMP[+FL.ERR] as the argument mask, set the buffer pointers to be at end of record and create the pointer to the constant value storage block. The record processing code calls $GETS to read a record.

3.2.1.6 List-Directed Output -- OSL$, OSLE$ - These routines have FL.WRT+FL.FMP[+FL.ERR] as the argument mask and initialize buffer pointers. The record processing code calls $PUTS to write a record, inserts a blank character for carriage control and sets the buffer length to 72 characters.

3.2.1.7 ENCODE Statement -- ENF$, ENFE$ - ENCODE is processed identically to formatted output. These routines have FL.ENC+FL.FMT[+FL.ERR] as the argument mask. A partial FFDB in the work area is initialized. The record processing code checks for multiple record requests.

3.2.1.8 DECODE Statement -- DEF$, DEFE$ - DECODE is processed identically to formatted input. These routines have the argument mask FL.ENC+FL.FMT[+FL.ERR]. A partial FFDB in the work area is initialized. The record processing code checks for multiple record requests.

3.2.1.9 PRINT, TYPE and ACCEPT Statements - These statements are compiled into equivalent READ and WRITE statements using default unit numbers. Default unit numbers are small negative integers, which are mapped by $FCHNL through the work area to actual unit numbers. The number of such unit numbers is the value of offset W.LNMP and the mapped values are contained at offsets W.PRNT for PRINT, W.TYPE for TYPE, W.ACPT for ACCEPT and W.READ for READ with no unit number.

```
PRINT   compiles into OSF$ with unit number = -1, maps to 6.
TYPE    compiles into OSF$ with unit number = -2, maps to 5.
ACCEPT  compiles into ISF$ with unit number = -3, maps to 5.
READ    compiles into ISF$ with unit number = -4, maps to 1.
```

3.2.2 Unformatted Sequential I/O

The I/O routines for unformatted sequential I/O perform the following operations:

1. Store the proper argument mask in R1 and call $INITIO to initialize the I/O system.
2. Miscellaneous operation-specific initialization.
3. Initiate the I/O list transmission co-routine via a JSR PC,@(R4)+ instruction.
4. For each I/O list element, move the data bytes to or from the record.
5. Call the appropriate I/O utility routine to transmit the record and update miscellaneous data such as record counts and buffer pointers.

FORTRAN unformatted sequential I/O utilizes a spanned record concept as described in the FORTRAN IV-PLUS User's Guide. Each I/O operation transmits a single FORTRAN record which may be one or more FCS physical records.

The FORTRAN to physical record mapping is provided by a 2-byte field at the beginning of each physical record. Bit 0 is set for the first segment and bit one is set for the last segment of a logical record. All 4-bit combinations are possible. A physical record has a maximum size of 126 bytes which contains 124 bytes of data.

3.2.2.1 Sequential Input -- ISU$, ISUE$ - These routines have FL.RD[+FL.ERR] as the argument mask. The element processing code moves the data bytes from the record to the list items. $GETS is called to read a physical record.

3.2.2.2 Sequential Output -- OSU$, OSUE$ - These routines have FL.WRT[+FL.ERR] as the argument mask. The element processing code moves the data bytes from the list items to the record. $PUTS is called to write a physical record.

3.2.3 Unformatted Direct Access I/O

The I/O routines for unformatted direct access I/O perform the following operations:

1. Store the proper argument mask in R1 and call $INITIO to initialize the I/O system.
2. Miscellaneous operation-specific initialization.
3. Initiate the I/O list transmission co-routine via a JSR PC,@(R4)+ instruction.
4. For each I/O list element, move the data bytes to or from the record.
5. Call the appropriate I/O utility routine to transfer the record.

3.2.3.1 Direct Access Input -- IRU$, IRUE$ - These routines have FL.RD+FL.REC[+FL.ERR] as the argument mask and call $GETR to read the record. The element processing code moves data bytes from the record to the list elements.

3.2.3.2 Direct Access Output -- ORU$, ORUE$ - These routines have FL.WRT+FL.REC[+FL.ERR] as the argument mask and call $PUTRI to initialize the record. The element processing code moves data bytes from the list elements to the record. $PUTR is called to write the record.

3.3 FILE PROCESSING SUPPORT

The following modules provide unit and file processing facilities.

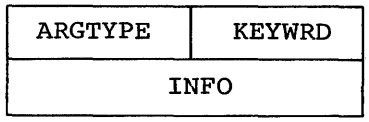
3.3.1 OPEN Statement

The OPEN statement provides the user program detailed control of file attributes and characteristics. The OPEN statement processor is called using the standard R5 calling sequence with a specialized argument list encoding. The argument list has the following form:

```
ARGLST:   .WORD    2*n
          KEY1
          ...
          KEYn
```

There is one argument for each keyword in the FORTRAN source statement. Duplicate keywords are not allowed. The order of arguments is immaterial.

Each argument consists of a 2-word block formatted as follows:



where

KEYWRD = keyword identification number

ARGTYPE = type of information in INFO word

INFO = use depends upon ARGTYPE value

ARGTYPE distinguishes among the following cases:

ARGTYPE=1 - Keyword value is an INTEGER*2 constant expression. In this case, the INFO word contains the value.

ARGTYPE=2 - Keyword value is an INTEGER*2 value. INFO contains the address of the value.

ARGTYPE=3 - Keyword value is an INTEGER*4 value. INFO contains the address of the value.

ARGTYPE=4 - Keyword value is an alphanumeric literal decodable by the compiler. INFO contains the keyword value encoded as a small integer.

ARGTYPE=5 - Keyword value is a variable, array, array element or alphanumeric literal terminated by an ASCII null character (zero-byte). INFO contains the address of the start of the string.

Table 3-2
Summary of Argument Blocks by Keyword

Keyword Name	Keyword Number	Allowed Argtypes	Literal Values	Literal Encoding
UNIT	1	1,2,3		
ACCESS	4	4	DIRECT SEQUENTIAL APPEND	1 2 3
ASSOCIATE VARIABLE	17	2,3		
BLOCKSIZE	18	1,2,3		
BUFFERCOUNT	9	1,2,3		
CARRIAGECONTROL	7	4	FORTRAN LIST NONE	1 2 3
DISPOSE	2	4	SAVE DELETE PRINT	1 2 3
ERR	3	(NOTE 1)	-	label address
EXTENDSIZE	11	1,2,3		
FORM	5	4	FORMATTED UNFORMATTED	1 2
INITIALSIZE	10	1,2,3		
MAXREC	16	1,2,3		
NAME	14	5		
NOSPANBLOCKS	12	--		
READONLY	8	--		
RECORDSIZE	6	1,2,3		
SHARED	13	--		
TYPE	15	4	OLD NEW SCRATCH UNKNOWN	1 2 3 4

(1) The ARGTYPE field for the ERR= keyword contains the number of bytes of temporary stack storage which must be deleted if an ERR= transfer occurs.

Table 3-2 provides a summary of the keywords and keyword values supported by the OPEN statement. The basic OPEN statement processing searches through the argument list locating each keyword in a prescribed order. The ordering is determined so that all relevant information required by a given keyword is available when that keyword is processed. If a keyword is not present in the list, an appropriate default is used. A running error count is maintained. At the end of keyword processing, if any errors have occurred, the actual OPEN is not attempted. If errors occur, the ERR= transfer is taken and the FFDB is zeroed. The processing for each keyword is described below in the order of processing. (Consult Section 2.3.2 for the definitions of the status bits DV.xxx.)

1. ERR - The ERR= transfer address is obtained and the stack adjustment value is saved in the work area at offset COUNT. The transfer address, if present, is stored at offset ERREX; if it is not present, ERREX is cleared.

2. UNIT - The unit number is obtained and $FCHNL is called to obtain the FFDB pointer. Fatal errors that immediately abort processing occur if there is no unit number, the unit number is invalid, or the unit is already open.

3. READONLY - If present, DV.RDO is set.

4. ACCESS - 'DIRECT' sets DV.DFD, 'APPEND' sets DV.APD. If DV.RDO is set and DV.APD is specified, an error occurs. If not specified, the default is 'SEQUENTIAL'.

5. TYPE - If not present, the default is 'NEW'. 'NEW' sets DV.NEW, 'OLD' sets DV.OLD, 'SCRATCH' sets DV.SCR and 'UNKNOWN' sets DV.UNK. If DV.RDO is set and DV.SCR, DV.NEW or DV.UNK is specified an error occurs. If DV.APD is set and DV.SCR or DV.NEW is specified an error occurs. The file access byte F.FACC is set up as follows:

```
DV.RDO      --> FO.RD
DV.APD      --> FO.APD
DV.SCR      --> FO.WRT + FA.TMP
DV.NEW      --> FO.WRT
DV.OLD      --> FO.UPD
DV.UNK      --> FO.UPD
```

6. DISPOSE - 'SAVE' sets DV.SAV, 'PRINT' sets DV.SPL and 'DELETE' sets DV.DEL. If DV.RDO is set, and DV.DEL or DV.SPL is specified, an error occurs. If a DISPOSE value is not specified, 'DELETE' is the default if DV.SCR is set. 'SAVE' is the default otherwise.

7. FORMAT - 'FORMATTED' sets DV.FMP, 'UNFORMATTED' sets DV.UFP. If not specified and DV.DFD is set then DV.UFP is the default, otherwise DV.FMP is the default.

8. RECORDSIZE - The value is stored at F.RSIZ. If the value is negative or larger than the user record buffer size (MAXBUF value), an error occurs. If DV.UFP (unformatted) is specified, the value is converted to bytes from storage units (four bytes per storage unit). If the value given does not equal the value for an existing file, an error occurs unless the system subroutine ERRSET has been called to set the continuation-type for Error 37 (Inconsistent Record Length) to a return continuation.

9. CARRIAGECONTROL - DV.CC is set; 'FORTRAN' sets FD.FTN in F.RATT, and 'LIST' sets FD.CR in F.RATT. If DV.CC is not set and DV.FMP is specified, FD.FTN is the default.

