DEC-11-LFFSA-A-D

PDP-11

FORTRAN IV

COMPILER
(V004A)

AND

OBJECT TIME SYSTEM
(V020A)

FUNCTIONAL SPECIFICATION

NOTES

The programs described in this specification are supported by DEC when used on 12K or larger systems only. (References to 8K systems are for information purposes only.)

The material in this specification, including but not limited to construction times and operating speeds, is for information purposes only. All such material is subject to change without notice. Consequently, Digital Equipment Corporation makes no claims and shall not be liable for its accuracy.

The detailed internal information contained herein is applicable for Version 4A of the Compiler and Version 20A of the Object Time System only.

pdp11
PROGRAM
LIBRARY

PART I

PDP-11

FORTRAN IV COMPILER

VERSION 4A

TABLE OF CONTENTS

1.0 OVERALL DESCRIPTION

1.1 Usage

FORTRAN IV is a well known algebraic language (originally designed by BACKUS, et al) in common use on most currently available computers. It is described in the American National Standards Institute FORTRAN IV Language Specification.

PDP-11 FORTRAN is a variant of ANSI Standard FORTRAN IV as will be described in this and associated documents.

The FORTRAN Compiler is used in conjunction with the remainder of the PDP-11 FORTRAN System under the Disk Monitor to allow users to write and run programs on the PDP-11.

1.2 Market

Potential users of FORTRAN are assumed to know FORTRAN and to be able to use the PDP-11 Disk Monitor. FORTRAN allows the user to take advantage of the Disk Monitor and the associated library without requiring knowledge of assembly language techniques. PDP-11 FORTRAN is designed to be saleable in the same market as the IBM 1130.

1.3 Design Philosophy

PDP-11 FORTRAN runs under the PDP-11 Disk Monitor, and will therefore require a system with at least 8K of core.

PDP-11 FORTRAN is ANSI FORTRAN IV compatible with added features to allow most IBM 1130 FORTRAN programs to run without change.

The compiler is designed to be as helpful to the user as possible in terms of diagnostic capabilities and operating characteristics.

1.4 References

A. ANSI FORTRAN IV Language Spec., X3.9, 1966.
B. FORTRAN Object Time System spec., 130-311-002.

C. PDP-11 FORTRAN Programming Manual.
D. Getting on the Air with FORTRAN.

2.0 HARDWARE ENVIRONMENT

2.1 Minimum Requirements

PDP-11 FORTRAN runs under the Disk Monitor, which requires a minimum of 8K of core, RF-11, RC-11, or RK-11 Disk, High Speed Reader/Punch or Dectape, and an ASR-33 TELETYPE. FORTRAN will run in only those configurations supported by DOS.

2.2 Options

FORTRAN supports all standard product line options which are supported by the Disk Monitor(DOS).

2.3 Future Considerations

The system must be highly modular to allow extensions in hardware configuration to be made with minimal effort.

At NO future time will FORTRAN be supported in a paper tape only environment.

3.0 SOFTWARE ENVIRONMENT

3.1 Minimum Requirements

FORTRAN will require the PDP-11 Disk Monitor. Under no circumstances will it run in the Paper Tape System.

The compiler will output code suitable for assembly by the Relocatable Assembler.

3.1.2 Object Time System Requirements

3.1.2.1. Arithmetic Routines

Each of the following is called using the POLISH call(section 4.8) which assumes that upon entry to a routine, register R4 points to a location containing the address of the next routine to be executed. Thus, R4 must be preserved by the routine and the exit from each routine is via a "JMP @(R4)+" which jumps to the next routine as well as advancing the R4 pointer. This way, the compiler need only generate a list of addresses for most arithmetic expressions which is only slightly (approximately 2-3%) slower than in-line JSR calls. The resulting code is usually between 5 and 15% shorter than the equivalent in-line form.

The value of each operand will be on the stack upon entry to a routine. A routine returns with the value of the result on top of the stack (the original operands must be removed from the stack).

The mode of the operands and the mode of the result will be defined implicitly by entry into the routine.

The compiler will distinguish modes by suffixing one character to the subroutine name.

On entry:

(SP)	LAST (SECOND) OPERAND
(SP+N)	FIRST OPERAND

Where N is the length in bytes of each operand.

Upon return:

(SP) RESULT.

The suffix will have the following significance:

SUFFIX	OPERANDS	RESULT
B	BYTE	BYTE
I	INTEGER	INTEGER
L	LOGICAL	LOGICAL
R	REAL	REAL
D	DOUBLE	DOUBLE
C	COMPLEX	COMPLEX

The Arithmetic Routines are:

NAME	FUNCTION
$AD	ADD FIRST OPERAND TO SECOND OPERAND
$SB	SUBTRACT THE SECOND OPERAND FROM THE FIRST OPERAND
$ML	MULTIPLY THE FIRST OPERAND BY THE SECOND OPERAND
$DV	DIVIDE THE FIRST OPERAND BY THE SECOND OPERAND
$PW	RAISE THE FIRST OPERAND TO THE POWER SPECIFIED BY THE SECOND OPERAND.

NOTE: $PW has a two character suffix, the first describes the type of the base, the second describes the exponent type. Byte mode exponentiation is illegal.

3.1.2.2. IF Functions

A. Arithmetic IF tests will be performed by a set of routines whose entry point names begin with $TS with suffixes as described in section 3.1.2.1. The return will be to one of three locations pointed to by the locations at (R4), (R4+2), and (R4+4) for negative, zero, and positive, respectively.

On entry: (SP) contains the operand to be tested. After return: the stack is clear.

B. Logical IF tests are performed by the routine $TRTST. Upon entry, (SP) contains the operand to be tested. Upon return the stack is clear. If the value on top of the stack is zero (false) control is transferred to the address specified at (R4); otherwise control will continue at the word following (R4).

3.1.2.3. Data Conversion Routines.

The operand on the stack is converted to the destination mode and replaces the source operand on the top of the stack.

ROUTINE	OPERAND	RESULTING MODE
$BI	BYTE	INTEGER
$BL	BYTE	LOGICAL
$BR	BYTE	REAL
$BD	BYTE	DOUBLE
$BC	BYTE	COMPLEX
$IB	INTEGER	BYTE
$IL	DEFAULT CONVERSION	
$IR	INTEGER	REAL
$ID	INTEGER	DOUBLE
$IC	INTEGER	COMPLEX
$LB	LOGICAL	BYTE
$LI	DEFAULT CONVERSION	
$LR	LOGICAL	REAL
$LD	LOGICAL	DOUBLE
$LC	LOGICAL	COMPLEX
$RB	REAL	BYTE
$RI	REAL	INTEGER
$RL	REAL	LOGICAL
$RD	REAL	DOUBLE
$RC	REAL	COMPLEX
$DB	DOUBLE	BYTE
$DI	DOUBLE	INTEGER
$DL	DOUBLE	LOGICAL
$DR	DOUBLE	REAL
$DC	DOUBLE	COMPLEX
$CB	COMPLEX	BYTE
$CI	COMPLEX	INTEGER
$CL	COMPLEX	LOGICAL
$CR	COMPLEX	REAL
$CD	COMPLEX	DOUBLE

3.1.2.4. I/O Routines

Each I/O operation involves an initialization call; then one or more calls to transmit one or more list variables per call, then a call to terminate the I/O operation.

Initialize:	(SP+2)	(SP)
$INFI,FORM. READ $OUTFI,FORM. WRITE	ADDRESS OF DEVICE NUMBER	ADDRESS OF 1ST CHAR. OF FORM.
$INI,UNFORM. READ $OUTI,UNFORM. WRITE	ADDRESS OF DEVICE NUMBER	
$INRI,DISK READ $OUTRI,DISK WRITE	ADDRESS OF DEVICE NUMBER	ADDRESS OF LOG. RECORD NUMBER

The ENCODE and DECODE initialization calls are similiar to the above forms except that the names called are $ENCD and $DECD respectively and there are three items on the stack. The top (SP) contains the array address specified, the second (SP+2) contains the format address, and the last (SP+4) contains the character count.

Each initialization call has two in-line parameters corresponding to the "END=" and "ERR=" conditions. If either of these parameters are zero, the condition is not specified. If non-zero, they are the address of the statement to be transferred to for either end-of-file or error conditions respectively. This transfer must be a POLISH mode transfer.

