PDP-11 FORTRAN-77 Object Time System Reference Manual

Order Number: AA-V195B-TK

August 1988

This document describes the object modules that are selectively linked with compiled PDP-11 FORTRAN-77 code by the appropriate operating system's task builder to produce an executable task.

Revision/Update Information:	This revised document supersedes PDP-11 FORTRAN-77 Object Time System Reference Manual, AA-V195A-TK
Operating System and Version:	RSX-11M Version 4.4 RSX-11M/M-PLUS Version 4.1 RSTS/E Version 9.6 VAX/VMS Version 4.7
Software Version:	Fortran-77 Version 5.3

digital equipment corporation
maynard, massachusetts

First Printing, July 1983
Revised, August 1988

The postpaid Reader's Comments forms at the end of this document request the user's critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DEC	DIBOL	UNIBUS
DEC/CMS	EduSystem	VAX
DEC/MMS	IAS	VAXcluster
DECnet	MASSBUS	VMS
DECsystem-10	PDP	VT
DECSYSTEM-20	PDT	
DECUS	RSTS	
DECwriter	RSX	digital™

ZK4337

Contents

FIGURES

TABLES

Preface

Manual Objectives

This manual contains detailed information about the FORTRAN–77 Object Time System (OTS) not contained in the *PDP–11 FORTRAN–77 User's Guide*. The information is not needed for typical use of FORTRAN–77; however, many users need to know more about the OTS for specialized applications. This manual is especially helpful to programmers interfacing MACRO–11 and FORTRAN–77 routines to the OTS.

Intended Audience

This manual assumes that the readers know MACRO and FORTRAN and are familiar with the information in the *PDP–11 FORTRAN–77 User's Guide*, their operating system's executive reference manual and I/O operations reference manual, and the *RMS–11 MACRO–11 Reference Manual*.

Internal OTS interfaces are *not* guaranteed to remain constant across releases of FORTRAN–77. Calling the OTS the same way as the compiled code is called and using the OTS named offsets ensure as much release-to-release compatibility as possible.

Document Structure

This manual contains 13 chapters and four appendixes.

- Chapter 1, "Object Time System Overview," provides a conceptual view of the structure of the OTS.
- Chapter 2, "Conventions and Standards," describes the calling sequences and naming conventions used by PDP-11 FORTRAN-77.
- Chapter 3, "Assembly Language Interfaces to the OTS," describes how to write MACRO-11 programs that interface with the OTS.
- Chapter 4, "Data Structures and Storage," describes the OTS work area and logical unit control table.
- Chapter 5, "I- and D-Space Tasks," explains how to increase the virtual task space available to your program.
- Chapter 6, "Overview of FORTRAN-77 Input/Output," provides a conceptual view of OTS I/O processing and describes the I/O modules that are accessed by both the FCS-11 and RMS-11 file management systems.
- Chapter 7, "FCS-11 Input/Output Support," describes the FCS-11 file management system operations that are used to implement PDP-11 FORTRAN-77 I/O operations.
- Chapter 8, "RMS-11 Input/Output Support," describes the RMS-11 file management system operations that are used to implement PDP-11 FORTRAN-77 I/O operations.
- Chapter 9, "Format Processing and Format Conversions," describes the internal form of format specifications, the format processing algorithm, and the format conversion routines.
- Chapter 10, "Error Processing and Execution Control," discusses execution control processing, the detection and processing of run-time errors, and the generation of error messages.
- Chapter 11, "Other Compiled-Code Support Routines," describes routines that support various arithmetic and housekeeping operations required by the compiled code.
- Chapter 12, "OTS System Generation and Tailoring," describes the OTS installation options.
- Chapter 13, "OTS Resident Libraries," describes how to create OTS libraries.
- Appendix A, "PDP-11 FORTRAN-77 Impure Area Definitions," shows the layout of the OTS work area described in Chapter 4.

- Appendix B, "FORTRAN-77 Logical Unit Control Block Definitions," describes the data structures used in OTS I/O processing.
- Appendix C, "OTS Size Summary," provides the approximate sizes of all the OTS modules.
- Appendix D, "Program Section Descriptions," describes the program sections (PSECTs) used by the OTS.

Associated Documents

The following documents provide related information:

- *PDP-11 FORTRAN-77 User's Guide*
- *PDP-11 FORTRAN-77 Language Reference Manual*
- *RMS-11 User's Guide*
- *RMS-11 MACRO-11 Reference Manual*
- *IAS/RSX-11 I/O Operations Reference Manual*

Conventions Used in this Document

The following syntactic conventions are used in this manual:

- All references to FORTRAN-77 denote PDP-11 FORTRAN-77, unless otherwise specified.
- Uppercase type is used in text to indicate system commands and command options.
- Lowercase letters are used in syntax specifications and examples to indicate variables; anything that is not a variable (for example, statement names and keywords) appears in uppercase.
- Brackets ([ ]) indicate optional elements within statements.
- Braces ({}) are used to enclose lists from which one element is to be chosen.
- Horizontal ellipses (. . .) indicate that the preceding item(s) can be repeated one or more times.

- "Real" (lowercase) is used to refer to the REAL*4 (REAL), REAL*8 data types as a group; likewise, "complex" (lowercase) is used to refer to COMPLEX*8; "logical" (lowercase) is used to refer to the LOGICAL*2 and LOGICAL*4 data types as a group; and "integer" (lowercase) is used to refer to the INTEGER*2 and INTEGER*4 data types as a group.
- Unless otherwise noted, numeric values are represented in decimal notation. Values in MACRO-11 examples are in octal notation.
- Unless otherwise specified, all commands end with a carriage return.

Chapter 1

Object Time System Overview

The FORTRAN–77 Object Time System (OTS) consists of assembly language modules that complement your compiled code. Most OTS routines are common to the RSX–11M/M–PLUS and RSTS/E operating systems, and the FCS-11 and RMS–11 file management systems. However, certain routines are supported only by a specific operating system or file management system. The FORTRAN–77 distribution kit allows you to configure systems individually for each file management or operating system.

The OTS has five main parts:

1. Tables, buffers, and impure storage that the OTS routines need
2. I/O processing routines
3. Task control and error-processing routines
4. Mathematical functions and system subroutines
5. Other compiled-code support routines

The rest of this chapter introduces and describes each of these parts of the OTS.

1.1 Tables, Buffers, and Impure Storage

The OTS uses data areas that include read-only constants, logical unit control tables, various buffers, and the impure storage area. Chapter 4 describes these data areas.

1.2 I/O Processing Routines

The I/O processing routines are a collection of small modules. Only those modules required by a given FORTRAN source program need to be linked into the user's task.

Chapter 6 describes the I/O system design and the I/O routines common to the FCS-11 and RMS-11 file management systems. Chapter 7 discusses FCS-11-specific routines, Chapter 8 discusses RMS-11-specific routines, and Chapter 9 contains information on format processing routines.

1.3 Task Control and Error-Processing Routines

For every FORTRAN main program, the compiler inserts a call to OTS initialization. You can control program termination by using the USEREX subroutine to set up a procedure that is called when a program terminates.

When the OTS detects an error, it executes a TRAP instruction with the error number in the low byte of the instruction. A service routine within the error-processing modules handles floating-point processor asynchronous traps.

There are two methods of error recovery: use an 'ERR=' transfer within an I/O statement, or a return to the error site for appropriate action. A byte in the OTS impure storage determines which action to take. Each defined error number corresponds to an error control byte that you can access using the FORTRAN-callable subroutines ERRSET, ERRTST, and ERRSNS.

For more information on these subroutines, see Chapter 10.

1.4 Mathematical Functions and System Subroutines

The *PDP-11 FORTRAN-77 User's Guide* describes how to use special names to call mathematical routines from compiled code. These routines are known as processor-defined functions. Appendix B of the *PDP-11 FORTRAN-77 User's Guide* describes the algorithms for these mathematical library routines, and Appendix D describes the system subroutines.

1.5 Compiled-Code Support Routines

Compiled-code support routines complement the compiled code by performing operations too complicated or cumbersome to perform with in-line code, such as array subscript checking, exponentiation, character assignment and comparison operations, and complex arithmetic.

For more information on these routines, see Chapter 11.

Chapter 2

Conventions and Standards

FORTRAN–77 has specific procedural and naming conventions. The following sections describe these conventions.

2.1 Registers

The eight processor general registers are referenced as follows:

R0 to R5 = Registers 0-5

SP = Register 6

PC = Register 7

The six floating-point processor accumulators are referenced as F0-F5.

2.2 Calling Sequences

FORTRAN–77 compiled code uses the following four calling sequence conventions to call components of the OTS:

1. R5 Calls—for all system subroutines, most processor-defined functions, and all user-routine calls
2. PC Calls—for I/O operations, system-dependent routines, and character assignment and comparison operations
3. R4 Calls—for out-of-line, stack-oriented arithmetic routines and certain compiled-code support routines
4. F0 Calls—for faster calls to certain processor-defined functions

The following sections describe these calls.

2.2.1 R5 Calls

This calling sequence convention is the standard for FORTRAN–77. The basic form for these calls 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
```

The argument list must reside in data-space and, except for label type arguments, addresses in the list must also refer to data-space.

User programs should not reference the byte at address LIST+1. It is reserved for future use by DIGITAL software. References to LIST + 1 may cause unpredictable results.

Control returns to the calling program by restoring (if necessary) the stack pointer (SP) to its value on entry and executing an RTS PC instruction.

Function subprograms return a single result in the processor general registers. The type of variable returned by the function determines which registers receive the result. Table 2–1 shows the variable types and their associated register assignments.

Table 2–1: Register Assignments for Subprogram Results (R5 Calls)

If the Result Type Is:	The Result Is 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

Calling programs use R0 through R5 to save values needed after a return from a subprogram. The argument list pointer value in register R5 may not be valid after return. Calling programs must save and restore the floating-point registers that they use. You cannot assume that the called routines will restore the floating-point status bits I/L (integer/long integer) or F/D (floating/double precision).

An address of -1 (177777 octal) represents a null argument in an argument list. The -1 address ensures that null arguments in subprograms that cannot handle them will result in an error when the routine is called. The errors most likely to occur are illegal memory references and word references to odd byte addresses.

For more information about this calling sequence convention, see the *PDP-11 FORTRAN-77 User's Guide*.

2.2.2 PC Calls

Use a JSR PC,xxx instruction to make PC calls. These calls pass all arguments on the stack and return with the arguments deleted from the stack. There are no changes to registers R0-R5, F0-F5, or the FPP status register. PC calls are used for the following operations:

- All I/O statements except OPEN and CLOSE
- STOP, PAUSE, computed GO TO, and assigned GO TO statements
- Character out-of-line support routines for assignment and comparison
- Array subscript checking, if enabled

For example,

The FORTRAN-77 statement

```
REWIND  3
```

is compiled into the code

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

2.2.3 R4 Calls

Use this convention 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. They delete the stack arguments and return a value on the stack. This type of routine is called by a JSR R4,xxx instruction. R4 calls modify the FPP status register and registers F0-F5 and R0-R4, but preserve R5. Chapter 10 describes the modules that use this convention.

For 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
```

2.2.4 F0 Calls

Commonly used processor-defined functions use this convention. It sets the FPP F/D status bit to the type of argument and loads the argument into F0. A JSR PC,xxx instruction calls this routine. It returns a result in F0 and preserves the FPP F/D status bit, but does not preserve registers R0-R5, F1-F5, and the FPP I/L status bit. The functions that use F0 calls are named $$xxxx, as shown in Table 2–2.

Table 2–2: Processor-Defined Functions

Name	Function
$$SIN	Real sine
$$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

For 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
```

2.2.5 Special Call Conventions

The following are exceptions to the four general calling conventions:

- OPEN (OPEN$) and CLOSE (CLOS$) statements use the R5 convention with a special argument list encoding.
- Run-time format compilation (FMTCV$) uses a PC call but returns a stack result for use in a subsequent I/O initialization call.
- Adjustable array initialization calls (MAK1$, MAK2$, MAKN$, and MAKV$) use a PC call but preserve only R5.
- Traceback name initialization (@NAM) uses a coroutine call.
- Virtual array processing ($VRTxy) uses a PC call that preserves all registers except R0.
- Task initialization ($OTI) uses a PC call that does not preserve the registers.
- The intrinsic function INDEX uses the R5 convention, but the addresses in the list point to 2-word (length, address) descriptors of the argument.

See the corresponding module descriptions in later chapters for more details on these special variants.

2.3 Labeling Conventions

The labels of OTS routines begin with a dollar sign ($) and are followed by the name or a contraction of the name. All external entry point names contain a dollar sign ($) as either the first or last character.

2.4 Context Save and Restore

The calling sequence determines the OTS register context conventions. Refer to Section 2.2.

Internal OTS calls use various conventions. In general, the calling routine saves those registers it requires. Registers not mentioned in the OTS routine descriptons are saved.

Chapter 3

Assembly Language Interfaces to the OTS

Chapter 2 describes how the compiled code that is output from your FORTRAN-77 source program compilation interfaces with the OTS. You also can write MACRO-11 programs that interface with the OTS. This chapter summarizes how you can set up that interface.

3.1 Writing a Fortran Main Program in Assembly Language

The following MACRO-11 code represents a hypothetical FORTRAN-77 main program:

```
START::
    JSR       PC, OTI$      ; Initialize the OTS and file management
                            ; system
        .
        .
        .
    MOV #^R<IN.>,-(SP)      ; Last 3 letters of name in RADIX-50
    MOV #^R<.MA>, R4        ; First 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
.GLOBL  ORGSQ$      ; RMS-11 sequential impure
.GLOBL  ORGRL$      ; RMS-11 relative impure
.GLOBL  ORGIX$      ; RMS-11 indexed impure
.END    START
```

Notes:

1. The call to OTI$ initializes the FPP (SFPA$S), the SST vector (SVTK$S), and FCS-11 (FINIT$) or RMS-11 ($INITIF).
2. The reference to $OTSVA loads the FORTRAN impure storage area.
3. The references to the FORMAT conversion routines are needed only if the desired conversion routine is required. (Note that a FORTRAN subprogram containing a FORMAT statement holds the required FORMAT conversion references.)
4. The RMS-11 impure storage references are needed only if RMS-11 is the file system your program needs to process a particular file organization.

3.2 Linkage to the FORTRAN Impure Storage Area

The FORTRAN impure storage area defines a global symbol $OTSVA, which is referenced by the compiled code in FORTRAN main programs. Subprograms do not reference this symbol. When the Task Builder processes a reference to $OTSVA, it loads the FORTRAN impure area and defines global symbol $OTSV in the task containing the address of the symbol $OTSVA. All FORTRAN OTS routines obtain the address of the impure area by referencing the location $OTSV (refer to the discussion of the $AOTS macro in Chapter 12).

Chapter 4

Data Structures and Storage

The OTS maintains two major areas of impure storage: the work area and the logical unit control table. This chapter describes these two areas.

4.1 Work Area Storage Description

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

For example, the work area contains:

- Named offsets

  The named offsets make up the first 120 words of the work area and have names of the form W.xxxx or xxxxxx. There are both word and byte offsets, and some of the offsets have an associated global symbol name.

- QIO directive parameter block

  The 12-word QIO directive parameter block (DPB) uses event flag 30 to send error messages to terminals. On RSX-11M/M-PLUS and RSTS/E systems, you use the DPB for all message output. The offset W.QIO points to the DPB.

- Error message text buffer

  The buffer for the error text message line is 70 bytes in RSX-11M /M-PLUS and RSTS/E. The offsets W.ERLN (start address) and W.ERLE (end address+1) point to the buffer.

- Error control table The error control table is 128 bytes, with one byte for each error. The error control table is an impure data area that the error-handling routines use and manipulate. The task initialization routine OTI$ copies a prototype version of the table into this area. The offset W.ERTB points to this table.
- Synchronous System Trap vector address table

  The Synchronous System Trap (SST) vector address table ($SST) contains an entry for each defined SST. The offset W.SST points to this table.
- Window block

  An 8-word address mapping window block is used by the virtual array processing routines. The virtual array initialization routine $VINIT initializes this window block. The offset W.WDB points to this window block.

In this section, the named offsets are organized into functional groups and described in Tables 4–1 through 4–8. The functional groups and their corresponding tables are as follows:

Task control—Table 4–1
I/O control—Table 4–2
Format control—Table 4–3
Run-time format control—Table 4–4
Error control—Table 4–5
Error message and traceback control—Table 4–6
Virtual array control—Table 4–7
Trap routines—Table 4–8

Table 4–1: Task Control Information

Global Symbol	Description	Global Name	Default
EXADDR	Address of USEREX routine or 0		
W.ACPT	Logical unit number for ACCEPT statements	$ACCPT	5
W.BEND	High address+1 of the user record buffer		
W.BFAD	Start address of the user record buffer		
W.BLEN	Length of the user record buffer; computed at task initialization time and equal to W.BEND - W.BFAD		133
W.DEV	Start address of the logical unit control table		
W.DEVL	For FCS-11, the high address+1 of the logical unit control table; for RMS–11, the address of the free storage		
W.END	Last word of named offsets		
W.EXST	Exit with status value		
W.EXTK	Size (in 64-byte units) of the task increment value for use in the EXTK$ directive	$EXTKL	16
W.FNML	Maximum length of file name strings nonblank characters	$MXFNL	80
W.FPPF	FP-11 flag byte; 0 if FP-11 present, 1 if not		
W.LIMT	Address of a .LIMIT directive block		
W.LNMP	Number of valid negative unit numbers		4
W.LUNS	Number of valid logical units	.NLUNS	
W.MO	Logical unit number for error messages	.MOLUN	
W.PRNT	Logical unit number for a PRINT statement	$PRINT	6
W.READ	Logical unit number for a READ statement	$READ	1
W.SKLM	Task's current stack overflow		
W.SST	Limit address of the SST table		
W.TKLM	Task current maximum virtual address		

Table 4–1 (Cont.): Task Control Information

Global Symbol	Description	Global Name	Default
W.TSKP	Address of the special PSECT, $$TSKP, containing task parameters		
W.TYPE	Logical unit number for a TYPE statement	$TYPE	5
W.LUN0	System logical unit number for FORTRAN-77 logical unit 0	$LUN0	0

Table 4–2: I/O Control Information

Global Symbol	Description
BLBUF	Address of next data byte in current I/O record
COUNT	Length of array in an I/O list
DENCWD	Maximum number of I/O records or 0 if no limit
ENDEX	Address of END= statement or 0
EOLBUF	End address+1 of current I/O record
ERREX	Address of ERR= statement or 0
FILPTR	Address of active I/O control block or 0
FMTCLN	Value of SP on entry to I/O processing
ITEMSZ	Size in bytes of current I/O list element
LNBUF	Start address of current I/O record
RACNT	Number of data bytes remaining in current I/O record
RECIO	Address of record-processing I/O routine (GET or PUT)
UNCNT	Number of data bytes remaining in record segment
UNFLGS	Segmented record control word
VARAD	Address of current I/O list element or 0
W.EXJ	Coroutine address of current I/O element processing routine
W.FDB1	Pseudo I/O control block for ENCODE/DECODE and internal files (word 1)
W.FDB2	Pseudo I/O control block for ENCODE/DECODE and internal files (word 2)
W.FPST	FP-11 status register at I/O entry

Table 4–2 (Cont.): I/O Control Information

Global Symbol	Description
W.KDSC	Character key descriptor address
W.KDTP	Key data type byte
W.KMAT	Key match criterion byte
W.KNUM	Integer key value (2 words)
W.KREF	Key-of-reference value
W.OPFL	Count of errors during OPEN or CLOSE statement processing
W.RECH	High-order direct access record number
W.RECL	Low-order direct access record number
W.UOPN	USEROPEN procedure address or 0
W.VTYP	Data type code of current I/O list element

Table 4–3: Format Control Information

Global Symbol	Description
D	Decimal fraction width of current format item
DOLFLG	Dollar sign format flag for the current I/O record
FMTAD	Address of current format byte
FMTLP	Infinite format loop flag
FMTRET	Address in format for format reversion
FSTK	Base of 16-word stack for format parenthesis nesting
FSTKP	Address in FSTK of current nesting level
LENGTH	Field width of current format item
PSCALE	P format value
REPCNT	Repeat count of current format item
TSPECP	Highest address used in current I/O record
TYPE	Current format code
W.CPXF	Complex data item flag: 1=real part; 0=not complex; -1=imaginary part
W.DFLT	Current default format code or 0

Table 4–3 (Cont.): Format Control Information

Global Symbol	Description
W.ELEM	Flag indicating data element has been processed
W.LICB	Base address of current list-directed data value control block in previous versions of PDP–11 FORTRAN IV-PLUS
W.LICP	Address in list-directed data value control block of current data value in previous versions of PDP–11 FORTRAN IV-PLUS
W.NULL	Flag indicating a slash separator character was seen during list-directed input processing
W.PLIC	Address in list-directed data value control block of current data value
W.PNTY	Variable format expression flag byte
W.R5	Saved R5 value for variable format expressions
W.SPBN	The SP/SS, BN/BZ, and T format flags

Table 4–4: Run-Time Format Control Information

Global Symbol	Description
NOARG	Number of arguments required by current format code
NUMFLG	Current numeric value
PARLVL	Current[parenthesis] level
W.OBFH	End address +1 of run-time format buffer
W.OBFL	Start address of run-time format buffer

Table 4–5: Error Control Information

Global Symbol	Description
W.ECNT	Task error limit count, global name: $ERCNT
W.ERNM	Last error number or 0
W.ERTB	Start address of error control table
W.ERUN	Logical unit number of last I/O error or 0
W.FERR	Primary I/O error code of last I/O error or 0
W.FER1	Secondary I/O error code of last I/O error or 0
W.IOEF	Special I/O error processing flag
W.PC	PC value of SST and FP-11 errors
W.QIO	Address of error message QIO DPB

Table 4–6: Error Message and Traceback Control Information

Global Symbol	Description
W.ERLE	End address+1 of error message text buffer
W.ERLN	Start address of error message text buffer
W.MOA1	MO first text segment address
W.MOA2	MO second text segment address
W.MOPR	Address of MO parameter list
W.MOTC	MO traceback count
W.MOTR	MO traceback list head
W.MOTY	Error message type byte: 0=MO, 1=QIO
W.MOT2	MO traceback current statement number
W.MOV1	MO first text segment length
W.MOV2	MO second text segment length
W.NAMC	Traceback chain list head, global name: $NAMC
W.QIO	Address of error message QIO DPB
W.SEQC	Traceback current statement number, global name: $SEQC
W.TKNP	Address of task name in error message text buffer

Table 4–7: Virtual Array Control Information

Global Symbol	Description
W.WDB	Address of window block for mapping
W.WNHI	Current high-window address+1
W.WNLO	Current low-window address

Table 4–8: Trap Routine Information

Global Symbol	Routine Whose Address Contained
W.ERXT	$ERXIT
W.ERLG	$ERRLG
W.FIN	$EXIT
W.FPER	$FPERR
W.NAM	NAM$
W.IOXT	$IOEXIT
W.RLME	RLMEM$
W.RQME	RQMEM$
W.GSA	RMSQL$

4.2 Logical Unit Control Table

The logical unit control table contains a block of storage for each logical unit allocated to the FORTRAN–77 OTS. Each block contains complete information required by the OTS to perform I/O to the associated unit.

All FORTRAN–77 I/O is performed on logical units. Each logical unit has a control block (LUB). The allocation and manipulation of the control blocks depends on the file system in use: FCS-11 or RMS–11.

The following sections describe the LUB symbolic names and their use. Appendix B contains the offset values for each symbolic name.

4.2.1 Common LUB Definitions

The first two words of each control block are status words (D.STAT and D.STA2). Certain bits in these status words are defined the same way in both the FCS-11 and RMS–11 file systems to support common I/O processing as much as possible.

The following bits in D.STAT are defined the same way in both file systems:

DV.FAK— partial control block for ENCODE/DECODE and internal file usage
DV.FMP— formatted logical unit
DV.FRE— free format disallowed ("short field termination")
DV.OPN— open logical unit
DV.RW — current operation type: 0= READ, 1= WRITE
DV.UFP— unformatted logical unit

The following bit definition in D.STA2 is common to both file systems:

DV.BN — BLANK = 'NULL' specified

4.2.2 LUB Definitions For FCS-11 Support

Each logical unit has a LUB in the $$DEVT program section (PSECT). There is one LUB allocated for each unit declared in the task builder UNITS= statement (if the UNITS= parameter is not specified, the default value is six logical units). Each LUB is a fixed-length block consisting of an FCS-11 File Descriptor Block (FDB) and a 6-word header for FORTRAN usage. At task initialization time, each LUB is set to zero (0). A close operation also sets each LUB to zero (0).

Offsets of the form D.xxxx describe the FORTRAN header portion of the LUB, as follows:

D.STAT — status word 1 (see below)
D.STA2 — status word 2 (see below)
D.RCNM — direct access record limit (low order)
D.RCN2 — direct access record limit (high order)
D.RCCT — record count for BACKSPACE (low order)
D.RCC2 — record count for BACKSPACE (high order)
D.AVAD — address of associated variable address or 0
D.RSIZ — maximum record length
D.FDB — start of FCS-11 FDB

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

The FORTRAN header portion of the LUB contains two status words. The bits in these status words have symbolic names of the form DV.xxxx. These bits are defined as follows:

Status Bits for Word 1 (D.STAT)

Symbol	Value	Description
DV.FAK	20	Partial LUB for ENCODE/DECODE and internal files
DV.FNB	4	File Name Block initialized
DV.DFD	10	Direct access unit
DV.FACC	40	File attributes byte of FDB (F.FACC) defined
DV.OPN	200	Unit open
DV.FMP	2000	Formatted unit
DV.UFP	4000	Unformatted unit
DV.ASGN	10000	File name defined
DV.CLO	20000	Close in progress
DV.FRE	40000	Free format prohibited (short field termination)
DV.RW	100000	Input or output operation (0 = read, 1 = write)
DV.FIX	2	Fixed-length records
DV.VAR	400	Variable-length records
DV.SEG	1000	Segmented records

Status Bits for Word 2 (D.STA2)

Symbol	Value	Description
DV.AI4	2	Associated variable is INTEGER*4 data type
DV.CC	10	Explicit carriage control specified
DV.SPL	20	DISP = 'PRINT' specified
DV.DEL	40	DISP = 'DELETE' specified

Symbol	Value	Description
DV.SAV	40000	DISP = 'SAVE' specified
DV.RDO	400	READONLY specified
DV.UNK	1000	TYPE = 'UNKNOWN' specified
DV.OLD	2000	TYPE = 'OLD' specified
DV.NEW	4000	TYPE = 'NEW' specified
DV.SCR	10000	TYPE = 'SCRATCH' specified
DV.APD	20000	ACCESS = 'APPEND' specified
DV.RSZ	4	Explicit RECORDSIZE specified
DV.BN	100000	BLANK = 'NULL' specified

4.2.3 LUB Definitions For RMS–11 Support

Each open logical unit using RMS–11 has a LUB, which comes from the storage pool. There is a 1-word pointer, or 0, to the LUB, which is allocated in the $$DEVT PSECT. A LUB contains the following information:

- An RMS–11 Record Access Block (RAB)
- FORTRAN control information
- Storage for the file specification string for use during error reporting.
- Allocation of the LUB occurs at the first reference to the logical unit. Deallocation occurs at the close of the unit.

RMS–11 also requires these additional control blocks: the File Access Block (FAB), Extended Attributes Block (XAB), and the Name Block. The OTS allocates these as required.

Offsets of the form D.xxxx describe the FORTRAN header portion of the LUB as follows:

D.STAT — status word 1 (bits defined below)
D.STA2 — status word 2 (bits defined below)
D.LUN — logical unit number
D.NAMC — length of name string
D.IFI — RMS internal file identifier value
D.PFAB — pointer to FAB or 0
D.RSIZ — maximum record length
D.RCNM — direct access record limit (word 1)
D.RCN2 — direct access record limit (word 2)

D.RCCT — record count for BACKSPACE (word 1)
D.RCC2 — record count for BACKSPACE (word 2)
D.AVAD — address of associated variable or 0
D.STS — RMS status code
D.STV — RMS secondary status code
D.RNUM — current direct access record number (2 words)
D.SPAR — spare word (reserved)
D.RAB — start of RMS RAB
D.NAM — start of file name string save area

Several of these words have different uses depending on the kind of I/O operation.

The FORTRAN header portion of the LUB contains two status words. The bits in these status words have symbolic names of the form DV.xxxx. All unused bit positions are reserved. These bits are defined as follows:

Status Bits for Word 1 (D.STAT)

Symbol	Value	Description
DV.SEQ	1	Sequential access
DV.DIR	2	Direct access
DV.KEY	4	Keyed access
DV.FIX	10	Fixed-Length records
DV.FAK	20	Partial LUB for ENCODE/DECODE and internal files
DV.FACC	40	File access set by CALL FDBSET
DV.VAR	100	Variable-Length records
DV.OPN	200	Unit open
DV.FMP	2000	Formatted unit
DV.UFP	4000	Unformatted unit
DV.SEG	10000	Segmented records
DV.CLO	20000	Close in progress
DV.FRE	40000	Free format prohibited (short field termination)
DV.RW	100000	Input or output operation (0=read, 1=write)

Status Bits for Word 2 (D.STA2)

Symbol	Value	Description
DV.SEQ	1	Sequential organization
DV.REL	2	Relative organization
DV.IDX	4	Indexed organization
DV.CC	10	Explicit carriage control specified
DV.SPL	20	DISP = 'PRINT' specified
DV.DEL	40	DISP = 'DELETE' specified
DV.AI4	100	Associated variable is INTEGER*4 data type
DV.RDO	400	READONLY specified
DV.UNK	1000	TYPE = 'UNKNOWN' specified
DV.OLD	2000	TYPE = 'OLD' specified
DV.NEW	4000	TYPE = 'NEW' specified
DV.SCR	10000	TYPE = 'SCRATCH' specified
DV.APD	20000	ACCESS = 'APPEND' specified
DV.SAV	40000	DISP = 'SAVE' specified
DV.RSZ	200	Explicit RECORDSIZE specified
DV.BN	100000	BLANK = 'NULL' specified

You need an RMS–11 FAB for file open and close operations. FORTRAN allocates the FAB and additional control information. The LUB offset D.PFAB points to this FAB.

The FORTRAN header portion contains eight words of control information, including a 10-byte default file name string 'FOR0nn.DAT', where nn is the logical unit number.

The information in the FORTRAN header portion is as follows:

F.STAT— FAB status byte
F.KYCT— number of keys in the OPEN statement KEY parameter
F.PXAB— pointer to key definition XAB control block
F.SPAR— spare (word reserved for future use by DIGITAL)
F.DNAM— start of default file name string
F.FAB — start of RMS FAB

FORTRAN uses XABs for key definitions when opening an indexed file. The KEY parameter of the OPEN statement specifies the number of key definition XABs to allocate. FORTRAN allocates a single large block of memory for all key definition XABs, and FAB block offset F.PXAB points to this block.

Each XAB in the block of key definition XABs contains two words appended to the RMS XAB. These words contain FORTRAN information used to check the key definitions of an existing file. The definition of these words is as follows:

X.XAB—start of RMS XAB
X.POS—start position of key specification
X.SIZ—size of key specification
X.DTP—data type of key specification

Chapter 5

I- and D-Space Tasks

I- and D-space ("I" represents "instruction" and "D" represents "data") is an advanced programming technique that allows you to effectively double your virtual task space from 32K words to 64K words. More specifically, you can have up to 32K words of instructions and 32K words of data associated with a task. Without I- and D-space, only 32K total words are available for your tasks.

5.1 Support for I- and D-Space Tasks

Before using the I- and D-space technique, you must establish that I- and D-space support is available for your task by making sure that:

- Your processor and operating system support I- and D-space tasks. (See Section 5.1.1.)
- The D-space parameter within the configuration file equals 1. (See Section 5.1.2.)
- You have included the /ID switch in the task-build command line for your programs. (See Section 5.1.3.)
- If your task uses a FORTRAN-77 OTS resident library, that this library's relationships to supervisor-mode libraries, resident commons, and virtual arrays respect restrictions.

5.1.1 Processor and Operating System Support

I- and D-space is supported by the following processors and operating systems:

- Processors:

PDP-11/44	PDP-11/70
PDP-11/45	PDP-11/73
PDP-11/50	
PDP-11/53	PDP-11/83
PDP-11/55	PDP-11/84
	J-11

- Operating systems (version noted or higher):

RSX-11M/M-PLUS V2.0	RSTS/E V9.0
Micro/RSX V3.0	Micro/RSTS V2.0

5.1.2 D-Space Parameter Requirement for Support

See Chapter 1 of the *PDP-11 FORTRAN-77 User's Guide* for more information about the D-space parameter and I- and D-space support.

5.1.3 /ID Switch Requirement for Support

All I- and D-space tasks require an /ID switch in the Task Builder command line. For example:

```
TKB MYPROG/ID=MYPROG,LB:[1,1]F77FCS/LB
```

(On RSTS/E systems, replace LB:[1,1] with LB:.)

Instead of explicitly supplying the command line to the Task Builder, you can use an indirect command file. In this case, edit your command file to include the /ID switch on the command line.

Chapter 6

An Overview of FORTRAN Input/Output

There are three kinds of OTS input/output (I/O) support modules:

- Those that are independent of a file system
- Those that use the FCS-11 file system
- Those that use the RMS–11 file system

This section describes some of the independent I/O modules (see Chapter 9 for the format-processing routines) and provides an overview of the I/O subsystem. Chapter 7 describes the FCS-11-specific modules and Chapter 8 describes the RMS–11-specific modules.

FORTRAN I/O processing consists of three layers or levels:

- Compiled-code interface (see Section 6.1)
- Data formatting (see Section 6.2)
- Record processing (see Section 6.3)

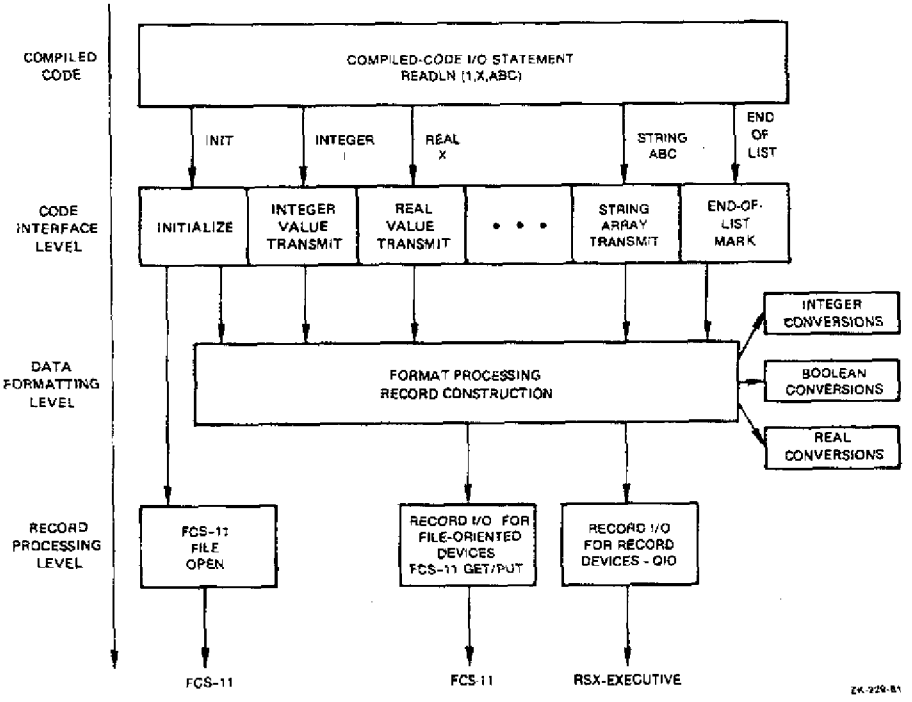
The compiled-code interface level consists of the routines called directly by the compiled code. The routines listed in Table 6–1 take the compiled-code arguments, transform them into OTS standard form, and pass them to the data-formatting level.

The data-formatting level accepts the standard I/O arguments and produces I/O records as specified by the data elements and format control. The records are then passed to or received from the record-processing level.

The record-processing level interfaces with the file management systems to read and write logical records. It is the only level dependent on a particular file system.

Figure 6–1 illustrates the I/O subsystem.

Figure 6–1: The I/O Subsystem



6.1 Compiled-Code Interface

The compiled-code interface is the external interface for the OTS I/O subsystem.

I/O statements produce three types of subroutine calls in the compiled code:

- Initialization calls—set up the I/O system for the specific I/O requested, open the specified logical unit if necessary, and declare the I/O system to be active
- Element transmission calls (if any)—generate calls to the OTS for entities in the I/O list
- Termination calls—complete the I/O operation and declare the I/O system inactive

For example, the FORTRAN statements

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

are compiled into the following code:

```
MOV   #2,-(SP)       ;Unit number
JSR   PC,ISU$        ;Initialize READ
MOV   #I,-(SP)       ;Address of I
JSR   PC,IOAI$       ;Transmit integer
MOV   #A$ADB,-(SP)   ;Address of array descriptor for A
JSR   PC,IOAA$       ;Transmit array A
MOV   #B,-(SP)       ;Address of B
JSR   PC,IOAR$       ;Transmit real
JSR   PC,$EOLST      ;End-of-list
```

6.1.1 Initialization Processing

There is a separate initialization-processing routine for each compiled FORTRAN I/O statement. These routines take the I/O statement-specific arguments, construct a mask word describing the arguments, and pass them to the I/O statement initialization module $INITIO.

6.1.1.1 The Routines

Table 6–1 lists the entry point names for the initialization-processing routines. Each routine has two entry points:

- xxxx$—for I/O statements that do not use END= or ERR=
- xxxxE$—for I/O statements that do use END= or ERR=

Table 6–1: I/O Initialization Entries

Entry Name	Arguments	Function
ISF$ ISFE$	u,f u,f,e	Sequential formatted input
ISU$ ISUE$	u u,e	Sequential unformattted input
IRF$ IRFE$	u,r,f u,r,f,e	Direct formatted input
IRU$ IRUE$	u,r u,r,e	Direct unformatted input
IKF$[1] IKFE$[1]	u,f,k,kr,km u,f,k,kr,km,e	Character keyed formatted input
IKU$[1] IKUE$[1]	u,k,kr,km u,k,kr,km,e	Character keyed unformatted input
ILF$[1] ILFE$[1]	u,f,l,kr,km u,f,l,kr,km,e	Integer keyed formatted input
ILU$[1] ILUE$[1]	u,l,kr,km u,l,kr,km,e	Integer keyed unformatted input
OSF$ OSFE$	u,f u,f,e	Sequential formatted output
OSU$ OSUE$	u u,e	Sequential unformatted output
ORF$ ORFE$	u,r,f u,r,f,e	Direct formatted output
ORU$ ORUE$	u,r u,r,e	Direct unformatted output
RSF$[1] RSFE$[1]	u,f u,f,e	Formatted rewrite

[1] These entries are supported only by RMS–11 versions.

Table 6–1 (Cont.): I/O Initialization Entries

Entry Name	Arguments	Function
RSU$[1]	u	Unformatted rewrite
RSUE$[1]	u,e	
ENF$	c,f,a	ENCODE
ENFE$	c,f,a,e	
DEF$	c,f,a	DECODE
DEFE$	c,f,a,e	
ISL$	u	List-directed input
ISLE$	u,e	
OSL$	u	List-directed output
OSLE$	u,e	
DLS$[1]	u	Sequential DELETE
DLSE$[1]	u,s	
DLR$[1]	u,r	Direct DELETE
DLRE$[1]	u,r,e	
FDR$	u,r	Direct FIND
FDRE$	u,r,e	
ENDF$	u	ENDFILE
ENDFE$	u,s	
REWI$	u	REWIND
REWIE$	u,s	
UNLK$[1]	u	UNLOCK
UNLKE$	u,s	
DEFF$	u,mr,rl,v,vf	DEFINEFILE
IIF$	d,f	Internal file read
IIFE$	d,f,e	
IIFA$	adb,f	
IIFAE$	adb,f,e	
OIF$	d,f	Internal file write
OIFE$	d,f,e	
OIFA$	adb,f	
OIFAE$	adb,f,e	

[1]These entries are supported only by RMS–11 versions.

Arguments:

u Logical unit number—INTEGER*2 value.

r Direct access record number—INTEGER*4 value.

f Format specifier—address of compiled format text.

k Character key specifier—address of key descriptor, which has the following form:

```
length of string

address of string   <---------address of descriptor
```

Note that the address points to the second word of the descriptor.

adb Address of the array descriptor block.

kr Key-of-reference number—INTEGER*2 value; if no KEYID is specified, a value of -1 is supplied.

e END=/ERR= specifier—address of END= label, followed by address of ERR label. If one of the labels is missing, a 0 address is supplied for that label.

l Integer key specifier—INTEGER*4 value.

km km Key match criterion—INTEGER*2 value encoded as follows:

```
0 -- equal match
1 -- greater than or equal match
2 -- greater than match
```

a ENCODE/DECODE buffer—address of buffer.

c ENCODE/DECODE buffer—INTEGER*2 value.

d Address of the character descriptor. The first word of the descriptor contains the length of the string; the second word contains the address of the string.

s ERR= statement label address.

mr Maximum direct access record number – INTEGER*4 value.

rl Record length in 16-bit words—INTEGER*2 value.

v Address of associated variable.

vf Associated variable data type flag—INTEGER*2 value encoded as follows:

```
 0 = INTEGER*2 data type
-1 = INTEGER*4 data type
```

NOTE

If a run-time format is specified, the run-time format compiler FMTCV$ overwrites the source address of the run-time format array with the address of the compiled format string.

6.1.1.2 $INITIO

The $INITIO routine performs specific functions based on the arguments passed by the initialization-processing routines described in Section 6.1.1.1. In addition, $INITIO paves the way for the remaining processing levels by storing the appropriate data-formatting routine address in the impure area offset W.EXJ, and the appropriate record-processing routine address in the impure area offset RECIO.

The routines that pass arguments to $INITIO use a bit-encoded mask to indicate what operations need to be performed. When $INITIO is called, R0 points to the stack arguments and R1 contains the bit-encoded mask.

Tables 6–2 and 6–3 describe the symbols and argument masks used by the routines. Table 6–4 describes the operations $INITIO performs based on the bit settings.

Table 6–2: I/O Initialization Symbols

Symbol	Value	Description
FL.ERR	100000	END= # and ERR= present
FL.INB	40000	Internal files passed by ADB
FL.IND	20000	Internal files passed by descriptor
FL.ENC	11000	ENCODE statement
FL.FMT	4200	Format present
FL.REC	2400	Direct access record number present
FL.FMP	200	Formatted operation permitted
FL.WRT	140	WRITE operation (with implied OPEN)
FL.RD	40	Read operation (with implied OPEN)
FL.EDA	10000	ENCODE/DECODE buffer address
FL.FMA	4000	Format address
FL.RNM	2000	Record number
FL.EDL	1000	ENCODE/DECODE buffer length
FL.DIR	400	Direct access
FL.OUT	100	Output operation

Table 6–2 (Cont.): I/O Initialization Symbols

Symbol	Value	Description
FL.OPN	40	OPEN required
FL.IGN	20	Ignore format and record type checks
FL.KEY	10	Keyed access
FL.REW	4	REWRITE
FL.DEL	2	DELETE
FL.KIN	1	Integer key value

Table 6–3: I/O Initialization Argument Masks

Mask	Meaning
ISF$	Sequential formatted input: FL.FMT+FL.RD
OSF$	Sequential formatted output: FL.FMT+FL.WRT
ISU$	Sequential unformatted input: FL.RD
OSU$	Sequential unformatted output: FL.WRT
ISL$	Sequential list-directed input: FL.FMP+FL.RD
OSL$	Sequential list-directed output: FL.FMP+FL.WRT
RSF$	Sequential formatted rewrite: FL.FMT+FL.WRT+FL.REW
RSU$	Sequential unformatted rewrite: FL.WRT+FL.REW
IRF$	Direct formatted input: FL.FMT+FL.REC+FL.RD
ORF$	Direct formatted output: FL.FMT+FL.REC+FL.WRT
IRU$	Direct unformatted input: FL.REC+FL.RD
ORU$	Direct unformatted output: FL.REC+FL.WRT
IKF$	Character keyed formatted input: FL.KEY+FL.FMT+FL.RD
IKU$	Character keyed unformatted input: FL.KEY+FL.RD
ILF$	Integer keyed formatted input: FL.KEY+FL.FMT+FL.RD+FL.KIN
ILU$	Integer keyed unformatted input: FL.KEY+FL.RD+FL.KIN
ENF$	ENCODE statement: FL.FMT+FL.ENC
DEF$	DECODE statement: FL.FMT+FL.ENC
ENDF$	ENDFILE statement: FL.WRT+FL.IGN

Table 6–3 (Cont.): I/O Initialization Argument Masks

Mask	Meaning
DLS$	Sequential delete: FL.WRT+FL.IGN+FL.DEL
DLR$	Direct delete: FL.WRT+FL.REC+FL.IGN+FL.DEL
FDR$	FIND statement: FL.RD+FL.REC+FL.IGN
ILF$	Internal file read: FL.IND+FL.FMT
IIFA$	Internal file read with address of ADB passed as the Internal logical unit number: FL.INB+FL.FMT
OIF$	Internal file write: FL.IND+FL.FMT
OIFA$	Internal file write with address of ADB passed as the Internal logical unit specifier: FL.INB+FL.FMT

NOTE

If the corresponding END=/ERR= entry point is called (for instance, ISFE$ rather than ISF$), the argument mask includes FL.ERR.

Table 6–4: I/O Initialization Routine Functions

Function	Description
FL.DEL	If the file organization is sequential, issue OTS error 55, DELETE statement error.

FL.DIR	Compare the access mode of the I/O statement with the FL.KEY access mode of the logical unit; issue OTS error 31 if the access mode does not match. Issue OTS error 26 if direct or keyed access is required but has not been specified. Valid combinations are:
	`Direct access I/O -- direct access unit`
	`Sequential access I/O -- sequential or keyed access unit`
	`Keyed access I/O -- keyed access unit`

FL.EDA	Save the ENCODE/DECODE buffer address in the impure area offsets, LNBUF (start address) and BLBUF (current address).
FL.EDL	Add the ENCODE/DECODE buffer length to the start address to determine the end address of the buffer. Save this value in impure area offset EOLBUF.

Table 6–4 (Cont.): I/O Initialization Routine Functions

Function	Description
FL.ERR	Save the END= address in impure area offset ENDEX, and the ERR= address in impure area offset ERREX.
FL.FMA	Save the format address in impure area offset FMTAD.
FL.FMP	Compare formatting type specified with format type of the logical unit. Mixed formatted and unformatted operations are not permitted. Issue OTS error 31 if the format types do not match.
FL.IGN	Ignore the format checks for ENDFILE, FIND, and DELETE since both formatted and unformatted are permitted. Ignore the record type check since record type depends on format.
FL.INB	Save the format address in impure area offset FMTAD. Save the internal logical unit address in the impure area offsets LNBUF (start address) and BLBUF (current address). Add the bytes per element from offset A.BPE in the array descriptor block to offset LNBUF to determine the end address of the internal logical unit. Save this value in impure area offset EOLBUF. Divide the total size of the array in bytes (offset A.SIZB in the ADB) by the bytes per element (offset A.BPE) to determine the number of records and store this value in the impure area offset DENCWD.
FL.IND	Save the format address in impure area offset FMTAD. Save the internal logical unit address in the impure area offsets LNBUF (start address) and BLBUF (current address). Add the length of the internal logical unit specifier to offset LNBUF to determine the end address of the internal logical unit. Save this value in impure area offset EOLBUF.
FL.KEY	Save the key value descriptor address in impure area offset W.KDSC, the key-of-reference number in impure area offset W.KREF, and the key match value in impure area offset W.KMAT.
FL.KIN	Save the integer key value in impure area offset W.KNUM and W.KNUM+2. Set W.FDTP to integer.
FL.OPN	If the logical unit is not yet open, open it using the default open processor $OPEN.
FL.OUT	Set the logical unit status to input or output as appropriate. If output is specified and the logical unit is declared read-only, issue OTS error 47.
FL.REW	If the file organization is sequential or relative, issue OTS error 54 (REWRITE statement error).
FL.RNM	Save the direct access record number in impure area offsets W.RECL and W.RECH as an INTEGER*4 value.

6.1.2 List Element Transmission

The compiled code makes one data transmission call to the OTS for each data item in the I/O list. Data transmission entry points have the form:

```
IOat$
```

where:

a Designates whether the argument is an address or a value; can be A, for address, or V, for value.

t 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
```

The following additional entry points are used only for arguments that are addresses:

IOAH$ – Transmits a Hollerith constant (output only). The argument is the address of the first byte of the constant as an ASCIZ string.

IOAA$ – Transmits an entire array by name. The argument is the address of the array descriptor block. For formatted I/O, each array element is passed individually to the data-formatting level. For unformatted I/O, the entire array is passed as a single large data item.

IOAVA$ — Transmits an entire virtual array by name. The argument is the address of the array descriptor block.

One entry is used for an argument that is two words (length, address descriptor):

IOACH$ – Transmits a character string. The argument is the length of the character string and the address of the first byte of the ASCIZ string.

The routines at each of these entry points set up impure area offsets and then invoke the data-formatting level of processing at impure area offset W.EXJ. The impure area offsets set up are as follows:

ITEMSZ –	Size in bytes of the data item.
VARAD –	Address of the first byte of the data item, or 0 if at end of list.
W.VTYP –	Data type code of data item.
W.CPXF –	Complex data type flag. Complex data items are passed as a pair of real values. W.CPXF=0 indicates a noncomplex item; +1 indicates the real part of a complex item; -1 indicates the imaginary part of a complex item.

6.1.3 Termination Call

The routine at entry point EOLST$ is called to specify the end of the I/O list. No arguments are required.

6.2 Data-Formatting Level

The compiled-code interface level calls data-formatting routines to transmit data between records and I/O list items, including any common operations that are required.

For formatted I/O, there are three routines:

$FIO—format processor
$LSTI—list-directed input processor
$LSTO—list-directed output processor

These routines are called with no register arguments; on return all registers are undefined.

Conversion is not needed for unformatted I/O so the appropriate initialization modules maintain the transfer code as routines.

The data-formatting routines accept data item descriptions from the impure area offsets ITEMSZ, VARAD, W.VTYP, and W.CPXF. On input, the routines read the next field of the record and transfer data to the item. On output, the data item value is transferred to the record. The following impure area offsets describe the record being processed:

LNBUF —start-of-buffer address
BLBUF —address of next data byte
EOLBUF—end-of-buffer address

When a new record must be read, or an output record is full, the record-processing routine specified by impure area offset RECIO is called to process the record. On input, the old record is discarded, a new record is read, and the impure area record description is updated. On output, the record is written and a new buffer area is set up.

6.3 Record-Processing Level

The record-processing routines are called to transfer records to and from the file system. The record-processing routines are:

$GETS —sequential input
$PUTS —sequential output
$GETR —direct input
$PUTR —direct output
$GETK —keyed input
$UPDATE—rewrite

6.4 PRINT, TYPE, and ACCEPT Statements and Logical Unit 0

The PRINT, TYPE, and ACCEPT statements compile into equivalent READ and WRITE statements using default unit numbers. Default unit numbers are small negative integers, which $FCHNL maps through a table in impure storage to actual unit numbers. This table has global names for each statement to allow modification of the mapping. The global names are:

$PRINT for PRINT
$TYPE for TYPE
$ACCPT for ACCEPT
$READ for READ

The unit number value is at impure area offset W.LNMP. The mapped values are 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

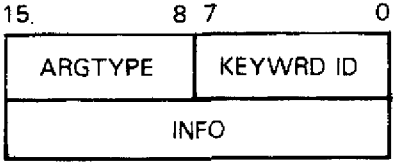
If you specify logical unit 0, you must use the GBLPAT option of the Task Builder to associate a valid logical unit number (1-99) with global $LUN0. $FCHNL (see Section 6.6.1) uses the value of $LUN0 to change FORTRAN logical unit 0 to a valid system logical unit number. If you attempt to use logical unit 0 without moving a valid number to $LUN0, an error occurs.

6.5 OPEN and CLOSE Statements

The OPEN and CLOSE source statements allow your programs to control the attributes and characteristics of files. The compiled code for these statements uses the standard R5 calling sequence with a special argument list encoding, as follows:

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

There is one argument for each keyword in the FORTRAN source statement. Each argument consists of a 2-word block, formatted as follows:



ZK-7630-HC

KEYWRD ID

The low-order byte of the first word contains the keyword identification number associated with the keyword name in the source statement (see Table 6–5).

ARGTYPE

The high-order byte of the first word contains the argument type. It is used in conjunction with the INFO word to identify the keyword's value.

INFO

The second word is called the information word; its use depends on the ARGTYPE value.