10. BUFFERCOUNT - The value specified is stored at F.MBCT. If the value is negative or greater than 127, an error occurs. Note that the actual number of buffers used depends upon the FCS version in use and the number of buffers available at file open. If a buffer count of -1 is specified, the unit will be opened in block (READ$/WRITE$) mode rather than in record (GET$/PUT$) mode. Normal FORTRAN I/O is then not permitted but the user can perform asynchronous block mode I/O through the FORTRAN Special Subroutines provided by the operating system.

11. INITIALSIZE - The value specified is stored at F.CNTG. If the value is positive, a contiguous allocation is made; if negative, a non-contiguous allocation is made. If the value is greater than 32767 or less than -32767, an error occurs.

12. EXTENDSIZE - The value specified is stored at F.ALOC. If the value is positive, a contiguous extend is made; if negative, a non-contiguous extend is made. If the value is greater than 32767 or less than -32767 an error occurs.

13. NOSPANBLOCKS - If specified, FD.BLK is set in F.RATT.

14. SHARED - If specified, FA.SHR is set in F.FACC.

15. NAME - If specified, $FNBST is called to initialize the file name block and DV.ASGN is set. $FNBST returns an error if the string is incorrect.

16. MAXREC - The value specified is stored at D.RCNM and D.RCN2. If the value is negative, an error occurs.

17. ASSOCIATE VARIABLE - The variable address is stored at D.AVAD. If the variable is INTEGER*4 type, DV.AI4 is set.

18. BLOCKSIZE - The value specified is stored at F.OVBS. If the value is negative or greater than 32767, an error occurs.

The FORTRAN statement:

```
OPEN (UNIT = I, ERR = 99, NAME = 'A.DAT', TYPE = 'OLD',
INITIALSIZE = I**J)
```

is compiled into the code:

```
        MOV    I, -(SP)          ; I
        JSR    R4, PWIIC$        ; compute I**J on stack
        .WORD    J
        MOV    SP, ARGLST+24     ; save address in argument list
        MOV    #ARGLST, R5       ; address of arg list
        JSR    PC, OPEN$         ; open the file
        TST    (SP)+             ; delete stack temp

ARGLST: .WORD      12            ; 5 arguments
        .BYTE      1,2           ; UNIT, arg type = 2
        .WORD       I            ; address of I
        .BYTE     3,2            ; ERR, 2 bytes of stack temp
        .WORD     .99            ; address of label
        .BYTE     16,5           ; NAME, arg type = 5
```

```
        .WORD   STRING          ; address of ASCIZ string
        .BYTE    17,4           ; TYPE, arg type = 4
        .WORD      1            ; 'OLD' encoded value
        .BYTE    12,2           ; INITIALSIZE, arg type = 2
        .WORD      0            ; filled-in address of I**J

STRING: .BYTE     101,56,104,101,124,0     ; 'A.DAT'
```

3.3.2 CLOSE Statement

The CLOSE statement provides the user program flexibility in file processing and logical unit utilization based upon run-time events. The CLOSE statement is compiled using an encoded argument list similar to that for the OPEN statement; only the UNIT, ERR and DISPOSE keywords are allowed. Processing is similar to OPEN: the argument list is searched for each allowed keyword and appropriate actions are taken. If any errors are encountered, the CLOSE is not attempted and the FFDB is NOT zeroed.

1. ERR - The ERR= transfer address is obtained and the stack adjustment value is saved at offset COUNT. The address is stored at offset ERREX if present.

2. UNIT - The unit number is obtained and $FCHNL is called to obtain the FFDB address. If no unit number is present or an invalid unit number is specified, a fatal error occurs.

3. DISPOSE - If not present, the existing disposition is used. The existing disposition is superseded by the CLOSE statement specification. 'SAVE' sets DV.SAV, 'PRINT' sets DV.SPL and 'DELETE' sets DV.DEL. If DV.SCR is set and DV.SPL or DV.SAV is specified an error occurs. If DV.RDO is set and DV.SPL or DV.DEL is specified, an error occurs.

3.3.3 DEFINEFILE Statement

The DEFINEFILE statement is compiled as follows:

1. Convert unit number to Integer*2 (if needed) and push onto the stack.

2. Convert number of records to Integer*4 (if needed) and push.

3. Convert words per record to Integer*2 (if needed) and push.

4. Push address of associated variable.

5. If associated variable is type Integer*2 then push zero; if type Integer*4 then push the value -1.

6. Call DEFF$.

The unit number is obtained and $GETFILE is called to obtain the FFDB address. If the unit is open, an error occurs. The number of records is stored at D.RCNM and D.RCN2 in the FFDB. The recordsize is converted to bytes and stored at F.RSIZ in the FDB. The associated variable address is stored at D.AVAD and DV.AI4 is set if the associated variable is Integer*4. DV.DFD and DV.UFP are set. If DV.DFD was previously set then an error occurs. If the number of records or record size is negative, an error occurs.

The FORTRAN statement:

```
DEFINE FILE 2(N, 100, U, IVAR)
```

is compiled into the code:

```
MOV   #2, -(SP)       ; unit number
MOV   N+2, -(SP)      ; high order value of record number
MOV   N, -(SP)        ; low order value of record number
MOV   #144, -(SP)     ; record length
MOV   #IVAR, - (SP)   ; associated variable address
CLR   -(SP)           ; associated variable is INTEGER*2
JSR   PC, DEFF$
```

3.3.4 FIND Statement

The FIND statement is contained in the same module as the DEFINEFILE statement. It is compiled as follows:

1. Convert unit number to Integer*2 (if needed) and push.
2. Convert record number to Integer*4 (if needed) and push.
3. Call FIND$.

The argument mask for $INITIO is set to FL.REC!FL.RD and $INITIO is called. The associated variable, if present, is set to the record number.

The FORTRAN statement:

```
FIND (4' 131073)
```

is compiled into the code:

```
MOV   #4, -(SP)       ; unit number
MOV   #2, -(SP)       ; high order value of record number
MOV   #1, -(SP)       ; low order value of record number
JSR   PC, FIND$
```

3.3.5 BACKSPACE Statement

The BACKSPACE statement is compiled as follows:

1. Push unit number (Integer*2) on the stack.
2. Call BKSP$.

The unit number is obtained and $GETFILE is called to obtain the FFDB address. If the file is closed or direct access the operation is ignored. If the file is opened for append, an error occurs. A call to the FCS routine .POINT is made to position the file at the beginning (virtual block 1, byte 0). The record count is obtained from D.RCCT and D.RCC2 in the FFDB. The record count is decremented by 1, and then n-1 reads are performed. Note that the count is the logical record count, hence multiple physical reads may be required for the unformatted spanned records.

3.3.6 REWIND Statement

The REWIND statement is compiled as follows:

1. Push unit number (Integer*2) on the stack.
2. Call REWI$.

The unit number is obtained and $GETFILE is called to obtain the FFDB address. If the file is closed or direct access, the operation is ignored. The append bit is cleared and the record count D.RCCT and D.RCC2 is zeroed. A call to the FCS routine .POINT is made to position the file at the beginning (virtual block 1, byte 0).

3.3.7 ENDFILE Statement

The ENDFILE statement is compiled as follows:

1. Push unit number (Integer*2) on the stack.
2. Call ENDF$.

The unit number is obtained, and $GETFILE is called to obtain the FFDB address. If the file is direct access, an error occurs and the operation is ignored. The file is opened by $OPEN (default open) for write if not open. A 1-byte record, containing an octal 32 (CTRLZ) is output to the file, using $PUTS.

3.4 I/O PROCESSING UTILITIES

These low-level OTS routines are called by the I/O statement processors and format processors to perform the actual calls to FCS for record transfer and to perform miscellaneous utility tasks. These routines are called with the work area address in R3.

3.4.1 Sequential Input -- $GETS

The FFDB pointer is obtained from offset FILPTR. The FCS macro call GET$S is made to get a record. If FCS error IE.EOF is returned or an ENDFILE record is read, the END= transfer is made; any other error causes the ERR= transfer to be taken. The record count D.RCCT and D.RCC2 is incremented.

3.4.2 Sequential Output -- $PUTS

The FFDB pointer is obtained from offset FILPTR. The FCS macro call PUT$S is made to output the record. The record count D.RCCT and D.RCC2 is incremented. This routine is called with the record length in R1.

3.4.3 Direct Access Input -- $GETR

The FFDB pointer is obtained from offset FILPTR, and $CKRCN is called to verify the validity of the record number and return the record number in R1 and R2. The FCS macro call GET$R is made to read the record. $ASVAR is called to update the associated variable.

3.4.4 Direct Access Output -- $PUTR and $PUTRI

$PUTRI is called to initialize a direct access write operation. The FFDB pointer is obtained from offset FILPTR, and $CKRCN is called to verify the record number. The record number is stored at F.RCNM and F.RCNM+2 in the FDB. The FCS routine .POSRC is called to position the file to the desired record. If FCS error IE.EOF is returned, it is ignored. All other errors cause the ERR= transfer to be taken.

$PUTR is called to write the record. The FFDB pointer is obtained from FILPTR. The number of unfilled bytes in the record is computed and the record is padded with blanks for formatted records and zero bytes for unformatted records to the correct length. The FCS macro call PUT$R is made to write the record. $ASVAR is called to update the associate variable.

3.4.5 Direct Access Record Number Checking -- $CKRCN

$CKRCN verifies the validity of the current record number by comparing it against the maximum record number for the file. The current record number is stored at offsets W.RECL (low order) and W.RECH (high order). The maximum record number, if it exists, is at D.RCNM in the FFDB (low order) and D.RCN2 (high order). The record number, if valid, is returned in R1 (high order) and R2 (low order). This routine is called with the FFDB address in R0.

3.4.6 Associated Variable Update -- $ASVAR

The current record number is obtained from offsets W.RECL and W.RECH, incremented by one and stored in the associate variable at the address in D.AVAD in the FFDB.