Transmit list variables:

$IOA - TRANSMIT ARRAYS
$IOB - TRANSMIT BYTE LENGTH ITEM
$IOI - TRANSMIT SINGLE WORD INTEGER OR LOGICAL
$IOJ - TRANSMIT TWO WORD INTEGER
$IOR - TRANSMIT REAL
$IOD - TRANSMIT DOUBLE
$IOC - TRANSMIT COMPLEX

One or more parameters may be transmitted, (SP) contains the number of parameters, (SP+2) - (SP+N) contain the addresses of each data item. Note that all consecutive items of the same type will be transferred together.

For array I/O, all consecutive arrays will have their ADB addresses pushed on the stack followed by the number of arrays. A call to $IOA will then be made to cause the array transfer. The OTS may determine the data type of the array by examining the ADB(see section 3.1.2.6).

Terminate:

$IOF, TERMINATE I/O LIST

(See section 7.4 for a complete POLISH example.)

3.1.2.5. Printing of Formatted Records.

The OTS will maintain a File/Device table (which should be modifiable by the user) which, along with other functions, indicates for each logical file whether the first character in a formatted record should be transmitted literally or should be interpreted for print control purposes.

When the table so indicates, the vertical spacing character should be interpreted and therefore converted into an output string as follows:

Vert. spacing char.	meaning	substituted
BLANK	SPACE ONE LINE	<CR,LF>
0	SPACE TWO LINES	<CR,LF,LF>
1	SPACE TO FIRST LINE OF NEXT PAGE	<CR,FF>
+	NO ADVANCE	<CR>
$	NO ADVANCE	<CR>

The $ carriage control acts like a blank for the beginning of line control, but it has the additional characteristic of forcing a <VT> for the end of line character instead of a <CR>, thus leaving the carriage at the end of line rather than at the left margin. This is useful when it is desired to make teleprinter responses on the same line as the query.

The monitor provides device independence as regards the Line Printer and the TELETYPE; therefore the same substituted output string applies to either device.

3.1.2.6. Subscript Computation.

There will be three subscript computation routines, one for each of 1, 2, and 3 dimensional arrays. The call to a subscript routine will have as a single in-line parameter an Array Descriptor Block (generated by the compiler, and for adjustable arrays, initialized by the object code itself), as well as to the current index values on the stack. The subscript routine will return with the address of the desired array item in R0.

The Array Descriptor Block (ADB) format is as follows(see figure 1):

WORD 0: ADDRESS OF THE FIRST ELEMENT OF THE ARRAY.
(ADDR)

WORD 1: BITS 15-14: NUMBER OF DIMENSIONS
BITS 13-11: DATA TYPE
BITS 7-0: DATA ELEMENT SIZE IN BYTES

WORD 2: NUMBER OF INDEX ITEMS FOR THE FIRST
DIMENSION(ABBR.:A)

WORD 3: NUMBER OF INDEX ITEMS FOR THE SECOND
DIMENSION
(ABBR.: B) (PRESENT FOR 2 AND 3 DIMENS. ARRAYS)

WORD 4: NUMBER OF INDEX ITEMS FOR THE THIRD
DIMENSION
(ABBR.:C) (PRESENT FOR 3 DIMENSIONAL ARRAYS)

The subscript routines may, with the information supplied, if desired check for references exceeding the boundaries of the array. This is currently effected by specifying the /CK switch at compile time which forces calls to special bounds checking routines to be generated instead of the standard routines(these routines are prefixed $SBX instead of $SBS).

ROUTINE NAME	$SBS3 $SBX3	$SBS2 $SBX2	$SBS1 $SBX1
(SP) ON ENTRY	VAL OF 3RD INDEX(K)	VAL OF 2ND INDEX(J)	VAL OF 1ST INDEX(I)
(SP+2) ON ENTRY	VAL OF 2ND INDEX(J)	VAL OF 1ST INDEX(I)	
(SP+4) ON ENTRY	VAL OF 1ST INDEX(I)		
R0 AFTER RET CONTAINS THE ADDRESS:	ADDR+((I-1)+A*((J-1)+B*(K-1)))*SIZE	ADDR+((I-1)+A*(J-1))*SIZE	ADDR+(I-1)*SIZE

3.2 Options

Additional discussion occurs in section 8.0.

3.2.1 Switch Options

A. /ON

The compile option to select one or two word integers is /ON. When selected on the input specification, it sets the compile mode to use only one-word integers instead of the normal two. Note that two word integers as described here do not imply two words of precision.

B. /SU

There is also a compile option to suppress the sequencing generated for trace-back. When the switch /SU is specified on the input file name, the sequencing is suppressed from the source listing and the object code. This causes a saving of two words per statement, but eliminates the comprehensive traceback normally available.

C. /CK

The /CK switch may be specified to force the compiler to generate special calls for all subscript references to force run time array boundary checking. This is especially useful when benchmarks are to be run or early debugging is to be performed. When the /CK switch is specified, the compiler generates calls to $SBX subscript routines instead of the $SBS routines.

D. /ER

The /ER switch is an optional switch, that when specified, flags "S" class errors. "S" errors are those errors which are not considered errors in normal usage, but may under certain conditions be considered to be errors. For instance, if the first statement of a program were "X=A", an error would be issued telling the user that A has not been previously defined.

E. /CO

The normal PDP-11 FORTRAN continuation line default is 5. If it is desired to allow other han 5 continuations in a particular compile, it is necessary to specify the /CO:nn switch where nn is a number of lines between 0 and 99 which is to be allowed. Thus if one desires to specify 19 continuation lines, a /CO:19 should be typed. Note that the reason for specifying only a five line default is to conserve core and that the space required in the continuation buffer is nn times 72 bytes where nn is the number of continuations required. Also note that if you run out of space during a compile, but do not need any continuations, that a /CO:0 will reduce the amount of core used by 360 bytes.

F. /GO

If it is desired to immediately execute the results of a compile, it is only necessary to specify the /GO switch. When this switch is specified, the compiler when the current compile is done will automatically invoke the linker to link the program, which will in turn cause the program to be automatically executed.

Note that the switches above, once set, stay set for all successive compiles. To clear either or both of the switches it is necessary to reload or restart the compiler.

G. /LI

In the 12K compiler, listing control is achieved with the /LI switch. This switch may have a value from 0 to 3. /LI:0 specifies the minimal listing, which consists only of error diagnostics and the block descriptor (section 4.11). No listing formatting is done. /LI:1 is the default value. It specifies normal listing formatting, source listing, and the block descriptor to be listed. /LI:2 specifies the above in addition to an assembly listing. /LI:3 adds the assembler symbol table listing to the above, thus giving all possible listing output. The /LI option must be specified every time it is needed, or else the listing will revert to the default value.

H. /AS

The 12K compiler has one additional switch which allows the assembler pass to be bypassed to allow the user to get the assembler output instead of the object output. When the switch /AS is specified,he object file specification defines the assembler output file and the assembler pass is avoided. The default extension for the file is also changed from .OBJ to .PAL . Note that since the 8K compiler does not have the assembler interface, it works as if the /AS switch is always specified.

An example of an compiler command specification having all of the above switches specified might be:

```
OUTPUT/AS,LIST/LI:1<INPUT/ON/SU/ER/CK/CO:19/GO
```

One minor point to be stressed is that if the /AS switch is specified, the /LI switch has no meaning for any values greater than one.

3.2.2 Compiler Options

The compiler for 12K and larger has several capabilities not included in the 8K compiler. First, the 12K version will be capable of direct binary generation without requiring a separate assembly process. Second, it will be capable of compiling multiple routines in a single input file. It will also have neater page and listing control.

3.3 Future Considerations

The compiler has been designed to be easily modifiable and/or extensible to allow for future changes in requirements.

4.0 CONVENTIONS AND STANDARDS

4.1 Labelling

4.1.1 Compiler internal labelling

Routine or module labels may consist of one to six characters and should, if possible, be self descriptive.

Example: the symbol "LPTINT" could refer to an interrupt address for a line printer routine.

4.1.2 Object output labelling conventions

A statement label generated by the compiler consists of a "." followed by the source statement label.

Example: Statement number 237 would generate an internal label of .237

A format label is similar but uses a "$" instead of a "." .

EXAMPLE: $10

Other compiler generated labels will consist of a "$" followed by a single alphabetic character describing the label followed by 4 numeric digits. The following is a list of allowed labels (nnnn refer to the four numeric digits):

$Innnn Integer or Logical constant
$Rnnnn Real constant
$Dnnnn Double or Complex constant

$Fnnnn Internal location labels (for example, DO loop termination label)

4.2 Registers

To avoid confusion, the general registers will always be referred to as follows:

REGISTERS 0-5 = R0-R5
REGISTER 6 = SP
REGISTER 7 = PC

Certain special registers will be referred to according to the following:

STATUS REGISTER (LOC. 177776) = PSW
SWITCH REGISTER (LOC. 177570) = SWR

In a FORTRAN compiled routine, no registers will ever be saved or restored upon calling or being called by an external routine. It is assumed that R5 is the subroutine call register, R4 is the threaded code pointer, and that R0-R3 may be used without prejudice.

