PDP-11 FORTRAN-77/RT-11

Object Time System Reference Manual

AA-BR71A-TC

March 1984

This document describes the object modules that are linked with compiled FORTRAN-77/RT-11 V5.0 code by the RT-11 linker to produce an executable job. Includes a description of the character-handling modules.

SUPERSESSION/UPDATE INFORMATION:	This is a new document for this release.
OPERATING SYSTEM AND VERSION:	RT-11 V5.1
SOFTWARE VERSION:	FORTRAN-77/RT-11 V5.0

Multiware, Inc.

Adapted from PDP-11 FORTRAN-77/RSX by Multiware, Inc.
139 G Street, Suite 161, Davis, California 95616

digital equipment corporation · maynard, massachusetts

First Printing, March 1984

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

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

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright © 1984 by Digital Equipment Corporation
All Rights Reserved

Printed in U.S.A.

The following are trademarks of Digital Equipment Corporation:

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EduSystem	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAX	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10
VAX	VMS	SBI
DECnet	IAS	PDT
DATATRIEVE	TRAX	digital

MW-F77-006

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

International orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532.

CONTENTS

FIGURES

TABLES

PREFACE

MANUAL OBJECTIVES

This manual contains detailed information about the FORTRAN-77/RT-11 Object Time System (OTS) not contained in the PDP-11 FORTRAN-77/RT-11 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 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/RT-11 User's Guide, the RT-11 Software Support Manual, and the RT-11 Programmer's Reference Manual.

Internal OTS interfaces are not guaranteed to remain constant across releases of FORTRAN-77/RT-11. 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.

STRUCTURE OF THIS DOCUMENT

This manual contains nine 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 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, "Overview of FORTRAN Input/Output," provides a conceptual view of OTS I/O processing and describes the I/O modules employed.
- Chapter 6, "I/O Support," addresses the specifics of FORTRAN-77 input/output -- OPEN, CLOSE, READ, WRITE and other operations.

- Chapter 7, "Format Processing and Format Conversions," describes the internal form of format specifications, the format processing algorithm, and the format conversion routines.
- Chapter 8, "Error Processing and Execution Control," discusses execution control processing, the detection and processing of run-time errors, and the generation of error messages.
- Chapter 9, "Other Compiled-Code Support Routines," describes routines that support various arithmetic and housekeeping operations required by the compiled code.
- Appendix A, "FORTRAN Impure Area Definitions," shows the layout of the OTS work area described in Chapter 4.
- Appendix B, "FORTRAN 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/RT-11 User's Guide
- PDP-11 FORTRAN-77 Language Reference Manual
- RT-11 System User's Guide
- RT-11 Software Support Manual
- RT-11 System Utilities Manual
- RT-11 Programmer's Reference Manual

CONVENTIONS USED IN THIS DOCUMENT

The manual follows these conventions:

- 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.
- The name FORTRAN-77 in the manual refers to PDP-11 FORTRAN-77/RT-11, unless otherwise specified.

CHAPTER 1

OBJECT TIME SYSTEM OVERVIEW

The FORTRAN-77 Object Time System (OTS) consists of assembly language modules that complement the user's compiled code. Most of the OTS routines are independent of the RT-11 operating system. However, certain routines access system functions by using RT-11 Programmed Requests. These calls are the same ones available to a MACRO programmer, and are described in the RT-11 Programmer's Reference Manual. They usually concern the performing of input/output functions, file management, and job control.

The OTS has five main parts:

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

The rest of this chapter introduces each of these parts.

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 job.

Chapters 5 and 6 describe the I/O system design and the I/O routines involved with file management support. Chapter 7 contains information on format processing routines.

1.3 JOB 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: 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 8.

1.4 MATHEMATICAL FUNCTIONS AND SYSTEM SUBROUTINES

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

Appendix D of the User's Guide describes the system subroutines.

1.5 COMPILED-CODE SUPPORT ROUTINES

These 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 9.

CHAPTER 2

CONVENTIONS AND STANDARDS

FORTRAN-77/RT-11 has specific procedural and naming conventions. The following sections describe those conventions.

2.1 REGISTERS

The eight [processor general registers] are referenced as follows:

- R0 - 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 sections that follow describe these calls.

2.2.1 R5 Calls

This calling sequence convention is the standard for PDP-11 FORTRAN-77.

Its basic form 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; thus, references to it could 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. The variable types and their associated register assignments are shown in Table 2-1.

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 they use, and they 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. It is used to ensure that using 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/RT-11 User's Guide.

2.2.2 PC Calls

PC calls are made with a JSR PC,xxx instruction. They 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

Example:

The FORTRAN statement

```
REWIND 3
```

is compiled into the code

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

2.2.3 R4 Calls

This convention is used for out-of-line, stack-oriented arithmetic routines and other compiled-code support. These routines receive argument values on the stack, or a pointer to an argument value as an in-line argument immediately following the call. 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 9 describes the modules that use this convention.

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

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$) use a PC call but preserve only R5.
- Traceback name initialization (@NAM) uses a co-routine call.
- Virtual array processing ($VRTxy) uses a PC call that preserves all registers except R0.
- Job 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 other chapters for more details on these special variants.