3.4.7 File Name Block Initialization -- $FNBST

This module sets up the Filename Block (FNB) of the FFDB pointed to by offset FILPTR. It is called from the ASSIGN subroutine if a file name argument is present, and from the NAME keyword processor of the OPEN statement processor. It invokes the command string interpreter routines (.CSI1 and .CSI2) and the FCS .PARSE logic to construct the FNB. If no file name is found in the name string, $FLDEF is called to fill in the default file name. This routine is called with R2 containing the length of the name string and R1 pointing to the start of the string.

3.4.8 Default File Name Generation -- $FLDEF

This routine stores the default FORTRAN file name and file type into the filename block of the FFDB pointed to by offset FILPTR. The FORTRAN default filename is FOR0nn.DAT where nn is the unit number.

3.4.9 Register Save and Restore -- $SAVPx

This routine provides the register save/restore and argument processing support for implementing the OTS PC call convention (Section 1.2.2.2). This convention pushes all arguments on the stack, calls the OTS routine via a JSR PC, xxx instruction and returns with arguments deleted and all context preserved. This register save/restore routine is called by the OTS routine. It saves all registers on the stack, sets R0 to point to the call arguments, and co-routine calls the OTS module. Upon return from the OTS routine, it restores the registers, deletes the stack arguments and returns to the original caller. Seven entry points are provided: $SAVP0-$SAVP6 for routines with zero to six argument words on the stack. For routines with more than six arguments or with a variable number of arguments, $SAVP0 is called to save the registers; when returning, R0 contains the number of arguments and $SAVPC is jumped to instead of returning. For ERR= transfers $SAVPX is jumped to with R4 containing the transfer address.

CHAPTER 4

FORMAT PROCESSING AND FORMAT CONVERSIONS

This chapter discusses the internal form of format specifications, the format processing algorithm and the format conversion routines.

4.1 COMPILED FORMAT LANGUAGE

The formats used by the I/O system for formatted I/O are compiled into a standard internal form, with all error checking performed either by the FORTRAN IV-PLUS compiler, or by the OTS routine FMTCV$ at run-time. This allows the format interpreter itself to be simpler and smaller.

Each format code has a unique 6 bit code and requires one or more bytes of format text. All format code parameters (field widths, repeat counts, etc.) are stored in single bytes following the format code. A variable format expression (VFE) is represented by the address of the compiled machine instructions for the VFE. Figure 4-1 shows the form of a format code. Note that only the format code byte is required, but that each format code requires a fixed number of additional values, which are described in the following sections.

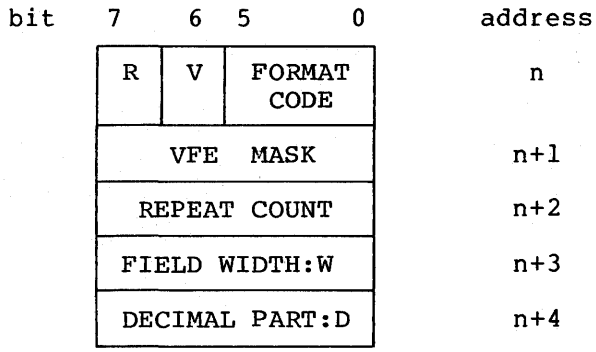


Figure 4-1 Format Code Form

4.1.1 Format Code Byte

The format code consists of a 6-bit format code, a 1-bit repeat count flag (R) and a 1-bit VFE flag (V). Table 4-1 lists the format codes and the additional values required. If the repeat count flag is 0, the repeat count is 1 and the repeat count byte is not present; if it is a 1 then the repeat count is not 1 or is a VFE. If the VFE flag is 0, no VFE's are present; if 1, the VFE mask byte is present and VFE addresses are required.

4.1.2 VFE Mask Byte

This byte describes which of the values required are VFE's. Bits starting at bit 7 from left to right describe the value; 1 means VFE, 0 means compiled constant.

4.1.3 Repeat Count Byte

This byte contains the optional repeat count value if the repeat count is not 1. The value is 1 less than the source code value.

4.1.4 Field Width Byte

This byte is the field width, tab position or scale factor in the range 1 to 255 (-128 to 127 for scale factor).

4.1.5 Decimal Part

This byte is the decimal field width for the floating point conversion codes, in the range 0 to 255.

4.1.6 VFE Implementation

If the repeat count, W value or D value is a Variable Format Expression (VFE) as indicated by the VFE Mask, the VFE address XXX begins on the next word boundary. The VFE is compiled as an unparameterized arithmetic statement function of type Integer*2 and is called by a JSR PC, xxx with R5 pointing to the program unit argument list. All range checking is done by the format interpreter on the result.

4.1.7 Hollerith Formats

Quoted format strings (alphanumeric literals) are compiled as Hollerith constants. The count for Hollerith constants may NOT be a VFE and the characters to be transmitted are contained in the compiled format following the repeat count.

4.1.8 Default Formats

Most format code field descriptors have a default value supplied if the numeric value is not present. These defaults are determined jointly from the format code and the data type of the corresponding list element as follows:

Format Code	Data Type	Default Values of W or W.D
I,O	I*2	7
I,O	I*4	12
D,E,F,G	R*4	15.7
D,E,F,G	R*8	25.16
L	all	2
A	all	Number of bytes in the variable
X	---	1

Table 4-1
Compiled Format Codes

Code	Source Form	Repeat Count	W	D	Notes
0*	--	--	--	--	format syntax error
2*	--	--	--	--	format too large
4	(	--	--	--	format reversion point
6	n(	n-1	--	--	left paren of repeat group
8	)	--	--	--	right paren of repeat group
10	)	--	--	--	end of format
12	/	--	--	--	record separator
14	$	--	--	--	
16	:	--	--	--	
18	sP	--	s	--	-128<s<127
20	Q	--	--	--	
22	Tn	--	n	--	1 < n < 255
24	nX	n-1	--	--	1 < n < 127
26	nHc1...cn	n-1	--	--	n not VFE, n chars follow
28	nAw	n-1	w	--	Standard conversions
30	nLw	n-1	w	--	
32	nOw	n-1	w	--	
34	nIw	n-1	w	--	
36	nFw.d	n-1	w	d	
38	nEw.d	n-1	w	d	
40	nGw.d	n-1	w	d	
42	nDw.d	n-1	w	d	
44	nA	n-1	--	--	Default formats
46	nL	n-1	--	--	
48	nO	n-1	--	--	
50	nI	n-1	--	--	
52	nF	n-1	--	--	
54	nE	n-1	--	--	
56	nG	n-1	--	--	
58	nD	n-1	--	--	

* These codes are generated only by FMTCV$.

The FORTRAN statement:

```
1     FORMAT(1X, F13.5, 'ABCDE', <K>I10, 3(2E15.7)/)
```

is compiled into the following:

```
.1:       .BYTE    30             ; 1X
          .BYTE  44,15,5          ; F13.5
          .BYTE    232            ; Hollerith code
          .BYTE     4             ; repeat count
          .BYTE  101,102,103,104,105 ; 'ABCDE'
          .BYTE    342            ; I format code
          .BYTE    200            ; VFE mask
          .BYTE     0             ; make an even address
          .WORD   L$VFE           ; VFE address
          .BYTE    12             ; I10
          .BYTE     4             ; reversion point
          .BYTE   206,2           ; left paren and repeat count
          .BYTE   246,1           ; E format code and repeat count
          .BYTE    17,7           ; E15.7
          .BYTE    10             ; right paren
          .BYTE    14             ; / code
          .BYTE    12             ; end-of-format
          .EVEN

L$VFE:    MOV      K, R0
          RTS      PC
```

4.2 FORMAT PROCESSING PSECTs

Six PSECTs are used for format and list-directed processing. $$FIOC contains the pure code of the format processor ($FIO) and the list-directed processors ($LSTI and $LSTO). $$FIOD contains pure data (constants and dispatch tables) used by these processors. $$FIOI contains the code for integer and octal conversions; $$FIOL contains the code for logical conversions; $$FIOR contains the code for floating point conversions. $$FIO2 contains only the addresses of the conversion routine entry points. Each module stores only its own entry point addresses in $$FIO2. The processing routines pick up the address of the conversion routine; if that address is 0, an error occurs.

None of the actual conversion routines reference the work area or any other portion of the OTS. They preserve R5 and the FPP registers and leave all other registers undefined.

4.3 FORMAT PROCESSOR -- $FIO

This module, in conjunction with the format conversion modules, processes formatted I/O. It operates as a co-routine with the I/O transmission operators. It is called at the end of formatted I/O initialization and continues to process list items and formats until called by $EOLST with offset VARAD equal to 0. $FIO processes through the format, calling an internal routine for each format code. Variable format expressions are called as encountered with all context saved and R5 restored to the user code value. When a format requiring a list item is encountered, if no element remains (offset VARAD = 0) then processing terminates. If an element exists, a call is made to a conversion routine. 'A' format is handled within $FIO. Nested group repeat specifications are handled by a pushdown stack contained within the work area. Offset FSTKP points to the current position; offset FSTK is the base of the pushdown stack.

4.4 LIST-DIRECTED INPUT PROCESSOR -- $LSTI

This module, in conjunction with the format conversion modules, processes list-directed input. It acts as a co-routine with the I/O transmission operators. It is called at the end of I/O initialization by ISL$ and processes list items until called with offset VARAD equal to 0.

$LSTI lexically scans the external record and delimits a field of input characters, determines the data type of the field and calls the appropriate input conversion routine to perform the conversion. The resulting internal data value is converted to the type of the list element and moved to the list element. The currently active data value is stored in the work area at offset W.LICB, which is pointed to by offset W.LICP.

The parameters passed to the format conversion modules are the buffer pointer, the actual field width as determined by the delimiter scan, and a decimal part of 0 and scale factor of 0 for floating point conversions.

4.5 LIST-DIRECTED OUTPUT PROCESSOR -- $LSTO

This module, in conjunction with the format conversion modules, processes list-directed output. It acts as a co-routine with the I/O transmission operators. It is called at the end of I/O initialization by OSL$ and processes list items until called with offset VARAD equal 0.