4.3 The FORTRAN internal documentation will consist of:

A. ANSI FORTRAN IV Specification, X3.9, 1966.
B. This document
C. Specification for the Object Time System, 130-311-002.
D. Pertinent and profuse comments on the source listings.

Other associated documentation includes: PDP-11 FORTRAN Programming Manual, Getting on the air with FORTRAN.

4.4 Operating Conventions

All operating conventions and command strings, etc. must conform to those of the disk monitor.

4.5 I/O

All input and output for FORTRAN will be done using the standard provisions supplied by the Disk Monitor.

4.6 Character Set/Codes

The character set/codes for FORTRAN will be completely compatible with the ANSI ASCII conventions.

The compiler will not convert lower case to upper case, though lower case may be used only in Hollerith constants and literal strings. Illegal non-printing characters are printed as a circumflex (up arrow) followed by the alphabetic equivalent of the offending character (e.g. 001=^A). The OTS will allow all ASCII characters to be input or output under "A" format. The compiler recognizes as meaningful only the ANSI standard compiler input characters.

4.7 Calling Conventions

FORTRAN callable subroutines and functions will obey the following object code calling conventions:

All argument addresses will be placed in a list following the subprogram call. The standard sequence will be:

```
                .GLOBL  SUBR
                JSR     R5,SUBR
                BR      XX
                A
                B
                .
                .
                .
                Z
        XX:
```

Note: The even byte of the branch instruction following the JSR contains the number of arguments and is pointed to by R5 after the JSR is executed.

Subprograms are responsible for not altering the contents of register R5 since it is the parameter list pointer.

Function subprograms, in addition to the above, will return the result in registers R0-R3(number of registers used is dependent on type, e.g. - integer uses R0, real uses R0 and R1, etc.)

4.8 Fortran POLISH Calls (Threaded Code)

This call is designed to take advantage of the simple POLISH method for evaluating expressions. It assumes that a typical expression consists of a large number of very simple operations done in a linear sequence.

The implementation of this technique as described below makes several assumptions:

1. The first operation done in a POLISH sequence is invariably a "push".
2. It is not necessary to place breakpoints(as in ODT or DDT) in the middle of an arithmetic statement.
3. Speed will not suffer by assignment of a register for special purposes.

Description of this mode is best done using a simple example:

The statement A=B+C would generate code similar to the following (section 4.8.5 describes the entry into POLISH mode).

```
$P0001          ;THE VARIABLE B IS PUSHED
                ;ON THE STACK
$P0002          ;EACH OPERATION CONSISTS OF THE
$ADR            ; ADDRESS OF THE ROUTINE TO
                ; BE EXECUTED.  A PUSH PLACES
$POP3,A         ; A VALUE ON THE STACK, A POP
                ; REMOVES A VALUE
.+2             ;THIS LINE WILL CAUSE POLISH MODE
                ;TO BE EXITED,
                ;AND NORMAL EXECUTION
                ;RESUMED.
```

The subroutines called would be as follows:

```
$P0001:  MOV   #B+4,R0     ;GET THE ADDRESS OF B
         BR    $F0001      ;JUMP TO COMMON PUSH
$P0002:  MOV   #C+4,R0     ;GET THE ADDRESS OF C
$F0001:  MOV   -(R0),-(SP) ;PUSH
         MOV   -(R0),-(SP) ; TWO WORDS ON STACK
         JMP   @(R4)+      ;JUMP TO NEXT ROUTINE
$POP3:   MOV   (R4)+,R3    ;GET ADDRESS OF
                           ;VARIABLE DESTINATION
         MOV   (SP)+,(R3)+ ;POP A VALUE
         MOV   (SP)+,(R3)+ ; TO THE VARIABLE
         JMP   @(R4)+      ;GO TO NEXT ROUTINE
```

($ADR is an OTS routine to add two floating point numbers. See section 3.1.2).

Note that the JMP @(R4)+ jumps to the next routine in the list as well as incrementing R4 over that item in the thread.

All internal functions are called in this manner and must exit using a JMP @(R4)+ and must clear any stack space used (except for the return value which is left on the top of the stack).

All routines explicitly callable by the user (i.e. - subroutines, external functions) are called using the PDP-11 subroutine calling convention(section 4.7).

4.8.1 The following basic decisions were made:

1. R4 will be used as the threaded code pointer.

2. All code (including Integer arithmetic) will be handled in POLISH mode except for actual subroutine (and function) linkage which will be executed in-line.

3. A majority of the service routines detailed herein can actually be made .GLOBL to the compilation and hence occur only once in a core load (rather than included in each compiled module).

4. Any routines not explicitly mentioned as being locally generated are assumed to be global to the program and available from the FORTRAN library (e.g. - $MLF, etc.)

4.8.2 On return from a subroutine or function the value is in R0, R0 and R1, or R0 thru R3. Service routines to move these to stack are:

```
$PSHR4: MOV     R3,-(SP) ;PUSH FOUR WORDS
        MOV     R2,-(SP)
$PSHR2: MOV     R1,-(SP) ;PUSH TWO WORDS
$PSHR1: MOV     R0,-(SP) ;PUSH ONE WORD
        JMP     @(R4)+
```

4.8.3 Getting and putting to the stack given an address which results from subscript calculation proceeds as follows:

```
$GET4:  MOV     6(R0),-(SP)    ;FOUR WORD CASE
        MOV     4(R0),-(SP)
$GET2:  MOV     2(R0),-(SP)    ;TWO WORD CASE
$GET1:  MOV     @R0,-(SP)      ;ONE WORD CASE
        JMP     @(R4)+
$PUT4:  MOV     (SP)+,(R0)+
        MOV     (SP)+,(R0)+
$PUT2:  MOV     (SP)+,(R0)+
$PUT1:  MOV     (SP)+,(R0)+
        JMP     @(R4)+
```

4.8.4 Explicit exit from POLISH mode can be made with a word containing the address of the following word, E.G.:

```
...     ;IN POLISH MODE
.WORD   .+2     ;LEAVE POLISH MODE
...     ;CONTROL PASSES TO HERE
```

4.8.5 Entry to POLISH mode is made via a special routine

```
$POLSH: TST     (SP)+     ;DELETE OLD VALUE OF
                          ;R4 PUSHED ON ENTRY
        JMP     @(R4)+    ;AND WE'RE OFF!!
```

4.8.6 Finally we come to the handling of formal parameters. The POLISH mode string is essentially the same. The service sections look like the following:

```
;A IS FLOATING
$PNNNN: MOV    N(R5),R0 ;GET THE ADDRESS FROM
                        ;CALLING SEQUENCE WHERE "N"
                        ;DEPENDS ON POSITION IN
                        ;FORMAL PARAMETER LIST
        JMP    $GET2    ;NOW JUST LIKE BEFORE
```

The one word and four word forms differ only in which $PUTx is invoked.

Assignment to a formal parameter is:

```
$POPP2: MOV     (R4)+,R0        ;GET DISPLACEMENT OF
                                ;THIS FORMAL IN CALL
        ADD     R5,R0           ;ADDRESS OF ADDRESS
        MOV     @R0,R0          ;ADDRESS OF PARAMETER
        JMP     $PUT2           ;STORE TWO WORDS FROM
                                ;STACK
```

Similarly for one and four word data. Assignments to formals are like local assignments except that the displacement in the call sequence to the actual address must be given to the service routine instead of the actual base address.

4.9 Data Conventions

4.9.1 Integer format (Figure 2)

An Integer number is a 16-bit signed quantity. When in two word format, it is assigned two words, with only the high order word (i.e., the word with the lower address) being significant.

The result of any operation which would exceed 16 bits will cause a diagnostic to be issued by the OTS.

4.9.2 Real format (Figure 3)

The Real number format consists of two words of data as follows:

```
    WORD N - BIT 15 - SIGN OF MANTISSA
             BITS 7-14 - BINARY EXCESS 128 EXPONENT
             BITS 0-6 - HIGH ORDER MANTISSA
    WORD N+2 - BITS 0-15 - LOW ORDER MANTISSA
```

This is a sign-magnitude format with binary normalization.

This format is limited to normalized numbers and the high order bit of the mantissa is always 1, therefore this bit is discarded in this format giving an effective precision of 24 Bits.