2.3 LABELING CONVENTIONS

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

2.4 CONTEXT SAVE AND RESTORE

The calling sequence determines the OTS register context conventions. See 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 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
   .END     START
```

Notes:

1. The call to OTI$ initializes the FPP (SFPA$S).
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 that contains a FORMAT statement contains the required FORMAT conversion references.)

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. Note that subprograms do not reference this symbol. When the linker processes a reference to $OTSVA, it loads the FORTRAN impure area and defines global symbol $OTSV in the job that contains the address of the symbol $OTSVA. All FORTRAN OTS routines obtain the address of the impure area by referencing the location $OTSV.

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 those two areas.

4.1 WORK AREA STORAGE DESCRIPTION

The work area contains job-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 113 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.
- Error message text buffer -- The buffer for the error text message line is 70 bytes. 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 job initialization routine OTI$ copies a prototype version of the table into this area. The offset W.ERTB 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:

Job 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: Job 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 job initialization time and equal to W.BEND - W.BFAD		133
W.DEV	Start address of the logical unit control table		
W.DEVL	The high address+1 of the logical unit control table		
W.END	Last word of named offsets		
W.EXST	Exit with status value		
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.PRNT	Logical unit number for PRINT statement	$PRINT	6

(continued on next page)

Table 4-1 (Cont.): Job Control Information

Global Symbol	Description	Global Name	Default
W.READ	Logical unit number for READ statement	$READ	1
W.SKLM	Job's current stack overflow		
W.SST	Limit address of the SST table		
W.TYPE	Logical unit number for TYPE statement	$TYPE	5

Table 4-2: I/O Control Information

Global Symbol	Description
BLBUF	Address of next data byte in current I/O record
CHNATB	Address of Channel Table
COUNT	Length of array in an I/O list
DENCWD	Maximum number of I/O records or 0 if no limit
DEVHDR	Address of lowest device handler
ENDEX	Address of END= statement or 0
ENMLNK	Pointer to last name on entry names queue
EOLBUF	End address+1 of current I/O record
ERREX	Address of ERR= statement or 0
FILETB	Address of first Logical Unit Control Table (LUB)
FILPTR	Address of active LUB or 0
FMTAD	Current pointer into format string
FMTCLN	Value of SP on entry to I/O processing
FREESP	Pointer to free memory
ITEMSZ	Size in bytes of current I/O list element
PLNBUF	Start address of current I/O record

(continued on next page)

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

Global Symbol	Description
QELEM	Address of queue element flag
RACNT	Number of data bytes remaining in current I/O record
RECIO	Address of record-processing I/O routine (GET or PUT)
RTCNLS	RT-11 channel-usage bitmap
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	Co-routine 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
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.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

(continued on next page)

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

Global Symbol	Description
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
W.ELEM	Flag indicating data element has been processed
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	Job 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 FP-11 errors

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.NAMC	Traceback chain list head, global name: $NAMC
W.SEQC	Traceback current statement number, global name: $SEQC

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

4.2 LOGICAL UNIT CONTROL TABLE

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

FORTRAN I/O is performed either with logical units, or through RT-11 channels and calls to SYSLIB. The logical unit method is best suited for applications requiring sequential I/O.

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

4.2.1 LUB Definitions

Each logical unit has a LUB allocated dynamically when a job is executed. There is one LUB allocated for each unit declared in the compiler's /N:m or /UNITS= qualifier (if neither the /N nor the /UNITS= qualifier is not specified, the default value is six logical units). Each LUB is a fixed-length block consisting of 20 decimal 16-bit words. At job initialization time, each LUB is set to 0. A close operation also sets each LUB to 0.

Offsets of the form D.xxxx and xxxxxx describe portions of each LUB, as follows:

ASSOCV -- Address of associated variable

BLKNO -- Current block number

BUFNO -- Number of buffers, one byte

BUFRAD -- Address of start of buffer

BUFRSZ -- Size of buffer in words

CHNLNO -- Channel number, one byte

DATAD -- Address of data pointer

DEVNM -- Device name (RAD50)

FILNM -- File name (RAD50), two words

FILSZ -- Number of blocks for file creation

EXTEN -- File type (RAD50)

HIGHBL -- Highest block number written

RECMAX -- Number of records in file

RECSZ -- Record size

D.FDB -- status word 0 (see below)

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

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

Each LUB contains three status words; D.FDB, D.STAT and D.STA2. The bits in these status words have symbolic names of the form DV.xxx or xxxxxx. These bits are defined as follows:

Status Bits used in D.FDB

Symbol	Value	Description
BUFBIT	1	Double buffering flag bit
EOF	10000	End of file reached
KB	20000	File open to KB

Status Bits used in D.FDB (cont.)

Symbol	Value	Description
LISTMD	20	Force output for listing device
LP	2000	File open to LP:
LSTFMT	40	File is open for listing format
MWRB	2	Modified random block flag
OLD	400	'OLD' flag for RT-11 SYSLIB
OPNBIT	4000	'OPEN' flag for RT-11 SYSLIB
RDBFWT	1000	Read before write
RDO	100	'READONLY' flag for RT-11 SYSLIB
SCR	4	'SCRATCH' flag for RT-11 SYSLIB
TT	200	File open to console terminal
UNLIST	10	Forced for FORTRAN output
WRITE	100000	Writes have been performed on file

Status Bits used in D.STAT

Symbol	Value	Description
DV.FIX	2	Fixed-length records
DV.FNB	4	File Name Block initialized
DV.DFD	10	Direct access unit
DV.FAK	20	Partial LUB for ENCODE/DECODE and internal files
DV.FACC	40	File attributes defined
DV.OPN	200	Unit open
DV.VAR	400	Variable-length Records
DV.SEG	1000	Segmented Records
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)

Status Bits used in D.STA2

Symbol	Value	Description
DV.AI4	2	Associated variable is INTEGER*4 data type
DV.RSZ	4	Explicit RECORDSIZE specified
DV.CC	10	Explicit carriage control specified
DV.SPL	20	DISP = 'PRINT' specified
DV.DEL	40	DISP = 'DELETE' 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.SAV	40000	DISPOSE = 'SAVE' specified
DV.BN	100000	BLANK = 'NULL' specified

CHAPTER 5

OVERVIEW OF FORTRAN INPUT/OUTPUT

This section describes some of the independent I/O modules and provides an overview of the I/O subsystem. Input/output support provided by PDP-11 FORTRAN-77/RT-11 uses the RT-11 file system. The I/O support modules in OTS reduce all input/output calls to RT-11 programmed requests for logical block read or write functions. Special functions such as OPENing, CLOSing, REWINDing or writing ENDFILE marks are accomplished through other appropriate RT-11 programmed requests. See Chapter 7 for descriptions of the format-processing routines.

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

- Compiled-code interface
- Data formatting
- Record processing

The compiled-code interface level consists of the routines called directly by the compiled code. The routines (listed in Table 5-1) 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. Then the records are passed to or received from the next level -- 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 the RT-11 file system.

Figure 5-1 illustrates the I/O subsystem.

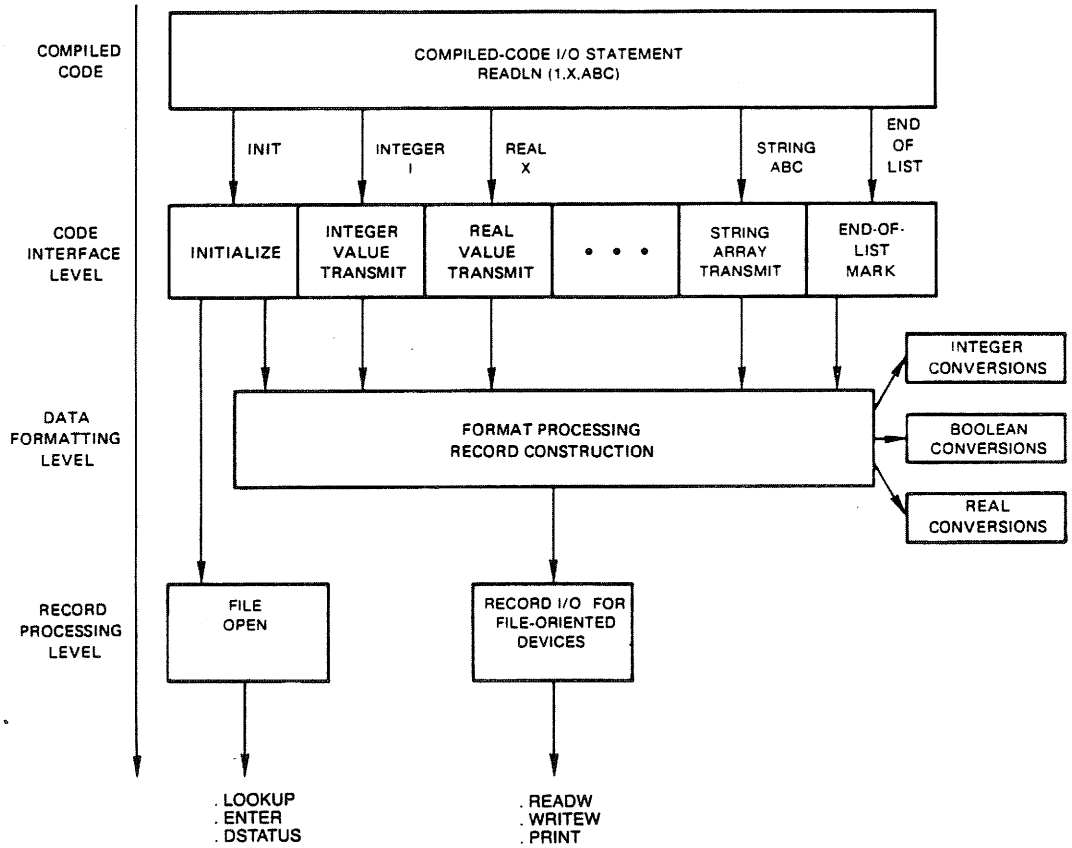

Figure 5-1: The I/O Subsystem

5.1 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
```