$LSTO accepts the list element and determines a format based on the list element data type as follows:

```
BYTE          I5
LOGICAL*2     L2
LOGICAL*4     L2
INTEGER*2     I7
INTEGER*4     I12
REAL*4        1PG15.7
REAL*8        1PG25.16
COMPLEX*8     1X,'(',1PG14.7,',',1PG14.7,')'
Hollerith     1X, nA1 where n is the string length.
```

If the field length thus computed is longer than the remaining characters in the record, the current record is written and a new record begun. Each item is contained in a single record except for alphanumeric literals that are longer than a single record. A space is inserted at the front of each record for carriage control. The record length is fixed at 73 bytes, which yields 72 print positions.

4.6 OBJECT-TIME FORMAT COMPILER -- FMTCV$

Format specifications stored in arrays must be compiled during execution into the required form prior to use. This is accomplished as follows:

1. Push the address of the beginning of the array specification.

2. JSR PC, FMTCV$

This routine does not delete the stack argument but rather replaces its value with the address of the compiled format.

Object time formats are compiled into an internal buffer within the OTS whose length may be controlled by the FMTBUF option during task building. The format buffer address is stored at offset W.OBFL and its high address+1 is stored at W.OBFH. Offset FMTAD points to the current entry in the output format buffer.

Within the FMTCV$ processing routines, R5 points to the source characters; R0 contains the current source byte; R2 contains any numeric value being accumulated; offset NOARG notes the number of expected arguments for this code; offset PARLVL notes the parentheses depth encountered; and offset NUMFL notes that a number is available in R2.

The module proceeds through the source characters. If the character is a digit, a number is accumulated; if a letter, a dispatch is made to process the format code; if a special character, a dispatch is made to process the format code.

If the buffer space is exhausted, the FMTBIG format code (2) is stored in the first byte of the compiled format and processing ceases. If a format syntax error is detected, the FMTBAD format code (0) is stored in the first byte and processing ceases.

4.7 INTEGER AND OCTAL CONVERSIONS

For input, OCI$ is called for octal conversion and ICI$ for integer conversion. The calling sequence is:

1. push address of input string
2. push number of input characters
3. call ICI$ or OCI$

The return is made with arguments deleted and a 2-word value on the stack in Integer*4 format. If an error occurs, the C-bit is set and the value returned is 0. An entry point $ECI is called by the floating point conversions to input the exponent field.

For output, OCO$ is called for octal conversion and ICO$ is called for integer conversion. The calling sequence is:

1. push address of output field
2. push width of output field
3. push Integer*4 value
4. call ICO$ or OCO$

The return is made with arguments deleted. If an error occurs, the C bit is set and the output field is filled with asterisks.

4.8 LOGICAL CONVERSIONS

For input, LCI$ is called as follows:

1. push address of input field
2. push width of input field
3. call LCI$

The return is made with arguments deleted and a 1-word value on the stack; 0 for .FALSE., and -1 for .TRUE. If an error occurs, the C-bit is set and .FALSE. is returned.

LCO$ is called for output as follows:

1. push address of output field
2. push width of output field
3. push 1-word word logical value
4. call LCO$

The return is made with arguments deleted and C-bit clear.

4.9 REAL, DOUBLE PRECISION AND COMPLEX CONVERSIONS

For input, RCI$ is called for all formats (D, E, F and G format codes) as follows:

1. push address of input field
2. push width of input field
3. push decimal part width
4. push scale factor (P format)
5. call RCI$

The return is made with arguments deleted and a 4-word double precision value on the stack. If an error occurs, the C-bit is set and the value returned is 0.0. If an exponent subfield is encountered, a call is made to $ECI in the integer input conversion routine to handle the conversion. The conversion is done entirely in software; the FPP unit is not used.

For output, the call is made as follows:

1. push address of output field
2. push width of output field
3. push decimal part width
4. push scale factor
5. push 4-word double precision value
6. call DCO$ or ECO$ or FCO$ or GCO$

The return is made with arguments deleted. If an error occurs, the C-bit is set and the output field is filled with asterisks. The conversion is done in software, without using the FPP unit.

The optional module provided, F4PCVF, is an FPP implementation that is significantly faster but slightly less accurate. The entire FPP state is conserved.

4.10 FORMAT CONVERSION ERROR PROCESSING

When a format conversion error occurs, both methods of error continuation, ERR=transfer and return, are generally supported. The action taken for each error that supports return continuation is as follows:

Error 59 - List-Directed I/O Syntax Error
-- Result value is null (no change).

Error 61 - Format/Variable Type Mismatch Error
-- Value is used as is.

Error 63 - Input Conversion Error
-- Result value is 0, 0. or 0D0.

Error 64 - Output Conversion Error
-- Field is filled with asterisks.

Table 2-7 in the FORTRAN IV-PLUS User's Guide gives the initial settings of the error continuation-type. The system subroutine ERRSET is used to change the initial settings.

CHAPTER 5

ERROR PROCESSING AND EXECUTION CONTROL

This chapter discusses the detection and processing of run-time errors, and the generation of error message output.

5.1 TRAP INSTRUCTION PROCESSING

The OTS reports errors using a TRAP instruction with the error number contained in the low byte of the instruction. The error number internally is 128(decimal) larger than the reported number, i.e., error 21 is represented internally as 149. The first 128 TRAP values are available to the user; see Section 5.6.

Upon execution of a TRAP instruction, the operating system transfers control to the TRAP instruction processor, $SST6. This routine checks the range of the error number and, if it is valid, calls $ERRAA to do the error analysis and reporting. If invalid, an invalid error number error (Error 1) is reported. All processor registers are preserved by the error processing routines.

5.2 ERROR CONTROL BYTE PROCESSING

$ERRAA obtains the error control byte from the OTS impure area and does the processing based upon the contents of that byte. If offset FILPTR is non-zero and the F.ERR byte of the corresponding FDB is negative, then the values F.ERR, F.ERR+1 and F.LUN are saved for the error report.

The error-occurred bit is set in the error byte and the error analysis is performed as follows. If the continue bit is off, the error report is printed with the exit flag. If the count bit is on, offset W.ECNT is decremented; if it is less than or equal to zero, the report is printed with the exit flag. If the continue-type bit indicates an ERR= transfer and no ERR= address exists, the error report is printed with the exit flag. If the log bit is set, the error report is printed with the no-exit flag. If the task exits, the message is always logged.

$ERRLG is called to log all terminal messages, both error reports and the messages from STOP and PAUSE statements.

5.2.1 Continuation-type Processing

Two types of continuation after an error are supported: return to the source of the error for corrective action; or transfer to an ERR= address. Most I/O errors provide ERR= support, but not return, while most other errors provide return.

5.2.2 W.IOEF Error Processing

For certain I/O errors, it is convenient for the ERR= transfer not to be initiated by the error processor but for the I/O routine itself to take the ERR= transfer. For example, OPEN statement processing processes all keywords even though an error occurs for a keyword and then takes the ERR= exit. Work area offset W.IOEF is used to obtain this special processing. If W.IOEF is 0, default processing is enabled. If W.IOEF is negative, default processing is performed except that the ERR= transfer is not made and a return is made to the source of the error for the ERR= transfer. Note that regardless of the W.IOEF setting, if no ERR= address exists, the task will exit. If W.IOEF is set positive, a return continuation will always be executed. W.IOEF is initially zero and is reset to zero before exiting from a routine which utilizes it.

5.3 ERROR MESSAGE PROCESSING

The error message construction and processing is performed by numerous small routines. Message processing is begun by a call to $ERRLG. This routine controls the flow of message processing and calls the message utilities as required. It prints the task name and error number on the first line, message text if available on the second line, the program counter value at the time of the error on line three, the error count exceeded message on line four, the FCS data on line five, followed by the program unit traceback. Any line that is not available or is inappropriate is not printed. Offset W.PC contains the saved program counter value and controls the third line. Offset W.ECNT contains the error limit count and controls line 4. Offset W.FERR contains the F.ERR field of the FFDB and controls line 5. Messages are printed via the message output task (MO) on RSX-11D and IAS. On RSX-11M messages are output by issuing QIO's to the user's terminal (TI:).

The message construction process requires R3 to point to the work area, and R5 to point to the current position in the message text being constructed. Offset W.MOTY is 0 if MO is being used, and non-zero if QIO's to the terminal are being performed. Offset W.ERLN points to the beginning of the error message buffer.

$ERRLG is also called to output the messages from STOP and PAUSE statements. The type of message being generated is determined from the values of R0 and R1. A STOP or PAUSE message is signaled by R1=0 and R0 pointing to the message text block. An error message is signaled by R1 being non-zero and then R0=-1 if the task is exiting or R0=0 if the task is continuing.

5.3.1 Message Construction Utilities

These routines are used to construct the error report text in the error text buffer. These routines operate identically regardless of whether the message output task (MO) is in use or not.

Terminal QIO - Perform a QIO of message to the user's terminal. Compute the message length; set MO LUN number (offset W.MO, global symbol .MOLUN) in the QIO DPB. Issue the QIO. Wait for the QIO to complete.

$ATT - Initialize R5 to error message buffer and store a carriage return-line feed (CR-LF) as the first two

characters. Set R5 into offset W.MOA1.

$ERRNL - Start a new line. Stores a CR-LF into buffer.

$ERRZA - Perform a GTSK$S directive to obtain the task name. Call $ATT and call $R50AB to decode the Radix-50 task name.

$BINAS - Convert a binary number to decimal ASCII.

$FILL - Move ASCIZ text pointed to by R1 to error message buffer pointed to by R5.

$R50AS,$R50AB - Convert Radix-50 value to ASCII by calling $R50.

5.3.2 Message Output Task (MO) Utilities

These routines in RSX-11D and IAS construct the MO parameter block (global symbol .MOPRM) and perform the QIO to MO.

$ERRW1 - Write the first segment. Store length of string in W.MOV1 and set W.MOA2 equal to R5.

$DETIC - Output in-core message text. Store no error number, call $ERRW1 to set up MO pointers and branch to $DET.

$DET - Output the message. Call MO if present, on error call $REAMO to reassign the MO lun to the terminal. If MO not present, call Terminal QIO. Go to $ATT.