4.9.3 Double precision format (Figure 3)

WORD N - SAME AS 4.9.2
WORD N+2 - SAME AS 4.9.2
WORD N+4 - LOWER ORDER MANTISSA
WORD N+6 - LOWEST ORDER MANTISSA

The effective precision is 56 bits.

4.9.4 Complex format (Figure 4)

WORD N AND N+2 - REAL PART (FORMAT AS IN 4.9.2)
WORD N+4 AND N+6 - IMAG PART (FORMAT AS IN 4.9.2)

4.9.5 Byte format

Each data item in this format is 8 bits long. Logical, masking, and arithmetic operations are allowed.

Any arithmetic operations done in byte mode must take into account the limited size that any value may have. The range of numbers from +127 to -128 may be represented. Any arithmetic operations will be accomplished by taking both 8-bit operands, extending the sign to a full word, doing the desired operation, and then truncating the result to 8 bits. No diagnostic will be issued for overflow errors. The result of such an operation, thus, is allowed to be at most 8 bits long. Logical and masking operations will work with the whole byte at a time.

4.9.6 Character strings

A character string is defined to be a string of byte length elements. The length of this form will be limited to 255. If the string length is odd, a blank will be appended to fill out to a word boundary.

4.9.7 Logical values

A Logical value as represented by .TRUE. will be equal to the integer value -1. A logical value as represented by .FALSE. will be equal to the integer value 0.

4.10 File conventions

See Object Time System specification 130-311-002.

4.11 Compiler listing format with summaries

The following is a short example showing the various kinds of information supplied to the user by the compiler. This example is complete, including the heading as well as all other information supplied in a normal compile.

```
FORTRAN V004A                       10:26:05    16-JUN-72    PAGE      1

        C  LISTING SUMMARY EXAMPLE
                DIMENSION X(10)
                COMMON X
                COMMON /ABC/Y(4)
                X(1)=A(1.,2.)
                CALL B
                CALL EXIT
                END

        ROUTINES CALLED:
        A      , B      , EXIT

        SWITCHES = /ON,/CK,/SU

        BLOCK        LENGTH
        MAIN.   47      (000136)*
        .$$$$.  20      (000050)
        ABC     8       (000020)

        **COMPILER ----- CORE**
           PHASE        USED  FREE
        DECLARATIVES 00366 17853
        EXECUTABLES  00458 17761
        ASSEMBLY     00899 20183
```

Several items of interest are described above.

The heading line, which is printed at the top of every page of the listing, shows the page number, the date and time of the compile and the version number of the compiler used.

The listing proper follows.

If subroutines or functions are called by the routine, they are summarized immediately after the end statement is processed. This is of interest mainly in large programs where there may be some confusion as to which routines are required by what.

The switch summary describes what switches were used in the compile. In this particular example, the /ON (one-word integers), /CK (array bounds checking), and the /SU (suppress traceback) switches were set. A side note should be noted that the /SU switch also causes suppression of the sequence numbers which are normally printed on the left margin of the listing.

The block summary describes in decimal words and octal bytes the length of the compiled program and the length of each COMMON block used in the program. The program name entry is flagged with an "*" following the entry. A name of MAIN. describes the main program name. A name of .$$$$. describes blank Common.

The final section of the summary details how much storage was used by each of the three phases of the compiler in decimal words.

5.0 DATA STRUCTURES

The tables used by the compiler, as described below, are dynamically allocated at the time the compiler is first started (see figure 8). The procedure is as follows:

1. Find out the total amount of space used by the monitor and its buffers.
2. Allocate the compiler stack area immediately above the monitor area.
3. Allocate a minimal symbol table area above the stack.
4. Set up the statement buffer just below the compiler overlay area.
5. Set up the COMMON/EQUIVALENCE table.

If a table overflow in the COMMON/EQUIVALENCE table occurs it will expand downwards. If an overflow occurs in the symbol table occurs, it expands upwards. An irrecoverable error occurs if these two tables meet (SYMBOL TABLE OVERFLOW).

After the executable statements have started, the COMMON/EQUIVALENCE table is no longer needed and it is discarded, giving more room for the symbol table.

5.1 Compiler Data Structures

Each internal table will initially be assigned a block of core storage. Each block will be managed by its own set of special routines. Overflow of any block will result in an aborted run.

5.2 Main Symbol Table (Figure 5)

The symbol table consists of a linked chain of entries in free core.

5.2.1 Entry format

Word 0

Bits 15-14 entry type:
00 Data items, including all user and compiler defined variables and literals.
01 Statement functions.

10 External functions.

Bits 13-11 Data type:
000 Logical-1
001 Logical-2
010 Integer
011 Real
100 Double precision
101 Complex
110 Hollerith
111 Unassigned

Bit 10 Adjustable array flag
Bit 9 Set if entry is program name
Bit 8 Constant bit
Bits 7-0 Length of data item in bytes(if Constant bit=1)
Parameter list index(if constant bit=0 and parameter=1).

Word 1

Bit 15 Common indicator (=1 if item is in common)

NOTE: This indicator and the parameter indicator will never both = 1 simultaneously.

Bit 14 Adjustable array indicator (=1 if item defines an adjustable array).
Bit 13 Equivalence indicator (=1 if item appeared in an EQUIVALENCE statement).
Bit 12 Parameter indicator (=1 if variable is a parameter to be accessed by indexed addressing through R5).

NOTE: This indicator and the COMMON indicator will never both = 1 simultaneously.

Bits 11-0 Serial number of entry

Word 2:
Bits 15-0 address of next symbol table entry. Equal to -1 if this is the last entry in the table.

Word 3:
Bits 15-0 1st three characters of symbol name (RADIX 50)

Word 4:
Bits 15-0 second three characters of symbol name.

Word 5

Bit 15 Single reference bit
Bit 14 Assign bit (use in ASSIGN statement)
Bit 13 Explicit bit (explicit typing)
Bit 12 - Used in expression bit
Bit 11 - Generate Push flag
Bits 10-9 Dimensions of item
Bit 8 - Unused
Bits 7-0 - Common block sequence (position in common chain)
0 => not in COMMON
1 => blank COMMON

Word 6:
Bits 15-0 ADB pointer. Points to the associated Array Descriptor Block if this item is dimensioned (I.E., if the dimension indicator is non-zero). The ADB occupies space in the main symbol table area. The pointer from the symbol table entry to the ADB is relative to the start of the ADB.

Additional words - value of entry.
Present only if the constant indicator is set. The value will be represented as the binary equivalent of the original source number, or as a string of ASCII bytes terminated by at least one byte = 0 if the item is a Hollerith constant. The length of this field (not including padded blanks and terminating zeros) in bytes is contained in the data item length (section 5.2.1.0).

5.3 Common Table.

A single contiguous area will be used to accomodate information collected from both COMMON and EQUIVALENCE statements. This area is allocated at the high end of memory, below the compiler. It grows "down" toward the top of the symbol table. After the last declaration statement is processed it will be possible to perform storage allocation for all variables involved in either COMMON or EQUIVALENCE whereupon the area used can be recovered for other use by the compiler.

Within this area, COMMON will use two data structures:

5.3.1 COMMON block header

A 6 word item:
Word 0 - Link to next block header (if any).

Word 1 - 1st two characters of block name (ASCII).
Word 2 - 2nd two characters of block name (ASCII).
Word 3 - Last two characters of block name (ASCII).
Word 4 - 0 terminator for name.
Word 5 - Link to COMMON block list.

5.3.2 COMMON block list

A variable length block containing:

Word n+1 - Link to next group in this block (if any).
Word n - Serial number of variable/array.
.
.
.
Word 1 - Serial number of variable/array.
Word 0 - Zero terminator.

5.4 Array Descriptor Block Table (ADB) (Figure 1)

This table has an entry for each array defined in the subprogram being compiled. The ADB is available to the compiler so that it may compute and fix array entry references when the subscripts are constants, and so that it may reserve the appropriate amount of memory for each array during "END" processing. The format of the compile time and the object time ADB's are not the same. See also section 3.1.2.6.

Word 0:
Link to next ADB relative to the base of the symbol table (=0 if this is the last ADB).

Word 1
Bits 15-14 Number of dimensions in this array
Bits 13-11 Data type (see section 5.2.1)
BitS 7-0 size in bytes of a data element.

Word 2:
Number of index items, first dimension

Word 3:
Number of index items, 2nd dimension

Word 4:
Number of index items, 3rd dimension

5.4.2 Words 4 and 3, or just word 3, are present for 3 dimensional and for 2 dimensional arrays, respectively. If any dimension is adjustable, the corresponding index item word will contain a zero.