The possible ARGTYPE values are 1 through 7. The meanings of each ARGTYPE are as follows:

ARGTYPE Value	Meaning
1	The keyword's value is an INTEGER*2 constant expression. The INFO word contains the value.
2	The keyword's value is an INTEGER*2 variable. The INFO word contains the address of the variable.
3	The keyword's value is an INTEGER*4 variable. The INFO word contains the address of the variable.
4	The keyword's value is an alphanumeric literal that can be decoded by the compiler. The INFO word contains the keyword's value encoded as a small integer.
5	The keyword's value is a variable, array, array element, or character constant terminated by an ASCII null character (zero-byte). The INFO word contains the address of the start of the string.
6	The keyword's value is the address of an external procedure. The INFO word contains the address.
7	The keyword's value is the address of a 2-word descriptor. The first word of the descriptor contains the length of the string; the second word contains the address of the string. The INFO word contains the address of the first word of the descriptor.

A statement's keywords can be in any order, but there cannot be any duplicates. Table 6–5 lists the keyword names, their associated identification numbers, and the ARGTYPES permissible with each keyword. The table also lists the literal values and associated literal encoding possible for keywords whose ARGTYPES are 4.

Table 6–5: Summary of Argument Blocks by Keyword

Keyword Name	Keyword Number	Allowed Argtypes[2]	Literal Values	Literal Encoding
ACCESS	4	4	DIRECT SEQUENTIAL APPEND KEYED	1 2 3 4
ASSOCIATE VARIABLE	17	2,3		
BLANK	25	4	NULL ZERO	1 2
BLOCKSIZE	18	1,2,3		
BUFFERCOUNT	9	1,2,3		
CARRIAGECONTROL	7	4	FORTRAN LIST NONE	1 2 3
CHARKEY[1]	23	1,2,3		
DISPOSE	2	4	SAVE DELETE PRINT	1 2 3
ERR	3		–	Label address
EXTENDSIZE	11	1,2,3		
FILE or NAME	14	5,7		
FORM	5	4	FORMATTED UNFORMATTED	1 2
INITIALSIZE	10	1,2,3		
INTKEY[1]	24	1,2,3		
KEY[1]	22	1,2,3		
MAXREC	16	1,2,3		
NOSPANBLOCKS	12	–		

[1]KEY occurs one time and gives the number of key specifications. KEY is followed by n pairs of CHARKEY or INTKEY keywords giving the start and end positions of each key specification. The pseudo-keywords CHARKEY and INTKEY denote the data type of the key specification.

[2]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 6–5 (Cont.): Summary of Argument Blocks by Keyword

Keyword Name	Keyword Number	Allowed Argtypes[2]	Literal Values	Literal Encoding
ORGANIZATION	19	4	SEQUENTIAL RELATIVE INDEXED	1 2 3
READONLY	8	–		
RECORDSIZE or RECL	6	1,2,3		
RECORDTYPE	20	4	FIXED VARIABLE SEGMENTED	1 2 3
SHARED	13	–		
STATUS or TYPE	15	4	OLD NEW SCRATCH UNKNOWN	1 2 3 4
UNIT	1	1,2,3		

[2]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.

As an example, consider the following FORTRAN source statement:

```
OPEN (UNIT=I, ERR=99, NAME='A.DAT')
```

When it is compiled, the code (in part) looks like the following:

```
                MOV   ARGLST,R5  ;Address of arg list
                JSR   PC,OPEN$   ;Open the file
                  .
                  .
                  .
ARGLST:    .WORD   6             ;3 args
           .BYTE   1,2           ;UNIT, ARGTYPE=2
           .WORD   I             ;Address of I
           .BYTE   3,2           ;ERR, 2 bytes of stack temp
           .WORD   99            ;Address of label
           .BYTE   14,5          ;NAME, ARGTYPE=5
           .WORD   STRING        ;Address of string
                  .
                  .
                  .
STRING:    .BYTE   101,56,104,101,124,0     ;'A.DAT'
```

6.6 Other Internal Support Routines

The following sections describe several other internal support routines used by the OTS.

6.6.1 $FCHNL, $GETFILE, and $IOEXIT

The $FCHNL, $GETFILE, and $IOEXIT routines serve as the common entrance and exit to the I/O system.

$FCHNL locates the LUB for a given logical unit number and issues an error for invalid units. It is called with the logical unit number in R2 and returns the address of the associated LUB in R0. The PSW C-bit is used as an error flag on return. The flag is set if there is an error, and is cleared if there is no error. On return, registers R1 and R2 are undefined, R3 contains the impure area pointer, and R4 and R5 are preserved.

$GETFILE executes a $FCHNL, sets the FILPTR impure area offset, and checks the status of the unit. It is called the same way as $FCHNL. It does not return the C-bit error flag; however, its register returns are identical to those for $FCHNL.

$IOEXIT restores the user-level status and register state and executes the ERR= transfer. It is called with the ERR= transfer address in R4 and the work area pointer in R3.

6.6.2 Default File Open Processing—$OPEN

A default open is the implicit opening of a logical unit due to executing an I/O statement on a closed logical unit. If a READ or FIND statement is executed, the default open is equivalent to the following OPEN statement (unless a DEFINEFILE has been executed):

```
OPEN (UNIT=unit, TYPE='OLD', ORGANIZATION= 'SEQUENTIAL', BLANK='ZERO',
     FORM= "form of the I/O statement", ACCESS='SEQUENTIAL')
```

If a WRITE statement is executed, the default open is equivalent to the following OPEN statement (unless a DEFINEFILE has been executed):

```
OPEN (UNIT=unit, TYPE='NEW', ORGANIZATION= 'SEQUENTIAL', BLANK='ZERO',
     FORM= "form of the I/O statement", ACCESS= 'SEQUENTIAL')
```

All other OPEN statement parameters assume their default values as described in Chapters 7 and 8, respectively.

The default file open processor is called with R0 pointing to the LUB and R3 pointing to the impure area. On return, all registers are preserved.

6.6.3 Default File Close Processing—$CLOSE

The file close processor is invoked when any one of the following occurs:

- A CLOSE statement is executed.
- A CALL CLOSE subroutine is executed.
- A program terminates.
- A file open fails.

The $CLOSE routine implements the DISPOSE= parameter set by the OPEN or CLOSE statement, and invokes the appropriate routine to close, delete, or print the file.

This routine is called with the logical unit number in R2. On return, R0, R1, R2, and R4 are undefined; R3 points to the impure area; R5 is preserved; and the processor C-bit is set to indicate whether an error occurred during the close operation.

6.6.4 Direct Access Record Number Checking—$CKRCN

$CKRCN compares the current record number with 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 (low order) and D.RCN2 (high order) in the LUB. If the record number is valid, it is returned in R1 (high order) and R2 (low order). This routine is called with the LUB address in R0, and the impure area pointer in R3. Registers R4 and R5 are preserved.

6.6.5 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 LUB.

6.6.6 Keyed I/O Specifier Checking—$CKKEY

$CKKEY verifies the key specification in a keyed I/O statement and sets the proper control information in the LUB. This routine is called with the LUB pointer in R0 and the impure area pointer in R3. Registers R1 and R2 are destroyed; all other registers are preserved.

6.6.7 Register Save and Restore—$SAVPx

The $SAVPx routine provides the register save/restore and argument processing support for implementing the OTS PC call convention (see Section 2.2.2), which pushes all arguments on the stack, calls the OTS routine by 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, the register save/restore routine restores the registers, deletes the stack arguments, and returns to the original caller. Nine entry points are provided: $SAVP0-$SAVP8 for routines with zero to eight argument words on the stack. For routines with more than eight arguments or with a variable number of arguments, $SAVP0 is called to save the registers; upon return to the OTS module, R0 contains the number of arguments and a jump to $SAVPC is executed at the completion of the OTS module, rather than a return to the register restore portion of the $SAVPx routine. For ERR= transfers, $SAVPx is jumped to with R0 containing the transfer address.

6.6.8 Register Save and Restore—.SAVR1

Several OTS routines call the FCS-11 register save co-routine .SAVR1 to save and restore registers R1 through R5 in co-routine fashion.

6.7 FORTRAN File and Record Formats

This section describes the file and record formats that are processed by the FORTRAN-77 I/O system.

6.7.1 Sequential Organization Files

You can process sequential files on all devices. Records may be fixed length, variable length, or FORTRAN segmented. Fixed-length records have no control information and are packed densely into blocks by the file system. Variable-length records have a count field in front of each record. For ANSI magnetic tape, the count field is a 4-byte decimal ASCII number. For all other devices, the count field is a 2-byte binary value.

A segmented record is a single logical record consisting of one or more variable-length records. Each variable-length record constitutes a segment. Segmented records are useful when you want to write exceptionally long records, and are used for unformatted sequential files.

Because the size of a segmented record is unlimited, each variable-length record in the segmented record contains control information to indicate that it is one of the following:

- The first segment in the segmented record
- The last segment in the segmented record
- The only segment in the segmented record
- None of the above

This control information is in the first two bytes of the record after the count field. Only two bits of the first byte are used; all other bits must be zero.

If both bits are set, the segment is the only segment in the record. If only bit 0 is set, the segment is the first segment. If only bit 1 is set, the segment is the last segment. If neither bit is set, the segment is not the first, last, or only segment in the record.

This control information is transparent to the user program; it is interpreted only by the FORTRAN–77 I/O system. If unformatted sequential files are to be processed in any other way, the files must be created with either fixed- or variable-length records.

The FORTRAN–77 I/O system does not support system files or files with variable fixed control (VFC) format records.

6.7.2 Relative Organization Files

You can process relative files only on disk devices. Records may be fixed length or variable length. Each record has a control byte used for deletion control. Variable-length records are stored in fixed-length cells that are the size of the largest record, as specified by the RECORDSIZE parameter. Variable-length records also contain a 2-byte binary count field that specifies the current length.

6.7.3 Indexed Organization Files

You can process indexed files only on disk devices. Records may be fixed length or variable length. Each record has a control byte used for deletion control. Variable-length records also have a 2-byte binary count field. Each record contains additional bytes of RMS–11 control information. Additional RMS–11 control information is stored for each bucket. Additional buckets are required for index areas, alternate key areas, and RMS–11 control information.

Chapter 7

FCS-11 Input/Output Support

This chapter discusses the FCS-11-specific portions of the OTS. In particular, it describes the explicit FCS operations used to implement FORTRAN–77 I/O operations.

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

- R0—address of the FCS File Descriptor Block (FDB)
- R1—address of the Logical Unit Block (LUB)
- R3—address of the work area
- R2 and R4—scratch registers

PC,xxx instruction calls all routines except the co-routine calls. R5 is generally preserved.

7.1 FCS-11 I/O CONTROL BLOCK

The FCS-11 I/O system associates a single control block, called the File Descriptor Block (FDB), with each open unit. The FDB is incorporated within the FORTRAN–77 Logical Unit Block (LUB). See Section 4.2 and Appendix B for more information about the LUB. See the *IAS/RSX–11 I/O Operations Reference Manual* for more information on FCS data structures.

7.2 Open Processing

Default file open processing and OPEN statement processing merge into a single common routine, $OPEN$ (see Section 7.2.3), for a file open. $OPEN$ invokes the macro OFNB$, which performs all file open operations. If file name parsing logic is not required—that is, if FORTRAN-77 default file names are used—the routines for file name parsing are not included in the task. Table 7-1 shows the OPEN statement keywords, the possible values of the keywords, and the FDB settings associated with the keyword values.

Table 7-1: Summary of OPEN Statement Keywords and FDB Settings

Keyword Name	Value	FDB Setting
ACCESS	'DIRECT' 'SEQUENTIAL' 'APPEND'	Set FD.RAN in F.RACC - Set FO.APD in F.FACC
ASSOCIATEVARIABLE	v	-
BLANK	'NULL' 'ZERO'	- -
BLOCKSIZE	n	Set n in F.OVBS
BUFFERCOUNT	n	Set n in F.MBCT
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Set FD.FTN in F.RATT Set FD.CR in F.RATT
DISPOSE	'SAVE' 'DELETE' 'PRINT'	Use CLOSE$ at file close Call .DLFNB at file close Call .PRINT at file close
ERR	s	-
EXTENDSIZE	n	Set n in F.ALOC
FILE or NAME	f	Call $FNBST to set File Name Block of FDB
FORM	'FORMATTED' 'UNFORMATTED'	- -
INITIALSIZE	n	Set n in F.CNTG
MAXREC	n	-

Table 7–1 (Cont.): Summary of OPEN Statement Keywords and FDB Settings

Keyword Name	Value	FDB Setting
NOSPANBLOCKS	-	Set FD.BLK in F.RATT
READONLY	-	Set FO.RD in F.FACC
RECORDSIZE or RECL	n	Set n in F.RSIZ
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'	Set R.FIX in F.RTYP Set R.VAR in F.RTYP Set R.VAR in F.RTYP
SHARED	-	Set FA.SHR in F.FACC
STATUS or TYPE	'OLD' 'NEW' 'SCRATCH' 'UNKNOWN'	Use OPEN$U Use OPEN$W Use OPNT$D Try OPEN$U, if no such file, then OPEN$W
UNIT	n	Set n in F.LUN
USEROPEN	p	-

f
Is an array, array element, variable, or character constant.

n
Is an integer expression.

s
Is an executable statement label.

v
Is an integer variable name.

p
Is an external procedure name.

7.2.1 OPEN Statement Processing

In OPEN statement processing, an argument list is searched and each keyword is located in a prescribed order. All information required for each keyword is available when that keyword is processed. An appropriate default is used for keywords not in the list. If any errors occur during the search, the execution of the OPEN statement is not attempted, the ERR= transfer is executed, and the LUB is zeroed.

The processing for each keyword is as follows:

- ACCESS—'DIRECT' sets DV.DFD; 'APPEND' sets DV.APD. If DV.RDO is set and DV.APD is specified, an error occurs. If DV.APD is not specified, the default is 'SEQUENTIAL'.
- ASSOCIATEVARIABLE—The variable address is stored at D.AVAD. If the variable is type INTEGER*4, DV.AI4 is set.
- BLANK—'NULL' sets DV.BN. Note that if the /F77 switch is set and no BLANK= is specified, the compiler passes a BLANK='NULL' parameter.
- BLOCKSIZE—The value specified is stored at F.OVBS. An error occurs if the value is negative or greater than 32767.
- BUFFERCOUNT—The value specified is stored at F.MBCT. If the value is less than -1, or greater than 127, an error occurs. Note that the actual number of buffers used depends on which FCS version is used and on the number of buffers available when the file is opened. A buffer count of -1 means the unit is opened in block mode (READ$/WRITE$) rather than record mode (GET$/PUT$). In block mode, normal FORTRAN-77 I/O is not permitted but the user can perform asynchronous block mode I/O using the FORTRAN special subroutines provided by the operating system. Note that when you do block mode I/O, you should either specify the Task Builder option MAXBUF = recordsize or call the system subroutine ERRSET to avoid run-time error 37 (inconsistent record length).
- CARRIAGECONTROL—If 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.
- 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 and DV.SCR is set, 'DELETE' is the default; otherwise, 'SAVE' is the default.

- 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.
- EXTENDSIZE—The value specified is stored at F.ALOC. If it is positive, a contiguous extend is made; if it is negative, a noncontiguous extend is made. If the value is greater than 32767 or less than -32767, an error occurs.
- FILE or NAME—If a file is specified, $FNBST is called to initialize the file Name block and DV.ASGN is set. $FNBST returns an error if the string is incorrect.
- FORM—'FORMATTED' sets DV.FMP; 'UNFORMATTED' sets DV.UFP. If no value is specified and DV.DFD is set, DV.UFP is the default; otherwise, DV.FMP is the default.
- INITIALIZE—The value specified is stored at F.CNTG. If it is positive, a contiguous allocation is made; if it is negative, a noncontiguous allocation is made. If the value is greater than 32767 or less than -32767, an error occurs.
- MAXREC—The value specified is stored at D.RCNM and D.RCN2. If the value is negative, an error occurs.
- NOSPANBLOCKS—If this keyword is specified, FD.BLK is set in F.RATT.
- READONLY—If this keyword is present, DV.RDO is set.
- RECORDSIZE or RECL—The value is stored at F.RSIZ. If it is negative or is larger than the user record buffer size (MAXBUF value for TKB), 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.
- RECORDTYPE—'FIXED' sets DV.FIX; 'VARIABLE' sets DV.VAR; and 'SEGMENTED' sets DV.SEG.

  The defaults are 'FIXED' for direct access; 'VARIABLE' for formatted sequential access; and 'SEGMENTED' for unformatted sequential access. For direct access, 'VARIABLE' or 'SEGMENTED' is an error; for formatted, 'SEGMENTED' is an error.
- SHARED—If this keyword is specified, FA.SHR is set in F.FACC.

- STATUS or TYPE—If STATUS is not present, the default is 'NEW'. Note, however, that if the /F77 switch is set and no STATUS = parameter is specified in the source code, the compiler passes a STATUS = 'UNKNOWN' parameter. '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
```

- UNIT—The unit number is obtained and $FCHNL is called to obtain the LUB pointer. Processing is aborted immediately if there is no unit number, the unit number is invalid, or the unit is already open.
- USEROPEN—The external procedure name is saved at offset W.UOPN for use by $OPEN$.

7.2.2 Default OPEN Processing

If DV.FACC is not set, default OPEN processing performs the following operations:

- For input, it sets DV.OLD and FO.UPD.
- For output, it sets DV.NEW and FO.WRT.

Other fields and values may have been set by CALL ASSIGN, CALL FDBSET, or DEFINEFILE statements.

7.2.3 $OPEN$ Procedure

The $OPEN$ procedure opens the file and performs the checks and computations common to OPEN statement processing (Section 7.2.1) and default OPEN processing (Section 7.2.2).

Before the file is opened, $OPEN$ performs the following operations:

- If no user file specification is provided (DV.ASGN is not set), the default File Name Block of the FDB is set up. The routine $FLDEF is called to make the file name FOR0nn.DAT (nn is the logical unit number).
- If no directory is specified for the file, the FCS routine .GTDIR is called to set the default directory.
- If DV.CC is not specified and the file is formatted, FD.FTN is set in the F.RATT field of the FDB.
- The user record buffer description in the FDB, F.URBD, is initialized with the address specified by the impure area offset W.BFAD and the length specified by W.BLEN.
- FD.PLC is set in the F.RACC field of the FDB to specify locate mode I/O operations.
- A record format is set as follows: If DV.FIX is set, R.FIX is set in the F.RTYP field of the FDB; otherwise, R.VAR is set.
- If DV.DFD is set, FD.RAN is set in the F.RACC field of the FDB to specify direct access.
- A record length is computed. If a user-specified value is available, that value is used; otherwise, one of the following values is used:

  ```
  133 for formatted files

  128 for unformatted files of fixed-length records

  126 for other unformatted files
  ```

  If DV.FIX is set, the record length value is set in the F.RSIZ field of the FDB.

If impure area offset W.UOPN is nonzero, the user's routine is called to perform the FCS OPEN operation; otherwise, .OPFNB is called to open the file by file name block. If the open operation fails because the file cannot be found, and DV.UNK is set, the operation is retried with DV.NEW set, and FO.WRT set in the F.FACC field of the FDB.

After the file is open, the following operations are performed:

- DV.OPN is set to indicate that the file is open.
- The record format is checked for consistency; if the user-specified record type does not match the file's record format, an error occurs.
- The record length, D.RSIZ, is checked for consistency.

  - If the user-specified length does not match the file's record length for fixed-length records, an error occurs. If the error continuation bit specifies "RETURN", the user-specified length is used.

  - For variable-length records, the record length is set to the maximum of the user-specified length and the file's maximum size.
- The user record buffer description in the FDB, F.URBD, is initialized with the address specified by the impure area offset W.BFAD and the length specified by D.RSIZ.
- If D.RSIZ is larger than the user record buffer, as specified by impure area offset W.BLEN, a record size error occurs.

If any errors occur, either reported by the FCS or resulting from the consistency checks, the file is closed. If the file was just created, it is deleted as well.

7.2.4 USEROPEN Interface Specification

The USEROPEN parameter of the OPEN statement gives you a way to access special FCS processing options not explicitly available in the FORTRAN language. The value of the USEROPEN parameter is the name of a user-written MACRO-11 routine that the OTS calls to open a file. To use the special FCS processing options, you must do the following:

- Using the MACRO-11 language, write a routine that opens the file.
- In your FORTRAN program, add the statement:

  ```
  EXTERNAL filename
  ```

  where "filename" is the name of the MACRO-11 routine you wrote to open the file.
- In the OPEN statement in your FORTRAN program, add the keyword parameter USEROPEN=filename, where "filename" is the name of your MACRO-11 routine.

Although the MACRO-11 routine is called by the OTS (not your FORTRAN program), write it as if it is being called by a FORTRAN program. You must report the status of the open operation in R0. The OTS invokes the routine as a standard FORTRAN function of one argument using this standard FORTRAN calling convention:

```
ISTS = userprocedure (FDB)
```

FDB The address of the FCS FDB for the logical unit.

ISTS The INTEGER*2 error status to be returned. The value is expected to be the F.ERR FCS completion status and to follow the FCS conventions (positive numbers indicate success, negative numbers failure). The status is returned only to the OTS, not to the FORTRAN program.

The following limits and constraints are imposed on the user-written procedure:

- All FORTRAN processing is completed prior to the call.
- The FDB address specified is valid until the logical unit is closed. You do not have access to the FDB in the FORTRAN program. You can access the FDB in a MACRO-11 program; the FDB address is at 2(R5).

The following sample FORTRAN program and user-open procedure specify that an existing file of the same name should not be superseded by a create operation:

```
EXTERNAL NOSUP
OPEN (UNIT = 1, USEROPEN=NOSUP,TYPE='NEW')
    .
    .
    .
END
            .MCALL  OFNB$

NOSUP::  MOV   2(R5),R0          ; Get FDB addr
         BISB #FA.NSP,F.FACC(R0) ; Set no supersede
         OFNB$                   ; Open the file
         MOV  F.ERR(R0),R0       ; Return completion status
         RETURN
```

7.2.5 File Name Processing

Two routines—$FNBST and $FLDEF—are used to process file name strings and supply FORTRAN default file names.

The File Name Block Initialization module, $FNBST, sets up the File Name Block (FNB) of the LUB.

If there is a file name argument (the NAME keyword is used), $FNBST is called from the ASSIGN subroutine and uses the command string interpreter routines (.CSI1 and .CSI2) and the FCS-11 .PARSE logic to construct the FNB. $FNBST is called with R3 containing the impure area pointer, R2 containing the length of the name string, and R1 pointing to the start of the string. Registers R0, R1, and R2 are destroyed; R3, R4, and R5 are preserved.

If no file name is provided, the Default File Name Generation module, $FLDEF, is called to fill in the default file name. It stores the default FORTRAN file name and file type in the FNB. On input, the FORTRAN default file name is FOR0nn.DAT, where nn is the unit number. R3 points to the impure area. All registers are preserved.

The following FCS-11 utility routines are invoked during file name processing:

- The Default Directory Processing routine, .GTDID, is called by $OPEN$ to obtain the default directory used in constructing the File Name Block.
- The File Name Block Processing routines, .PARSE, .CSI$1, and .CSI$2, are called by $FNBST when a user file specification is to be used rather than the FORTRAN default file names. Refer to the *IAS/RSX I/O Operations Manual* for more information on these routines.

7.3 File CLOSE Processing

File CLOSE processing is performed by the OTS routine $CLOSE, which uses the following FCS-11 routines:

- The File Close Processing routine, CLOSE$, to close files
- The File Deletion routine, .DLFNB, to delete files
- The File Printing routine, .PRINT, to print and optionally delete files

The CLOSE source statement is compiled using an encoded argument list similar to that for the OPEN statement; however, only the UNIT, ERR, and DISPOSE keywords are allowed. The processing used is also similar: 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 LUB is not zeroed.

The processing for each keyword is described below, in order of execution:

1. ERR—The ERR= transfer address is obtained and the stack adjustment value is saved at offset COUNT. If present, the address is stored at offset ERREX.
2. UNIT—The unit number is obtained, and $FCHNL is called to obtain the LUB address. If no unit number is present, or if an invalid unit number is specified, a fatal error occurs.
3. DISPOSE—If not present, the existing disposition is used. '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.

7.4 Sequential Input/Output Processing

This section describes low-level OTS routines called by the I/O statement processors and format processors to perform the actual calls to FCS-11 for sequential record transfers, and to perform miscellaneous utility tasks. The routines are called with the work area address in R3.

The Sequential Input Routine, $GETS, does the following:

- Obtains the LUB pointer from offset FILPTR.

- Calls FCS macro GET$S to get a record. If FCS error IE.EOF is returned, or an ENDFILE record is read, the END= transfer is executed. If IE.EOF is returned and no END= transfer address is given, an error occurs. Errors cause the ERR= transfer to be executed.
- Increments the record count in D.RCCT and D.RCC2.
- Returns the actual record length in R1, and returns the start address of the record in R2 (R0 is undefined).

The Sequential Output routine, $PUTS, proceeds as follows:

- Obtains the LUB pointer from offset FILPTR
- Calls the PUT$S macro to output the record
- Increments the record count in D.RCCT and D.RCC2

$PUTS is called with the record length in R1. Registers R0, R1, and R2 are undefined upon return.

7.5 Direct Access Input/Output Processing

This section describes low-level OTS routines called by the I/O statement processors and format processors to perform the actual calls to FCS for direct access record transfers, and to perform miscellaneous utility tasks. The routines are called with the work area address in R3.

The Direct Access Input routine, $GETR, proceeds as follows:

- Obtains the LUB pointer from offset FILPTR, and calls $CKRCN to verify the record number and return it in R1 and R2
- Calls FCS macro GET$R to read the record
- Calls $ASVAR to update the associated variable

Registers R0, R1, and R2 are undefined.

The Direct Access Output routines, $PUTR and $PUTRI, proceed as follows:

- $PUTRI is called to initialize a direct access write operation.
- Obtains the LUB pointer from offset FILPTR and calls $CKRCN to verify the record number.
- Stores the record number at F.RCNM and F.RCNM+2 in the FDB.

- Calls the FCS routine .POSRC 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 executed.
- $PUTR is called to write the record.
- Obtains the LUB pointer from FILPTR.
- Computes the number of unfilled bytes in the record. The record is padded to the correct length with blanks for formatted records and zero bytes for unformatted records.
- Calls the FCS macro PUT$R to write the record and $ASVAR to update the associate variable.

Registers R0, R1, and R2 are undefined.

The Direct Access Record Number Checking routine, $CKRCN, verifies 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 (low-order) and D.RCN2 (high-order) in the LUB. The record number, if valid, is returned in R1 (high-order) and R2 (low-order).

$CKRCN is called with the LUB address in R0, and the impure-area pointer in R3. Registers R4 and R5 are preserved.

The Associated Variable Update routine, $ASVAR, obtains the current record number from offsets W.RECL and W.RECH, increments it by 1, and stores it in the associate variable at the address in D.AVAD in the LUB. $ASVAR is called with R0 pointing to the FCS portion of the FDB. Registers R1 and R2 are undefined.

7.6 Auxiliary Input/Output Operations

This section identifies and explains the routines that perform the operations of the following FORTRAN source statements: BACKSPACE, REWIND, ENDFILE, DEFINEFILE, and FIND.

BACKSPACE—BKSP$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the file is closed or is a direct access file, 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 LUB. The record count is decremented by 1, and n-1 reads are performed. The count is the logical record count, and therefore that multiple physical reads may be required for unformatted segmented records.

REWIND—REWI$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the file is closed or is a direct access file, 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).

ENDFILE—ENDF$

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

DEFINEFILE—DEFF$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the unit is open, an error occurs. The number of records is stored at D.RCNM and D.RCN2 in the LUB. 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, an error occurs. If the number of records or record size is negative, an error occurs.

FIND—FIND$

The FIND statement is contained in the same module as that of the DEFINEFILE statement. 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. No FCS-11 call is required.

7.7 Input/Output-Related Subroutines

This section describes the operation of three I/O-related subroutines. The *PDP-11 FORTRAN-77 User's Guide* describes these subroutines in greater detail.

ASSIGN

The unit number is placed in R2 and $GETFILE is called to get the LUB address. The file specification string address is placed in R1. If no string length is present, it is computed by scanning for a zero-byte. $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 LUB 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. Table 7–2 summarizes the argument processing.

Table 7–2: FDBSET Argument Summary

Call FDBSET Arguments	FDB Setting
Argument 1 = n	Set n in F.LUN
Argument 2 = 'READONLY'	Set FO.RD in F.FACC
= 'NEW'	Set FO.WRT in F.FACC
= 'OLD'	Set FO.UPD in F.FACC
= 'APPEND'	Set FO.APD in F.FACC
= 'UNKNOWN'	Set FO.UPD in F.FACC; if no such file, then set FO.WRT in F.FACC.
Argument 3 = 'SHARE'	Set FA.SHR in F.FACC
Argument 4 = n	Set n in F.MBCT
Argument 5 = n	Set n in F.CNTG
Argument 6 = n	Set n in F.ALOC

n

Is an integer expression.

Chapter 8

RMS-11 Input/Ouput Support

This chapter discusses the RMS-11-specific portions of the OTS. Particularly the explicit RMS-11 operations used to implement FORTRAN I/O operations.

8.1 RMS-11 I/O Control Blocks

RMS-11 uses two primary and several secondary control blocks to control I/O operations. The primary control block for file functions (open, close, and so forth) is the File Access Block (FAB). The primary control block for record functions (read, write, and so forth) is the Record Access Block (RAB). RMS-11 uses auxiliary control blocks for file name parsing (NAM block) and indexed file key specification (XAB blocks).

The FORTRAN-77 I/O system uses one Logical Unit Control Block (LUB) to control I/O for each logical unit. The LUB contains FORTRAN control information, a RAB, and a file-name-string save area whose size is specified by the impure area offset W.FNML. The RMS FAB, NAM, and XAB control blocks are allocated and deallocated as needed.

Indexed file key specifications are processed by the OPEN statement and are allocated as a single block containing n key-definition XAB control blocks. Each XAB has two extra words containing position and size values used for consistency checks for an existing file. The file open processor $OPEN$ uses an RMS NAM control block to obtain the expanded file name string that is used for error reporting and file deletion.

For more information about RMS-11 control blocks, see Section 4.2 and Appendix B.

8.1.1 Dynamic Storage Allocation for Control Blocks

All OTS and RMS–11 control blocks and I/O buffers are dynamically allocated and deallocated from a central storage pool. The size of the pool is determined by the Task Builder option EXTTSK or the /INC option of the INSTALL or RUN commands.

Two OTS procedures manage the storage: RQMEM$ allocates storage and RLMEM$ deallocates storage. RMS–11 obtains storage from the OTS by using the option $SETGSA, and the storage allocation algorithm uses the operating system procedure $RQLCB. The OTS storage pool listhead address is contained at offset W.DEVL. Best-fit allocation is used.

The three storage management procedures are:

- RQMEM$ (allocate storage)

  On input – R0 contains the size of the request.

  On output – R0 contains the address of a successful allocation. A Condition-bit error flag is returned. If unsuccessful, all other registers are preserved.

- RLMEM$ (deallocate storage)

  On input – R0 contains the address of the storage to deallocate. R1 contains the size of the storage.

  On output – R0 and R1 are undefined; all other registers are preserved.

- RMSQL$ (RMS-called GSA (get-space-available) routine to request and release storage)

  On input – R0 contains the address of an RMS pool (ignored). R1 contains the size of the block to allocate or deallocate. R2 contains 0 for allocation request; address of the block if release request.

  On output – R0 contains the address of a successful request. A Condition-bit error flag is returned if unsuccessful.

  RMSQL$ calls either RQMEM$ or RLMEM$ to process the request or release.

8.2 Open Processing

Default file open processing and OPEN statement processing merge into a single common routine, $OPEN$ (see Section 8.2.3), for a file open.

8.2.1 OPEN Statement Processing

Table 8–1 shows the OPEN statement keywords, the possible values of the keywords, and the RMS FAB and RAB settings associated with those values.

Table 8–1: FAB/RAB Settings for OPEN Statement

Keyword Name	Value	FAB/RAB Setting
ACCESS	'DIRECT' 'SEQUENTIAL' 'APPEND'	– – RB$EOF in O$ROP in RAB
ASSOCIATEVARIABLE	v	–
BLANK	'NULL' 'ZERO'	– –
BLOCKSIZE	n	Set n in O$BLS in FAB; set (max(n,RECORDSIZE)+511 /512) in O$BKS in FAB and O$MBC in RAB
BUFFERCOUNT	n	Set n in O$MBF in RAB
CARRIAGECONTROL	'FORTRAN' 'LIST' 'NONE'	Set FB$FTN in O$RAT in FAB Set FB$CR in O$RAT in FAB –
DISPOSE	'SAVE' 'DELETE' 'PRINT'	Use $CLOSE at file close Use $ERASE at file close Use $CLOSE at file close
ERR	s	-
EXTENDSIZE	n	Set IABS(n) in O$DEQ in FAB
FILE or NAME	f	Call $FWBST to initialize O$FNA and O$FNS in FAB

Table 8–1 (Cont.): FAB/RAB Settings for OPEN Statement

Keyword Name	Value	FAB/RAB Setting
FORM	'FORMATTED' 'UNFORMATTED'	– If positive, set FB$CTG in O$FOP in FAB;
INITIALSIZE	n	Set IABS(n) in O$ALQ in FAB.
KEY	n1:n2	Set n1 in O$POS0 in XAB; Set n2-n1+1 in O$SIZ0 in XAB; Set XB$STG in O$DTP in XAB; Set XB$CHG!XB$DUP in O$FLG in XAB if not primary key.
KEYCNT	n	Allocate n XABs and set address in O$XAB in FAB
Keyword Name	Value	FAB/RAB Setting
MAXREC	n	– (Depends on device type)
NOSPANBLOCKS	–	Set FB$BLK in O$RAT in FAB
ORGANIZATION	'SEQUENTIAL' 'RELATIVE 'INDEXED'	Set FB$SEQ in O$ORG in FAB Set FB$REL in O$ORG in FAB Set FB$IDX in O$ORG in FAB
READONLY	–	Set FB$GET in O$FAC in FAB
RECORDSIZE or RECL	n	Set n in O$MRS in FAB if needed
RECORDTYPE	'FIXED' 'VARIABLE' 'SEGMENTED'	Set FB$FIX in O$RFM in FAB Set FB$VAR in D$RFM in FAB Set FB$VAR in O$RPM in FAB
SHARED	–	Set FB$WRI in O$SHR in FAB

Table 8–1 (Cont.): FAB/RAB Settings for OPEN Statement

Keyword Name	Value	FAB/RAB Setting
STATUS or TYPE	'OLD'	Use $OPEN
	'NEW'	Use $CREATE
	'SCRATCH'	Use $CREATE, set FB$TMD in O$FOP in FAB
	'UNKNOWN'	Try $OPEN. If no such file, then $CREATE
UNIT	n	Set n in O$LCH in FAB
USEROPEN	p	-

f
Is an array, array element, variable, or character constant.

n
Is an integer expression.

s
Is an executable statement label.

v
Is an integer variable name.

p
Is an external procedure name.

In basic OPEN statement processing, an argument list is searched and each keyword is located in a prescribed order. All information required by a given keyword is available when that keyword is processed. An appropriate default is used for keywords not in the list. If any errors occur during the search, the OPEN is not attempted, the ERR= transfer is taken, and the LUB is released.

The processing for each keyword is described below.

- ACCESS—The default access is 'SEQUENTIAL'. The FORTRAN LUB fields are set as follows:

  SEQUENTIAL - DV.SEQ
  DIRECT - DV.DIR
  APPEND - DV.SEQ and DV.APD
  KEYED - DV.KEY

  If DV.RDO and DV.APD are set, an error occurs.
- ASSOCIATEVARIABLE—The variable address is stored at D.AVAD. If the variable is type INTEGER*4, DV.AI4 is set.
- BLANK—'NULL' sets DV.BN. If the /F77 switch is set and no BLANK= is specified, the compiler passes a BLANK='NULL' parameter.
- BLOCKSIZE—The value specified sets the following values:

  - The physical blocksize for sequential tape files.

  - The multiblock count for sequential disk files.

  - The bucketsize for relative and indexed files.

  The values set are as follows for a BLOCKSIZE value of n (n must be positive and less than 32767):

  - Set n in O$BLS in the FAB for magtape.

  - Set (511+MAX(n,RECORDSIZE))/512 in O$BKS in the FAB for bucketsize and in O$MBC in the RAB for multiblock count.
- BUFFERCOUNT—The value specified is stored at O$MBF in the RAB. If the value is negative or greater than 127, an error occurs.
- CARRIAGECONTROL—If specified, DV.CC is set. 'FORTRAN' sets FB$FIN in O$RAT and 'LIST' sets FB$CR in O$RAT in the FAB. If DV.CC is not set and DV.FMP is specified, FD.FTN is the default.
- 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 and DV.SCR is set, 'DELETE' is the default; otherwise, 'SAVE' is the default.
- 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.

- EXTENDSIZE—The absolute value specified is stored at O$DEQ in the FAB. If the value is greater than 32767 or less than -32767, an error occurs.
- FILE or NAME—If specified, $FNBST is called to initialize the file name specification in the FAB. $FNBST returns an error if the string is incorrect.
- FORM—'FORMATTED' sets DV.FMP, 'UNFORMATTED' sets DV.UFP. If not specified and DV.DIR or DV.KEY is set, then DV.UFP is the default; otherwise, DV.FMP is the default.
- INITIALSIZE—The absolute value specified is stored at O$ALQ in the FAB. If the value was positive, FB$CTG is set in O$FOP in the FAB to indicate contiguous allocation.
- KEY—Each key entry is processed as follows:

  - n1 is set in O$POS0 of the XAB and X.POS in the FORTRAN portion of the XAB.

  - n2-n1+1 is set in O$SIZ0 of the XAB and X.SIZ in the FORTRAN portion of the XAB.

  - An error occurs if nl is greater than 32767 or if n2-n1+1 is greater than 255.
- KEYCOUNT—n XABs are allocated as a single control block. The XABs are linked together, and the XABBLK address is stored at O$XAB in the FAB and F.FXAB in the FORTRAN FABBLK. The XABs are initialized with XB$STG as the data type and XB$CHG and XB$DUP as options for all but the primary key. An error occurs if n is greater than 255 or negative.
- MAXREC—The value specified is stored at D.RCNM and D.RCN2. If the value is negative, an error occurs.
- NOSPANBLOCKS—If specified, FB$BLK is set in O$RAT in the FAB.
- ORGANIZATION—The default organization is 'SEQUENTIAL'. The FAB organization field is initialized, the LUB organization field is initialized, and the file access field, O$FAC in the FAB, is initialized as follows:

  If DV.RDO is set, then FB$GET is set; otherwise:

  SEQUENTIAL —FB$GET!FB$PUT!FB$TRN

  RELATIVE —FB$GET!FB$PUT!FB$UPD!FB$DEL

  INDEXED —FB$GET!FB$PUT!FB$UPD!FB$DEL

- READONLY—If present, DV.RDO is set.
- RECORDSIZE or RECL—The value is stored at D.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.
- SHARED—If specified, FB$WRI is set in O$SHR in the FAB.
- STATUS or TYPE—If not present, the default is 'NEW'. Note, however, that if the /F77 switch is set and no STATUS = parameter is specified in the source code, the compiler passes a STATUS = 'UNKNOWN' parameter. '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.
- UNIT—The unit number is obtained and $FCHNL is called to obtain the LUB pointer. Possible fatal errors include: no unit number, invalid unit number, or unit already open. A FAB is allocated and its address is stored in D.PFAB.
- USEROPEN—The address of the procedure is saved at F.UOPN in the FORTRAN portion of the FAB.

After all keywords are processed, $OPEN$ is called to perform the actual file open. If successful, then the key specifications, if present, are checked for consistency. The FAB and XAB control blocks are released.

8.2.2 Default OPEN Processing

Default OPEN processing sets the following values and then calls $OPEN$:

- Sets organization to SEQUENTIAL; sets DV.SEQ and FB$SEQ in O$ORG in the FAB
- If DV.FACC is not set and the I/O statement is an input operation, sets DV.OLD; otherwise, sets DV.NEW

Other fields and values may have been set by CALL ASSIGN, CALL FDBSET, or DEFINEFILE statements.

8.2.3 $OPEN$ Routine

The $OPEN$ routine opens the file and performs the various checks and computations common to OPEN statement processing (see Section 8.2.1) and default OPEN processing (see Section 8.2.2).

Before file open, $OPEN$ performs the following operations:

- If no user file specification is provided, uses the default file string for error reports
- If append access is not specified, sets FB$NEF in O$FOP in the FAB to inhibit positioning to end-of-file for magnetic tape files
- Sets the RMS record format to FB$FIX if DV.FIX is set; otherwise, sets the record format to FB$VAR
- Computes a record length as follows:

  If a user-specified value is available, uses that value and moves it to O$MRS in the FAB; otherwise, uses 133 for formatted files, 128 for unformatted files with fixed-length records, and 126 for unformatted files with variable or segmented records; if DV.FIX or DV.REL is set, sets record length in O$MRS in the FAB
- If DV.CC is not specified, sets FB$FTN in O$RAT in the FAB if formatted
- Saves the organization type for consistency checks
- Sets RB$LOC and RB$UIF in O$ROP in the RAB to enable locate mode I/O and to permit WRITE statements to update records in relative files
- Sets O$UBF and O$USZ in the RAB to reflect the user record buffer as specified by impure area offsets W.BFAD and W.BLEN
- Creates an RMS NAM block to obtain the expanded file name string for error reports and file deletion

If impure area offset W.UOPN is nonzero, $OPEN$ calls the user's routine to perform the RMS OPEN and CONNECT. If F.UOPN is not set, $OPEN$ calls $CREATE, if DV.NEW is set, or $OPEN$ if DV.NEW is not set. If DV.UNK is set and $OPEN$ fails with error ER$FNF, $CREATE is tried with DV.NEW set.

After the $CREATE or $OPEN routine is executed, the following operations are performed:

- The expanded file name string from the NAM block is copied to the LUB name string save area and the NAM block is deleted.

- DV.OPN is set to indicate that the file is open.
- The file organization is checked for consistency.
- The record format is checked for consistency; if the user-specified record type does not match the file's record format, an error occurs.
- The record length is checked for consistency as follows:

  If the user-specified length does not match the file size, an error occurs. If the default length is used, the value is set to the maximum of the file and default values.
- The RMS $CONNECT operation is performed.

If any errors occur, either reported by RMS or as a result of the consistency checks, the file is closed. If DV.NEW is set, the file is deleted as well.

8.2.4 USEROPEN Interface Specification

The USEROPEN parameter of the OPEN statement allows you to access special RMS processing options not explicitly available in the FORTRAN-77 language. The value of the USEROPEN parameter is the name of a user-written MACRO-11 routine that the OTS calls to open a file. To use this facility, you must do the following:

- Using the MACRO-11 language, write a routine that opens the file.
- In your FORTRAN-77 program, include the statement

  ```
  EXTERNAL filename
  ```

  where "filename" is the name of the MACRO-11 routine you wrote to open the file.
- In the OPEN statement in your FORTRAN program, include the keyword parameter USEROPEN=filename, where, again, "filename" is the name of your MACRO-11 routine.

Although the MACRO-11 routine is called by the OTS (not your FORTRAN-77 program), you should write it as if it were being called by a FORTRAN-77 program. You must report the status of the open operation in R0. The OTS invokes the routine as a standard FORTRAN function of two arguments using the standard FORTRAN calling convention:

```
ISTS= userprocedure (FAB,RAB)
```

FAB
The address of the RMS FAB for the logical unit.

RAB
The address of the RMS RAB for the logical unit.

ISTS
The error status to be returned. This value is expected to be the RMS completion status (STS value) and follows the RMS conventions (positive numbers indicate success and negative numbers indicate failure.) Note that the status is returned only to the OTS, not to the FORTRAN program.

The following limits and constraints are imposed on the user procedure:

- All FORTRAN processing is completed prior to the call. All nonzero fields containing addresses must be preserved so that postprocessing will operate correctly and storage for control blocks and scratch areas can be returned.
- If additional XABs are used, they must be included in the XAB list at the front. XAB is the address of the RMS key access block for the logical unit.
- All control blocks allocated by the procedure must be deallocated as well. Only control blocks and storage allocated by the OTS are deallocated by the OTS.
- The FAB address specified is not valid after the file open is completed. The RAB address specified is valid until the logical unit is closed. Care must be taken to ensure that an invalid FAB address is not saved for later use.
- The user procedure must perform both the $CREATE/$OPEN function and the $CONNECT function.

The following sample FORTRAN program and user-open procedure specify that bucket fill numbers are to be used when records are inserted.

```
EXTERNAL USROPN
OPEN    (UNIT=1,    ORGANIZATION='INDEXED',    ACCESS='KEYED',
1  USEROPEN=USROPN)
     .
     .
     .
END
```

```
USROPN::  MOV      2(R5),R2            ;Get FAB pointer
          MOV      4(R5),R1            ;Get RAB pointer
          $OPEN    R2                  ;Open the file
          MOV      O$STS(R2),R0        ;Get error status
          BLE      1$                  ;Quit on error
          BIS      #RB$LOA, O$ROP(R1)  ;Use bucket fill numbers
          $CONNECT R1                  ;Connect the RAB
          MOV      O$STS(R1),R0        ;Get error status
    1$:   RETURN                       ;Return with status in R0
```

8.2.5 File Open Utility Routines

The following procedures are used internally as part of file open processing:

FABRQ$

Allocates and initializes a FORTRAN FAB block. The address of the FAB is set in D.PFAB in the LUB and in O$FAB in the RAB. The string FOR0nn.DAT is set up as the default file name string by initializing O$DNS and O$DNA in the FAB.

This procedure is called with the LUB pointer in R0. It returns with the FAB pointer in R1 and all other registers preserved. A Carry-bit (C-bit) error is returned if no storage is available.

FABRL$

Deallocates a FAB. In addition, deallocates the XAB block connected to the FAB through offset F.PXAB. LUB offset D.PFAB is set to 0. This routine is called with the LUB pointer in R0. All registers are preserved. If no FAB is currently allocated, this routine has no effect.

$FNBST

Initializes the user file name specification for the LUB at offset FILPTR. The string is copied to the name string area of the LUB. Scanning ceases when the user count is exhausted or an ASCII null byte is scanned. The string is converted to uppercase and space characters are removed.

On input, this routine has the following register assignments:

- R1—address of file name string
- R2—length of string or 0
- R3—impure area pointer

On output, R0, R1, and R2 are undefined; R3, R4, and R5 are preserved. A Carry-bit (C-bit) error is returned if the string does not fit in the name string save area, or if the length of the string is zero.

8.3 File Close Processing

File close processing is performed by the routine $CLOSE. The following functions are performed:

- $DISCONNECT is executed if O$ISI in the RAB is nonzero.
- $CLOSE is executed.
- If DV.DEL is set, $ERASE is executed. The file name specification is taken from the name string save area.
- If DV.SPL is set, no operation is performed.

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 LUB is *not* zeroed.

The following list describes the processing for each keyword:

- ERR—The ERR= transfer address is obtained and the stack adjustable value is saved at offset COUNT. The address is stored at offset ERREX, if present.
- UNIT—The unit number is obtained and $FCHNL is called to obtain the LUB address. If no unit number is present or an invalid unit number is specified, a fatal error occurs.
- DISPOSE—If not present, the existing disposition is used. '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.

8.4 Sequential Input/Output Processing

The following sections describe low-level OTS routines called by the I/O statement processors and format processors to perform the actual calls to RMS for sequential record transfers, and to perform miscellaneous utility tasks. These routines are called with the work area address in R3.

8.4.1 Sequential Input ($GETS)

The LUB pointer is obtained from offset FILPTR. The RMS $GET operation is executed to get a record. If RMS error ER$EOF is returned or if an ENDFILE record is read, the END= transfer is made; any other error transfers control to the ERR= address. The record count at D.RCCT and D.RCC2 is incremented.

8.4.2 Sequential Output ($PUTS)

The LUB pointer is obtained from the offset FILPTR. The RMS operation $PUT is executed to output the record. If RMS error ER$NEF is returned for a sequential organization file, the file is not positioned at end-of-file and must be truncated before the record can be written. This is done by performing RMS sequential $FIND and $TRUNCATE operations before re-executing the $PUT operation.

If the file contains fixed-length records, the record is padded with spaces for formatted files and nulls for unformatted files. The record count at D.RCCT and D.RCC2 is incremented. This routine is called with the record length in R1. On return, R0 contains the LUB pointer; R1 and R2 are undefined; and R3, R4, and R5 are preserved.

8.5 Direct Access Input/Output Processing

The following sections describe low-level OTS routines called by the I/O statement processors and format processors to perform the calls to RMS–11 for direct access record transfer, and to perform miscellaneous utility tasks. These routines are called with the impure area address in R3. For sequential organization files, the RMS $SETRECN operation is executed to include the RMS routine that converts a relative record number to an actual record file address.

8.5.1 Direct Input ($GETR)

The $GETR routine proceeds as follows:

- Obtains the LUB pointer from offset FILPTR
- Calls $CKRCN to verify the record number and sets the KRF and KSZ fields in the RAB
- Executes the RMS $GET operation to read the record
- Calls $ASVAR to update the associated variable

8.5.2 Direct Output ($PUTR And $PUTRI)

$PUTRI initializes a direct access write operation. It proceeds as follows:

- Obtains the LUB pointer from offset FILPTR.
- Calls $CKRCN to verify the record number and initialize the KRF and KSZ fields of the RAB.
- For sequential files, executes $UPDATE to update an existing record, or $PUT to write a new record.
- Executes an RMS $FIND operation to position the file to the desired record. If RMS error ER$EOF is returned, the record does not exist within the current file storage allocation. In that case, the sequential $PUT operation automatically extends the file to the correct size to accommodate the record.

$PUTR is called to write the record. It proceeds as follows:

- Obtains the LUB pointer from FILPTR
- If the file contains fixed-length records, pads the record (with spaces for formatted records and zero bytes for unformatted records) to the correct length
- Calls $ASVAR to update the associate variable

8.5.3 Direct Delete ($DELETE)

The LUB pointer is obtained from offset FILPTR. The RMS $DELETE operation is executed to delete the record.

8.5.4 Direct Access Record Number Checking ($CKRCN)

$CKRCN verifies the current record number by comparing it with 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 (low-order) and D.RCN2 (high-order) in the LUB. The record number, if valid, is returned in R1 (high-order) and R2 (low-order). This routine is called with the LUB address in R0. Registers R4 and R5 are preserved.

8.5.5 Associated Variable Update ($ASVAR)

The current record number is obtained from offsets W.RECL and W.RECH, incremented by 1 and stored in the associate variable at the address in D.AVAD in the LUB. $ASVAR is called with the LUB pointer in R0. Registers R1 and R2 are undefined.

8.6 Keyed Input/Output Processing

The following sections describe low-level OTS routines called by the I/O statement processors and format processors to perform the calls to RMS for keyed record transfer, and to perform miscellaneous utility tasks. These routines are called with the work area address in R3.

8.6.1 Keyed Input ($GETK)

The LUB pointer is obtained from offset FILPTR, and $CKKEY is called to verify the validity of the key expression and initialize the KRF and KSZ fields in the RAB. The RMS $GET operation is executed to read the record. On output, R0 contains the LUB pointer; R1 is undefined.

8.6.2 Keyed Output ($PUTS)

Keyed output is performed identically to sequential output.

8.6.3 Keyed Rewrite ($UPDATE)

The LUB pointer is obtained from offset FILPTR. If the file contains fixed-length records, the record is padded with spaces for formatted files or nulls for unformatted files. The RMS $UPDATE operation is executed to update the record.

8.6.4 Keyed I/O Specifier Checking ($CKKEY)

$CKKEY verifies the key specification in a keyed I/O statement and sets the proper control information in the LUB. It is called with the LUB pointer in R0 and the impure area pointer in R3. Registers R1 and R2 are destroyed; all other registers are preserved.

8.7 Auxiliary Input/Output Operations

This section identifies and explains the routines that perform the operations of the FORTRAN statements BACKSPACE, REWIND, ENDFILE, UNLOCK, DEFINEFILE, FIND, and DELETE.

BACKSPACE—BKSP$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the file is closed or is a direct access file, the operation is ignored. If the file is opened for append, an error occurs. An RMS $REWIND operation is executed to position the file at its beginning. The record count is obtained from D.RCCT and D.RCC2 in the LUB. 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 segmented records.

REWIND—REWI$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the file is closed or is a direct access file, the operation is ignored. The append bit is cleared and the record count at D.RCCT and D.RCC2 is zeroed. An RMS $REWIND operation is executed to position the file at its beginning.

ENDFILE—ENDF$

$INITIO is called with argument mask FL.IGN + FL.WRT to open the file, if necessary, and prepare for the output operation. If the file has relative or indexed organization, or contains fixed-length records, an error occurs. A 1-byte record containing octal 32 (CTRL/Z) is output to the file using $PUTS.

UNLOCK—UNLK$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the file is closed, the operation is ignored. An RMS $FREE operation is executed to unlock the currently locked record bucket. If RMS error ER$RNL is returned, indicating that no record was locked, the error is ignored.

DEFINEFILE—DEFF$

The unit number is obtained and $GETFILE is called to obtain the LUB address. If the unit is open, an error occurs. A FAB is allocated if one is not already present. The number of records is stored at D.RCNM and D.RCN2 in the LUB. The record size is converted to bytes and stored at D.RSIZ in the LUB. The associated variable address is stored at D.AVAD and DV.AI4 is set if the associated variable is INTEGER*4. DV.DIR and DV.UFP are set. If DV.DIR was previously set, or if the number of records or record size is negative, an error occurs.

FIND—FIND$

$INITIO is called with argument mask FL.RD+FL.REC+FL.IGN to open the file, if necessary, and prepare for an input operation. $CKRCN is called to initialize the RAB KRF and KSZ fields and verify the validity of the record number. The RMS $FIND operation is executed to locate the record. $ASVAR is called to update the associated variable.

DELETE—DLS$ and DLR$

$INITIO is called to prepare for the I/O operation. The argument mask is as follows:

```
Sequential DELETE: FL.WRT+FL.DEL+FL.IGN
Direct DELETE:     FL.WRT+FL.DEL+FL.IGN+FL.REC
```

For sequential DELETE, RMS $DELETE is executed to delete the record.

For direct DELETE, $CKRCN is called to initialize the record number. If the current record number equals the requested record number, RMS $DELETE is executed to delete the current record without unlocking it. If the operation fails with the error NO CURRENT RECORD, then $GETR is called to locate and lock the record, and RMS $DELETE is called to

delete the record. The $ASVAR routine is called to update the associated variable.

8.8 Input/Output-Related Subroutines

This section describes the operations of the three I/O-related subroutines ASSIGN, CLOSE, and FDBSET. The subroutines are described in detail in the *PDP-11 FORTRAN-77 User's Guide*.

ASSIGN

The unit number is placed in R2 and $GETFILE is called to get the LUB address. A FAB is allocated if one is not present. The file specification string address is placed in R1. If no length is present, the string length is computed by scanning for a zero-byte. $FNBST is called to store the file name string in the LUB and set up the file name specification in the FAB.

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 LUB address. A FAB is allocated if one is not present. The first character of the access mode string is checked against the list, the corresponding file access fields are stored in the FAB and LUB, and DV.FACC is set in the LUB.

Parameter Value	O$FAC in the FAB	LUB Status
'NEW'	FB$GET!FB$PUT!FB$UPD!FB$TRN	DV.NEW
'OLD'	FB$GET!FB$PUT!FB$UPD!FB$TRN	DV.OLD
'READONLY'	FB$GET	DV.OLD!DV.RDO
'APPEND'	FB$GET!FB$PUT!FB$UPD!FB$TRN	DV.OLD!DV.APD
'MODIFY'	FB$GET!FB$PUT!FB$UPD!FB$TRN	DV.OLD
'UNKNOWN'	FB$GET!FB$PUT!FB$UPD!FB$TRN	DV.UNK

If the third argument is 'SHARED', FB$WRI is set in the SHR field in the FAB. The fourth argument, if present, sets the MBF field in the RAB. The fifth argument, if present, sets the ALQ0 field in the FAB. The sixth argument, if present, sets the DEQ field in the FAB. The value is made positive.

Chapter 9

Format Processing and Format Conversions

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

9.1 Compiler Format Language

Format specifications are compiled into a standard internal form, which consists of a format code byte followed by one to five bytes of optional format code parameters. Figure 9–1 illustrates this form.

9.1.1 Format Code Byte

The format code byte consists of a 6-bit format code, a 1-bit Variable Format Expression (VFE) flag, and a 1-bit repeat count flag.

The flags indicate whether the VFE mask and repeat count bytes are included in the compiled code. If the VFE flag equals 0, no VFEs are present in the format. If the VFE flag equals 1, VFEs are present and the compiled code includes a VFE mask byte followed by VFE addresses. If the repeat count flag equals 0, the repeat count for the format specification is 1. If the repeat count flag equals 1, the repeat count for the specification is greater than 1 or is a VFE, and the repeat count byte is included in the compiled code.

Figure 9–1: Format Code Form

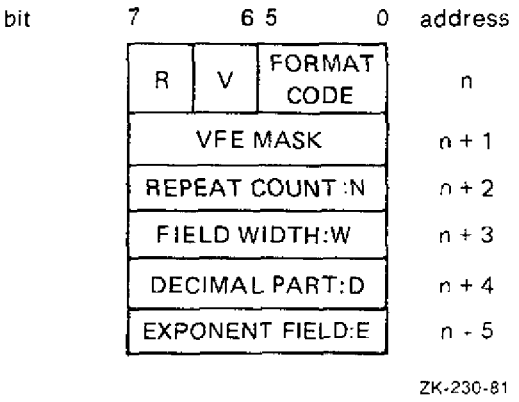


Table 9–1 lists the decimal value of each 6-bit format code, gives its source code form, and indicates whether it uses the field width and decimal part parameters.

Table 9–1: Compiled Format Codes

Decimal Code	Source Form	Repeat Count	W	D	E	Notes
0 - 3	–	–	–	–	–	Format error, only 0 and 2 are used currently; 0 means format syntax error; 2 means 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	/	–	–	–	–	
14	$	–	–	–	–	
16	:	–	–	–	–	
18	sP	–	s	–	–	
20	Q	–	–	–	–	
22	Tn	–	n	–	–	
24	nX	n-1	–	–	–	Previous PDP–11 FORTRAN IV-PLUS behavior for nX (Compiler does not generate this code for nX; OTS still includes routine for compatibility)
26	nHc1 . . . cn or 'c1..cn'	n-1	–	–	–	n not VFE; n characters 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	–	

Table 9–1 (Cont.): Compiled Format Codes

Decimal Code	Source Form	Repeat Count	W	D	E	Notes
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	–	–	–	
5	S	–	–	–	–	New format descriptors
7	SP	–	–	–	–	
9	SS	–	–	–	–	
11	BN	–	–	–	–	
13	BZ	–	–	–	–	
15	TLn	–	n	–	–	
17	TRn or nX	–	n	–	–	
19	nZw	n-1	w	–	–	
21	nZ	n-1	–	–	–	Default Z format
23	nEw.dEe	n-1	w	d	e	E format descriptor with exponent component
25	nGw.dEe	n-1	w	m	–	G with e component
27	nOw.m	n-1	w	m	–	O, Z, I with m component
29	nZw.m	n-1	w	m	–	
31	nIw.m	n-1	w	m	–	

9.1.2 Format Code Parameters

Up to five bytes of format code parameters may appear in the compiled code for a format specification. The parameters are:

- VFE Mask Byte—indicates whether the other format code parameters are VFEs or compiled constants. Bits 7, 6, and 5 are associated with the repeat count, field width, and decimal part parameters, respectively. A bit setting of 1 means that the associated parameter is a VFE; a 0 setting means that the associated parameter is a compiled constant.
- Repeat Count Byte—contains the repeat count value when the repeat count is not 1. This value is 1 less than the source code value. It must be in the range 1 to 255.
- Field Width Byte—contains the field width or tab position in the range 1 to 255, or the scale factor in the range -128 to +127.
- Decimal Part Byte—contains the decimal field width for the floating-point conversion codes, in the range 0 to 255; or contains the significant digit part for the I, O, and Z formats in the form Iw.m, Ow.m, or Zw.m.
- Exponent Field Width Byte—contains the optional exponent field width value, in the range 0 to 255. The default value is 2.

When the repeat count, field width, or decimal part is a VFE, the VFE address begins on the next word boundary after the VFE mask byte. The VFE is compiled as an unparameterized arithmetic statement function of type INTEGER*2 and is called by the instruction JSR PC,xxxx, with R5 pointing to the program unit argument list. The format interpreter performs all range checking on the result.

9.1.3 Hollerith Formats

Quoted format strings (character constants) are compiled as Hollerith constants. The characters to be transmitted are included in the compiled code following the repeat count. The repeat count cannot be a VFE.

9.1.4 Default Formats

Most format code field descriptors have default values that are supplied if no numeric value is present. The defaults are determined from the format code and the data type of the corresponding list element, as follows:

Format Code	Data Type	Default Values of W, W.D, or W.DE
I	INTEGER*2	7
I	INTEGER*4	12
E,G	REAL*4	15.7(E=2)
E,G	REAL*8	25.16(E=2)
D,F	REAL*4	15.7
D,F	REAL*8	25.16
O,Z	All	W=MAX(7,MIN(255(8*ELEM_SIZE)/3+2))
L	All	2
A	All	Number of bytes in the variable
X	—	1

9.1.5 Format Compiled Code Example

This section gives an example of the code resulting from the compilation of a FORMAT source statement.

The FORTRAN statement:

```
1   FORMAT(1X, F13.5, 'ABCDE', <K>I10, 3(2E15.7E4)/)
```

is compiled into the following:

```
.1:   .BYTE   21,1          ; 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
```

```
                .WORD   L$VFE           ; VFE address
                .BYTE     12            ; I10
                .BYTE      4            ; Reversion point
                .BYTE   206,2           ; Left paren and repeat count
                .BYTE   227,1           ; E format code and repeat count
                .BYTE   17,7,4          ; E15.7E4
                .BYTE     10            ; Right paren
                .BYTE     14            ; / code
                .BYTE     12            ; End-of-format

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

9.2 Format Processing PSECTs

The OTS uses the following program sections (PSECTs) 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 $FIO, $LSTI, and $LSTO
- $$FIOI—contains the code for integer conversions
- $$FIOL—contains the code for logical conversions
- $$FIOR—contains the code for floating-point conversions
- $$FIOS—contains the list-directed input constant storage block
- $$FIOZ—contains the code for octal and hexadecimal conversions
- $$FIO2—contains the addresses of the conversion routine entry points

Each module stores its own entry point address in $$FIO2. The processing routines pick up the addresses of the appropriate conversion routines as needed (if that address is 0, an error occurs). The PSECTs have the GBL attribute so that the Task Builder can correctly build overlaid tasks.

None of the 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.

9.3 Format and List-Directed Processors

The format and list-directed processors—$FIO, $LSTI, and $LSTO – operate as coroutines with the I/O transmission operators. They are called at the end of I/O initialization, and process formats and list items until called with offset VARAD equal to 0.

9.3.1 Format Processor—$FIO

$FIO processes through the format, calling an internal routine for each format code. It calls VFEs as encountered, with all context saved and R5 restored to the user code value. When $FIO encounters a format requiring a list item, it calls the appropriate conversion routines (except the 'A' format, which is handled within $FIO) until no elements remain in the list (offset VARAD = 0). For nested group repeat specifications, $FIO uses a pushdown stack in the work area. Offset FSTKP points to the current position; offset FSTK is the base of the pushdown stack.

9.3.2 List-Directed Input Processor—$LSTI

$LSTI lexically scans the external record, delimits a field of input characters, determines the data type of the field, and calls the appropriate input conversion routine. It converts the resulting internal data value to the appropriate type and moves it to the list element. The currently active data value is stored at the address in PSECT $$FIOS pointed to by the work area offset W.PLIC.

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

9.3.3 List-Directed Output Processor—$LSTO

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

Data Type	Format
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,')'
CHARACTER*n or Hollerith	nA1 where n is the string length.

If the computed field length is longer than the number of remaining characters in the record, $LSTO writes the current record and begins a new record. Each item is contained in a single record except for character constants that are longer than a single record. $LSTO inserts a space at the front of each record for carriage control. The record length is the record size specified in the RECORDSIZE parameter of the OPEN statement. If no RECORDSIZE parameter is specified, the default is 81 bytes, which yields 80 print positions.

9.4 Run-Time Format Compiler—FMTCV$

Format specifications stored in arrays are converted into the required form during execution. This is done by the following:

1. Pushing the address of the array specification
2. Executing JSR PC,FMTCV$

FMTCV$ does not delete the stack argument; it replaces its value with the address of the compiled format.

Object time formats are compiled into a buffer in the OTS, whose length is controlled by the Task Builder option FMTBUF. The buffer's 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 bytes.
- R2 contains any numeric value being accumulated.
- Offset NOARG indicates the number of expected arguments for the code.
- Offset PARLVL specifies the parentheses depth encountered.
- Offset NUMFLG indicates whether a number is available in R2.

The module examines each source character. If the character is a digit, a number is accumulated; if it is a number or a special character, a dispatch is made to process the format code.

If the buffer space is exhausted, FMTCV$ stores the FMTBIG format code (2) in the first byte of the compiled format and returns an error. If a format syntax error is detected, FMTCV$ stores the FMTBAD format code (0) in the first byte and returns an error.

9.5 Integer and Octal Conversions

For input, the routines called are OCI$ for octal conversions (F4P V3 version) and ICI$ for integer conversions. The calling sequence is:

1. Push the address of the input string
2. Push the number of input characters (high bit of this word indicates BN/BZ; 0=BZ and 1=BN)
3. Call ICI$ (or OCI$)

The routines return a 2-word result on the stack in INTEGER*4 format. The calling arguments are deleted. If an error occurs, the C-bit is set and the value returned is 0. The floating-point conversions call the routine at entry point $ECI to input the exponent field.

For output, the routines called are OCO$ for octal conversions (previous PDP–11 FORTRAN IV-PLUS version), and ICO$ for integer conversions. The calling sequence is:

1. Push the address of the output field
2. Push the width of the output field (high bit of this word indicates SP/SS; 0=SS and 1=SP)
3. Push the INTEGER*4 value
4. Call ICO$ (or OCO$)

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

Also for output, IMO$ is called for integer conversions of the form Iw.m. The calling sequence is:

1. Push the address of the output field
2. Push the width of the output field (high bit of this word indicates SP/SS; 0=SS and 1=SP)
3. Push the INTEGER*4 value
4. Push the least number of digits to be output
5. Call IMO$

NOTE

The OTS no longer uses the entry points OCI$ and OCO$ for octal conversions. They are included for compatibility purposes.

9.6 Hexadecimal and New Octal Conversions

The hexadecimal and new octal conversions apply to all data types in PDP–11 FORTRAN–77. The calling sequence uses descriptors instead of values on the stack. For input, the routines called are ZCI$ for hexadecimal conversions and NOCI$ for octal conversions. The calling sequence is:

1. Push the address of the input string.
2. Push the number of input characters (high bit of this word indicates BN/BZ; 0=BZ and 1=BN).
3. Push the variable address.
4. Push the variable length.

5. Call ZCI$ or NOCI$.

The return is made with the arguments deleted and the value loaded into the variable whose address was given. If an error occurs, the C-bit is set and the value returned is 0.

For output, the routines called are ZMO$ for hexadecimal conversions and OMO$ for octal conversions. The calling sequence is:

1. Push the address of the output field.
2. Push the width of the output field (high bit of this word indicates SP/SS; 0=SS and 1=SP).
3. Push the least number of digits to be output (for Zw.m and Ow.m).
4. Push the variable address.
5. Push the variable length.
6. Call ZMO$ or OMO$.

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

9.7 Logical Conversions

The input logical conversion routine, LCI$, is called as follows:

1. Push the address of the input field.
2. Push the width of the input field.
3. Call LCI$.

LCI$ returns a 1-word result on the stack: 0 for .FALSE and -1 for .TRUE. The calling arguments are deleted. If an error occurs, the C-bit is set and .FALSE is returned.

The output logical conversion routine, LCO$, is called as follows:

1. Push the address of the output field.
2. Push the width of the output field.
3. Push the 1-word logical value.
4. Call LCO$.

The return is made with the calling arguments deleted and the C-bit cleared.

9.8 Real, Double-Precision, and Complex Conversions

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

1. Push the address of the input field.
2. Push the width of the input field (high bit of this word indicates BN/BZ; 0=BZ and 1=BN).
3. Push the decimal part width.
4. Push the scale factor (P format).
5. Call RCI$.

RCI$ returns a 4-word, double-precision result on the stack. The calling arguments are deleted. If an error occurs, the C-bit is set and the value returned is 0.0. If an exponent subfield is encountered, $ECT is called in the integer input conversion routine to handle the conversion.

The output conversion routines, DCO$, ECO$, FCO$, and GCO$, are called as follows:

1. Push the address of the output field.
2. Push the width of the output field (high bit of this word indicates SP/SS; 0=SS and 1=SP).
3. Push the decimal part width (high byte of this word contains the value of e for forms Ew.dEe or Gw.dEe).
4. Push the scale factor.
5. Push the 4-word, double-precision value.
6. Call DCO$, ECO$, FCO$, or GCO$.

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

The real, double-precision, and complex conversions are done in the software; the FPP unit is not used.

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

9.9 Format Conversion Error Processing

When a format conversion error occurs, both methods of error continuation, ERR=transfer and return (see Section 10.2.2.1), are generally supported. The actions taken for these errors are as follows:

Error 59	– List-directed I/O syntax error. The result value is null (no change).
Error 61	– Format/variable type mismatch error. The value is used as is, without conversion.
Error 63	– Output conversion error. The field is filled with asterisks.
Error 64	– Input conversion error. The result value is 0, 0. or 0.D0.
Error 68	– Variable format expression value error. A value of 1 is used for repeat count or field width; a value of 0 is used for the decimal part or scale factor.

For more information on format conversion error processing, see the *PDP-11 FORTRAN-77 User's Guide*.

Chapter 10

Error Processing and Execution Control

This chapter discusses execution control processing, detecting and processing run-time errors, and generating error messages.

10.1 Task Initialization

The first instruction of every FORTRAN main program calls the OTS initialization routine, as follows:

```
JSR  PC,OTI$
```

The following operations are performed:

- An SVTK$S executive directive initializes the synchronous trap vector.
- $STFPP is called to initialize the FP-11 floating-point processor or the KEF11A floating-point microcode option (unless F4PEIS is used).
- The error control byte table is copied into impure storage.
- The number of available logical units is computed as the minimum of the size of the device table program section (PSECT) and the value of impure area offset W.LUNS. The device table PSECT is set to zero.
- The user record buffer PSECT size is computed and stored at impure area offset W.BLEN.
- Miscellaneous impure area offsets are set to zero.
- The task error count limit is set to 15.
- $VINIT is called to initialize the virtual array mapping window if virtual arrays are used.
- If the FCS-11 files system is being used, FINIT$ is executed to initialize FCS.

- If the RMS–11 file system is being used:

  – The free storage pool size is determined using the task .LIMIT directive, and the GPRT$S and GTSK$S system directives and free storage list head are initialized.

  – An RMS–11 $SETGSA operation is executed to initialize the RMS GSA storage allocation procedure.

10.2 Execution-Time Errors

The following sections describe the types of errors reported by the OTS.

10.2.1 TRAP Instruction Processing

The OTS uses TRAP instructions to report errors. FORTRAN error numbers range from 1 through 120 (decimal). Not all numbers have a definition; some are reserved for future error definitions. The error number is in the low byte of the TRAP instruction. Internally, it is 128 larger than the reported number; thus, error number 21 is internally represented as 149. The first 128 TRAP values are available to users (see Section 10.4).

When a TRAP instruction is executed, the operating system transfers control to the TRAP instruction processor, $SST6, which checks the range of the error number. If that is valid, $SST6 calls $ERRAA to do the error analysis and reporting. If the error number is invalid, $SST6 returns an error number 1.

$ERRAA's processing is based on the contents of an error control byte in impure storage. 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 of the error control byte has no name. It is tested and cleared by the ERRTST system subroutine. When it is clear, an error has not occurred; when it is set, an error has occurred.

The standard bit combinations are as follows:

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

10.2.2 Error Control Byte Processing

$ERRAA obtains the error control byte from the OTS impure area. The sign bit is set. $ERRAA examines other bits in the error control byte and acts as follows:

- If the continue bit is cleared, the error report includes the exit flag.
- If the count bit is set and no ERR=address exists, offset W.ECNT is decremented. If W.ECNT is less than or equal to zero, the report includes the exit flag.
- If the continue-type bit is set and no ERR= address exists, the error report includes the exit flag.
- If the log bit is set, the error report includes the no-exit flag. If the task exits, the message is always logged.

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

10.2.2.1 Continuation Processing

Two types of continuation after an error are supported:

- Transfer to an ERR= address. This type is used for most I/O errors.
- Return to the source of the error. This type is generally used for errors other than I/O errors.

10.2.2.2 W.IOEF Error Processing

For some I/O errors, it may be better if the ERR= transfer is initiated by the I/O routine itself, rather than by the error processor ($ERRAA). For example, when OPEN statement processing detects an error in a keyword, the transfer to the ERR= address is delayed until all of the statement's keywords are examined.

Work area offset W.IOEF is used to obtain this special error processing. The effects of W.IOEF's value are as follows:

- When it is 0, default processing is enabled.
- When it is negative, default processing is performed except that the ERR= transfer is not made; instead, control is returned to the source of the error and the ERR= transfer can be made from there.
- When it is positive, the return type of continuation is always executed.

W.IOEF is initially zero and is reset to zero before exiting from a routine that uses it. Regardless of the W.IOEF setting, if no ERR= address exists, the task will exit.

10.2.3 Floating-Point Processor Errors

All Floating-Point Processor (FPP) errors are processed as Asynchronous System Traps (AST) in routine $FPERR. When divide-by-zero, overflow, or underflow occurs, zero is supplied as the result of the operation that caused the trap. The AST procedure log uses the TRAP instruction to issue the error report.

10.2.4 Error Message Construction and Processing

Error message construction and processing is performed by many small routines. Message processing begins with a call to $ERRLG, which controls the flow of message processing, calling the appropriate message utilities as required. $ERRLG produces a 5-line error log containing the following:

- On line 1, the task name and error number.
- On line 2, the text of the message.
- On line 3, the value of the program counter at the time of the error. This is found at offset W.PC.

- On line 4, the error count exceeded message. This is based on the error limit count stored at offset W.ECNT.
- On line 5, the I/O error data, which is based on the primary error field of the LUB (referenced by offset W.FERR), followed by the program unit traceback.

The log does not include any line that is inappropriate or unavailable. On RSX-11M/M-PLUS and RSTS/E, messages are output by issuing QIOs to the user's terminal.

For message construction, R3 points to the work area, R5 points to the current position in the message text being constructed, and offset W.ERLN points to the beginning of the error message buffer.

Offset W.MOTY is zero if the message output task (MO) is being used, and nonzero if QIOs to the terminal are being performed.

$ERRLG is also called to output messages from STOP and PAUSE statements. It uses the values of R0 and R1 to determine the type of message being generated, as follows:

- If R1 is 0, the message is associated with a STOP or PAUSE statement, and R0 points to the message text block.
- If R1 is not 0, the message is an error message, and R0 is -1 if the task is exiting and 0 if the task is continuing.

10.2.4.1 Message Construction Utilities

The following routines build the error report text in the error text buffer. They operate the same way whether messages are output by the message output task (MO) or QIOs to the user's terminal.

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. Store a CR/LF in buffer.
$ERRZA	–	Perform a GTSK$S directive to obtain the task name. Call $ATT and $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.

10.3 STOP and PAUSE Statement Processing

STOP and PAUSE statements are compiled to calls as follows:

1. Push the address of the display (0 indicates no display).
2. Call the statement-specific entry:

   STOP$ for STOP

   PAUS$ for PAUSE

All context is saved. $ERRLG (see Section 10.2.4) is called to output the message. STOP then jumps to $EXIT; PAUSE issues a SPND$S directive and returns.