5.1.1 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.

5.1.1.1 The Routines - Table 5-1 lists the entry point names for the initialization-processing routines. Each routine has two entry points:

- XXX$ -- for I/O statements that do not use END= or ERR=
- XXXE$ -- for I/O statements that do use END= or ERR=

Table 5-1: I/O Initialization Entries

Entry Name	Arguments	Function
ISF$	u,f	Sequential formatted input
ISFE$	u,f,e	
ISU$	u	Sequential unformattted input
ISUE$	u,e	
IRF$	u,r,f	Direct formatted input
IRFE$	u,r,f,e	
IRU$	u,r	Direct unformatted input
IRUE$	u,r,e	
OSF$	u,f	Sequential formatted output
OSFE$	u,f,e	
OSU$	u	Sequential unformatted output
OSUE$	u,e	

ORF$ ORFE$	u,r,f u,r,f,e	Direct formatted output
ORU$ ORUE$	u,r u,r,e	Direct unformatted output
ENF$ ENFE$	c,f,a c,f,a,e	ENCODE
DEF$ DEFE$	c,f,a c,f,a,e	DECODE
ISL$ ISLE$	u u,e	List-directed input
OSL$ OSLE$	u u,e	List-directed output
FDR$ FDRE$	u,r u,r,e	Direct FIND
ENDF$ ENDFE$	u u,s	ENDFILE
REWI$ REWIE$	u u,s	REWIND
DEFF$	u,mr,rl,v,vf	DEFINEFILE
IIF$ IIFE$ IIFA$ IIFAE$	d,f d,f,e adb,f adb,f,e	Internal file read
OIF$ OIFE$ OIFA$ OIFAE$	d,f d,f,e adb,f adb,f,e	Internal file write

Arguments:

u Logical unit number -- INTEGER*2 value.

r Direct access record number -- INTEGER*4 value.

f Format specifier -- address of compiled format text.

adb Address of the array descriptor block.

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.

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.

5.1.1.2 $INITIO - The $INITIO routine performs specific functions based on the arguments passed by the initialization-processing routines described in Section 5.1.1.1. In addition, $INITIO paves the way for the remaining levels of processing 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.

As mentioned, 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.

The symbols and argument masks used by the routines are described in Tables 5-2 and 5-3, respectively. Table 5-4 describes the operations $INITIO performs based on the bit settings.