5.5 Equivalence Block

A block is created for each group of items which are equivalenced to each other. Format is:

Word 1,2 - A two word work area for this group
Word 3 - Link to next group (if any)
Word 3+N - Serial number of N-th item in this group
Word 4+N - Total offset (in bytes) of item in this group from the base of the items as defined.
Word M - Zero terminator

5.6 Implicit Table

A 26(10) byte table which relates letters of the alphabet to the variable type-modes to be selected whenever implicit mode assignment is called for.

If the first character of a symbol has octal representation N, the entry at relative byte position N-1 contains the mode to be implicitly used for that symbol.

5.7 Do Table (Figure 7)

Created upon processing of a DO statement (see section 7.2.21).

5.7.1 Table format

Words 0,1 - Statement number of terminal statement in RADIX 50.
Word 2 - Serial number of destination return label
Word 3 - Pointer to control variable symbol table entry
Word 4 - Pointer to initial parameter symbol table entry
Word 5 - Pointer to terminal parameter symbol table entry
Word 6 - Pointer to step value symbol table entry

5.8 Object Time Data Structures

See Object Time System Specification, 130-311-002.

5.9 Stack and Table structures in expression evaluation

Three stacks exist for evaluating arithmetic expressions. R4 is the operator stack, R5 is the mode stack, and SP (R6) is the final code stack.

Items on the R4 stack have the following format:

Bits 15-8 - Operator Value ID
Bits 7-0 - Operator Priority

OPERATOR	VALUE ID	PRIORITY
.OR.	1	0
.AND.	2	1
.NOT.	3	2
.LT.	12	3
.GT.	13	3
.EQ.	14	3
.NE.	15	3
.LE.	16	3
.GE.	17	3
+	4	4
-	5	4
*	6	5
/	7	5
**	10	6
UNARY -	11	7

Items on the R5 stack have the following format:

Bit 15 - Zero
Bits 14-12 - Mode of SP item
Bits 11-0 - position of item on SP stack relative to STKCNT.

MODE	VALUE
LOGICAL*1	0
LOGICAL*2	1
INTEGER	2
REAL	3
DOUBLE	4
COMPLEX	5

Items on the SP stack are of the following form:

Variables -

Bit 15 - Zero
Bits 14-12 - Mode change flag
Bits 11-0 - Variable serial number

Operators -

Bit 15 - One
Bits 14-12 - Mode change
Bit 11 -SVSP Flag
Bit 10 - FUNC Flag
Bit 9 - ARRY Flag
Bit 8 - FNEND Flag
Bits 7-0 Operator ID

If SVSP=1, the following word contains the $F label required and the ID contains the type.
If ARRY=1, the ID contains the number of subscripts for the array call and the next word contains the ADB serial number.
FUNC=1 defines the start of a function definition sub-mode. (bits 0-7 have the parameter count, the second word has the depth, and the third word has the serial).
FNEND=1 turns off function sub-mode

Note that normal mode conversions must be checked in the ADB serial number pointer when ARRY=1.

The function parameter lists have the serial numbers of variables.

If bits 15-12 are all = 1 the parameter is a substitution label.

If FNEND=1, the ID contains the number of bytes to pop from the stack.

The exponentiation operator has a second word which contains the type of the base in bits 14-12.

6.0 INPUT/OUTPUT

6.1 Compiler I/O

All compiler I/O is done in formatted ASCII through the Disk Monitor.

6.2 Object Time I/O

See 130-311-002

6.3 Diagnostic Output

A FORTRAN source diagnostic will consist of the following:

[XXXXXYXXXXX]
ERROR zzz
Message.

Immediately following the line in error, 5 characters (XXXXX) on either side of the current character position (Y) will be printed (with control characters interpreted also) inside the brackets, followed by the error number (ZZZ), and the text of the message (if any) corresponding to the error number as found in the disk error directory. The message may be any ASCII string less than 64 characters long and terminated by a <CR,LF>.

All error diagnostics will be printed on the source listing and will also be placed as comments in the object file.

Each error message will be prefixed by a single character describing the class of error. An "F" (Fatal) error will definitely cause improper execution, so execution should not be attempted. A "W" (Warning) error may cause improper execution, but is generally discretionary. An "I" (Informative) error should not affect program execution. The "S" error (issued only if the /ER switch was set) flags items which might be errors depending on the context of the item. (For instance, mixed mode arithmetic is noted using "S" errors.)

Each error diagnostic is described in detail in Appendix E of the FORTRAN manual.

7.0 LANGUAGE

7.1 Source Language

Items implemented for 1130 compatibility are tagged "(1130)".

7.1.1 Language exceptions and differences from ANSI FORTRAN IV.

The following are keyed to the corresponding section numbers in the ANSI FORTRAN IV specification.

<3.2> Line formats.

A TAB may be used in lieu of multiple spaces at the start of a line. If a numeric character follows the TAB, a continuation is assumed.

The default number of continuations is specified to be five. Any number from 0 to 99 may be optionally be declared at compile time.

<4.2> Data types.

Integers are 16 bit signed numbers. They are normally stored in a two word format where the first word is the value and the second is a filler word. Single word integers may be selected as described in section 3.2.

Real numbers use a two word format as described in section 4.9.2.

Double precision uses a four word format(section 4.9.3).

Complex uses a four word format(section 4.9.4).

Logical*1 is a special one byte format useable for alphanumeric and limited arithmetic manipulations.

<5.1.1> Constants.

Hex and octal constants will be allowed within DATA statements. The formats are:

```
ZNNNN   FOR A HEX CONSTANT(1130)
ONNNN   FOR AN OCTAL CONSTANT
```

where NNNN is the hex or octal value respectively. Note that hex values may not exceed FFFF and octal values may not exceed 177777.

Octal constants may also be specified anywhere in a program by specifying a " (double quote) followed by from one to six octal digits not exceeding 177777.

An alternate form of Hollerith constant is a string of characters surrounded by single quotes. Within the string, two consecutive single quotes denote a single quote in the string.

<6.1> Arithmetic expressions

General mixed mode expressions are allowed with no restrictions.

<7.1.2.1.2> Assigned GOTO

The label list is optional. A compiler diagnostic will be supplied if the variable is also used in an expression.

<7.1.2.1.3> Computed GOTO

When the value of the expressin falls outside the range of the supplied statements, an object time error diagnostic will be issued.

<7.1.2.7> STOP N and PAUSE N.

N may be a one to six digit octal constant (not larger than 177777).

The value specified will be typed on the teleprinter when the STOP or PAUSE is executed.

<7.1.2.8> DO Statement.

Integer variables or constants may be used for the do parameters(1130).

<7.1.3> I/O Statements.

The following statements are added to facilitate random access I/O to a fixed or moving head disk(1130).

A) DEFINE FILE a(m,l,U,v)

Set up a file as follows:

a - Logical unit number (an integer constant from 1 to 32767). This is the descriptor by which the file is recognized when a FIND, READ or WRITE is executed.

m - Maximum number of records in the file.

l - Length of each record in words.

U - The letter "U" which declares the file to be unformatted (e.g. - binary). No other designation is legal.

v - Associated variable name to be used for the record pointer.

B) FIND (a'b)

Position the disk head properly for record #b of file #a. This is a no-operation command for disks as currently used under DOS. "a" refers to the symbolic file designator and is described in section (A) above. "b" is a simple Integer variable or constant not greater than 32767.

C) READ (a'b) list
    WRITE (a'b) list

Read or write the b-th record of file a. "a" and "b" are as described above.

An I/O statement may optionally specify End-of-file and/or error conditions as shown in this example:

```
READ (1,100,END=10,ERR=20)LIST
```

where END=10 specifies statement 10 for End-of-file processing and ERR=20 specifies statement 20 for error processing.

D) ENCODE (cnt,fmt,array) list
   DECODE (cnt,fmt,array) list

Encode or Decode "cnt" characters from "array" using the format "fmt".

END= and/or ERR= conditions may be optionally specified but have no meaning and will cause no actions.

<7.2.1> Specification Statements.

Alternate forms of Type statements may be as follows(1130):

```
BYTE = LOGICAL*1
INTEGER = INTEGER*2
REAL = REAL*4
DOUBLE = REAL*8
```

An IMPLICIT type statement is also allowed. It causes all variables beginning with some specified letter to be considered as a given type, unless explicitly stated otherwise.

EXAMPLE:

```
IMPLICIT REAL*4 (M-P,R)
INTEGER MRR
```

Causes variable MRR to be an integer, while all other variables beginning with M-P and R are treated as real.