10.4 User Interfacing to Error Processing

The first 128 (0 to 127) trap codes are available to users. TRAP instructions transfer control to the OTS error processor by using a System Synchronous Trap Table located in the OTS impure work area. The first word of this table has the global symbol $SST. You can use coding similar to the following 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
```

You can use similar techniques to intercept the other synchronous traps.

10.5 User Interfacing to Terminal Message Output

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

In FORTRAN-77:

```
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 nonerror type message
        JSR PC,$ERRLG       ; Output the message
        CMP (SP)+, (SP)+    ; Delete message block
        RTS PC              ; Return
        .END
```

User text is preceded by the task name. Only a single line can be output.

10.6 Execution Control Subroutines

The following subroutines are described in detail in the *PDP-11 FORTRAN-77 User's Guide:*

ERRSET—The error number specified by the user is extracted and checked for validity. The 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 information saved 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 sign bit of the error control byte is tested and cleared, and the result is returned in the second argument.

EXIT—Performs a jump to $EXIT.

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

Chapter 11

Other Compiled-Code Support Routines

This chapter describes routines that support various arithmetic and housekeeping operations required by the compiled code.

11.1 Arithmetic Operations

All the routines follow a common naming convention in which:

- 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
- The next letter (next two, in the case of exponentiation) indicates the data types of the arguments, as follows:

  I—INTEGER*2

  J—INTEGER*4

  R—Real

D—Double precision

C—Complex

- The last letter indicates how to access either the single argument of a one-argument operation or the second (right hand) argument of a two-argument operation. For two-argument operations, the first (left hand) argument is always on the stack. The last letter can be one of the following:

  S—indicates the argument is at the top of the 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 routines are called using the R4 convention described in Chapter 2. In addition, they all delete their stack arguments, return their result on the stack, and preserve the contents of general register 5 (R5).

11.1.1 Exponentiation

The exponentiation routines are as follows:

Routine	Base	*Data Type Exponent	Result
PWIIx$	INTEGER*2	INTEGER*2	INTEGER*2
PWIJx$	INTEGER*2	INTEGER*4	INTEGER*4
PWJIx$	INTEGER*4	INTEGER*2	INTEGER*4
PWJJx$	INTEGER*4	INTEGER*4	INTEGER*4
PWRIx$	REAL*4	INTEGER*2	REAL*4
PWRJx$	REAL*4	INTEGER*4	REAL*4

Routine	Base	*Data Type Exponent	Result
PWDIx$	REAL*8	INTEGER*2	REAL*8
PWDJx$	REAL*8	INTEGER*4	REAL*8
RWRRx$	REAL*4	REAL*4	REAL*4
PWRDx$	REAL*4	REAL*8	REAL*8
PWDRx$	REAL*8	REAL*4	REAL*8
PWDDx$	REAL*8	REAL*8	REAL*8
PWCIx$	COMPLEX*8	INTEGER*2	COMPLEX*8
PWCJx$	COMPLEX*8	INTEGER*4	COMPLEX*8
PWCCx$	COMPLEX*8	COMPLEX*8	COMPLEX*8

x is S, C, or P.

NOTE

This table of routines shows only the entry points called by the compiled code; it is not a complete list of all the supported forms of exponentiation. For example, a base of complex and an exponent of REAL*4 is supported by converting the REAL*4 to a complex number and calling the entry point that supports a base and exponent of complex. For a complete list of the supported forms of exponentiation, see the *PDP-11 FORTRAN-77 User's Guide*.

11.1.2 Complex Arithmetic Operations

The following entries are used in complex arithmetic operations:

ADCx$ — complex addition
SBCx$ — complex subtraction
MLCx$ — complex multiplication
DVCx$ — complex division
TSCx$ — complex test for zero
NGCx$ — complex negation
CMCx$ — complex compare
x is S, C, or P.

11.1.3 INTEGER*4 Arithmetic Operations

The following entries are used in INTEGER*4 arithmetic operations:

MLJx$ — Multiplication
DVJx$ — Division
x is S, C, or P.

11.1.4 Stack Swap Operations SWPxy$

The stack swap routines are used in conjunction with the out-of-line arithmetic operation entries when the order of evaluation causes the two arguments of the operation to be on the stack in reverse order. Entry names have the form:

```
SWPlr$
```

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

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

The two arguments are swapped on the stack.

11.1.5 Character Operations

The character operations routines are called using the PC convention described in Chapter 2, with the modification that a descriptor (length, address pair) is pushed on the stack for each argument. The two character operations are character assignment (entry point $CHASN) and character comparison (entry point $CHCMP).

Character assignment is called as follows:

1. Push the length of the destination (in bytes).
2. Push the address of the first byte of the destination.
3. Push the length of the source (in bytes).
4. Push the address of the first byte of the source.
5. JSR PC,$CHASN.

On return, the stack arguments are deleted.

Character comparison is called as follows:

1. Push the length of the left side of the comparison operation (in bytes).
2. Push the address of the first byte of the left side of the comparison operation.
3. Push the length of the right side of the comparison operation (in bytes).
4. Push the address of the right side of the comparison operation (in bytes).
5. JSR PC,$CHCMP.

On return, the stack arguments are deleted and the condition codes are set for an unsigned branch (C and Z bits of the PSW are valid).

11.2 Array Processing Support

An Array Descriptor Block (ADB) is a data structure provided by the compiler to describe an array. FORTRAN-77 compiled code uses ADBs for the following:

- Array subscript calculations for dummy argument arrays
- I/O calls that transmit an entire array
- Array subscript limit checking when specified by the compiler /CK command switch
- Virtual array load and store operations

The compiler defines the constant parts of an ADB. The varying parts are initialized when the subprogram containing the array declaration is executed.

The offsets within the ADB are as follows:

A.ASTR – Actual base storage address (first element) or, for virtual arrays, the 64-byte block number of the array base in virtual storage.

A.ASUM – Assumed size array flag bit in code word A.CWRD.

A.A0 – Zeroth-element address (address of A (0,0,0 . . . 0)). This offset is ignored for virtual arrays.

A.CWRD – Code word containing the number of dimensions, data type, element size, and information denoting whether it is an assumed size array:

Assumed Size Array Flag	Data Type	Number of Dimensions	Element Size
1 bit	4 bits	3 bits	8 bits

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; that is, A.D1+2 is the second dimension span.)

A.SIZB – Total array size in bytes, A.SIZB = D1*D2* . . . Dn*BPE; or, for virtual arrays, the number of elements in the array.

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

A.PLYV – Addressing polynomial evaluated for the first element of a virtual array, 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; that is, A.UN-2 is the last lower bound, A.UN-4 is the next-to-last upper bound, and so on.

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.CHAR = Character
A.HOLL = Hollerith

I/O transmissions also use these codes to denote the list item data type.

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

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

The compiled code uses dimension spans to determine the subscript value. The ADB retains the upper and lower bounds for each array. The bounds determine the size and shape of arrays.

11.2.1 Adjustable Array Initialization

There are four routines for initializing the contents of ADBs for dummy argument adjustable arrays: MAK1$ for one-dimensional arrays, MAK2$ for two-dimensional arrays, MAKN$ for arrays with three to seven dimensions, and MAKV$ for virtual arrays. Only R5 is preserved by these routines. They are called as follows:

1. Push the dimension bounds for any nonconstant 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 the address of the array descriptor block onto the stack.
4. Execute a call in the form of JSR PC, to one of the following routines: MAK1$, MAK2$, MAKN$, or MAKV$.
5. On return, the stack arguments are deleted.

11.2.2 Array Subscript Checking

If the compiler switch option /CK is in effect, each array reference is 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 the array element address onto the stack.
2. Push the address of the array descriptor block.
3. Execute a call in the form of JSR PC,ARYCK$.

This call preserves all registers.

11.2.3 Virtual Array Processing

Virtual array elements are processed by out-of-line calls in all cases. The OTS call returns the mapped virtual address of the array element. Either the value of the array element is loaded into a register for use or a value is stored into the array element.

11.2.3.1 Calling Process

To process the call, perform the following steps:

1. Push the address of the array descriptor block on the stack.
2. Move the indexing expression into R0.
3. Call the routine:

```
VRTx$,   if /-CK was specified
VRTxC$,  if /CK was specified
```

   where x is one of the following data type code letters:

   B - LOGICAL*1
   L - LOGICAL*2
   M - LOGICAL*4
   I - INTEGER*2
   J - INTEGER*4
   R - REAL*4
   D - REAL*8
   C - COMPLEX*8

4. On return, the stack argument is deleted, R0 contains the virtual address of the element, and all other registers are preserved.

11.2.3.2 Virtual Arrays in Separate I- and D-Space

When programming virtual arrays, you should carefully consider virtual array interaction with other elements. This is particularly important when your programs run in separate I- and D-space.

I- and D-space is an advanced programming technique that allows you to effectively double your virtual task space. Normally, 32K words can be associated with a task. With I- and D-space, this number increases to 64K words. (See Chapter 5 for more information on I- and D-space.)

I- and D-space increases your virtual task space, but it complicates relationships among virtual arrays and resident libraries, resident commons, and other shared regions. You must pay attention to these relationships, or you might experience failing tasks and unreliable results.

NOTE

Be careful if you use virtual arrays in an I- and D-space task linked to a resident library that contains data. This configuration is not supported on all operating systems.

The Task-Builder for RSX–11M/M-Plus Version 4.1 supports the mapping of virtual arrays and resident libraries in an I- and D-space task. Refer to the *RSX–11M/M-Plus Version 4.1 Release Notes* and the *RSX–11M/M–PLUS Task-Builder Manual* for more information on using resident libraries and virtual arrays in I- and D-space.

11.2.4 Notes on ADB Usage

The following defines the array-addressing polynomial function, polyA, for a three-dimensional array:

```
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 both of the following:

- Greater than or equal to the base address, A.ASTR
- Less than the high address+1, A.ASTR+A.SIZB

Only the complete subscript value is within the array; individual dimensions are *not* checked against their corresponding dimension bounds.

For example, the FORTRAN statements

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

cause the following ADBs to be created for I and A:

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

          .WORD       12   ;   L1  = 10
          .WORD        0   ;   U1  = N-1
          .WORD        1   ;   L2  = 1
          .WORD        0   ;   U2  = N
          .WORD    12000   ; 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
```

11.3 GO TO Statement Support

The following sections describe the code that results from compiling FORTRAN-77 GO TO statements.

11.3.1 Computed GO TO Statement Support

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

1. Push the address of the label list.
2. Convert the index expression value to INTEGER*2 (if needed) and push it on the stack.
3. Execute a call in the form of JSR PC,CGO$.
4. On return, the stack arguments are deleted.
5. If the index value is less than 1 or greater than the number of labels in the list, no transfer takes place and all registers are preserved.

11.3.2 Assigned GO TO Statement Support

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

1. Push the assigned label address.
2. Push the address of the allowed label list.
3. Execute a call in the form of JSR PC,AGO$.
4. On return, the stack arguments are deleted.
5. If the assigned label value is not in the list, no transfer takes place and all registers are preserved.

11.3.3 Label List Argument Format

The label list for the assigned or computed GO TO statement has the following form:

```
ADDR:   .WORD   n
        .WORD   label1
          .
          .
          .
        .WORD   labeln
```

11.4 Traceback Chain Processing

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

The work area contains the list head and the current statement number. 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:

```
$NAMC ->   pointer to next
           statement number
             program unit
             name in RAD50
```

The list head points to the currently active program unit entry. This entry contains the following items:

- The currently active program unit name in Radix–50
- The current statement number in the calling program at the time of the call
- A pointer to the calling program list block

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.

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

1. Push the last three letters of the entry name (represented in Radix–50) onto the stack.
2. Load the first three letters of the entry name into register R4.
3. Execute a call in the form of JSR R4,@NAM.

The current statement number, $SEQC, is set to zero. The traceback information is maintained on the execution stack. When the program unit returns, it returns to the NAM$ routine, which resets the stack, removes the name chain link, and returns control to the caller.

If /TR:NAMES is specified, the current statement number is not updated ($SEQC remains zero).

If /TR:BLOCKS is specified, the current statement number is periodically updated by the compiler to contain the negative of the statement number, for instance, -21 for statement 21.

If /TR:LINES is specified, the current statement number is updated on every statement, maintaining a positive number.

Chapter 12

OTS System Generation and Tailoring

The *PDP-11 FORTRAN-77 Installation Guide* describes how the OTS is built during the installation process. This chapter describes options you can choose when building FORTRAN-77 into your system. It gives a more detailed explanation of the installation options, as well as information on building the OTS from sources. This chapter also includes factors affecting compiler performance and information about optional OTS modules that you can use to tailor FORTRAN-77 to your particular applications.

12.1 Assembly Options

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

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

12.1.1 Operating System Options

The three operating system option symbols are RSXD for IAS, RSXM for RSX-11M/M-PLUS and RSTS/E, and RSXS for RSX-11S. 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/M-PLUS.

12.1.2 File System Options

The two versions of I/O systems are maintained as separate sources, but three assembly options are maintained. The assembly options are:

- FCS specifies FCS-11 version.
- RMS specifies RMS-11 version.
- Nothing specifies the RSX-11S subset.

12.1.3 EIS Instruction Set Option

The two hardware options are defined by the symbol FPP. If FPP is not defined, then you can use the OTS on a PDP-11/45 or -11/40 with EIS, provided no floating point computations are attempted.

The modules affected are:

$MLJ —INTEGER*4 multiplication
$DVJ —INTEGER*4 division
$JMOD —INTEGER*4 modulo
$FPPUTI—FPP save/restore and initialization

12.1.4 Special Assembly Options

The following sections describe the two special assembly options.

12.1.4.1 Double-Precision Arithmetic Option

The symbol F77DP is used to assemble certain mathematical functions in double-precision mode.

The modules affected are:

$ASIN—arc sine
$ACOS—arc cosine
$TAN —tangent

12.1.4.2 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

12.2 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 F77.MAC:

- OTSWA Macro—Defines the work area offsets (see Appendix A).
- ERRDEF Macro—Defines the OTS errors, error control byte control bits, and error message text.
- $AOTS Macro—Obtains the impure area pointer from location $OTSV and places it in a register, usually R3.
- OTS$I Macro—Defines the OTS code PSECT $$OTSI.
- OTS$D Macro—Defines the OTS pure area PSECT $$OTSD.
- ADBDEF Macro—Defines the array descriptor block offsets and the data type codes. It is found in the parameter file ADBDEF.MAC.

- FBLOCK Macro—Defines the LUB control block offsets for the FCS or RMS versions of the I/O system. It is defined in the parameter files FCS and RMS.

12.3 Options Affecting Compile-Time Performance

The following two options affect compile-time performance:

- You can choose one, two, or three temporary disk files for the compiler to use to store information during the compilation process.
- You can change the size of the dynamic storage area in the compiler.

The PDP–11 FORTRAN–77 compiler uses temporary disk files for storing information during the compilation process. The compiler requires at least one temporary file, called the work file.

The work file contains information that the compiler normally accesses at random (for example, the symbol table and the constants table). The dynamic storage area within the compiler is used to manipulate this information. (Only part of the work file is in memory at any given time. Software paging techniques move information back and forth between the dynamic storage area and the work file.)

Information must be moved into the dynamic storage area when needed by the compiler. Increasing the size of the dynamic storage area increases compilation speed by reducing the number of disk I/O operations (see Section 12.3.2).

12.3.1 Number of Temporary Files

The /WF:w compiler switch specifies the number of temporary disk files that are available to the compiler. If you specify /WF:1, the compiler stores internal representations in just the work file. However, if you specify /WF:2 (or /WF:3), the compiler stores some (or all) of these representations in the one or two other temporary files. The /WF:2 option is the default.

Using additional temporary files slows the compilation process, but it significantly increases the capacity of the compiler. For instance, with three temporary files (/WF:3), the compiler can compile a program that is approximately three times larger than any it can compile with only one temporary file (/WF:1). No significant change occurs in the compilation

rate if you place the temporary files on a fixed-head disk, because these files are written and read sequentially.

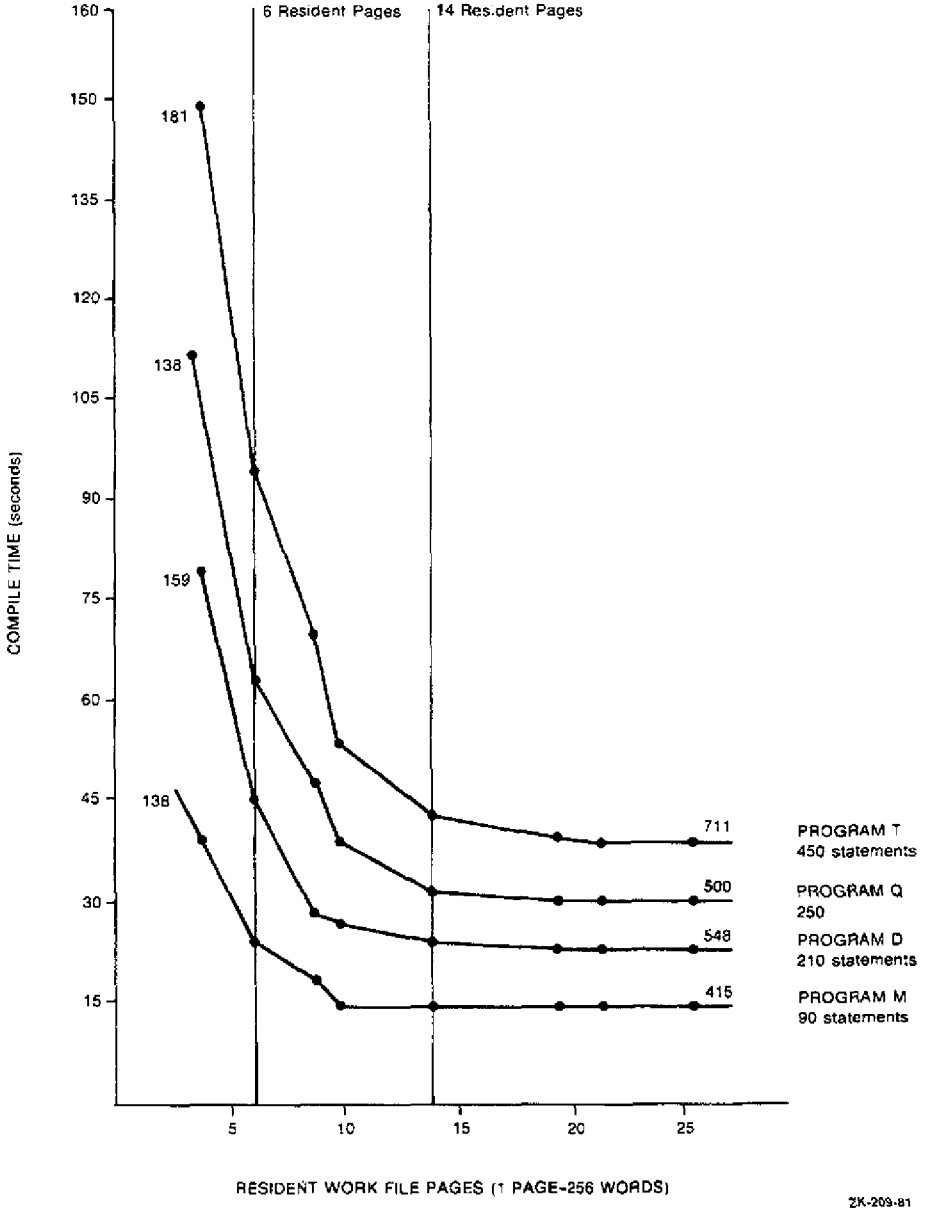
12.3.2 Size of the Dynamic Storage Area

Increasing the size of the dynamic storage area increases the rate of compilation. Figure 12–1 illustrates the correlation between compile time and the size of the dynamic storage area. The compile time of four different FORTRAN programs, varying in length from 90 to 450 statements, was measured on a PDP–11/60. The compiler used two temporary files (/WF:2), with the work file residing on the system moving-head disk (RP04). The dynamic storage area varied in size from 4 to 26 pages (when the listing file was suppressed).

The measurements at the end points of each curve denote the approximate compilation rate measured in statements compiled per minute. Continuation and comment lines were not counted.

Figure 12–1 shows that compilation speed is approximately three times greater when 26 pages of dynamic storage are used than it is when only 4 pages are used. However, using 14 pages results in optimal compiler performance. Building the compiler with more than 14 pages of dynamic storage achieves minimal improvement in the rate of compilation. The default size of the dynamic storage area is 12 pages.

Figure 12–1: Compiler Performance



12.3.2.1 Operating Systems Supporting Dynamic Memory Allocation

Under a RSTS/E, RSX–11M/M–PLUS, or RSX–11M system with dynamic memory allocation, you specify the size of the PDP–11 FORTRAN–77 compiler's dynamic storage area by using the EXTTSK option in the task-build command file. The value specified by EXTTSK is the size of the dynamic storage area in decimal words. The size of the dynamic storage area is computed as follows:

```
256*(n+w+1)
```

n
The number of pages for the dynamic storage area.

w
The value specified in the /WF:w switch.

You can override the dynamic storage area specified by EXTTSK at installation by means of the INC switch on the INSTALL (INS) command. The task extension size is specified in decimal words.

The following table shows the correlation between the compiler task size, the EXTTSK value, and the number of pages for the dynamic storage area under /WF:2.

Number of Pages	EXTTSK — Value INS/INC Value	Size of Compiler Task (Words)
4	1792	22K
8	2816	23K
12	3840	24K
16	4864	25K

For RSX–11M/M–PLUS installations whose default is the ANSI magnetic tape version of FCS-11 (LB:[1,1]ANSLIB.OLB), the compiler task size increases by approximately 500 words.

12.3.2.2 RSX–11M Without Dynamic Memory Allocation

On an RSX–11M system without dynamic memory allocation, the PDP–11 FORTRAN–77 compiler determines the size of the partition in which it is operating and uses all of the memory in that partition. Install the compiler in a partition large enough for the compiler to run with the desired number of pages of dynamic storage.

12.4 OTS Options

The distribution kit includes a number of optional OTS modules. After building the OTS library, you can add one or more of these optional modules to the library, or you can maintain these modules separately and refer to them only as needed. The installation procedures copy these modules to LB:[1,1] (or, on RSTS/E systems, to LB:). The PDP-11 FORTRAN-77 system does not require any of the optional modules for normal use. The following sections describe these modules.

12.4.1 F7711S

Module F7711S.OBS consists of a set of concatenated object modules containing alternate versions of FORTRAN sequential I/O support modules. These I/O support modules, designed for use with RSX-11S, provide sequential I/O to non-file-structured devices (for example, terminals, non-spooled card readers, and line printers). These modules do not use the file system but perform direct I/O operations; they reduce task size by approximately 2500 words.

You can use F7711S.OBS in two ways:

- You can include it as an object module at task-build time, as follows:

```
TKB>MAIN/FP=MAIN,LB:[1,1]F7711S.OBS
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

- You can build a separate F77 OTS library for RSX-11S use, LB:[1,1]F7711S.OLB, in addition to the host operating system's OTS library. To do this, when building the OTS, substitute module LB:[1,1]F7711S.OBS (or, for RSTS/E, LB:F7711S.OBS) for the file system module you selected in the installation procedure. For example, replace the reference to FCS11M.OBS with F7711S.OBS. Use this OTS library, rather than the host operating system's OTS library, when building tasks for RSX-11S, as follows:

```
TKB>MAIN/FP=MAIN,LB:[1,1]F7711S/LB
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

12.4.2 Short Error Text — RSX-11M/M-PLUS and RSTS/E Only

For error messages, the PDP-11 FORTRAN-77 OTS references an error-text module containing ASCII text. If your operating system is RSX-11M, RSX-11M/M-PLUS, or RSTS/E, you can use a long or a short error-text module. The long error-text module requires approximately 1000 words of memory, but the alternate version (SHORT.OBJ) requires only one word of memory.

A task with the short error-text module built into it generates complete error reports, but omits the one-line description of the error condition. The *PDP-11 FORTRAN-77 User's Guide* contains a complete list of OTS error numbers and message text. The F77 OTS uses the long error-text module by default. You can build a task using the short error-text module by loading module $SHORT from the library.

12.4.3 F77MAP

Module F77MAP.OBS consists of a set of concatenated object modules that you can use to transform intrinsic function names into internal names at task-build time. (The PDP-11 FORTRAN-77 compiler transforms intrinsic function names into internal names at compile time.)

Without F77MAP.OBS, if a program written in MACRO-11 attempts to reference a PDP-11 FORTRAN-77 intrinsic function with the FORTRAN name of the function instead of the internal name, an unresolved reference will occur during task build. For example, F77MAP.OBS maps the FORTRAN name SIN using the following module:

```
        .TITLE   $MSIN
SIN::   JMP      $SIN
        .END
```

F77MAP.OBS contains an object module similar to the preceding module for each of the PDP-11 FORTRAN-77 intrinsic functions.

You can build an F77MAP library as follows:

```
LBR> LB:[1,1]F77MAP.OLB/CR:40.=LB:[1,1]F77MAP.OBS
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

12.4.4 F77EIS

Module F77EIS.OBS consists of a set of concatenated object modules that contain extended instruction set (EIS) versions of certain integer functions that normally use a floating-point processor. This module allows FORTRAN programs that do not perform floating-point arithmetic to run on a machine that has the extended instruction set but not a floating-point processor. The modules provided in the F77 OTS use a floating-point processor for maximum efficiency in certain INTEGER*4 computations.

Use one of the following commands at task-build time to replace the normal modules in file INT with their EIS versions:

For FCS OTS

```
TKB> INT/-FP=INT,LB:[1,1]F77EIS.OBS,
                LB:[1,1]F77FCS/LB
```

For RMS OTS

```
TKB> INT/-FP=INT,LB:[1,1]F77EIS.OBS,
                LB:[1,1]F77RMS/LB,
                LB:[1,1]RMSLIB/LB
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

You may, instead, substitute the F77EIS module for the default conversion module as follows:

For FCS OTS

```
LBR>LB:[1,1]F77FCS/RP=LB:[1,1]F77EIS
```

For RMS OTS

```
LBR>LB:[1,1]F77RMS/RP=LB:[1,1]F77EIS
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

No changes in the Task Builder command line are necessary if you perform this substitution.

This module cannot be used with optional OTS module F77CVF.

12.4.5 F77CVF

Module F77CVF.OBJ is an alternative module for performing formatted output of floating-point values under control of the D, E, F, and G field specifiers. The standard module provided as part of the F77 OTS uses multiple-precision, fixed-point integer techniques to maintain maximum accuracy during the conversion of data (FPP hardware is not used). The alternative module performs the same functions using the FPP hardware. It is approximately twice as fast as, but in some cases slightly less accurate than, the standard module. Use one of the following commands at task-build time to replace the normal modules in file OUTR with their F77CVF FPP versions:

For FCS OTS

```
TKB> OUTR=OUTR,LB:[1,1]F77CVF,LB:[1,1]F77FCS/LB
```

For RMS OTS

```
TKB> OUTR=OUTR,LB:[1,1]F77CVF,LB:[1,1]F77RMS/LB,
        LB:[1,1]RMSLIB/LB
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

You may substitute the F77CVF module for the default conversion module as follows:

For FCS OTS

```
LBR> LB:[1,1]F77FCS/RP=LB:[1,1]F77CVF
```

For RMS OTS

```
LBR> LB:[1,1]F77RMS/RP=LB:[1,1]F77CVF
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

No changes in the Task Builder command line are necessary if you perform this substitution.

This module cannot be used with optional OTS module F77EIS.

12.4.6 F77NER

Module F77NER.OBS consists of a set of concatenated object modules for reporting run-time errors. Using this module suppresses the error-message text report. However, error processing and calls to ERRSET, ERRSNS, and ERRTST continue to operate normally; only the logging of the message on the user's terminal is suppressed. The STOP and PAUSE statement messages are also suppressed. F77NER.OBJ reduces task size by approximately 375 words less than the standard module.