$REAMO - Reassign the MO Lun. Perform ALUN$S of the MO unit number to the user terminal since MO is not present or not loaded and set the no MO switch (W.MOTY).

5.4 FLOATING POINT PROCESSOR ERRORS

All Floating Point Processor (FPP) errors are processed as Asynchronous System Traps in routine $FPERR. All possible FPP errors can be processed. For floating divide by zero, overflow and underflow, a value of zero is supplied as the result of the operation that caused the trap.

5.5 STOP AND PAUSE STATEMENT PROCESSING

STOP and PAUSE statements are compiled to calls as follows:

1. Push address of display (0 indicates no display)

2. Call statement specific entry:

```
STOP$ for STOP
PAUS$ for PAUSE
```

All context is saved. $ERRLG is called to output the message. STOP then jumps to $EXIT. PAUSE issues a SPND$S directive and returns.

5.6 USER INTERFACING TO ERROR PROCESSING

Users can use the first 128 (0 to 127) trap codes as follows. TRAP instructions transfer control to the OTS error processor by means of a System Synchronous Trap Table located in the OTS impure work area. The first word of this table has the global symbol $SST. Coding similar to the following can be used to intercept control:

```
;
; INITIALIZATION
;
INIT:     MOV      $SST+14,SST6        ;SAVE OTS TRAP ADDR
          MOV      #INTCEP,$SST+14     ;PUT NEW ADDR IN SST
                                       ;TABLE
          ...
SST6:     .WORD    0
          ...
; TRAP HANDLER
INTCEP:   CMP      #128.*2,@SP         ;LOW BYTE *2 OF TRAP
                                       ;INSTRUCTION FROM
                                       ;EXECUTIVE
          BHI      1$                  ;BRANCH IF USER CODE
          JMP      @SST6               ;GOTO OTS
1$:       ...                          ;USER TRAP
                                       ;PROCESSING CODE

          TST      (SP)+               ;DISCARD EXTRA WORD
                                       ;TRAP NUMBER
          RTI                          ;EXIT INTERRUPT
```

Similar techniques can be used to intercept the other synchronous traps.

5.7 USER INTERFACING TO TERMINAL MESSAGE OUTPUT

Users can utilize the error reporting message facility to write text on the user terminal without doing FORTRAN I/O. A message text block similar to that used for STOP and PAUSE statements is constructed as follows: R1 equal to 0; R0 points to a 2-word message block, the first word contains the address of an ASCIZ string (ASCII string terminated by a zero-byte), the second word is 0. The text is then output by executing a JSR PC, $ERRLG instruction.

Example:

The following prints "HELLO" on the user terminal:

In FORTRAN:

```
          CALL MSG ('HELLO')
          END
```

IN MACRO-11:

```
MSG::     CLR -(SP)            ; 2nd word of message block
          MOV 2(R5),-(SP)      ; address of ASCIZ text
          MOV SP,R0            ; R0 points to message block
          CLR R1               ; signal non-error type message
          JSR PC,$ERRLG        ; output the message
          CMP (SP)+, (SP)+     ; delete message block
          RTS PC               ; return
          .END
```

The user text will be preceded by the task name. Only a single line can be output. The message will appear as follows:

```
taskname  --  usertext
```

CHAPTER 6

MATHEMATICAL FUNCTIONS AND SYSTEM SUBROUTINES

This chapter summarizes the mathematical library functions and system subroutines. Detailed descriptions are contained in the FORTRAN IV-PLUS User's Guide.

6.1 PROCESSOR-DEFINED FUNCTIONS

Most of the processor-defined functions are called using the standard PDP-11 FORTRAN Calling Sequence Convention, (Section 1.2.2.1).

Some processor-defined functions are called using the F0 sequence described in Section 1.2.2.4. This calling sequence is used with the following internal entry names:

$$SIN	Real sine
$$DSIN	Double precision sine
$$SQRT	Real square root
$$DSQR	Double precision square root
$$ATAN	Real arctangent
$$DATN	Double precision arctangent
$$COS	Real cosine
$$DCOS	Double precision cosine
$$ALOG	Real logarithm (base e)
$$DLOG	Double precision logarithm (base e)
$$ALG1	Real logarithm (base 10)
$$DLG1	Double precision logarithm (base 10)
$$EXP	Real exponential (base e)
$$DEXP	Double precision exponential (base e)
$$TAN	Real tangent
$$DTAN	Double precision tangent

6.2 I/O-RELATED SUBROUTINES

ASSIGN-

The unit number is placed in R2 and $GETFILE is called to get the FFDB address. The file specification string address is placed in R1. The string length computed by scanning for a zero-byte if no length is present. $FNBST is called to parse the file specification and set up the file name block in the FDB.

CLOSE-

The unit number argument is moved to R2 and the OTS routine $CLOSE is called to close the file.

FDBSET-

The unit number is placed in R2 and $GETFILE is called to get the FFDB address. The first character of the access mode string is checked against the list and the corresponding file access is stored at F.FACC in the FDB:

'NEW'	-	FO.WRT
'OLD'	-	FO.UPD
'READONLY'	-	FO.RD
'APPEND'	-	FO.APD
'MODIFY'	-	FO.MFY
'ISUP'	-	FO.WRT + FA.NSP
'UNKNOWN'	-	FO.UPD

If the third argument begins with the character S, FA.SHR is set in F.FACC in the FDB. The fourth argument is stored at F.MBCT, the fifth at F.CNTG and the sixth at F.ALOC.

6.3 EXECUTION CONTROL SUBROUTINES

ERRSET-

The error number specified by the user is extracted and checked for validity. The following logical arguments are extracted and the appropriate bits in the error control byte are manipulated. If a limit count is provided, it is stored at offset W.ECNT.

ERRSNS-

This routine is called with zero to four integer arguments:

```
CALL ERRSNS (NUM, FERR, FER1, UNIT)
```

The saved information from the latest error is returned as follows:

offset	W.ERNM	into	NUM
offset	W.FERR	into	FERR
offset	W.FER1	into	FER1
offset	W.ERUN	into	UNIT

These offsets are then zeroed.

ERRTST-

The error number is retrieved and checked for validity. The error occurred bit of the error control byte is tested and cleared and the result returned in the second argument.

EXIT-

Performs a jump to $EXIT

USEREX-

Stores the argument address at work area offset EXADDR for use at task termination.

CHAPTER 7

COMPILED-CODE SUPPORT ROUTINES

This group of routines is used for performing arithmetic operations which are impractical to perform by in-line code sequences - notably exponentiation and complex arithmetic.

7.1 OUT-OF-LINE ARITHMETIC OPERATIONS

All of these entries follow a common naming convention in which:

1. The first two letters indicate the operation performed as follows:

   AD - addition
   SB - subtraction
   ML - multiplication
   DV - division
   PW - exponentiation
   CM - comparison
   TS - test for zero
   NG - negation

2. The next letter (two in the case of exponentiation) indicates the data type of the argument(s) as follows:

   I - Integer*2
   J - Integer*4
   R - Real
   D - Double Precision
   C - Complex

3. The last letter indicates how the argument for a 1-argument operation, or the second (right hand) argument for a 2-argument operation is accessed. For 2-argument operations, the first (left hand) argument is always on the stack.

   S - indicates the argument is at top of stack
   C - indicates that the following in-line word is the address of the argument
   P - indicates that the following in-line word is the offset in the parameter list (pointed to by R5) which contains the address of the argument.

All of these entries delete their stack arguments, return their result on the stack, and preserve the contents of general register 5 (R5).

7.1.1 Exponentiation -- PWxxt$

The following entries are used:

```
PW(I,J,R, or D)(I,J,R, or D)(S,C, or P)$
PWC(I or J)(S,C, or P)$
```

The FORTRAN statement:

```
I = (J**K)**(L**M)
```

is compiled into the code:

```
MOV   J, -(SP)       ; push J
JSR   R4, PWIIC$     ; compute J**K
.WORD K              ; result on stack = t1
MOV   L, -(SP)       ; push L
JSR   R4, PWIIC$     ; compute L**M
.WORD M              ; result on stack = t2
JSR   R4, PWIIS$     ; compute t1**t2
MOV   (SP)+, I       ; store result at I
```

7.1.2 Complex Arithmetic Operations

Complex Add, Subtract, Multiply, Divide, Test for Zero, Negate and Compare.

The following entries are used:

```
ADC(S,C, or P)$
SBC(S,C, or P)$
MLC(S,C, or P)$
DVC(S,C, or P)$
TSC(S,C, or P)$
NGC(S,C, or P)$
CMC(S,C, or P)$
```

Example:

The FORTRAN statements:

```
SUBROUTINE S(C)
COMPLEX A,B,C
A= -(B * C)
```

are compiled into the code:

```
MOV   B+6,-(SP)
MOV   B+4,-(SP)
MOV   B+2,-(SP)
MOV   B,-(SP)        ; push B
JSR   R4, MLCP$      ; multiply by C
.WORD 2              ; first argument
JSR   R4, NGCS$      ; negate result
MOV   (SP)+, A
MOV   (SP)+, A+2
MOV   (SP)+,A+4
MOV   (SP)+,A+6      ; store at A
```

7.1.3 INTEGER*4 Operations MLJt$ and DVJt$

The following entries are used:

```
MLJ(S,C, or P)$
DVJ(S,C, or P)$
```

Example:

The FORTRAN statements:

```
INTEGER * 4 I,J,K
I= J/K
```

are compiled into the code:

```
MOV   J+2,-(SP)
MOV   J,-(SP)
JSR   R4, DVJC$
.WORD K
MOV   (SP)+,I
MOV   (SP)+,I+2
```

7.1.4 STACK SWAP OPERATIONS SWPxy$

These routines are used in conjunction with the R4 entries for those cases in which the order of evaluation causes the two arguments of an R4 call to be on the stack in reverse order. Entry names are of the form

```
SWPlr$
```

where

l is the number of words the left argument occupies: 1, 2, or 4

r is the number of words the right argument occupies: 1, 2, or 4.

The two arguments are swapped on the stack.