E.G. - STANDARD FORTRAN WOULD IMPLY:

```
IMPLICIT REAL*4 (A-H,O-Z)
IMPLICIT INTEGER*2 (I-N)
```

<7.2.3> FORMATS.

The "O" (octal) field specification is allowed.

The "T" (TAB) specification is allowed.

An alternate form of the H specification is a string of characters surrounded by single quotes.

7.1.2 Statement Order Restrictions

Statements must occur in the order specified in the ANSI specification except that all DATA statements must occur after all other declaratives except ASF's.

ASF's must occur after all other declaratives and before any executables.

The IMPLICIT statement, when used, must be the first statement of any routine. It may be preceded only by the SUBROUTINE or FUNCTION statement.

7.2 Object Language Output

Any statement generating executable code will also generate a preamble consisting of the line number (see section 4.1.2) if any followed by:

```
$SEQ,nnnnnn
```

where nnnnnn is the numeric sequence of the statement in the program. $SEQ is a call to the trace-back handler which keeps track of where execution is occurring within the user program. If the /SU switch (section 3.2.1) is specified this preamble is not generated.

7.2.1 SUBROUTINE, FUNCTION Statements

The SUBROUTINE and FUNCTION statements cause internal flags and counters to be set describing:

A) The number of parameters and their position
B) Whether this is a function or a subroutine
C) The routine name and type

A non-fatal diagnostic is given if the statement has a line number.

The output generated is:

```
                .TITLE  XXX
                .CSECT
                .GLOBL  XXX
        XXX:    JSR     %4,$POLSH
                .GLOBL  $NAM,$POLSH,$SEQ
                $NAM,0,0,NNNNNN,MMMMMM
```

where XXX is the name of the routine. A main program is handled in the same fashion, except that the routine name is defined to be "MAIN.". The routine call $NAM is a Trace-Back function having four parameters, the final two (NNNNNN and MMMMMM) of which is the routine name in RADIX50. If the /SU switch is specified, the $SEQ global is deleted.

7.2.2 EXTERNAL Statement

A) Routine name cannot have been previously defined.
B) Doesn't allow a line number.

The output generated consists of a .GLOBL for every name declared external which is not a formal parameter. A flag is also set in the symbol table marking the entry as externally defined.

7.2.3 CALL

CALL XXX(X1,X2,X3,...XN) compiles as:

```
                .+2
                .GLOBL   XXX
                JSR      %5,XXX
                BR       $FNNNN
                X1
                X2
                .
                .
                .
                XN
        $FNNNN:
                JSR      %4,$POLSH
```

where XXX is the name of the subroutine to be called and X1, X2, ... XN are the arguments of the list.

In the case of compound parameters, the value is placed on the stack and its address is inserted in the list using the routine $SVSP. Upon return, the stack is cleared with an ADD.

Example:

```
            CALL ABC (A,B+C,D)
```

would generate the following:

```
                $P0002       ;PUSH VALUE OF B
                $P0003       ;PUSH VALUE OF C
                $ADF         ;ADD THEM
                $SVSP,$F0001 ;SAVE THE ADDRESS
                .+2
                .GLOBL    ABC
                JSR      %5,ABC
                BR       .+10
                A
       $F0001:  B
                D
                ADD      #4,%6
                JSR      %4,$POLSH
```

7.2.4 RETURN

Generates:

```
$RET
```

In a main program, a diagnostic is issued for any occurrence of RETURN. In a function subprogram, code is also generated to place the function result in R0-R3 before executing the $RET.

7.2.6 GOTO XXX

Generates:

```
$TR,.XXX
```

where XXX is the statement number in question and $TR is a POLISH mode jump.

Example:

```
GOTO 237

WOULD GENERATE:

$TR,.237
```

The computed GOTO form:

```
GOTO (10,20,30),I
```

compiles as

```
$P.I            ;VALUE OF I TO STACK
$TRX,3          ;COMPUTED GOTO ROUTINE
                ;AND NUMBER OF LABELS
.10             ;FOLLOWED BY ALL OF
.20             ;THE LABELS
.30             ;IN QUESTION
```

The simple assigned GOTO:

```
GOTO J
```

compiles as:

```
$P.J            ;VALUE OF J TO STACK
$TRA            ;SERVICE ROUTINE
```

Assigned GOTOs with a list

```
GOTO J,(10,20,30)
```

compile as

```
$P.J            ;VALUE OF J TO STACK
$TRAL           ;ASSIGNED GOTO
.10             ;LABEL STRING
.20             ;TERMINATED
.30             ;BY A
0               ;ZERO
```

7.2.6 ASSIGN

ASSIGN 15 TO I would generate

```
$AS,.15,I
```

if I is a local variable or

```
$ASP,.15,N
```

if I is a dummy argument with a parameter offser of "N".

When this statement is encountered, a flag is set to disallow using the variable assigned as a parameter in a call statement(or function call) and to disallow its use in arithmetic calculations.

7.2.7 CONTINUE

Generates no code!!

7.2.8 PAUSE XXX

Generates:

```
.GLOBL  $PAUSE
$PAUSE,XXX
```

where XXX is the octal constant to be printed on the console.

7.2.9 STOP XXX

Generates:

```
                .GLOBL  $STOP
                $STOP,XXX
```

Where XXX is the octal constant to be printed on the console.

7.2.10 FORMAT

10 FORMAT (XXXX) would compile as:

```
        .10:    $TR,$FNNNN
        $10:    .ASCII    ^(xxxx)^
                .EVEN
        $FNNNN:
```

Where XXXX is the contents of the FORMAT statement.

7.2.11 ENDFILE

Would generate:

```
                .GLOBL  $ENDFL
                $Pnnnn
                $ENDFL
```

Where nnnn is the desired unit number.

7.2.12 REWIND

As in 7.2.11 except use the routine "$RWIND".

7.2.13 BACKSPACE

As in 7.2.11 except use the routine "$BCKSP".

7.2.14 END

Generates a "RETURN" followed by all constants, data, and variables for the routine followed by a .END statement.

In a main program, a:

```
            .GLOBL  $EXIT
            $EXIT
```

Is generated instead of a "RETURN".

7.2.15 TYPE statements

This routine is entered with the type in R0.

The TYPE processor will make symbol table entries for each variable not already in the symbol table, and will set fields & indicators as follows:

```
    DATA TYPE
    DIMENSION (IF SPECIFIED BY PARENTHESES)
    LENGTH OF ITEM
    ADJUSTABLE ARRAY (IF A VARIABLE APPEARS BETWEEN
    PARENTHESES)
    SYMBOL
```

7.2.16 DIMENSION statements

This processor makes symbol table entries for each variable not already in the symbol table, and sets symbol table fields as follows:

```
DATA TYPE (IMPLICIT OFF IF NOT ALREADY DEFINED)
DIMENSION
LENGTH OF DATA ITEM (IMPLICIT IF NOT ALREADY DEFINED)
ADJUSTABLE ARRAY
SYMBOL
```

The "END" processor will output to the assembler all the ADB's in object-time format (see section 3.1.2.6) and will reserve space for all arrays.

7.2.17 COMMON

Each list item is placed in the main symbol table (if it is not already there) and, if dimensions are specified, appropriate ADB items are produced (see section 7.2.16).

The definitions will be made via the "=" operator in PAL-11R and will be relative to the base of the appropriately named CSECT.

7.2.18 EQUIVALENCE

7.2.18.1 The general form of the EQUIVALENCE statement is:

EQUIVALENCE (A1(I1),A2(I2)...AN(IN),(B1(J1),...),...

Where the "A" terms are equivalenced to each other, the "B" terms are equivalenced to each other, etc.

The "A" terms are array identifiers or single variables. For array identifiers the "I" terms are constant subscripts. The compiler requires the byte position of each item in the equivalence list, and will therefore replace the "I" term by the equivalent "index value" of the item. The index value is the address of the item relative to the start of the array is computed just as at object time (see section 3.1.2.6).

If a term is a simple variable, the I term will not appear in the source, and the compiler will take it to be 0.

In what follows the symbols I, I1, ..., IM, etc., indicate "index values". We then define the "offset" of AK:

OFFSET AK = MAX(IM) - IK

where I1, I2, ..., IM now denote the index values of the items, we can then say that:

AK(1) IS EQUIVALENT TO AI (OFFSET(AK))

Where the subscript I has been chosen such that IT=MAX(IM). This provides an equivalence between the start of each array and some relative byte position within one of the arrays, namely the array AI.

The actual equivalencing would take into account the size of an entry in AI.