Table 5-2: I/O Initialization Symbols

Symbol	Value	Description
FL.ERR	100000	END=/ERR= present
FL.INB	40000	Internal files passed by ADB
FL.IND	20000	Internal files passed by descriptor
FL.ENC	11000	ENCODE/DECODE 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
FL.OPN	40	OPEN required

Table 5-2 (cont.): I/O Initialization Symbols

Symbol	Value	Description
FL.IGN	20	Ignore format and record type checks
FL.KEY	10	Keyed access (not allowed in RT-11)
FL.REW	4	REWRITE (not allowed in RT-11)
FL.DEL	2	DELETE (not allowed in RT-11)
FL.KIN	1	Integer key value

Table 5-3: 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
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
ENF$	ENCODE statement: FL.FMT+FL.ENC
DEF$	DECODE statement: FL.FMT+FL.ENC
ENDF$	ENDFILE statement: FL.WRT+FL.IGN
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 5-4: I/O Initialization Routine Functions

Function	Description
FL.DIR	Compare the access mode of the I/O statement with the access mode of the logical unit; issue OTS error 31 if the access mode does not match. Issue OTS error 26 if direct access is required but has not been specified.

Table 5-4 (cont.): I/O Initialization Routine Functions

Function	Description
FL.EDA	Save the ENCODE/DECODE buffer address in the impure area offsets, PLNBUF (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.
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 PLNBUF (start address) and BLBUF (current address). Add the length of the internal logical unit specifier to offset PLNBUF to determine the end address of the internal logical unit. Save this value in impure area offset EOLBUF.
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 error OTS 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.

5.1.2 List Element Transmission

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

IOat$

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

There are additional entry points, used only for arguments that are addresses. They are defined as follows:

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 ASCII 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.

5.1.3 Termination Call

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

5.2 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

$STO -- list-directed output processor

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

For unformatted I/O, since conversion is not needed, 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:

PLNBUF -- 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.

5.3 RECORD PROCESSING LEVEL

The record-processing routines are called to transfer records to and from the RT-11 file system. The record-processing routines are:

$GETS -- sequential input

$PUTS -- sequential output

$GETR -- direct input

$PUTR -- direct output

5.4 PRINT, TYPE, AND ACCEPT STATEMENTS

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 also 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

5.5 OPEN AND CLOSE STATEMENTS

The OPEN and CLOSE source statements allow user 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:

15 8	7 0
ARGTYPE	KEYWRD ID
INFO	

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 5-5).

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 decodable 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 5-5 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 5-5: Summary of Argument Blocks by Keyword

Keyword Name	Keyword Number	Allowed Argtypes	Literal Values	Literal Encoding
ACCESS	4	4	DIRECT	1
			SEQUENTIAL	2
			APPEND	3
			KEYED	4
ASSOCIATE VARIABLE	17	2,3		
BLANK	25	4	NULL	1
			ZERO	2

(continued on next page)

Table 5-5 (Cont.): Summary of Argument Blocks by Keyword

Keyword Name	Keyword Number	Allowed Argtypes*	Literal Values	Literal Encoding
BLOCKSIZE	18	1,2,3		
BUFFERCOUNT	9	1,2,3		
CARRIAGECONTROL	7	4	FORTRAN LIST NONE	1 2 3
DISPOSE	2	4	SAVE DELETE PRINT	1 2 3
ERR	3		--	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		
MAXREC	16	1,2,3		
NOSPANBLOCKS	12	--		
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		

* 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'
```

5.6 OTHER INTERNAL SUPPORT ROUTINES

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

5.6.1 $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: it is set if there is an error, clear if there is not an 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 $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.

5.6.2 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 Chapter 7.

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.

5.6.3 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.

5.6.4 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.

5.6.5 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.

5.6.6 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. Seven 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.

5.6.7 Register Save and Restore -- .SAVR1

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

5.7 FORTRAN FILE AND RECORD FORMATS

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

5.7.1 Sequential Organization Files

You can process sequential files on all devices. Records may be fixed length, or variable length. Fixed-length records have no control information and are packed densely into blocks by the file system. Variable-length formatted sequential records are separated from one another by the control character sequence <CARRIAGE-RETURN><LINEFEED> (octal 15, octal 12).

Unformatted sequential records have a byte-count value prepended to the beginning of each record. In addition, each block that holds records of this type contains a list of record pointers at the block's end. This structure facilitates backspace operations through unformatted sequential files.

5.7.2 Random Organization Files

Files that are randomly organized are also called direct-access files. All records in a direct-access file have the same length. Random files can be opened only on block-replacable devices, such as disks. They cannot be used with sequential devices, such as terminals, printers or magnetic tape.

Direct-access records contain no system-produced control information and are packed densely into blocks by the file system.

CHAPTER 6

I/O SUPPORT

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

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

- R0 -- address of the Logical Unit Block (LUB)
- R1 -- address of the Logical Unit Block (LUB) (copy)
- R3 -- address of the work area
- R2 and R4 -- scratch registers

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

6.1 I/O CONTROL BLOCK

The I/O system associates a single control block, called the Logical Unit Block (LUB), with each open unit. See Section 4.2 and Appendix B for more information about the LUB.

6.2 OPEN PROCESSING

Default file open processing and OPEN statement processing merge into a single common routine, $OPEN$ (see Section 6.2.3), for a file open. $OPEN$ invokes RT-11's .LOOKUP programmed request for opening existing files, and the .ENTER request for opening new output files. In either case, a required device handler is .FETCHed if it is not already resident.

The RT-11 programmed requests .LOOKUP, .ENTER, and .FETCH make use of the User Service Routine (USR). This set of RT-11 file-services routines can be allowed to swap in and out of memory as required (SET USR SWAP), or it can be forced to remain resident (NOSWAP). See the _RT-11 User's Guide_ for more information about USR SWAPPING.

If the main module of a FORTRAN-77 program is compiled without the /U or /NOSWAP options, and the USR is set to SWAP, then the 2K-word USR will be allowed to become resident in memory temporarily in a pre-designated location in the FORTRAN-77 OTS. This location is a pure-code area in PSECT $$OTSI, where no USR programmed requests are made. In most programs, the OTS pure code region occupies at least 2K words, and will execute properly while USR swapping takes place. In very small programs that use very few OTS routines, it is possible to link a job with inadequate space in $$OTSI for the USR to swap over. Here it is recommended to SET USR to NOSWAP. If it is not possible to SET USR to NOSWAP, then you should expand the area by using the provided PSECT $SPACE and a short MACRO routine:

```
.TITLE  USRSPC
.PSECT  $SPACE
.BLKW   200.            ; Required extra space in words
.END
```

Assemble the routine with the MACRO assembler, and then link it with your small program.

6.2.1 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' (not supported) sets DV.APD, producing an error 48. If DV.APD is not specified, the default is 'SEQUENTIAL'.

- ASSOCIATEVARIABLE -- The variable address is stored at D.AVAD in the LUB. If the variable is type INTEGER*4, DV.AI4 is set in D.STA2.

- BLANK -- 'NULL' sets DV.BN in D.STA2. Note that if the /X (NO F77 syntax) switch is not set and no BLANK= is specified, the compiler passes a BLANK='NULL' parameter.

- BLOCKSIZE -- Causes run-time error 48 in RT-11 implementation.

- BUFFERCOUNT -- The value specified is stored at BUFNO in D.FDB. If the value is less than 1, or greater than 2, an error occurs.

- CARRIAGECONTROL -- If DV.CC is set, 'FORTRAN' sets UNLIST bit in D.FDB, and 'LIST' sets LISTMD in D.FDB. If DV.CC is not set and DV.FMP is specified, UNLIST is the default.

- DISPOSE -- 'SAVE' sets DV.SAV in D.STA2; 'PRINT' sets DV.SPL (not implemented); 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 -- Causes run-time error 45 in RT-11 implementation

- FILE or NAME -- If a file is specified, $FNBST is called to initialize the file Name block and DV.ASGN is set in D.STAT. $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 FILSZ in the LUB. If it is positive, a contiguous allocation is made. If the value is greater than 32767, an error occurs.

- KEY -- Causes run-time error 48 in RT-11 implementation.

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

- NOSPANBLOCKS -- Causes run-time error 48 in RT-11 implementation.

- ORGANIZATION -- If 'SEQUENTIAL', ignored. Otherwise, run-time error 48 is generated.

- READONLY -- If this keyword is present, DV.RDO is set in D.STA2.

- RECORDSIZE or RECL -- The value is stored at D.RSIZ, and bit DV.RSZ is set in D.STA2 of the LUB. If the size is negative or is larger than the user record buffer size, 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 -- Causes run-time error 48 in RT-11 implementation.

- STATUS or TYPE -- If STATUS is not present, the default is 'NEW'. Note, however, that if the /X (NO F77 syntax) switch is not 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. The resulting code is placed in D.STA2 in the LUB. 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. Processing is aborted immediately if there is no unit number, the unit number is invalid, or the unit is already open.

- USEROPEN -- Saves passed LUB address in work area's W.UOPN.

6.2.2 Default OPEN Processing

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

- For input, it sets DV.OLD.
- For output, it sets DV.NEW.

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

6.2.3 $OPEN$ Procedure

The $OPEN$ procedure opens the file and performs the checks and computations common to OPEN statement processing (6.2.1) and default OPEN processing (6.2.2).

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

- If no user file specification is provided (DV.ASGN is not set), a default file specification is set up. The routine $FLDEF is called to assemble the file dd:FOR0nn.DAT, where dd is the device name, and nn is the logical unit number.
- The user record buffer description in the LUB, BUFRAD and BUFRSZ, is initialized with the address specified by the impure area offset W.BFAD and the length specified by W.BLEN.
- A record length is computed. If a user-specified value is available, that value is used; otherwise, one of the following values is used:

  132 for formatted files

  128 for unformatted files of fixed-length records

  126 for other unformatted files

If impure area offset W.UOPN is nonzero, the user's routine is called to perform the OPEN operation; otherwise, .LOOKUP or .ENTER is called to open the file, given the specified or default name fields. If the open operation fails because the file cannot be found, and DV.UNK is set, the operation is retried with DV.NEW set.

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 LUB, BUFRAD and BUFRSZ, 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 RT-11 programmed request or resulting from the consistency checks, the file is closed. If the file was just created, it is deleted as well.

6.2.4 USEROPEN Interface Specification

The USEROPEN parameter of the OPEN statement gives you a way to access special 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 processing options, you must do the following:

* Using the MACRO-11 language, write a routine that opens the file.

* In your FORTRAN 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 program), you should write it as if it were 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 the standard FORTRAN calling convention:

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

FDB
: The address of the LUB for the logical unit.

ISTS
: The INTEGER*2 error status to be returned. The value is expected to be the F.ERR completion status and to follow the convention used for SYSLIB status returns (positive numbers indicate success, negative numbers 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-written procedure:

* All FORTRAN processing is completed prior to the call.

* The LUB address specified is valid until the logical unit is closed. Note, however, that you do not have access to the LUB in the FORTRAN program. You can access the LUB in a MACRO-11 program; the LUB 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  .LOOKUP, .ENTER
     ERRBYT=52
     DEVNM=6

     NOSUP:: MOV     2(R5),R0          ; Get LUB addr
             ADD     #DEVNM,R0         ; Point to name block
             .LOOKUP #AREA,0,R0        ; does it exist?
             CMPB    @#ERRBYT,#1
             BNE     ERROR             ; Error if anything but.
             .ENTER  #AREA,0,R0        ; Open the file
             MOVB    @#ERRBYT,R0       ; Return completion status
             NEG     R0                ; Make error code negative
             RETURN
     ;
     ERROR:  ...
             RETURN
```

6.2.5 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. $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.

6.3 FILE CLOSE PROCESSING

File close processing is performed by the OTS routine $CLOSE, which uses the following RT-11 programmed requests:

- The File Close Processing request, .CLOSE, to close files
- The File Deletion request, .DELETE, to 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. The address is stored at offset ERREX, if present.