Example:

The FORTRAN statements:

```
INTEGER*4 K,L
INTEGER*2 I,J
K= L**J**I
```

are compiled into the code:

```
MOV   J,-(SP)
JSR   R4,PWIIC$
.WORD I
MOV   L+2,-(SP)
MOV   L, -(SP)
JSR   PC,SWP21$
JSR   R4, PWJIS$
MOV   (SP)+K
MOV   (SP)+,K+2
```

7.2 ARRAY PROCESSING SUPPORT

An Array Descriptor Block (ADB) is a data structure provided by the compiler which describes an array (see Section 2.5). FORTRAN IV-PLUS compiled code uses ADBs for the following purposes:

1. Array subscript calculations for dummy argument arrays,
2. Input/Output calls that transmit an entire array, and
3. Array subscript limit checking when specified by the compiler /CK command switch.

Constant parts of an ADB are defined by the compiler. Varying parts are initialized at run-time upon entry to the subprogram which contains the array declaration.

7.2.1 Adjustable Array Initialization

Three entries are used for initializing the contents of ADB's for dummy argument adjustable arrays: MAK1$, MAK2$ and MAKN$. MAK1$ is called for 1-dimensional arrays, MAK2$ for 2-dimensional arrays and MAKN$ for arrays with three to seven dimensions. Only R5 is preserved by these routines which are called as follows:

1. Push the dimension bounds for any non-constant elements onto the stack in order of their appearance in the array declarator.
2. Push the base address of the dummy argument array passed in the subprogram call.
3. Push address of array descriptor block onto stack.
4. JSR PC, MAK1$, MAK2$ or MAKN$.

The FORTRAN statements:

```
SUBROUTINE X(A,N)
DIMENSION A(0:N-1,N)
```

are compiled into the code:

```
        MOV    @4(R5), R0       ; get N
        DEC    R0               ; compute N-1
        MOV    R0, -(SP)        ; push N-1
        MOV    @4(R5), -(SP)    ; push N
        MOV    2(R5), -(SP)     ; push address of A
        MOV    #A.ADB, - (SP)   ; push address of ADB
        JSR    PC, MAK2$        ; initialize 2-dimensional ADB

        .WORD      0            ; L1
        .WORD      0            ; U1
        .WORD      1            ; L2
        .WORD      0            ; U2
        .WORD    120000         ; A.PWRD
        .WORD      0            ; A.SIZB
A.ADB:  .WORD      0            ; A.ASTR
        .WORD      0            ; A.AO
        .WORD    31004          ; A.CWRD
        .WORD      0            ; D1
        .WORD      0            ; D2
```

7.2.2 Array Subscript Checking

If the compiler switch option /CK is in effect then each array reference will be checked to verify that the array element address is within the bounds established for the array by the array declarator.

The form of the call is:

1. Push array element address onto stack.
2. Push address of array descriptor block.
3. JSR PC,ARYCK$.

This call preserves all registers.

7.3 COMPUTED GO TO STATEMENT SUPPORT

A computed GO TO statement is compiled to a call as follows:

1. Push address of label list.
2. Convert index expression value to Integer*2 (if needed) and push.
3. JSR PC,CGO$.

The FORTRAN statement:

```
GO TO (20,1,99,30,10), I+4
```

is compiled into the code:

```
        MOV   #CGLST, -(SP)   ; address of label list
        MOV   I, R0
        ADD   #4, R0          ; compute I+4
        MOV   R0, -(SP)       ; push value
        JSR   PC, CGO$

CGLST:  .WORD   5
        .WORD   .20
        .WORD   .1
        .WORD   .99
        .WORD   .30
        .WORD   .10
```

7.4 ASSIGNED GO TO STATEMENT SUPPORT

An assigned GO TO statement is compiled to a call as follows:

1. Push assigned label address
2. Push address of allowed label list
3. JSR PC,AGO$.

The FORTRAN statement:

```
GO TO IV, (20,30,10)
```

is compiled into the code:

```
        MOV   IV, -(SP)       ; push value
        MOV   #AGLST, -(SP)   ; push address of label list
        JSR   PC, AGO$

AGLST:  .WORD   3
        .WORD   .20
        .WORD   .30
        .WORD   .10
```

7.5 TRACEBACK CHAIN PROCESSING

If the compiler command option /TR:NAMES, /TR:BLOCKS or /TR:ALL is in effect upon entry to the program unit, a call will be made to link the program unit name into the OTS name list for producing the error traceback information. The form of the call is:

1. Push last three letters of entry name (represented in Radix-50) onto stack.

2. Load first three letters of entry name into register R4.

3. JSR R4,NAM$

The traceback information is maintained on the execution stack. The NAM$ returns in a co-routine fashion so that, when the program unit finally returns, it actually returns to the NAM$ routine, which resets the stack, removes the name chain link, and then returns to the ultimate caller.

7.6 TASK INITIALIZATION

The first instruction of every FORTRAN main program is a call to the OTS initialization routine as follows:

```
JSR    PC,OTI$
```

This routine initializes the OTS. It issues an SVTK$S directive to initialize the synchronous trap table; it calls $STFPP to initialize the floating point processor; and it calls .FINIT to initialize FCS for I/O operations. It computes the number of available device table entries and zeroes them. It copies the error control byte table into impure storage. It computes the user I/O buffer length and clears miscellaneous words in the work area that must be zero initially. All FORTRAN programs are dynamically initialized so that tasks that are fixed in memory may be re-executed.

CHAPTER 8

OPERATING SYSTEM INTERFACES

This chapter describes the OTS interfaces to the operating system. The OTS performs input/output through File Control Services and relies on the Task Builder for allocation of impure storage and initialization of address pointers.

8.1 FILE CONTROL SERVICES (FCS)

All FORTRAN I/O is performed through FCS with synchronous record-oriented I/O so as to obtain FCS support of record blocking and deblocking.

Not all of the facilities available in FCS are used. The facilities that are used are described below.

Only 13 entries of FCS are called directly by the FORTRAN OTS. However, each of these modules may require additional FCS subroutines. The calls to FCS routines are always contained within a single module of the OTS. The OTS is partitioned so that no call is made to an FCS routine unless required by the FORTRAN program.

8.1.1 Direct Access Input -- GET$R

The GET$R macro is invoked from the OTS routine $GETR to perform direct access input.

8.1.2 Direct Access Output -- PUT$R

The PUT$R macro is invoked from the OTS routine $PUTR to perform direct access output.

8.1.3 Sequential Input -- GET$S

The GET$S macro is invoked from the OTS routine $GETS to perform sequential input. GET$S is also invoked by BKSP$ to read forward after rewinding the unit.

8.1.4 Sequential Output -- PUT$S

The PUT$S macro is invoked from the OTS routine $PUTS to perform sequential output.

8.1.5 File Open Processing -- OFNB$

All file open operations are performed by OFNB$. This means that if the filename parsing logic is not required, i.e., FORTRAN default file names are used, the routines for filename parsing are not included in the task. The OFNB$ macro is invoked from the OTS routine $OPEN$.

8.1.6 Default Directory Processing -- .GTDID

This routine is called by $OPEN$ to obtain the default directory for use in constructing the filename block for use by OFNB$.

8.1.7 File Name Block Processing -- .PARSE, CSI$1 and CSI$2

All of these routines are required if a user-specified file specification is to be used rather than the FORTRAN default filenames. These modules are called by $FNBST.

8.1.8 File Positioning -- .POINT

The OTS routines REWI$ and BKSP$ call .POINT to reposition the file to the beginning (byte 0, virtual block 1).

8.1.9 Direct Access Record Positioning -- .POSRC

The OTS routine $PUTRI calls .POSRC to position the file to a specified record for a direct access write.

8.1.10 File Close Processing -- CLOSE$

The OTS routine $CLOSE invokes the CLOSE$ macro to close a file.

8.1.11 File Deletion -- .DLFNB

The OTS routine $CLOSE calls .DLFNB to delete a file.

8.1.12 File Printing -- .PRINT

The OTS routine $CLOSE calls .PRINT to print and delete a file.

8.1.13 Register Save and Restore -- .SAVR1

Several OTS routines call .SAVR1 to save and restore registers R1 through R5 in co-routine fashion.

8.2 OVERLAYING THE OTS AND FCS

Because the I/O portion of the OTS is composed primarily of two parts, a natural overlay structure is to do file processing (open/close) on one branch and record processing (read/write) on the other. The following notes indicate potential problems and suggestions.

8.2.1 File Processing Overlay Notes

Because the OPEN statement (OPEN$) requires $CLOSE, these modules can not overlay each other, nor can the FCS routines .CLOSE, .PRINT, or .DLFNB overlay OPEN$. If OPEN$ is not used, $CLOSE may be on its own overlay branch.

$FNBST is the only routine that calls the system filename parsing routines, therefore a sub-tree with .CSI1 and .CSI2 on one branch and .PARSE on the other is an effective space saving arrangement.

8.2.2 Record Processing Overlay Notes

Because $INITIO is called by all I/O initialization modules and also references $OPEN, it should always be resident in the root.

$IOELEM and $IOARY should be root resident.

If formatted I/O is used, several constraints exist. The format processors ($FIO, $LSTI, $LSTO) must be in the same overlay segment as the format conversion routines ($CONVI, $CONVL, $CONVR). Either the record processors ($GETS, $PUTR, etc.) or the format processors must be root resident, or they must be in the same overlay segment.

If user code on more than one overlay does formatted I/O, the format processor should be root resident.

If an associated variable is declared for a direct access file, it must be root resident.

8.3 TASK BUILDER OPTIONS

At Task Build time, the Task Builder links in the FORTRAN impure area and, at the user's option, extends the impure storage.

8.3.1 UNITS=n

This specification causes the PSECT $$DEVT to have a size of n * F.FDB bytes. The value of n is stored at offset W.LUNS, the address of $$DEVT is at offset W.DEV and the address of the end of $$DEVT is at offset W.DEVL. The value of F.FDB is determined by the OTS to be the length of the FCS FDB (S.FDB) plus the length of the FORTRAN header (D.FDB). The Task Builder default value for n is 6.