7.2.18.2 When the compiler encounters an EQUIVALENCE statement the following actions will be taken:

A symbol table entry will be located or constructed for each item in the string, and the Equivalence bit will be set (section 5.2.1.1).

An "Equivalence block" for each set of equivalenced items will be established. The format of the equivalence block is defined in section 5.5.

7.2.18.3 After the last declarative statement is encountered, code will be generated to reserve space for all equivalenced variables. All equivalence chains will be resolved by means of the equivalance blocks, using an algorithm similar to that described in Arden, Galler and Graham, "AN ALGORITHM FOR EQUIVALENCE DECLARATIONS", ACM COMM., VOL 4, NO. 7.

7.2.19 Data statements DATA K1/D1/,K2/D2/,...

Processing of this statement assumes that no contradictory typing of the variables in the statement will occur afterwards.

As with the DIMENSION and COMMON processors, the variables in each list K are entered into symbol table if they are not already there (references to array elements with fixed subscripts will cause arrays to be defined with all dimension sizes=-1). In addition, a temporary list will be constructed consisting of pointers into the symbol table, one for each list item.

The constant data following the slash will then be matched to the list items and origined appropriately.

7.2.20 DO 10 I=J,K,L

Compiles as follows:

```
$P.J    ;PUSH J
$POP2,I ;POP TO I
$L:
```

A "DO TABLE" entry is constructed upon encountering the DO statement (see section 5.7). The "return jump" destination is the label generated following the MOV above. If there are any entries active (see section 5.7.1) in the DO table, it is searched after the code for each labelled statement is generated. When a match between statement label and terminal statement label occurs, two possibilities exist:

1. The corresponding DO table entry is not the last active entry in the table - in this case the DO's are not properly nested.

2. The corresponding DO table entry is the last active entry in the table - in this case the following code is generated:

```
$ENDDO,L,I,K,$L      for normal DO loops
      or
$ENDDP,L,I,K,$L      for DO loops con-
                     taining formal par-
                     ameters
```

7.2.21 IF

7.2.21.1 Logical IF

The logical expression within the outer parentheses is evaluated and the result is passed to the OTS "logical IF" routine. If the expression is "true" the conditional branch or statement is executed. If "false" the next FORTRAN statement is executed.

7.2.21.2 Arithmetic IF

The expression within the outer parentheses is evaluated and the result is passed to the OTS "arithmetic IF" routine. It returns indirectly to one of the three locations following the calling sequence.

See section 3.1.2.2.

7.2.22 READ, WRITE

Object code for READs and WRITEs is handled in a uniform manner, and is based on the use of an "initialize" call, several reads or writes to transmit list items to the OTS routines, and an "End" call when the list is depleted (see section 3.1.2.4).

The method for generating code for a typical formatted read is given as an example:

```
READ (M,100)A,B,(X(I),I=J,K,L)
```

A. Upon encountering the first right parenthesis, the compiler generates the initialization call:

```
$PSH,M
$PSH,$100
$INFI,0,2
```

B. If the next item in the statement is not a left parenthesis, the compiler puts a pointer to each list item into the read calling sequence, until a left parenthesis or end of statement is found:

```
$PSH,A
$PSH,B
$PSH,2
$IOR
```

C. When a left parenthesis is encountered the compiler looks ahead for the implied DO parameters, generates a label for the DO return jump destination ($F0002 in this example) and makes an entry in the DO table for an implied do-loop.

The DO processor is used to generate the loop initialization:

```
$P.J
$POP1,I
```

D. The list items are then passed to the READ routine as in part B (except that a subscripted item now appears in the list)

```
$F0002: <CALL TO SUBSCRIPT ROUTINE FOR X(I)>
        $PSHR1  ;PUSH REGISTER ON STACK
        $P.1
        $IOR
```

E. When the DO control variables are encountered in the list they are scanned over until a right parenthesis is found. The right parenthesis triggers a call to the DO processor to generate looping instructions from information on top of the DO table stack:

```
$ENDDO,L,I,K,$F0002
```

F. The DO loop finally falls through for transmission of the next list item; in this case there are no more items so the compiler generates code to terminate the operation:

```
$IOF
```

7.2.23 DEFINE FILE a(m,l,u,v)

See section 7.1.3. The generated code is:

```
$DEFIL,a,m,l,v
```

where a, m, l, and v are the addresses of the proper parameters.

7.2.24 FIND (u'b)

See section 7.1.3. The generated code is:

```
$PSH,u
$PSH,b
$FIND,0,0
```

7.2.25 IMPLICIT

The compiler will maintain a table from which all Implicit definitions of variables are made (see section 5.6).

This routine will adjust entries in both the implicit table and the symbol table according to the intent of the implicit statement.

7.2.26 BLOCK DATA

When this statement is encountered the compiler sets the Block Data switch, to be tested whenever an executable is encountered in the source program (an error).

7.2.27 Adjustable Arrays

When adjustable arrays are encountered in TYPE, DIMENSION or COMMON statements, the compiler will generate code to initialize the array by moving values of the appropriate parameters into the object time ADB. The call is of the form:

```
$ADJ,<ADB ADDR.>,<PAR. INDEX>,<1ST DIM>,<2ND
 DIM>,<3RD DIM>
```

7.2.28 Arithmetic Statement Functions

Arithmetic Statement Functions (ASF's) are compiled as standard functions except that the entry name is "non-.GLOBL."

The entry name has entry type 01.

Routine argument(s) have entry type 03 and are designated parameters as in subroutines or functions.

Deletion of arguments consists of zeroing the name part of the symbol table entry. The space is not reclaimed nor the entry unlinked - but no search will match the zero name.

A variable length table on the stack maintains a table of pointers to the symbol table entries for the parameters. At the end of the ASF compilation this allows going back and deleting argument names from the symbol table. The top of the stack contains the number of arguments. The next N words are the arguments (LIFO).

The structure of the compiled function is as follows:

```
            $TR,$Fnnnn            ;BRANCH AROUND THE
 ROUTINE
NAME:    JSR   %4,$POLSH        ;ROUTINE TO ENTER POLISH
```

```
[code for expression]             ;LEAVES VALUE ON STACK
        $POPRn                    ;MOVE VALUE INTO
 REGISTERS
        .+2                       ;EXIT POLISH
        RTS     %5                ;EXIT FUNCTION
$Fnnnn:                           ;TARGET FOR BRANCH
```

7.2.29 ENCODE, DECODE

Object code for ENCODE and DECODE is handled identically to READ and WRITE, except that three parameters are pushed and the routines called are $ENCD and $DECD respectively. (See section 3.1.2.4.)

7.3 Object Time System Exceptions and Differences From ANSI FORTRAN.

7.3.1 Library

The following FORTRAN library routines have been added to the ANSI list (see the user's manual for more detailed information):

1. DATE - returns the current date.
2. TIME - returns the current time of day.
3. SSWTCH - returns the contents of the switch register.
4. RAN - random number generator function.
5. RANDU - random number generator subroutine.
6. SETFIL - modify default device table entries.
7. PDUMP - dump core between specified limits.
8. EXIT - terminate program.
9. SETERR - modify default error handling.
10. TSTERR - test if an error has occurred.
11. LINK, RETURN - overlay handling.
12. RUN - load and execute another program.

7.3.2 FORTRAN overlays

As described in the OTS specification, 130-311-002.

7.3.3 Random access I/O

As described in the Object Time System specification, 130-311-002.