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.

6.4 SEQUENTIAL I/O PROCESSING

This section describes low-level OTS routines called by the I/O statement processors and format processors to perform sequential record transfers. 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.
- Maintains a pointer to next available byte
- Transfers bytes from the device buffer to the user buffer
- When an end of file condition is detected, the END= transfer is executed. 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
- Maintains an output pointer to the next available buffer position
- Transfers byte data from the user buffer to an appropriate device buffer.
- 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.

6.5 DIRECT ACCESS I/O PROCESSING

This section describes low-level OTS routines called by the I/O statement processors and format processors to perform the actual calls to the file system 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 $GETBL to read the required block when necessary
- 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 LUB.
- $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 $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 LUB. Registers R1 and R2 are undefined.

6.6 AUXILIARY I/O 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. The current block number and offset are reset to that the file is "rewound". 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. Note that 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.

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 D.RSIZ. 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.

6.7 I/O-RELATED SUBROUTINES

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

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.

CLOSE

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

CHAPTER 7

FORMAT PROCESSING AND FORMAT CONVERSIONS

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

7.1 COMPILER FORMAT LANGUAGE

Format specifications are compiled into a standard internal form. That form, which is illustrated in Figure 7-1, consists of a format code byte followed by one to five bytes of optional format code parameters.

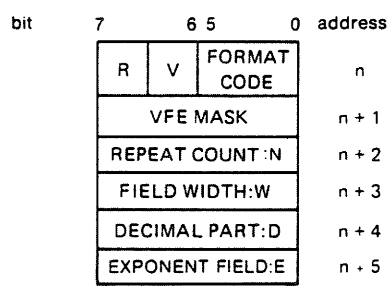

Figure 7-1: Format Code Form

7.1.1 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.

Table 7-1 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 7-1: 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) n not VFE
26 ;	nHcl...cn or 'cl..cn'	n-1	-	-	-	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	-	
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	-	

7.1.2 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 then 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,XXX, with R5 pointing to the program unit argument list. The format interpreter performs all range checking on the result.

7.1.3 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.

7.1.4 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	I*2	7
I	I*4	12
E,G	R*4	15.7(E=2)
E,G	R*8	25.16(E=2)
D,F	R*4	15.7
D,F	R*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

7.1.5 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
```

7.2 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 linker can correctly build overlaid job imagess.

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.

7.3 FORMAT AND LIST-DIRECTED PROCESSORS

The format and list-directed processors -- $FIO, $LSTI, and $LSTO -- operate as co-routines 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.

7.3.1 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 that 'A' format 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.

7.3.2 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.

7.3.3 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

Data Type	Format
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, whi ch yields 80 print positions.

7.4 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 set by the compiler's /R option (global $LRECL). 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.

7.5 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.

7.6 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.

7.7 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.

7.8 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, F77CVF, is an FPP implementation that is significantly faster but slightly less accurate. The entire FPP state is conserved.

7.9 FORMAT CONVERSION ERROR PROCESSING

When a format conversion error occurs, both methods of error continuation, ERR=transfer and return (see Section 9.2.2.1), are generally supported. The actions taken for these errors are as follows:

Error 51 -- list-directed I/O syntax error
The result value is null (no change).

Error 53 -- format/variable type mismatch error
The value is used as is, without conversion.

Error 55 -- output conversion error
The field is filled with asterisks.

Error 56 -- input conversion error
The result value is 0, 0. or 0.D0.

Error 60 -- 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/RT-11 User's Guide_.

CHAPTER 8

ERROR PROCESSING AND EXECUTION CONTROL

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

8.1 PROGRAM INITIALIZATION

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

```
JSR  PC,OTI$
```

The following operations are performed:

- $STFPP is called to initialize the FP-11 floating-point processor or the KEF11A floating-point microcode option (unless F77EIS 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 job error count limit is set to 15.
- $VINIT is called to initialize the virtual array mapping window if virtual arrays are used.

8.2 EXECUTION-TIME ERRORS

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

8.2.1 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 8.4).

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

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

8.2.2 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 decr emented. 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 re port includes the exit flag.
- If the log bit is set, the error report includes the no-exit flag. If the job 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.

8.2.2.1 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 error s other than I/O errors.

8.2.2.2 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 have been examined.

Work area offset W.IOEF is used to obtain this special error processing. The eff ects 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 job will exit.

8.2.3 Floating-Point Processor Errors

All Floating-Point Processor (FPP) errors are processed by routine $FPERR. When divide-by-zero, overflow, or underflow occurs, zero is supplied as the result of the operation that caused the trap.

8.2.4 Error Message 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 mess age processing, calling the appropriate message utilities as required. $ERRLG produces a 5-line error log containing the following:

- On line 1, the job name and error number.
- On line 2, message text.
- 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.

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.

$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 job is exiting and 0 if the job is continuing.

8.3 STOP AND PAUSE STATEMENT PROCESSING

STOP and PAUSE statements are compiled to calls as follows:

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

   STOP$ for STOP

   PAUS$ for PAUSE

All context is saved. $ERRLG (see Section 8.2.4) is called to output the message. STOP then jumps to $EXIT; PAUSE issues a .TTYIN request and then returns when a <CR><LF> sequence is encountered.

8.4 USER INTERFACING TO TERMINAL MESSAGE OUTPUT

The error-reporting message facility enables users to write text to their terminals 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.

Example:

The following prints 'HELLO' on the user terminal:

In FORTRAN:

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

IN MACRO-11