If you use F77NER with optional OTS modules F7711S or F77NIO, a multiply-defined symbol error may result during task-build. Two correct ways to use F77NER with F7711S or F77NIO are:

- Build F7711S (or F77NIO) and F77NER as separate libraries and use them as follows:

```
TKB> MAIN/FP=MAIN,LB:[1,1]F77NER/LB:$NERRL,
            LB:[1,1]F7711S/LB,LB:[1,1]F77FCS/LB
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

- Build an OTS by incorporating F7711S.OBS instead of FCS11M.OBS into F77FCS. Name the resulting library F7711S.OLB, and build F77NER as a separate library. Use those libraries as follows:

```
TKB> MAIN/FP=MAIN,LB:[1,1]F77NER/LB:$NERRL,
            LB:[1,1]F7711S/LB
```

NOTE

On RSTS/E systems, replace LB:[1,1] with LB:.

12.4.7 F77NIO

Module F77NIO.OBS consists of a set of concatenated object modules containing alternative versions of certain OTS routines that are always present in the user task and that provide support for FORTRAN–77 I/O operations. The alternate routines in F77NIO.OBS do not support FORTRAN–77 I/O and reduce task size by approximately 1000 words for programs not requiring FORTRAN–77 I/O (such as process control).

12.4.8 F77RAN

Module F77RAN.OBS consists of a set of concatenated object modules that contain an alternative random-number generator that is compatible with previous releases of PDP-11 FORTRAN. If you require this random-number generator for compatibility purposes, include file LB:[1,1]F77RAN.OBS (for RSTS/E systems, LB:F77RAN.OBS) at task-build time.

12.4.9 OTS Overlay Description Files

The two OTS overlay description files are:

FCS11M.ODL — FCS-11 support for RSX-11M/M-PLUS and RSTS/E
RMS11M.ODL — RMS support for RSX-11M/M-PLUS and RSTS/E

Each file is an ODL fragment file that you can use for overlaying the PDP-11 FORTRAN-77 OTS modules. Each file contains documentation describing OTS options and the procedures for using the file.

NOTE

If you are using the RSTS/E system, files FCS11M.ODL and RMS11M.ODL contain references to LB:[1,1]. Change all occurrences of LB:[1,1] to LB:.

12.4.10 OTS Modules Chart

The following chart lists the optional OTS modules and indicates whether they require a floating-point processor (FPP) and whether they can replace the standard OTS modules in your OTS library.

Module	Requires FPP	Can Replace Modules in Standard OTS
F7711S.OBS	NO	YES *
F77MAP.OBS	NO **	NO
F77EIS.OBS	NO ***	YES *
F77CVF.OBJ	YES	YES
F77NER.OBJ	NO	YES
F77NIO.OBS	NO	YES *
F77RAN.OBS	NO **	YES
FCS11M.ODL	NO	NO
RMS11M.ODL	NO	NO
SHORT.OBJ	NO	---

* - Use this module instead of FCS11M to build the OTS.

** - This module does not require an FPP, but a program using the results might.

*** - This module is for processors without an FPP and cannot be used with optional module F77CVF.OBJ.

Chapter 13

OTS Resident Libraries

This chapter describes how to create OTS resident libraries. First, it discusses factors that influence your choice of library. (Note in particular the new vectored default.) Then, it provides instructions for building the resident library of your choice.

A resident library has the following characteristics:

- It resides in memory and must be installed before a task that references it can be installed or run.
- It can be shared by multiple tasks. However, it occupies virtual address space in each task to which it is linked.

The FORTRAN-77 OTS has the following general limitations:

- It does not contain position-independent code (PIC) and, so it cannot be built into a PIC resident library.
- It cannot be built into a supervisor-mode library. (However, it can be linked to the FCSFSL supervisor-mode library; see Section 13.1.1.)

For more information on resident libraries, refer to the Task Builder manual for your operating system.

13.1 Types of Resident Libraries

There are two general types of resident libraries: noncluster and cluster. Within each of these two types, you can employ various schemes of organizing your libraries, and you can choose either File Control Services (FCS) or Record Management Services (RMS). The following sections describe the various combinations possible and some of the considerations involved in choosing a resident library organization.

13.1.1 Noncluster Libraries

In a simple noncluster library, all of the library code takes up virtual address space in the task. The size of the library (and the amount of address space it requires) consists of the total amount of space required by OTS routines and the file system (typically FCS) routines. A resident library of this type may be faster than an equivalent cluster library organization, but it also takes up much more space. You must use this organization, but, if you have an RSX system that contains no support for memory-mapping directives or for supervisor-mode libraries. (Section 13.4.2.2 contains a sample command file to build a nonclustered FCS library.)

For RSX-11M/M-PLUS or Micro/RSX systems that support supervisor-mode libraries, you can link the OTS resident library with the supervisor-mode FCS library (FCSFSL). With this organization, only the OTS library takes up virtual address space in your task. When available, this configuration is recommended because it is slightly faster than the equivalent cluster organization. However, you cannot link the OTS resident library with the RMS supervisor-mode library. (Section 13.4.2.3 describes how to build a nonclustered library linked to the supervisor-mode FCS library.)

13.1.2 Cluster Libraries

Cluster libraries are sets of two or more resident libraries that share the same portion of virtual memory. Conceptually, cluster libraries are like memory-resident overlays; the two or more cluster libraries form a single memory-resident overlay tree in your task's virtual address space.

When your task is linked with cluster libraries, only one of the two or more libraries is mapped by your task at one time. Therefore, the amount of virtual address space dedicated to libraries equals the largest of the cluster libraries, rather than their total. When a call is made to a routine

in a library other than the one currently mapped, the task automatically remaps to the new library. This process incurs some overhead; cluster libraries are slower than their noncluster or supervisor-mode counterparts.

If your task uses RMS, a cluster library organization is recommended. Using this scheme, you can include all of RMS and a great deal of the FORTRAN–77 OTS in 8K words of your task's virtual address space. (Section 13.4.2.4 shows how to build a resident library clustered with RMSRES.)

If your task uses FCS, you should use a cluster library organization for these same reasons, unless your system supports supervisor-mode libraries. The supervisor-mode library (see Section 13.1.1) gains for your task the same advantages as those of a cluster library and is faster. (Section 13.4.2.5 shows how to build a resident library clustered with FCSRES.)

13.2 Support for Resident Libraries

The chart below indicates the support available for the types of OTS libraries described in Section 13.1. Unless noted, support is extended regardless of operating system.

	FCS OTS	RMS OTS
NONCLUSTER LIBRARY	SUPPORTED	NOT SUPPORTED
CLUSTER LIBRARY	SUPPORTED	SUPPORTED
LINKED AGAINST SUPERVISOR-MODE LIBRARY	SUPPORTED*	NOT SUPPORTED

* This combination is not supported on the RSTS/E operating system.

13.3 Vectored Resident Libraries

No matter which type of resident library you favor for your task, it will be a vectored library. A vectored resident library references a fixed entry point to an OTS routine, not the address to an OTS routine.

This means that tasks built using a vectored resident library do not need to be rebuilt if routines in the resident library change. When modifications to the OTS resident library routines occur, the OTS routine addresses may shift, but the vectored entry pointers in the resident library do not.

It also means that resident libraries must remain the same size (see Section 13.4.2.1 for information about optimal sizes). A task linked against a resident library of one size fails when run against a resident library of another size.

13.4 Creating an OTS Resident Library

After you identify the type of OTS resident-library needed, you must choose between a default or a tailored version of this library:

- A default version offers you ease and speed in building, as well as a standard library size (8K-words).
- A tailored version offers you your choice of library modules and some choice in library size (a 4K-word multiple).

Section 13.4.1 provides instructions for building a default library; Section 13.4.2 provides instructions for building a tailored library. Section 13.4.2 provides Task Builder command file examples for each type of resident library; these examples will build appropriate libraries in most situations.

NOTE

The command files listed in this chapter do not link the two OTS error message modules, $ERTXT and $SHORT, into the resident library. You may include one of these modules when building a library if you wish to force long or short error-message text to be used by programs that link to that library.

13.4.1 The Default Library

Your FORTRAN–77 installation kit contains the command files identified in the following list. You build a default OTS resident library by invoking the command file corresponding to the type of library you desire.

Command File Name	Type of OTS Resident Library Built
F7FRES.BLD	Noncluster with FCS Routines Included
F7SRES.BLD	Noncluster Linked to FCSFSL
F7RCLS.BLD	Cluster with RMSRES
F7FCLS.BLD	Cluster with FCSRES

These command files perform the following steps:

1. Compile the vector module twice, once to create F77VEC (without OTS routine pointers) and once to create F77REC (with OTS routine pointers).
2. Replace the .OLB vector module in the OTS resident library with F77VEC.
3. Build an 8K-word resident library.
4. Install the library in a partition.

These steps make a default OTS resident library available to you.

13.4.2 The Tailored Library

Your FORTRAN–77 installation kit contains the MACRO–11 files identified in the following list. You can build a tailored OTS resident library by editing the MACRO–11 file corresponding to the type of library you desire.

MACRO-11 File	Type of OTS Resident Library Built
F7FRES.MAC	Noncluster with FCS routines included
F7SRES.MAC	Noncluster linked to FCSFSL
F7RCLS.MAC	Cluster with RMSRES
F7FCLS.MAC	Cluster with FCSRES

Section 13.4.2.1 provides more information on editing these files.

Depending on the degree of tailoring desired, you can then invoke the appropriate command file (see Section 13.4.1), or you can issue commands and create command files of your own as follows:

1. Compile the MACRO-11 file you just edited. Use .OBJ as the file extension in your output file.
2. Use a text editor to create an appropriate Task Builder command file to build your resident library. (The command files in Sections 13.4.2.2 through 13.4.2.5 should work as shown or with slight modifications.)

   Make sure the number in your EXTSCT=$$OTSI:n statement is correct; it should extend your task's size to 8K.

   Make sure that the OTS object module library you specify contains file system modules (FCS or RMS) that match the file system you intend to use.
3. Invoke the Task Builder and pass to it the command file you just created.
4. Inspect the map file resulting from the task-build. If the resident library is too large or is not large enough, edit (or re-edit) your MACRO-11 file and repeat the steps above.
5. Purge any task, map, or STB files resulting from previous task-builds.
6. Install the library in memory, following the instructions in the documentation for your particular operating system.

These steps make a tailored OTS resident library available to you.

13.4.2.1 Editing the MACRO–11 File

The four MACRO–11 files (F7FRES.MAC, F7SRES.MAC, F7FCCLS.MAC, and F7RCLS.MAC) contain global references to OTS entry points. The modules referenced in the MACRO–11 file you choose will make up your OTS resident library. You can edit these files to include modules that your tasks use frequently, or to exclude modules that are used infrequently. (Editing instructions are included in the file.) You can also use each file as is.

If you edit one of these files, you should create an OTS resident library that balances the requirements of size and functionality. If your library is large, the virtual address space available for your task may be small. (A library size of 8K words is recommended.) The Task Builder places in your task the object code for any modules it references that are not in the OTS resident library so it does not make sense to exclude commonly used modules from the library.

To make the best use of available virtual memory, the OTS resident library should be nearly equal to, but slightly below, a multiple of 4K words. Each time the size of the library exceeds a 4K multiple, an additional APR is required; this has the effect of reducing the virtual address space available to the task by 4K words. The following table illustrates this relationship:

Size in Words	Number of APRs	Size in Octal Bytes
4096	1	20000
8192	2	40000
12288	3	60000
16384	4	100000

In the case of an OTS resident library to be clustered with a file system library, there is an additional consideration. Remember that with clustered libraries, the virtual address space occupied by the libraries is equal to the size of the largest of the libraries. For example, if your OTS resident library occupies 4K words and the file system library occupies 8K words, the libraries occupy a full 8K words of virtual address space. In this situation, there is no advantage to limiting the size of your OTS resident library to 4K words; use the full 8K words for a richer library.

The file system cluster libraries occupy space as follows:

File System	Library Name	Number of APRs
FCS	FCSRES	1
RMS	RMSRES	2

When you create the command file to build the OTS resident library, include a PAR option as follows:

```
PAR=pname:base:length
```

The pname is the partition name; it must be the same as the name of the resident library. For RSX–11M systems, this partition must exist in the system; for RSX–11M/M–PLUS, the INSTALL command places the library in another partition if the specified partition does not exist. The values you supply for base and length depend on the number of APRs that the resident library occupies, as follows:

Number of APRs	Base	Length
1	160000	20000
2	140000	40000
3	120000	60000
4	100000	100000

When you use these values, the OTS resident library occupies the highest virtual addresses of your task.

13.4.2.2 Building a Noncluster Library with FCS Routines

The command files in this section build 8K-word OTS resident libraries that include FCS modules. You can tailor your library by modifying F7FRES.MAC.

RSX Systems

The following command file builds an 8K-word OTS resident library (F7FRES.TSK) that includes FCS modules referenced by the OTS:

```
F7FRES/-HD/LI/-PI,F7FRES/-SP/MA,F7FRES=F77RES
LB:[1,1]F77FCS/LB
/
STACK=0
UNITS=0
PAR=F7FRES:140000:40000
EXTSCT=$$OTSI:1000
@LB:[1,1]F77GBL.XCL
//
```

The number 1000 in the EXTSCT statement is a variable, not a constant. It represents the difference between actual library size and the recommended 8K library size.

Note that the FORTRAN–77 OTS library (LB:[1,1]F77FCS.OLB) is referenced in this command file. This library is the FCS version of the FORTRAN–77 OTS.

When you link a task to this library, include LB:[1,1]F77FCS/LB:[1,1]F77VEC in the task's root and use the following Task Builder option:

```
LIBR=F7FRES:RO
```

RSTS/E Systems

The following command file builds an 8K-word OTS resident library (F7FRES.TSK) that includes FCS modules referenced by the OTS:

```
F7FRES/-HD/LI/-PI,F7FRES/-SP/MA,F7FRES=F77RES
LB:F77FCS/LB
/
STACK=0
UNITS=0
PAR=F7FRES:140000:40000
EXTSCT=$$OTSI:1000
@LB:F77GBL.XCL
//
```

The number 1000 in the EXTSCT statement is a variable, not a constant. It represents the difference between actual library size and the recommended 8K library size.

Note that the FORTRAN–77 OTS library (LB:F77FCS.OLB) is referenced in this command file. This library is the FCS version of the FORTRAN–77 OTS.

When you link a task to this library, include LB:F77FCS/LB:F77VEC in the task's root and use the following Task Builder option:

```
LIBR=F7FRES:RO
```

13.4.2.3 Building a Noncluster Library Linked to FCSFSL

The following command file builds an OTS resident library (F7SRES.TSK) for which FCS routines reside in a separate supervisor-mode library. As supplied, F7SRES.MAC builds a 8K-word library to link to FCSFSL. You can tailor your library by modifying F7SRES.MAC.

You can use this configuration only on RSX-11M/M-PLUS systems that support supervisor-mode libraries.

```
F7SRES/-HD/LI/-PI,F7SRES/-SP/MA,F7SRES=F77RES
LB:[1,1]F77FCS/LB
/
STACK=0
UNITS=0
SUPLIB=FCSFSL:SV
PAR=F7SRES:140000:40000
GBLDEF=.FSRCA:0
GBLXCL=.FSRCA
EXTSCT=$$OTSI:1000
@LB:[1,1]F77GBL.XCL
//
```

Note that the number 1000 in the EXTSCT statement is a variable, not a constant. It represents the difference between actual library size and the recommended 8K library size.

Note also that the FORTRAN-77 OTS library (LB:[1,1]F77FCS.OLB) is referenced in this command file. This library is the FCS version of the FORTRAN-77 OTS.

When you link a task to this library, make sure that you include LB:[1,1]F77FCS/LB:F77VEC in the task's root. Then, use the following Task Builder option:

```
LIBR=F7SRES:RO
```

13.4.2.4 Building a Library to Cluster with RMSRES

The command files in this section build 8K-word OTS resident libraries that cluster with RMSRES, the RMS file system resident library. You can tailor your library by modifying F7RCLS.MAC

RSX Systems

The following command file builds an 8K-word OTS resident library (F7RCLS.TSK) that clusters with RMSRES:

```
F7RCLS/-HD/LI/-PI,F7RCLS/-SP/MA,F7RCLS=F77RES
LB:[1,1]F77RMS/LB
LB:[1,1]RMSLIB/LB:RORMSC
/
STACK=0
UNITS=0
PAR=F7RCLS:140000:40000
GBLXCL=.SAVR1
EXTSCT=$$OTSI:10000
@LB:[1,1]F77GBL.XCL
//
```

The number 1000 in the EXTSCT statement is a variable, not a constant. It represents the difference between actual library size and the recommended 8K library size.

Note that the FORTRAN–77 OTS library (LB:[1,1]F77RMS.OLB) is referenced in this command file. This library is the RMS version of the FORTRAN–77 OTS.

When you link a task to this library, include the following modules in the task's root:

```
LB:[1,1]F77RMS/LB:F77VEC
LB:[1,1]RMSLIB/LB:ROAUTL:ROIMPA:ROEXSY:RMSSYM
```

Then, use the following Task Builder option:

```
CLSTR=F7RCLS,RMSRES:RO
```

RSTS/E Systems

The following command file builds an 8K-word OTS resident library (F7RCLS.TSK) that clusters with RMSRES:

```
F7RCLS/-HD/LI/-PI,F7RCLS/-SP/MA,F7RCLS=F77RES
LB:F77RMS/LB
LB:RMSLIB/LB:RORMSC
/
STACK=0
UNITS=0
PAR=F7RCLS:140000:40000
GBLXCL=.SAVR1
EXTSCT=$$OTSI:1000
@LB:F77GBL.XCL
//
```

The number 1000 in the EXTSCT statement is a variable, not a constant. It represents the difference between actual library size and the recommended 8K library size.

Note that the FORTRAN–77 OTS library (LB:F77RMS.OLB) is referenced in this command file. This library is the RMS version of the FORTRAN–77 OTS. When you link a task to this library, include the following modules in the task's root:

```
LB:F77RMS/LB:F77VEC
LB:RMSLIB/LB:ROAUTL:ROIMPA:ROEXSY:RMSSYM
```

Then, use the following Task Builder option:

```
CLSTR=F7RCLS,RMSRES:RO
```

13.4.2.5 Building a Library to Cluster with FCSRES

The following command file builds an 8K-word OTS resident library (F7FCLS.TSK) that clusters with FCSRES, the FCS file system resident library. As is, F7FCLS.MAC builds a 8K-word library to cluster with FCSRES. You can tailor your library by modifying F7FCLS.MAC.

This command file is applicable to RSX systems only.

```
F7FCLS/-HD/LI/-PI,F7FCLS/-SP/MA,F7FCLS=F77RES
LB:[1,1]F77FCS/LB
LB:[1,1]SYSLIB/LB:FCSVEC
/
STACK=0
UNITS=0
PAR=F7FCLS:140000:40000
EXTSCT=$$OTSI:1000
@LB:[1,1]F77GBL.XCL
GBLINC=.FCSJT
GBLXCL=.ASLUN
GBLXCL=.CLOSE
```

```
GBLXCL=.CSI1
GBLXCL=.CSI2
GBLXCL=.CSI4
GBLXCL=.DELET
GBLXCL=.DLFNB
GBLXCL=.ENTER
GBLXCL=.EXPLG
GBLXCL=.EXTND
GBLXCL=.FCTYP
GBLXCL=.FIND
GBLXCL=.FINIT
GBLXCL=.FLUSH
GBLXCL=.GET
GBLXCL=.GETSQ
GBLXCL=.GTDID
GBLXCL=.GTDIR
GBLXCL=.MARK
GBLXCL=.MRKDL
GBLXCL=.OPEN
GBLXCL=.OPFID
GBLXCL=.OPFNB
GBLXCL=.PARSE
GBLXCL=.POINT
GBLXCL=.POSIT
GBLXCL=.POSRC
GBLXCL=.PRINT
GBLXCL=.PRSDI
GBLXCL=.PRSDV
GBLXCL=.PRSFN
GBLXCL=.PUT
GBLXCL=.PUTSQ
GBLXCL=.READ
GBLXCL=.REMOV
GBLXCL=.RENAM
GBLXCL=.SAVR1
GBLXCL=.TRNCL
GBLXCL=.WAIT
GBLXCL=.WRITE
//
```

The number 1000 in the EXTSCT statement is a variable, not a constant. It represents the difference between actual library size and the recommended 8K library size.

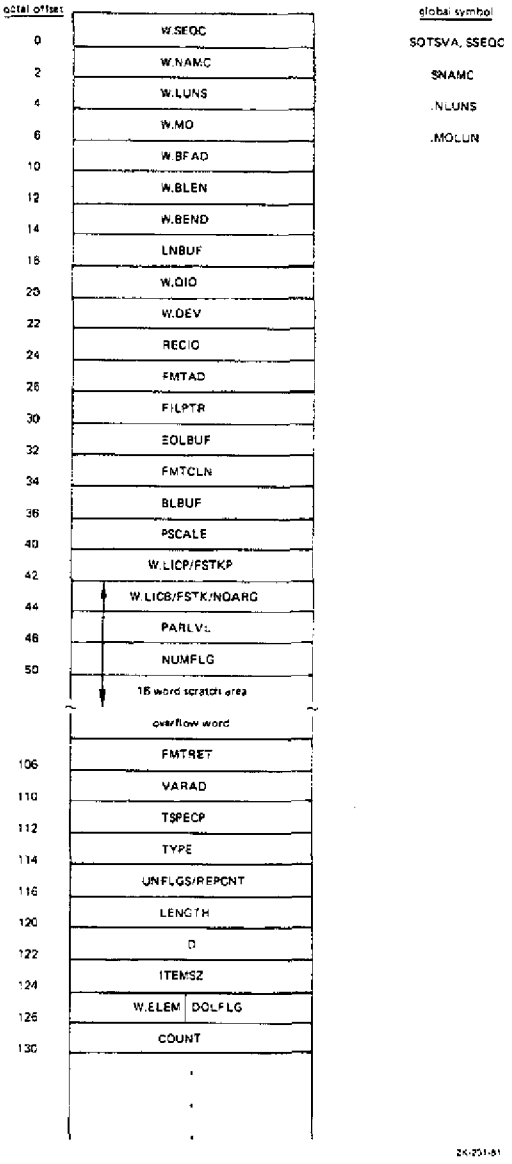
Note that the FORTRAN–77 OTS library (LB:[1,1]F77FCS.OLB) is referenced in this command file. This library is the FCS version of the FORTRAN–77 OTS.

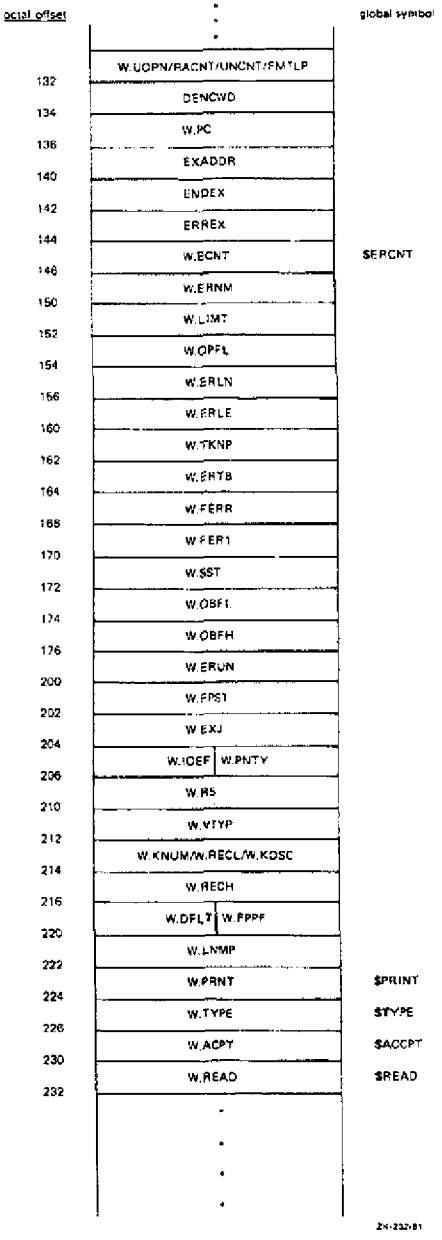
When you link a task to this library, include LB:[1,1]F77FCS/LB:F77VEC in the task's root and then use the following Task Builder option:

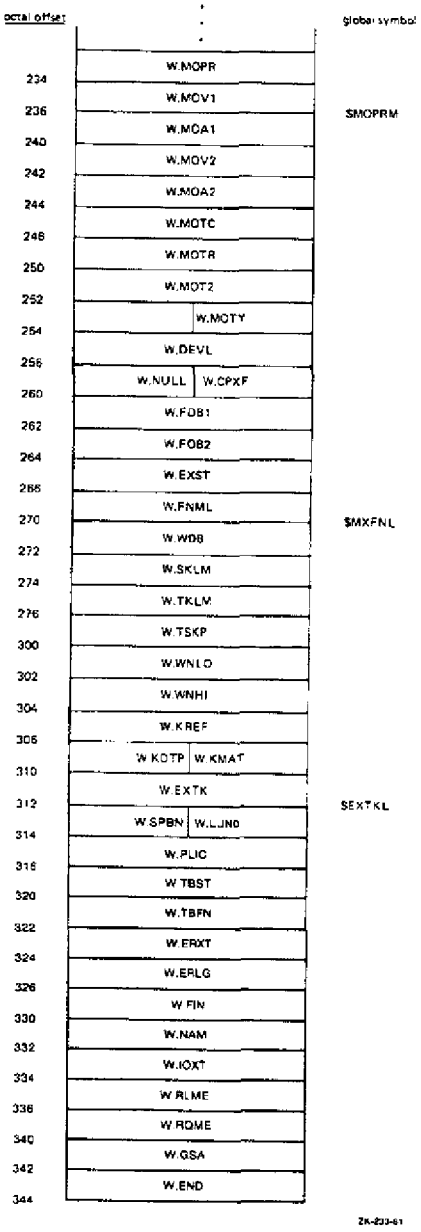
```
CLSTR=F7FCLS,FCSRES:RO
```

Appendix A

FORTRAN Impure Area Definitions


octal offset
global symbol
0
W.SEQC
SOTSVA, SSEQC
2
W.NAMC
$NAMC
4
W.LUNS
.NLUNS
6
W.MO
.MOLUN
10
W.BFAD
12
W.BLEN
14
W.BEND
16
LNBUF
20
W.QIO
22
W.DEV
24
RECIO
26
FMTAD
30
FILPTR
32
EOLBUF
34
FMTCLN
36
BLBUF
40
PSCALE
42
W.LICP/FSTKP
44
W.LICB/FSTK/NOARG
46
PARLVL
50
NUMFLG
16 word scratch area
overflow word
106
FMTRET
110
VARAD
112
TSPECP
114
TYPE
116
UNFLGS/REPCNT
120
LENGTH
122
0
124
ITEMSZ
126
W.ELEM
DOLFLG
130
COUNT
ZK-231-81

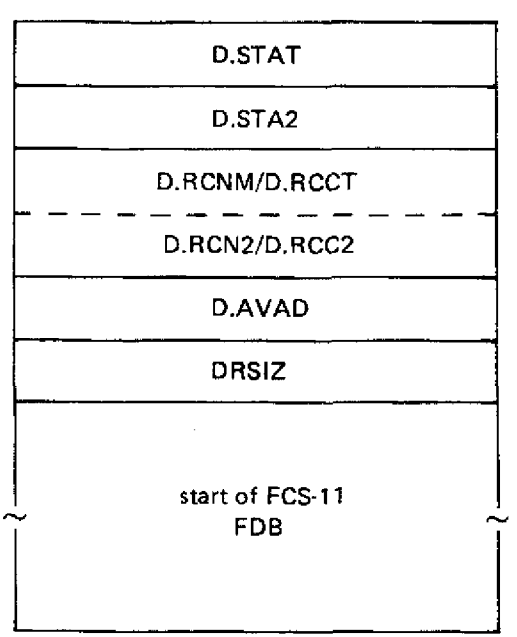

octal offset
global symbol
W.UOPN/RACNT/UNCNT/FMTLP
132
DENCWD
134
W.PC
136
EXADDR
140
INDEX
142
ERREX
144
W.ECNT
$ERCNT
146
W.ERNM
150
W.LIMT
152
W.OPFL
154
W.ERLN
156
W.ERLE
160
W.TKNP
162
W.ERTB
164
W.FERR
166
W.FER1
170
W.SST
172
W.OBF1
174
W.OBFH
176
W.ERUN
200
W.FPS1
202
W.EXJ
204
W.IOEF
W.PNTY
206
W.R5
210
W.VTYP
212
W.KNUM/W.RECL/W.KOSC
214
W.RECH
216
W.DFLT
W.FPPF
220
W.LNMP
222
W.PRNT
$PRINT
224
W.TYPE
$TYPE
226
W.ACPY
$ACCPT
230
W.READ
$READ
232
ZK-232-81


octal offset
global symbol
W.MOPR
234
W.MOV1
236
$MOPRM
W.MOA1
240
W.MOV2
242
W.MOA2
244
W.MOTC
246
W.MOTR
250
W.MOT2
252
W.MOTY
254
W.DEVL
256
W.NULL
W.CPXF
260
W.FOB1
262
W.FOB2
264
W.EXST
266
W.FNML
270
$MXFNL
W.WDB
272
W.SKLM
274
W.TKLM
276
W.TSKP
300
W.WNLO
302
W.WNHI
304
W.KREF
306
W.KOTP
W.KMAT
310
W.EXTK
312
$EXTKL
W.SPBN
W.LUN0
314
W.PLIC
316
W.TBST
320
W.TBFN
322
W.ERXT
324
W.ERLG
326
W.FIN
330
W.NAM
332
W.IOXT
334
W.RLME
336
W.RQME
340
W.GSA
342
W.END
344
ZK-233-81

Appendix B

FORTRAN Logical Unit Control Block Definitions

B.1 FCS-11 LUB Control Block Format



ZK-234-81

Status Bit Definition

D.STAT - Word 1