8.3.2 ACTFIL=n

This specification causes the PSECT $$FSR1 to have a size of n*F.BFHD bytes. The value of F.BFHD is determined by the OTS to be the length of the FCS buffers (S.BFHD). The Task Builder default value for n is 4.

8.3.3 MAXBUF=n

This specification causes the PSECT $$IOB1 to be n bytes long. The address of $$IOB1 is stored at offset W.BFAD and the end address at offset W.BEND. The minimum and default value for n is 132. This value is the user record buffer length.

8.3.4 FMTBUF=n

This specification causes the PSECT $$OBF1 to be n bytes long. The address of $$OBF1 is stored at offset W.OBFL and the end address at offset W.OBFH. The minimum and default value for n is 64.

8.3.5 ASG=dv:n

This specification causes logical unit n to be initially assigned to device dv.

CHAPTER 9

OTS SYSTEM GENERATION AND TAILORING

The OTS is built during the installation process as described in the FORTRAN IV-PLUS Installation Guide. The material in this Chapter gives a more detailed explanation of the installation options, as well as information on building the OTS from sources.

9.1 ASSEMBLY OPTIONS

All assembly options are determined by the definition or non-definition of a symbol.

There are two operating system assembly options, two hardware assembly options and three special assembly options. No two options affect the same module, thus options can be combined.

9.1.1 Operating System Options

The two system option symbols are RSXD for RSX-11D V6 and IAS V1, and RSXM for RSX-11M V2. The following modules are affected:

```
$OTV    - impure area allocation
$ERRMO  - error report interface
$ERRLOG - error report construction
$ERRPT  - error processor
```

The modules $ERTXT and $SHORT are used only with RSX-11M.

9.1.2 EIS Instruction Set Option

The two hardware options are defined by the symbol FPP. If FPP is not defined, then the OTS can be used on an 11/45 or 11/40 with EIS, provided no floating point computations are attempted. (See Section 4.4.1 of the FORTRAN IV-PLUS User's Guide.)

The modules affected are:

```
$MLJ    - Integer*4 multiplication
$DVJ    - Integer*4 division
$JMOD   - Integer*4 modulo
$FPPUTI - FPP save/restore and initialization
```

9.1.3 Double Precision Arithmetic Option

The symbol F4PDP is used to assemble certain mathematical functions in double precision mode.

The modules affected are:

```
$ASIN  - arc sine
$ACOS  - arc cosine
$TAN   - tangent
$SINH  - hyperbolic sine
$COSH  - hyperbolic cosine
$TANH  - hyperbolic tangent
```

9.1.4 Floating Point Format Conversion Option

The symbol FPP is also used to define the floating point output conversion module that utilizes the FPP.

The module affected is:

```
$CONVR - floating point format conversion
```

9.1.5 No-I/O OTS Subset Option

The symbol NONIO is used to define a subset of the OTS that cannot perform FORTRAN input/output.

The modules affected are:

```
$OTV   - impure area allocation
$OTI   - task initialization
$CLOSE - file close processor
```

9.2 OTS ASSEMBLY MACROS

The OTS data base, PSECT attributes and errors are defined at assembly time by the following macros contained in the parameter file F4P.MAC.

9.2.1 OTSWA Macro

This macro defines the work area offsets. These offsets are described in Appendix A.

9.2.2 ERRDEF Macro

This macro defines the OTS errors, error control byte control bits and the error message text.

9.2.3 FBLOCK Macro

This macro defines the FFDB offsets, the FCS FDB offsets and the QIOSYM values. The FFDB offsets are described in Appendix B.

9.2.4 $AOTS Macro

This macro obtains the impure area pointer from location $OTSV and places it in a register, usually R3.

9.2.5 OTS$I Macro

This macro defines the OTS code PSECT $$OTSI.

9.2.6 OTS$D Macro

This macro defines the OTS pure area PSECT $$OTSD.

9.2.7 ADBDEF Macro

This macro, defined in the parameter file ADBDEF.MAC, defines the array descriptor block offsets and the data type codes.

APPENDIX A

IMPURE STORAGE OFFSET DEFINITIONS

This Appendix briefly describes each of the named offsets in the FORTRAN impure storage area.

OFFSET	SIZE (Decimal Bytes)	MEANING
W.SEQC	2	Current sequence number
W.NAMC	2	Listhead of traceback chain
W.LUNS	2	Number of FFDBs
W.MO	2	LUN used for error reporting
W.BFAD	2	User record buffer address
W.BLEN	2	User record buffer length
W.BEND	2	User record buffer end address +1
LNBUF	2	Address of active buffer
W.QIO	2	Address of QIO DPB used for error reports
W.DEV	2	Address of FFDB table
RECIO	2	Address of active I/O record processing routine
FMTAD	2	Current address in format
FILPTR	2	Address of active FFDB or 0
EOLBUF	2	Current buffer end address +1
FMTCLN	2	Value of SP at entry to current I/O operation, used for error recovery
BLBUF	2	Address of next data byte in current buffer
PSCALE	2	Floating point scale factor
FSTKP	2	Pointer to current format pushdown entry
W.LICP	2	Pointer to current data value for list-directed input
FSTK	32	Base of 16-word pushdown stack for format processing
W.LICB	10	List-directed input current data value block
NOARG	2	Object time format compiler argument count
PARLVL	2	Object time format compiler parentheses depth
NUMFLG	2	Object time format compiler argument number flag word
FMTRET	2	Format reversion pointer
VARAD	2	Current I/O list element address
TSPECP	2	Current maximum line length
TYPE	2	Format code type flag
REPCNT	2	Repeat count for current format code
UNFLGS	2	Flag word for unformatted I/O
LENGTH	2	Format width (w of w.d)
D	2	Format decimal part (d of w.d)
ITEMSZ	2	Current I/O list element size in bytes
DOLFLG	1	Dollar format encountered flag
COUNT	2	No. of array elements to transfer or no. of stack arguments for ERR= during OPEN and CLOSE statements
RACNT	2	No. of bytes left in direct access unformatted record
FMTLP	2	Infinite format loop flag

IMPURE STORAGE OFFSET DEFINITIONS

OFFSET	SIZE (Decimal Bytes)	MEANING
UNCNT	2	Unformatted sequential read record size
DENCWD	2	No. of records permitted for this I/O operation; 0 if no limit
W.PC	2	Saved PC value for traps
EXADDR	2	User exit routine address
ENDEX	2	Address for END= return
ERREX	2	Address for ERR= return
W.ECNT	2	Task error limit count
W.ERNM	2	Last error number that occurred
W.MAIN	2	FOR traceback word - spare but reserved
W.OPFL	2	OPEN/CLOSE statement error flag
W.ERLN	2	Address of error message text buffer
W.ERLE	2	Address of end of error message text buffer
W.TKNP	2	Address of task name - used by GTSK$S directive
W.ERTB	2	Address of error control byte table
W.FERR	2	F.ERR value of latest I/O error
W.FER1	2	F.ERR+1 value of latest I/O error
W.SST	2	Address of SST table
W.OBFL	2	Address of object time format buffer
W.OBFH	2	Address of end of object time format buffer
W.ERUN	2	Unit number of latest I/O error
W.FPST	2	FPP status save word
W.EXJ	2	I/O co-routine exchange jump location
W.PNTY	1	VFE mask byte
W.IOEF	1	Special error handling flag
W.R5	2	User's R5 value
W.VTYP	2	I/O list element data type code
W.RECL	2	Direct access record number (low order)
W.RECH	2	Direct access record number (high order)
W.FPPF	1	FPP present flag byte
W.DFLT	1	Default format code byte
W.LNMP	2	Number of negative mappable LUNS (4)
W.PRNT	2	Actual LUN for LUN -1
W.TYPE	2	Actual LUN for LUN -2
W.ACPT	2	Actual LUN for LUN -3
W.READ	2	Actual LUN for LUN -4
W.MOPR	2	Address of MO parameter list
W.MOV1	2	1st string length in MO parameter list
W.MOA1	2	1st string address in MO parameter list
W.MOV2	2	2nd string length in MO parameter list
W.MOA2	2	2nd string address in MO parameter list
W.MOTC	2	No. of MO traceback levels
W.MOTR	2	Start of MO traceback chain (wd 1)
W.MOT2	2	Start of MO traceback chain (wd 2)
W.MOTY	1	Error message mode byte
W.DEVL	2	Address of device table end
W.CPXF	1	Complex I/O list item flag byte
W.NULL	1	List-directed input null flag (slash seen)
W.END	-	End of named work area offsets

Offsets FSTKP and W.LICP occupy the same memory location.
Offsets FSTK and W.LICB begin at the same memory location.
Offsets NOARG, PARLVL and NUMFLG begin at offset FSTK.
Offsets RACNT, FMTLP and UNCNT occupy the same memory location.
Offsets REPCNT and UNFLGS occupy the same memory location.

APPENDIX B

FFDB OFFSET DEFINITIONS

This Appendix summarizes the FORTRAN header portion of the FORTRAN File Descriptor Block. Consult the operating system's I/O Operations Reference manual for the description of the FCS FDB.

NAME	SIZE (Bytes)	MEANING
D.STAT	2	Status word 1
D.STA2	2	Status word 2
D.RCNM	2	Direct access record count (low order)
D.RCCT	2	Sequential record count (low order)
D.RCN2	2	Direct access record count (high order)
D.RCC2	2	Sequential record count (high order)
D.AVAD	2	Associate variable address
D.SPAR	2	Reserved spare word
D.FDB	-	Start of FCS FCB

Offsets D.RCNM and D.RCCT occupy the same memory location.
Offsets D.RCN2 and D.RCC2 occupy the same memory location.

APPENDIX C

OTS SIZE SUMMARY

This Appendix is a guide to the approximate sizes of all the modules in the FORTRAN IV-PLUS OTS. Modules are grouped by related function, and identified by the TITLE as shown in Task Builder storage allocation maps. All object module sizes are shown in decimal words.