```
MSG::   CLR -(SP)           ; 2nd word of message block
        MOV 2(R5),-(SP)     ; Address of ASCIZ text
        MOV SP,R0           ; R0 points to message block
        CLR R1              ; Signal non-error type message
        JSR PC,$ERRLG       ; Output the message
        CMP (SP)+, (SP)+    ; Delete message block
        RTS PC              ; Return
        .END
```

Only a single line can be output.

8.5 EXECUTION CONTROL SUBROUTINES

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

ERRSET -- The error number specified by the user is extracted and checked for v alidity. The logical arguments are extracted and the appropriate bits in the error contro l byte are manipulated. If a limit count is provided, it is stored at offset W.EC NT.

ERRSNS -- This routine is called with zero to two integer arguments:

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

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

offset	W.ERNM	into	NUM
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 job termination.

CHAPTER 9

OTHER COMPILED-CODE SUPPORT ROUTINES

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

9.1 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 1-argument operation or the second (right hand) argument of a 2-argument operation. For 2-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).

9.1.1 Exponentiation

The exponentiation routines are as follows:

	Data Type		
Routine	Base	Exponent	Result
PWIIx$	I*2	I*2	I*2
PWIJx$	I*2	I*4	I*4
PWJIx$	I*4	I*2	I*4
PWJJx$	I*4	I*4	I*4
PWRIx$	R*4	I*2	R*4
PWRJx$	R*4	I*4	R*4
PWDIx$	R*8	I*2	R*8
PWDJx$	R*8	I*4	R*8
RWRRx$	R*4	R*4	R*4
PWRDx$	R*4	R*8	R*8
PWDRx$	R*8	R*4	R*8
PWDDx$	R*8	R*8	R*8
PWCIx$	C*8	I*2	C*8
PWCJx$	C*8	I*4	C*8
PWCCx$	C*8	C*8	C*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/RT-11 User's Guide.

9.1.2 Complex Arithmetic Operations

The following entries are used:

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

9.1.3 INTEGER*4 Arithmetic Operations

The following entries are used:

MLJx$ -- Multiplication

DVJx$ -- Division

x is S, C, or P

9.1.4 Stack Swap Operations SWPxy$

These 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 are of 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.

9.1.5 Character Operations

These 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).

9.2 ARRAY PROCESSING SUPPORT

An Array Descriptor Block (ADB) is a data structure the compiler provides 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 /I 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 that contains 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.

9.2.1 Adjustable Array Initialization

Four routines are used 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.

9.2.2 Array Subscript Checking

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

The form of the call is:

1. Push 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.

9.2.3 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.

The form of the call is:

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 /I was not specified

   VRTxC$, if /I 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.

9.2.4 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

Note that 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.A0
        .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   120000  ; A.PWRD
        .WORD      0    ; A.SIZB
A.ADB:  .WORD      0    ; A.ASTR
        .WORD      0    ; A.A0
        .WORD    31004  ; A.CWRD
        .WORD      0    ;   D1
        .WORD      0    ;   D2
```

9.3 GO TO STATEMENT SUPPORT

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

9.3.1 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.

9.3.2 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.

9.3.3 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
```

9.4 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

Note that the statement number pertains to the program unit of the NEXT list block, since the current program unit statement number is maintained at the fixed global location $SEQC.

If the compiler command option /S:NAM, /S:BLO, or /S: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 /S:NAM is specified, the current statement number is not updated ($SEQC remains zero).

If /S:BLO 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 /S:LIN is specified, the current statement number is updated on every statement, maintaining a positive number.

APPENDIX A

FORTRAN IMPURE AREA DEFINITIONS

000	W.SEQC	= $OTSVA, $SEQC
002	PLNBUF	
004	CHNATB	
006	FILETB	(Points to LUB's)
010	QELEM	
012	DEVHDR	
014	FREESP	
016	ENMLNK	
020	RTCNLS	
022	FMTAD	
024	FILPTR	
026	W.NAMC	= $NAMC
030	W.LUNS	= .NLUNS
032	W.MO	
034	W.BFAD	
036	W.BLEN	
040	W.BEND	
042	RECIO	
044	EOLBUF	
046	FMTCLN	
050	BLBUF	
052	PSCALE	

```
|                         |
+-------------------------+
|  054  W.LICP/FSTKP      |
+-------------------------+
|  056  W.LICB/FSTK       |
|       NOARG=FSTK        |
|       PARLVL=FSTK+2     |
|       NUMFLG=FSTK+4     |
|                         |
|  16-word scratch area   |
|                         |
|     overflow word       |
|                         |
+-------------------------+
|  120  FMTRET            |
+-------------------------+
|  122  VARAD             |
+-------------------------+
|  124  TSPECP            |
+-------------------------+
|  126  TYPE              |
+-------------------------+
|  130  REPCNT/UNFLGS     |
+-------------------------+
|  132  LENGTH            |
+-------------------------+
|  134  D                 |
+-------------------------+
|  136  ITEMSZ            |
+-------------------------+
|  140  DOLFLG (byte)     |
+-------------------------+
|  141  W.ELEM (byte)     |
+-------------------------+
|  142  COUNT             |
+-------------------------+
|  144  RACNT             |
|       FMTLP=  RACNT     |
|       UNCNT=  RACNT     |
|       W.UOPN= RACNT     |
+-------------------------+
|  146  DENCWD            |
+-------------------------+
|  150  W.PC              |
+-------------------------+
|  152  EXADDR            |
+-------------------------+
|  154  ENDEX             |
+-------------------------+
|  156  ERREX             |
+-------------------------+
|  160  W.ECNT            |        = $ERCNT
+-------------------------+
|  162  W.ERNM            |
+-------------------------+
|  164  W.LIMT            |
+-------------------------+
|  166  W.OPFL            |
+-------------------------+
|  170  W.ERLN            |
+-------------------------+
|  172  W.ERLE            |
+-------------------------+
|  174  W.ERTB            |
|                         |
```