```
DV.FIX   =2     RECORD TYPE ='FIXED'
DV.FNB   =4     FILE NAME BLOCK INITIALIZED
DV.DFD   =10    DEFINE FILE DONE DIRECT ACCESS UNIT
DV.FAK   =20    PARTIAL FDB FLAG FOR ENCODE/DECODE
DV.FACC  =40    FILE ATTRIBUTES:   0 - DEFAULT
                                   1 - CALL FDBSET
DV.OPN   =200   UNIT OPEN MUST BE 200'S BIT
DV.VAR   =400   RECORDTYPE='VARIABLE'
DV.SEG   =1000   RECORDTYPE='SEGMENTED'
DV.FMP   =2000  FORMATTED ACCESSED UNIT
```

D.STAT - Word 1

```
DV.UFP   =4000   UNFORMATTED ACCESSED UNIT
DV.ASGN  =10000  FILESPEC:              0 - USE DEFAULT
                                        1 - FROM CALL ASSIGN
DV.CLO   =20000  CLOSE IN PROGRESS
DV.FRE   =40000  FREE FORMAT ALLOWED
DV.RW    =100000 CURRENT OPERATION:     0 -READ
                                        1 - WRITE
```

D.STA2 - Word 2

```
DV.AI4   =2      DEFINEFILE ASSOC VAR: 0 - I*2
                                       1 - I*4
DV.RSZ   =4      EXPLICIT RECORDSIZE SPECIFIED
DV.CC    =10     EXPLICIT CARRIAGE CONTROL SPECIFIED
DV.SPL   =20     DISPOSE = 'PRINT'
DV.DEL   =40     DISPOSE = 'DELETE'
DV.RDO   =400    READONLY
DV.UNK   =1000   TYPE = 'UNKNOWN'
DV.OLD   =2000   TYPE = 'OLD'
DV.NEW   =4000   TYPE = 'NEW'
DV.SCR   =10000  TYPE = 'SCRATCH'
DV.APD   =20000  ACCESS ='APPEND'
DV.SAV   =40000  DISPOSE='SAVE'
DV.BN    =100000 BLANK  = 'NULL'
```

B.2 RMS-11 Control Block Formats

Status Bit Definitions

D.STAT - Word 1

```
DV.SEQ  =1       SEQUENTIAL ACCESS
DV.DIR  =2       DIRECT ACCESS
DV.KEY  =4       KEYED ACCESS
DV.FIX  =10      FIXED LENGTH RECORDS
DV.PAK  =20      PARTIAL FDB FLAG FOR ENCODE/DECODE
DV.FACC =40      FILE ATTRIBUTES:   0 - DEFAULT
                                    1 - CALL FDBSET
DV.VAR  =100     VARIABLE LENGTH RECORDS
DV.OPN  =200     UNIT OPEN MUST BE 200'S BIT
DV.FMP  =2000    FORMATTED ACCESSED UNIT
```

D.STAT - Word 1

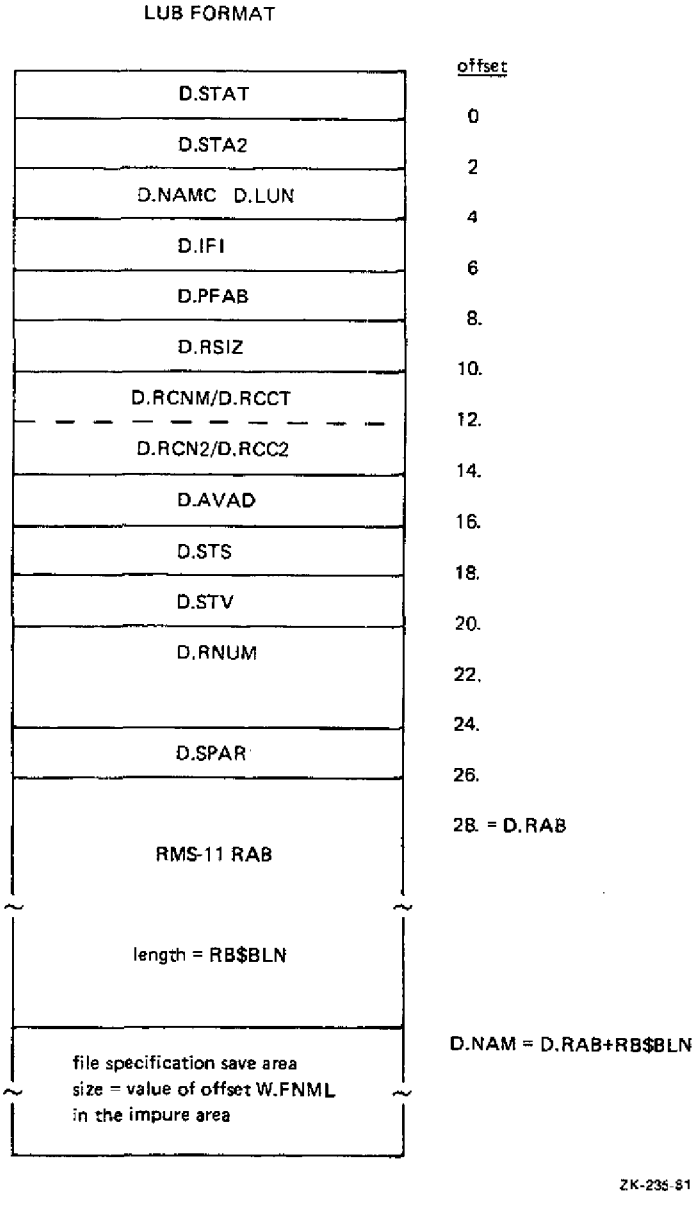
```
DV.UFP  =4000    UNFORMATTED ACCESSED UNIT
DV.SEG  =10000   SEGMENTED RECORDS (UNFORMATTED SEQ ONLY)
DV.CLO  =20000   CLOSE IN PROGRESS
DV.FRE  =40000   FREE FORMAT ALLOWED
DV.RW   =100000  CURRENT OPERATION:   0 - READ
                                      1 - WRITE
```

D.STA2 - Word 2

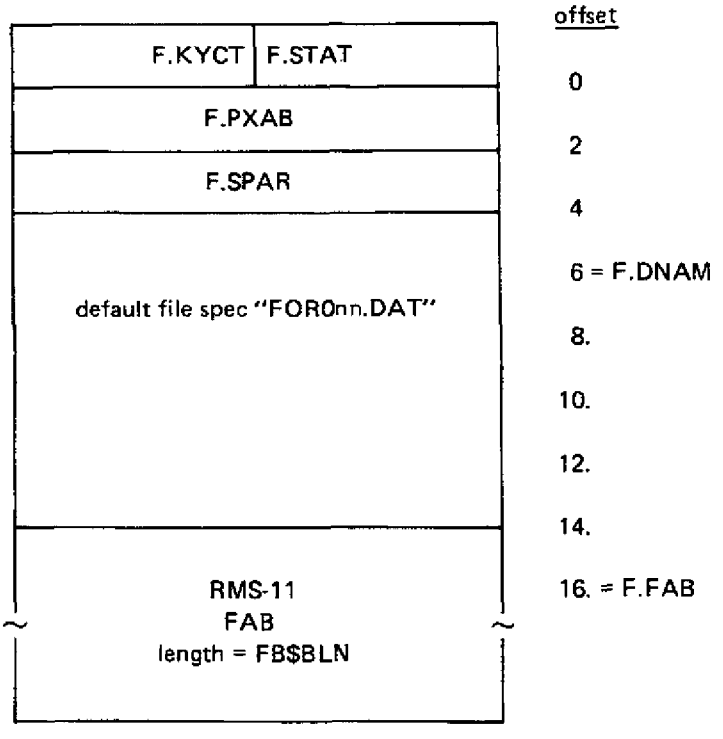
```
DV.SEQ  =1       ORGANIZATION=SEQUENTIAL
DV.REL  =2       ORGANIZATION=RELATIVE
DV.IDX  =4       ORGANIZATION=INDEXED
DV.CC   =10      EXPLICIT CARRIAGE CONTROL SPECIFIED
DV.SPL  =20      DISPOSE = 'PRINT'
DV.DEL  =40      DISPOSE = 'DELETE'
DV.AI4  =100     DEFINEFILE ASSOC VAR: 0 - I*2
                                       1 - I*4
DV.RSZ  =200     EXPLICIT RECORDSIZE SPECIFIED
DV.RDO  =400     READONLY
DV.UNK  =1000    TYPE = 'UNKNOWN'
DV.OLD  =2000    TYPE = 'OLD'
DV.NEW  =4000    TYPE = 'NEW'
DV.SCR  =10000   TYPE = 'SCRATCH'
DV.APD  =20000   ACCESS='APPEND'
DV.SAV  =40000   DISPOSE='SAVE'
DV.BN   =100000  BLANK  ='NULL'
```

FORTRAN FAB block definition

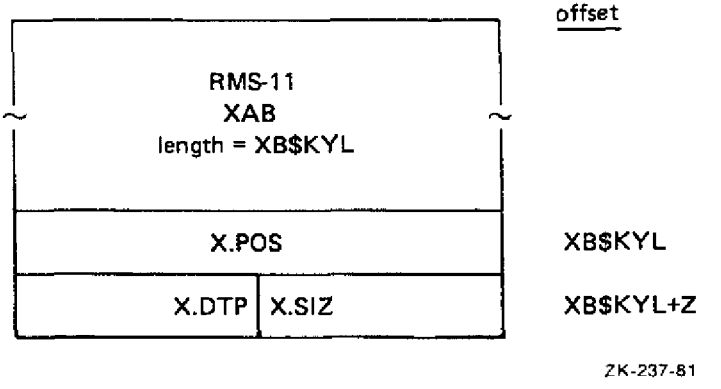
Figure B–1: LUB Format



FORTRAN Key Definition XAB blocks



ZK-236-81



ZK-237-81

Appendix C

OTS Size Summary

This appendix is a guide to the approximate sizes of all the modules in the PDP-11 FORTRAN-77 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.

C.1 Modules Always Present

C.1.1 FCS-11 Support

	Module Name	Module Size in Decimal Words
$CLOSE	Close files	44
$ERRLO	Error message construction	303
$ERRMO	Error message I/O	37/97
$ERRPT	Error control processing	252
$ERTXT	Error message text	1128/0
$FCHNL	LUB processing	67
$FPERR	FPP interrupt processor	54
$FPUTI	FPP utilities	37
$OTI	OTS initialization	84
$R50	Radix-50 to ASCII conversion	44
$SAVRG	Register save co-routine	59
$VINIT	Virtual array initialization	42

$OTV OTS Impure area (by PSECT)		
$$AOTS	Common work area	274
$$DEVT	Logical unit control table [Size=UNITS*54]	
$$FSR1	FCS buffer area [Size=ACTFIL*264]	1056
$$IOB1	I/O buffer [Size=max(MAXBUF,67)]	67
$$OBF1	Object time format buffer [Size=max(FMTBUF,32)]	32
$$FSR2	FCS impure area	75
$$OTSI	Mixed OTSs trap	2

C.1.2 RMS-11 Support

	Module Name	Module Size in Decimal Words
$CLOSE	Close files	98
$ERRLO	Error message construction	292/230
$ERRMO	Error message I/O	37/97
$ERRPT	Error control processing	249/259
$ERTXT	Error message text	1128/0
$FABUT	FAB control block processing	87
$FCHNL	LUB processing	101
$FPERR	FPP interrupt processing	54
$FPUTI	FPP utilities	37
$OTI	OTS initialization	130
$R50	Radix-50 to ASCII conversion	44
$RQLME	Dynamic memory allocation	58
$SAVRG	Register save co-routine	59
$VINIT	Virtual array initialization	42

	Module Name	Module Size in Decimal Words
$OTV OTS Impure area (by PSECT)		
$$AOTS	Common work area	279
$$DEVT	Logical unit control table [Size=UNITS*1]	6
$$FSR1	Dynamic memory area [Size=ACTFIL*(S.LUB+S.FAB+704)]	1816
$$IOB1	I/O record buffer [Size=max(MAXBUF,66)]	66
$$OBF1	Run-Time format buffer [Size=max(FMTBUF,32)]	32
$$DEVU	Dynamic memory listhead	2
$$OTSI	Mixed OTSs Trap	2

C.2 Common I/O Support

The following modules are common to all I/O operations.

	Module Name	Module Size in Decimal Words
$CONVI	Integer format conversions[1]	225
$CONVL	Logical format conversions[1]	49
$CONVR	Real format conversions[1]	680
$CONVZ	Octal and hexadecimal format conversions[2]	335
$FIO	Format processor	1045
$FMTCV	Run-time format compiler	532
$IOARY	Array I/O transmission	71
$IOELE	I/O element transmission	164
$IOVAR	Virtual array I/O transmission	94

[1] Loaded only if needed, or if list-directed or run-time format processing is used.

[2] Loaded only if needed, or if run-time format processing is used.

	Module Name	Module Size in Decimal Words
$LSTI	List-directed input processor	484
$LSTO	List-directed output processor	282
LICSB$	List-directed input constant storage block[3]	129

[3] Loaded only if list-directed input processing is used.

C.2.1 FCS-11 Support

	Module Name	Module Size in Decimal Words
$FCSRM	Dummy RMS-11-only entries	61
$FNBST	File specification processing	75
$INITI	I/O statement initialization	210
$OPEN	File open processing	318

C.2.2 RMS-11 Support

	Module Name	Module Size in Decimal Words
$FNBST	File specification processing	43
$INITIO	I/O statement initialization	309
$OPEN	File open processing	381

C.3 Sequential Input/Output

The following modules are used for sequential access I/O.

C.3.1 FCS-11 Support

	Module Name	Module Size in Decimal Words
$ISU	Sequential unformatted READ	81
$OSU	Sequential unformatted WRITE	47
$ISF	Sequential formatted READ[1]	26
$OSF	Sequential formatted WRITE[1]	37
$ISL	List-directed READ[1]	36
$OSL	List-directed WRITE[1]	54
$GETS	Get sequential record	40
$PUTS	Put sequential record	34

[1]Requires format processing routines.

C.3.2 RMS-11 Support

	Module Name	Module Size in Decimal Words
$ISU	Sequential unformatted READ	80
$OSU	Sequential unformatted WRITE	95
$ISF	Sequential formatted READ[1]	26
$OSF	Sequential formatted WRITE[1]	37
$ISL	List-directed READ[1]	36
$OSL	List-directed WRITE[1]	54
$GETS	Get sequential record	62
$PUTS	Put sequential record	78

[1]Requires format processing routines.

C.4 Direct Input/Output

The following modules are used for direct access I/O.

C.4.1 FCS-11 Support

	Module Name	Module Size in Decimal Words
$IRU	Direct access unformatted READ	40
$ORU	Direct access unformatted WRITE	42
$IRF	Direct access formatted READ[1]	29
$ORF	Direct access formatted WRITE[1]	43
$GETR	Get direct access record	22
$PUTR	Put direct access record	50
$CKRCN	Check record number, update associated variable	40
$DEFF	DEFINEFILE and FIND statements	71

[1] Requires format processing routines.

C.4.2 RMS-11 Support

	Module Name	Module Size in Decimal Words
$IRU	Direct access unformatted READ	40
$ORU	Direct access unformatted WRITE	47
$IRF	Direct access formatted READ[1]	29
$ORF	Direct access formatted WRITE[1]	43
$GETR	Get direct access record	42
$PUTR	Put direct access record	94
$CKRCN	Check record number, update associated variable	50
$DEFF	DEFINEFILE statement	54
$FOR	FIND statement	58
$DLR	DELETE statement	62

[1]Requires format processing routines.

C.5 Keyed Input/Output

The following modules are used for keyed access I/O.

C.5.1 RMS-11 Support

	Module Name	Module Size in Decimal Words
$IKF	Formatted keyed READ[1]	68
$IKU	Unformatted keyed READ	70
$CKKEY	Key description setup	45
$DLS	Sequential DELETE	44
$RSF	Formatted REWRITE[1]	37
$RSU	Unformatted REWRITE	43
$UPDAT	Update record	58
$GETK	Get keyed record	47

[1]Requires format processing routines.

C.6 Miscellaneous I/O Support

C.6.1 FCS-11 Support

	Module Name	Module Size in Decimal Words
$ASSIG	ASSIGN subroutine	41
$BACKS	BACKSPACE statement	90
$CLSCA	CLOSE subroutine	9
$CLSST	CLOSE statement	150
$ENCDE	ENCODE/DECODE and internal file statements[1]	143
$ENDF	ENDFILE statement	57
$FDBSET	FDBSET subroutine	90
$OPNST	OPEN statement	530
$REWIN	REWIND statement	51

[1] Requires format processing routines.

C.6.2 RMS-11 Support

	Module Name	Module Size in Decimal Words
$ASSIGN	ASSIGN subroutine	43
$BACKS	BACKSPACE statement	100
$CLSCA	CLOSE subroutine	9
$CLSST	CLOSE statement	154
$ENCDE	ENCODE/DECODE and internal file statements[1]	143
$ENDF	ENDFILE statement	40
$FDBSET	FDBSET subroutine	98
$OPNST	OPEN statement	709
$REWIN	REWIND statement	59
$UNLOC	UNLOCK statement	47

[1]Requires format processing routines.

C.7 Miscellaneous Compiled-Code Support

	Module Name	Module Size in Decimal Words
$AGO	Assigned GO TO statement	12
$ARYCK	Array subscript checking	17
$CGO	Computed GO TO statement	18
$MADB1	1-Dimensional adjustable array	44
$MADB2	2-Dimensional adjustable array	63
$MADBN	N-Dimensional adjustable array	69
$MADBV	Adjustable virtual array	62

	Module Name	Module Size in Decimal Words
$NAM	Traceback chain processing	15
$STPPA	STOP/PAUSE statements	31
$VIRT	Virtual array addressing	84

C.8 Processor-Defined Functions

	Module Name	Module Size in Decimal Words
$ABS	Real absolute value	7
$ACOS	Arc cosine	49
$AIMAG	Imaginary part	6
$AINT	Real truncation	9
$ALOG	Real log	87
$AMIX0	Real max/min of INTEGER*2	23
$AMIX1	Max/min of REALS	33
$AMOD	Real modulo	15
$ANINT	Real and DOUBLE nearest integer	24
$ASIN	Arc sine	41
$ATAN	Arc tangent	121
$CABS	Complex absolute value	33
$CEXP	Complex exponential	26
$CLOG	Complex logarithm	26
$CMPLX	Complex from reals	9
$CONJG	Complex conjugate	10
$COSH	Hyperbolic cosine	71
$CSIN	Complex sine	53
$CSQRT	Complex square root	42
$DABS	Double absolute value	9

	Module Name	Module Size in Decimal Words
$DACOS	Double arc cosine	53
$DASIN	Double arc sine	45
$DATAN	Double arc tangent	159
$DBLE	Double from REAL	7
$DCOSH	Double hyperbolic cosine	91
$DDIM	Double positive difference	14
$DIM	Positive difference	12
$DINT	Double truncation	11
$DLOG	Double logarithm	117
$DMIX1	Max/Min of DOUBLES	25
$DMOD	Double modulo	17
$DPROD	Double product of REALS	12
$DSIGN	Double transfer of sign	13
$DSIN	Double Sine	119
$EXP	Real exponential	120
$FCAL	Internal service entry	7
$FLOAT	INTEGER*2 to Real	8
$FLOTJ	INTEGER*4 to Real	17
$ICHAR	Character to integer conversion	4
$INDEX	Match a substring in a string	40
$LEN	Length of a character element	3
$LGE	Lexical greater than or equal to character comparison	16
$LGT	Lexical greater than character comparison	16
$LLE	Lexical less than or equal to character comparison	16
$LLT	Lexical less than character comparison	16
$I4FIX	Real to INTEGER*4	12
$IABS	INTEGER*2 absolute value	8

	Module Name	Module Size in Decimal Words
$IAND	INTEGER*2 AND	7
$IDIM	INTEGER*2 positive difference	10
$IEOR	INTEGER*2 exclusive OR	6
$IFIX	Real to INTEGER*2	8
$MOD	INTEGER*2 modulo	7
$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	44
$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	21
$MAXO	INTEGER*2 maximum	10
$MINO	INTEGER*2 minimum	10
$NINT	Nearest integer	19
$REAL	Real from Complex	5
$RJMIX	Real max/min of INTEGER*4	27

	Module Name	Module Size in Decimal Words
$SIGN	Real transfer of sign	11
$SIN	Real sine	99
$SINH	Hyperbolic sine	73
$DSINH	Double hyperbolic sine	93
$SNGL	Real from Double	14
$SQRT	Square root	78
$TAN	Real tangent	36
$DTAN	Double tangent	40
$TANH	Hyperbolic tangent	68
$DTANH	Double hyperbolic tangent	102

C.9 Compiled-Code Arithmetic Support (R4 Calls)

	Module Name	Module Size in Decimal Words
$ADC	Add/subtract Complex	29
$CMC	Compare Complex	22
$DVC	Divide Complex	50
$DVJ	Divide INTEGER*4	26
$MLC	Multiply Complex	27
$MLJ	Multiply INTEGER*4	21
$NGC	Negate Complex	16
$PWCC	Complex to Complex exponentiation	71
$PWCJ	Complex to Integer exponentiation	115
$PWDD	Floating to floating exponentiation	68
$PWII	INTEGER*2 to INTEGER*2 exponentiation	54
$PWJJ	INTEGER*4 TO INTEGER*4 exponentiation	133
$PWRI	Floating to integer exponentiation	81
$PWRR	Real to Real exponentiation	40
$SWPXY	Stack swap	95
$TSC	Test Complex	16

C.10 Compiled-Code Character Support

	Module Name	Module Size in Decimal Words
$CHASN	Character assignment	42
$CHCMP	Character comparison	65

C.11 Service Subroutines

	Module Name	Module Size in Decimal Words
$DATE	DATE	68
$ERRSE	ERRSET	72
$ERRSN	ERRSNS	22
$ERRTS	ERRTST	22
$EXIT	EXIT	13
$IDATE	IDATE	29
$IRAD5	IDATE50	15
$R5OAS	R5OASC	6
$RAD50	RAD50	11
$RAN	RAN	19
$RANDO	Random number generation	53
$RANDU	RANDU	18
$SECND	SECNDS	49
$TIME	TIME	41
$USERE	USEREX	11

C.12 Optional Modules

	Module Name	Module Size in Decimal Words
$CONVR	Real format conversions (FPP version)	587
$FPPUT	EIS version	7
$SHORT	Null error message text	1
$ERRLO	Null error message logging	1
$MLJ	EIS version	57
$DVJ	EIS version	74
$JMOD	EIS version	25

C.13 RSX-11S Subset Support

	Module Name	Module Size in Decimal Words
$CLOSE	Close files	2
$ERRLO	Error message construction	262
$ERRMO	Error message I/O	48
$ERRPT	Error control processing	228
$FCHNL	LUB processing	63
$FCS11	Dummy ECS entry points	61
$GETS	Sequential input	39
$INITIO	I/O statement initialization	179
$ISF	Sequential formatted input[1]	26
$ISL	List-directed input[1]	41
$ISU	Sequential unformatted input	57
$OSF	Sequential formatted output[1]	37
$OSL	List-directed output[1]	40
$OSU	Sequential unformatted output	47
$OTI	OTS initialization	68
$PUTS	Sequential output	27
$OTV OTS Impure area (by PSECT)		
$$AOTS	Common work area	266
$$IOB1	I/O buffer (Size=max(MAXBUF,67))	67
$$OBF1	Run-Time format buffer (Size=max(FMTBUF,32))	32
$$OTSI	Mixed OTSs traps	2
NAM		1

[1]Requires format processing routines.

Appendix D

Program Section Descriptions

This appendix 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.

$$OTSI—OTS Instructions

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

$$OTSD - OTS Pure Data

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

$$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.

$$DEVT—Logical Unit Device Table

$$DEVT defines the FORTRAN logical unit device table. The entries in this table are fixed-length FORTRAN Logical Unit Blocks (LUB). A LUB is composed of a File Control Services (FCS) FDB or an RMS RAB and FAB, and a header for use by FORTRAN. At task start-up, the actual number of LUBs available to the FORTRAN task is determined from the size of $$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.

$$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 133 (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.

$$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.

$$TSKP—Task Information

$$TSKP contains five 2-byte fields that provide the OTS with information about the task. The information is supplied by the Task Builder.

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 is placed in the lowest common segment 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
$$FIOI—contains the integer conversions
$$FIOL—contains the logical conversions
$$FIOR—contains the floating-point conversions
$$FIOS—contains the list-directed constant storage block
$$FIOZ—contains the octal and hexadecimal conversions
$$FIO2—contains the conversion dispatch table

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W