If the module size for RSX-11M is different from that for RSX-11D, the RSX-11M value is enclosed in parentheses.

C.1 MODULES ALWAYS PRESENT

		Module Size in Decimal Words
$OTI	OTS Initialization	76
$CLOSE	Close files	46
$ERRPT	Error reporting	244 (321)
$ERRLO	Error message logging	219 (296)
$ERRMO	Error message output	113 (36)
$ERTXT	Error message ASCII text	- (933 or 1)
$FPERR	FPP Interrupt handler	54
$FPPUT	FPP Utilities	37
$R50	Radix-50 to ASCII	44
$OTV	OTS Impure area (by PSECT)	
	$$AOTS Common Work Area	256 (231)
	$$DEVT Logical Unit Control Table (Size=UNITS*54)	324
	$$FSR1 FCS Buffer area (Size=ACTFIL*264)	1056
	$$IOB1 I/O Buffer (Size=max(MAXBUF,66))	66
	$$OBF1 Object Time Format Buffer (Size=max(FMTBUF,32))	32
	$$FSR2 FCS impure area	21
	$$OTSI Mixed FORTRANs trap	1
	$OTV Total (maximum)	1756 (1732)

(If UNITS=0 and ACTFIL=0, minimum $OTV size is 376 (352) words)

C.2 COMMON I/O SUPPORT

The following modules are used by all I/O operations.

		Module Size in Decimal Words
$OPEN	Default File Open	246
$INITI	Initialize Read/Write	138
$IOELE	I/O Element Transmission	155
$FCHNL	Find Logical Unit Control Block	61
$SAVRG	Register Save/Restore	53
	Total	653
$IOARY	Array I/O Transmission	72

C.3 FORMAT PROCESSING ROUTINES

The following routines are used by formatted I/O.

		Module Size in Decimal Words
$FIO	Format Interpreter	724
$LSTI	List-Directed Input	421
$LSTO	List-Directed Output	205
$CONVL	Logical Conversions (1)	35
$CONVI	Integer and Octal Conversions (1)	162
$CONVR	Real Conversions (1)	558
$FMTCV	Object Time Format Compiler (used only for formats stored in arrays)	326

C.4 SEQUENTIAL INPUT/OUTPUT

The following modules are needed for sequential input/output statements.

		Module Size in Decimal Words
$ISU	Sequential Unformatted READ	76
$OSU	Sequential Unformatted WRITE	78
$ISF	Sequential Formatted READ (2)	29
$OSF	Sequential Formatted WRITE (2)	36
$ISL	List-Directed READ (2)	46
$OSL	List-Directed WRITE (2)	38
$GETS	Get Sequential Record	34
$PUTS	Put Sequential Record	16

(1) Loaded only if needed or if list-directed or object time format is used.

C.5 DIRECT ACCESS INPUT/OUTPUT

The following modules are used for direct access input/output statements.

		Module Size in Decimal Words
$IRU	Direct Access Unformatted READ	38
$ORU	Direct Access Unformatted WRITE	40
$IRF	Direct Access Formatted READ (2)	33
$ORF	Direct Access Formatted WRITE (2)	35
$GETR	Get Direct Access Record	17
$PUTR	Put Direct Access Record	50
$CKRCN	Check Record Number, Update Associated Variable	42

C.6 OTHER I/O SUPPORT

		Module Size in Decimal Words
$BACKS	BACKSPACE Statement	75
$CLSST	CLOSE Statement	152
$DEFF	DEFINEFILE/FIND Statements	64
$ENCDE	ENCODE/DECODE Statements (2)	43
$ENDF	ENDFILE Statement	32
$FNBST	File Name Block Setup	70
$OPNST	OPEN Statement	433
$REWIN	REWIND Statement	35

C.7 I/O RELATED SUBROUTINE CALLS

		Module Size in Decimal Words
$ASSIG	ASSIGN Subroutine	49
$CLSCA	CLOSE Subroutine	9
$EXIT	EXIT Subroutine	2
$FDBSE	FDBSET Subroutine	95

C.8 MISCELLANEOUS COMPILED-CODE SUPPORT

		Module Size in Decimal Words
$AGO	Assigned GO TO Statement	12
$ARYCK	Array Subscript Checking	12
$CGO	Computed GO TO Statement	18
$MADB1	1-Dimensional Adjustable Array	34
$MADB2	2-Dimensional Adjustable Array	56
$MADBN	N-Dimensional Adjustable Array	58
$NAM	Traceback Chain Processing	15
$STPPA	STOP/PAUSE Statements	31

(2) Requires format processing routines.

C.9 PROCESSOR-DEFINED FUNCTIONS

		Module Size in Decimal Words
$ABS	Real Absolute Value	7
$ACOS	Arc Cosine	52
$AIMAG	Imaginary Part	6
$AINT	Real Truncation	9
$ALOG	Real Log	66
$AMAX1	Maximum of Reals	49
$AMIN0	Minimum of Reals	24
$AMOD	Real Modulo	15
$ANINT	Real and Double Nearest Integer	24
$ASIN	Arc Sine	47
$ATAN	Arc Tangent	120
$CABS	Complex Absolute Value	51
$CEXP	Complex Exponential	38
$CLOG	Complex Logarithm	28
$CMPLX	Complex From Reals	9
$CONJG	Complex Conjugate	10
$COSH	Hyperbolic Cosine	77
$CSIN	Complex Sine	51
$CSQRT	Complex Square Root	66
$DABS	Double Absolute Value	9
$DACOS	Double Arc Cosine	54
$DASIN	Double Arc Sine	49
$DATAN	Double Arc Tangent	158
$DBLE	Double From Real	7
$DCOSH	Double Hyperbolic Cosine	79
$DDIM	Double Positive Difference	17
$DIM	Positive Difference	13
$DINT	Double Truncation	7
$DLOG	Double Logarithm	96
$DMIN1	Minimum of Doubles	36
$DMOD	Double Modulo	17
$DPROD	Double Product of Reals	12
$DSIGN	Double Transfer of Sign	15
$DSIN	Double Sine	116
$EXP	Real Exponential	146
$FCALL	Internal Service Entry	7
$FLOAT	Integer*2 to Real	8
$FLOTJ	Integer*4 to Real	17
$I4FIX	Real to Integer*4	12
$IABS	Integer*2 Absolute Value	8
$IAND	Integer*2 AND	7
$IDIM	Integer*2 Positive Difference	10
$IEOR	Integer*2 Exclusive OR	6
$IFIX	Real to Integer*2	8
$IMOD	Integer*2 Modulo	11
$INOT	Integer*2 NOT	4
$IOR	Integer*2 Inclusive OR	5
$ISHFT	Integer*2 Shift	7
$ISIGN	Integer*2 Transfer of Sign	12
$JABS	Integer*4 Absolute Value	11
$JAND	Integer*4 AND	13
$JDIM	Integer*4 Positive Difference	23
$JEOR	Integer*4 Exclusive OR	11
$JMIX	Integer*4 Minimum and Maximum	46
$JMOD	Integer*4 Modulo	22
$JNOT	Integer*4 NOT	7
$JOR	Integer*4 Inclusive OR	9
$JSHFT	Integer*4 Shift	30
$JSIGN	Integer*4 Transfer of Sign	27

$MAX0	Integer*2 Maximum	10
$MIN0	Integer*2 Minimum	10
$NINT	Nearest Integer	19
$REAL	Real From Complex	5
$RJMIX	Real Maximum or Minimum of Integer*4	27
$SIGN	Real Transfer of Sign	11
$SIN	Real Sine	94
$SINH	Hyperbolic Sine	77
$DSINH	Double Hyperbolic Sine	79
$SNGL	Real From Double	14
$SQRT	Square Root	43
$TAN	Real Tangent	20
$DTAN	Double Tangent	22
$TANH	Hyperbolic Tangent	76
$DTANH	Double Hyperbolic Tangent	78

C.10 COMPILED-CODE ARITHMETIC SUPPORT (R4 CALLS)

		Module Size in Decimal Words
$ADC	Add/Subtract Complex	29
$CMC	Compare Complex	22
$DVC	Divide Complex	38
$DVJ	Divide Integer*4	26
$MLC	Multiply Complex	27
$MLJ	Multiply Integer*4	24
$NGC	Negate Complex	16
$PWCJ	Complex to Integer Exponentiation	158
$PWDD	Floating to Floating Exponentiation	71
$PWII	Integer*2 to Integer*2 Exponentiation	53
$PWJJ	Integer*4 to Integer*4 Exponentiation	139
$PWRI	Floating to Integer Exponentiation	111
$PWRR	Real to Real Exponentiation	55
$SWPXY	Stack Swap	95
$TSC	Test Complex	16

C.11 SERVICE SUBROUTINES

		Module Size in Decimal Words
$DATE	DATE	70
$ERRSE	ERRSET	95
$ERRSN	ERRSNS	30
$ERRTS	ERRTST	22
$IDATE	IDATE	29
$IRAD5	IRAD50	15
$R5OAS	R5OASC	6
$RAD50	RAD50	11
$RAN	RAN	7
$RANDU	RANDU	11
$SECND	SECNDS	37
$TIME	TIME	43
$USERE	USEREX	6

C.12 OPTIONAL MODULES

		Module Size in Decimal Words
$CONVR	Real Format Conversions(FPP Version)	523
$FPPUT	EIS Version	
$SHORT	Null Error Message Text	1
$ERRLO	Null Error Message Logging	1
$CLOSE	No I/O OTS Version	2
$OTV	No I/O OTS Version	274
$OTI	No I/O OTS Version	50
$MLJ	EIS Version	58
$DVJ	EIS Version	75
$JMOD	EIS Version	25

FORTRAN IV-PLUS Object
Time System Reference
Manual
DEC-11-LFPOA-A-D

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

_____

_____

_____

_____

_____

_____

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

_____

_____

_____

_____

_____

_____

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

_____

_____

_____

_____

_____

_____

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

Please cut along this line.

---------- Fold Here ----------

---------- Do Not Tear - Fold Here and Staple ----------

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

digital

digital equipment corporation

Printed in U.S.A.

INDEX