```
|                        |
+------------------------+
|  176  W.OBFL           |
+------------------------+
|  200  W.OBFH           |
+------------------------+
|  202  W.ERUN           |
+------------------------+
|  204  W.FPST           |
+------------------------+
|  206  W.EXJ            |
+------------------------+
|  210  W.PNTY (byte)    |
+------------------------+
|  211  W.IOEF (byte)    |
+------------------------+
|  212  W.R5             |
+------------------------+
|  214  W.VTYP           |
+------------------------+
|  216  W.RECL           |
|       W.KNUM=W.RECL    |
|       W.KDSC=W.RECL    |
|       W.SLEN=W.RECL    |
+------------------------+
|  220  W.RECH/W.SADR    |
+------------------------+
|  222  W.FPPF (byte)    |
+------------------------+
|  223  W.DFLT (byte)    |
+------------------------+
|  224  W.LNMP           |
+------------------------+
|  226  W.PRNT           |        = $PRINT
+------------------------+
|  230  W.TYPE           |        = $TYPE
+------------------------+
|  232  W.ACPT           |        = $ACCPT
+------------------------+
|  234  W.READ           |        = $READ
+------------------------+
|  236  W.MOTY           |
+------------------------+
|  240  W.DEVL           |
+------------------------+
|  242  W.CPXF (byte)    |
+------------------------+
|  243  W.NULL (byte)    |
+------------------------+
|  244  W.FDB1           |
+------------------------+
|  246  W.FDB2           |
+------------------------+
|  250  W.EXST           |
+------------------------+
|  252  W.FNML           |
+------------------------+
|  254  W.WDB            |
+------------------------+
|  256  W.TKLM           |
+------------------------+
|                        |
```

260	W.WNLO
262	W.WNHI
264	W.KREF
266	W.KMAT (byte)
267	W.KDTP (byte)
270	W.LUN0 (byte)
271	W.SPBN (byte)
272	W.PLIC
274	W.TBST
276	W.TBFN
300	W.ERXT
302	W.ERLG
304	W.FIN
306	W.FPER
310	W.NAM
312	W.IOXT
314	TTYRWF
316	NBLOCK
320	8 words extra . . .
340	W.END

APPENDIX B

FORTRAN LOGICAL UNIT CONTROL BLOCK DEFINITIONS

B.1 LUB CONTROL BLOCK FORMAT

000	D.FDB	Status word 0
002	BUFRAD	
004	BUFRSZ	
006	DEVNM	File Name Block (FNB)
010	FILNM(1)	
012	FILNM(2)	
014	EXTEN	
016	DATAD	
020	BUFNO (byte)	
021	CHNLNO (byte)	
022	ASSOCV/D.AVAD	
024	RECSZ/D.RSIZ	
026	BLKNO	
030	FILSZ	
032	HIGHBL	
034	D.STA2	Status word 2
036	D.STAT	Status word 1
040	RECMAX	
042	D.RCCT/D.RCNM	
044	D.RCC2/D.RCN2	
046	F.LUN	

B.2 STATUS BIT DEFINITIONS

D.FDB - Status Word 0

```
BUFBIT  =1        DOUBLE BUFFERING FLAG BIT
MWRB    =2        MODIFIED RANDOM BLOCK FLAG
SCR     =4        'SCRATCH' FLAG FOR RT SYSLIB
UNLIST  =10       FORCED FOR FORTRAN OUTPUT
LISTMD  =20       FORCED FOR LISTING OUTPUT
LSTFMT  =40       FILE IS OPEN FOR LISTING FORMAT
RDO     =100      'READONLY' FLAG FOR RT SYSLIB
TT      =200      FILE OPEN TO CONSOLE TERMINAL
OLD     =400      'OLD' FLAG FOR RT SYSLIB
RDBFWT  =1000     READ BEFORE WRITE
LP      =2000     FILE OPEN TO LP:
OPNBIT  =4000     'OPEN' FLAG FOR RT SYSLIB
EOF     =10000    END OF FILE REACHED
KB      =20000    FILE OPEN TO KB
WRITE   =100000   WRITES HAVE BEEN PERFORMED
```

D.STAT - Status 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
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 - Status 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'
```

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 linker's storage maps. All object module sizes are shown in decimal words.

C.1 MODULES ALWAYS PRESENT

C.1.1 File I/O Support

Module Name		Module Size in Decimal Words
$CLOSE	Close files	135
$ERRLO	Error message construction	303
$ERRMO	Error message I/O	37/97
$ERRPT	Error control processing	252
$ERTXT	Error message text	1004/0
$FCHNL	LUB processing	49
$FPERR	FPP interrupt processor	58
$FPUTI	FPP utilities	37
$OTI	OTS initialization	84
$R50	Radix-50 to ASCII conversion	40
$SAVRG	Register save co-routine	59
$VINIT	Virtual array initialization	65

$OTV OTS Impure area (by PSECT)

$$AOTS	Common work area	224
$$OBF1	Object time format buffer	32
$$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

	Module Name	Module Size in Decimal Words
$IOELE	I/O element transmission	164
$IOVAR	Virtual array I/O transmission	94
$LSTI	List-directed input processor	484
$LSTO	List-directed output processor	282
LICSB$	List-directed input constant storage block (3)	129

(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.
(3) Loaded only if list-directed input processing is used.

C.3 COMPILED-CODE CHARACTER SUPPORT

	Module Name	Module Size in Decimal Words
$CHASN	Character assignment	42
$CHCMP	Character comparison	65

C.4 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
$R50AS	R50ASC	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.5 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

OTS SIZE SUMMARY

Module Name		Module Size in Decimal Words
$OTV OTS Impure Area (by PSECT)		
$$AOTS	Common work area	266
$$OBF1	Run-Time format buffer	32
$$OTSI	Mixed OTSs traps	2
NAM		1

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 linker when constructing an executable job image.

$$OTSI -- OTS Instructions

This PSECT contains all of the executable code in the OTS except the formatted and list-directed I/O processors. When RT-11's USR is set to SWAP, it is this region that shares space with the USR. This PSECT has the attributes: RO,I,CON,LCL.

$$OTSD - OTS Pure Data

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

$$AOTS -- OTS Impure Storage

$$AOTS contains the FORTRAN work area impure storage associated with each job. It must be contained in the job'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,LCL.

$$OBF1 -- Object-Time Format Buffer

$$OBF1 defines the FORTRAN object time format buffer. The length is fixed at 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.

Format Conversion PSECTs

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

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