7.4 Code Generation Example

The following is an example of a FORTRAN program listing which includes the assembly listing. This listing is not exactly like a normal program listing because the normal page headings have been removed to reduce confusion, and because descriptive comments have been added to the assembly listing to describe what is going on and why.

```
        C  ASSEMBLY CODE EXAMPLE
0001             DIMENSION X(10),Y(10)
0002             COMMON X
0003             EQUIVALENCE (X,Y)
0004             DATA A/1.0/,I,J/1,2/
0005             ASF(Q)=Q+1.
0006             D=1.
0007             B=D+1-A
0008             C=ASF(B)
0009    10       X1=SIN(C)
0010             X2=(A-(B**2-4.*A*C))/(2.*A)
0011             WRITE (1,100)A,B,(X(I),I=3,8)
0012    100      FORMAT (8F10.3)
0013             IF (A.EQ.1.0)GO TO 10
0014             STOP 123
0015             END

        ROUTINES CALLED:
        SIN

        SWITCHES = /LI

        BLOCK      LENGTH
        MAIN.   205    (000632)*
        .$$$$.  20     (000050)
```

```
                ;C  ASSEMBLY CODE EXAMPLE
                ;         DIMENSION X(10),Y(10)
                          .TITLE  MAIN.
                ;         FORTRAN  V004A.05
       000000'            .CSECT
                          .GLOBL  MAIN.
 000000 004467 MAIN.:     JSR     %4,$POLSH         ;ENTER POLISH MODE
                          .GLOBL  $POLSH,$NAM
 000004 000000G           $NAM,0,0,050561,055740   ;SET UP TRACEBACK LINKAGE
                          .GLOBL  $SEQ
                ;         COMMON X
                ;         EQUIVALENCE (X,Y)
```

```
                 ;          DATA A/1.0/,I,J/1,2/
          000000'           .CSECT  .$$$$.            ;ALLOCATE BLANK COMMON
          000000'X=.+000000
          000050'.=.+000050
                            .EVEN
          000016'           .CSECT
          000050'           .CSECT  .$$$$.            ;SET UP EQUIVALENCES
          000020'Y          =X+000020
          000050'.          =X+000050
          000016'           .CSECT
                            .GLOBL  $TR
 000016 000000G             $TR,$F0001                ;JUMP AROUND DATA
ALLOCATION
          000022'A          =.
          000022'.          =A
 000022 040200              .WORD   040200,000000     ;A HAS A VALUE OF 1.0
          000026'.          =A+000004
          000026'I          =.
          000026'.          =I
 000026 000001              .WORD   000001            ;I HAS A VALUE OF 1
 000030 000000              0
          000032'.          =I+000004
          000032'J          =.
          000032'.          =J
 000032000002  .WORD         000002                   ;J HAS A VALUE OF 2
 000034 000000              0
          000036'.          =J+000004
                            .EVEN
          000036'           .CSECT
 000036          $F0001:                              ;END OF DATA AREA
                 ;          ASF(Q)=Q+1.
 000036 000000G             $TR.$F0002                ;JUMP AROUND THE ASF
 000042 004467 ASF:         JSR     %4,$POLSH         ;ENTER POLISH MODE
 000046 000454'             $P0010                    ;PUSH Q
 000050 000462'             $P0011                    ;PUSH 1
                            .GLOBL  $ADR
 000052 000000G             $ADR                      ;ADD 1 TO Q
                            .GLOBL  $POPR3
 000054 000000G             $POPR3                    ;POP THE RESULT
 000056 000060'             .+2                       ;EXIT POLISH MODE
 000060 000205              RTS     %5                ;RETURN TO CALLER
 000062          $F0002:
                 ;          D=1.
 000062 000000G             $SEQ,000006               ;TRACEBACK SEQUENCE 6


 000066 000462'             $P0011                    ;GET 1
                            .GLOBL  $POP3
 000070 000000G             $POP3,D                   ;STORE IT IN D
                 ;          B=D+1-A
 000074 000000G             $SEQ,000007               ;TRACEBACK SEQUENCE 7
 000100 000474'             $P0012                    ;GET D
```

```
 000102 000506'          $P0013                  ;GET INTEGER 1
                         .GLOBL  $IR
 000104 000000G          $IR                     ;CONVERT TO REAL
 000106 000000G          $ADR                    ;ADD 1 TO D
 000110 000440'          $P0003                  ;GET A
                         .GLOBL  $SBR
 000112 000000G          $SBR                    ;SUBTRACT IT
 000114 000000G          $POP3,B                 ;STORE RESULT IN B
                ;        C=ASF(B)
 000120 000000G          $SEQ,000010             ;TRACEBACK SEQUENCE 8
 000124 000126'          .+2                     ;EXIT POLISH MODE
 000126 004567           JSR     %5,ASF          ;CALL ASF
 000132 000401           BR      .+000004        ;WITH THE PARAMETER
 000134 000524'          +B                      ;B
 000136 004467           JSR     %4,$POLSH       ;RE-ENTER POLISH MODE
                         .GLOBL  $PSHR3
 000142 000000G          $PSHR3                  ;PUT RESULT ON STACK
 000144 000000G          $POP3,C                 ;RESULT GOES TO C
                ;10      X1=SIN(C)
 000150 000000G.10:      $SEQ,000011             ;TRACEBACK SEQUENCE 9
                         .GLOBL  SIN
 000154 000156'          .+2                     ;EXIT POLISH
 000156 004567           JSR     %5,SIN          ;CALL SIN
 000162 000401           BR      .+000004
 000164 000536'          +C
 000166 004467           JSR     %4,$POLSH       ;RE-ENTER POLISH
 000172 000000G          $PSHR3                  ;PUT THE RESULT
 000174 000000G          $POP3,X1                ;IN X1
                ;        X2=(A-(B**2-4.*A*C))/(2.*A)
 000200 000000G          $SEQ,000012             ;TRACEBACK SEQUENCE 10
 000204 000440'          $P0003                  ;PUSH A
 000206 000516'          $P0014                  ;PUSH B
 000210 000546'          $P0020                  ;PUSH 2
                         .GLOBL  $PWRI
 000212 000000G          $PWRI                   ;SQUARE B
 000214 000556'          $P0021                  ;PUSH 4
 000216 000440'          $P0003                  ;PUSH A
                         .GLOBL  $MLR
 000220 000000G          $MLR                    ;MULTIPLY 4.*A
 000222 000530'          $P0015                  ;PUSH C
 000224 000000G          $MLR                    ;MULTIPLY IT
 000226 000000G          $SBR                    ;SUBTRACT B**2
 000230 000000G          $SBR                    ;SUBTRACT FROM A
 000232 000570'          $P0022                  ;PUSH 2
 000234 000440'          $P0003                  ;PUSH A
 000236 000000G          $MLR                    ;MULTIPLY 2.*A
                         .GLOBL  $DVR
 000240 000000G          $DVR                    ;DIVIDE THE TWO EXPRESSIONS
 000242 000000G          $POP3,X2                ;PUT THE RESULT IN X2
                ;        WRITE (1,100)A,B,(X(I),I=3,8)
 000246 000000G          $SEQ,000013             ;TRACEBACK SEQUENCE 11
                         .GLOBL  $PSH
 000252 000000G          $PSH,$I0002             ;ADDRESS OF UNIT NUMBER
```

```
000256 000000G        $PSH,$100               ;ADDRESS OF FORMAT
                      .GLOBL  $OUTFI
000262 000000G        $OUTFI          ;INITIALIZE FORMATTED OUTPUT,0,0
000270 000000G        $PSH,A                  ;GET ADDRESS OF A
000274 000000G        $PSH,B                  ;GET ADDRESS OF B
000300 000000G        $PSH,000002             ;2 PARS. TO BE OUTPUT
                      .GLOBL  $IOR
000304 000000G        $IOR                    ;OUTPUT THE REAL PARAMETERS
000306 000606'        $P0024                  ;PUSH 3
                      .GLOBL  $POP2
000310 000000G        $POP2,I                 ;STORE IT IN I
000314        $F0003:
000314 000446'        $P0004                  ;PUSH I
                      .GLOBL  $SBS1
000316 000000G        $SBS1,$A0001            ;COMPUTE SUBSCRIPT ADDRESS
                      .GLOBL  $PSHR1
000322 000000G        $PSHR1                  ;PUT I'TH ITEM ON STACK
000324 000000G        $PSH,000001             ;ONE PARAMETER
000330 000000G        $IOR                    ;DO REAL OUTPUT
                      .GLOBL  $ENDDO
000332 000000G        $ENDDO,$I0002,I,$I0006,$F0003 ;TERM. LOOP
                      .GLOBL  $IOF
000344 000000G        $IOF                    ;I/O TERMINATION
              ;100    FORMAT (8F10.3)
000346 000000G.100:   $SEQ,000014             ;TRACEBACK SEQUENCE 12
000352 000000G        $TR,$F0004              ;SKIP AROUND FORMAT
000356    050 $100:   .ASCII  (8F10.3)        ;ASCII FORMAT STRING
                      .EVEN
000366        $F0004:
              ;       IF (A.EQ.1.0)GO TO 10
000366 000000G        $SEQ,000015             ;TRACEBACK SEQUENCE 13
000372 000440'        $P0003                  ;PUSH A
000374 000462'        $P0011                  ;PUSH 1.0
                      .GLOBL  $CMR
000376 000000G        $CMR                    ;COMPARE THE TWO
                      .GLOBL  $EQ
000400 000000G        $EQ                     ;DO EQUALITY CHECK
                      .GLOBL  $TRTST
000402 000000G        $TRTST                  ;SKIP
000404 000412'        $F0005                  ;IF FALSE
000406 000000G        $TR                     ;GO TO STATEMENT 10
000410 000150'        .10
000412        $F0005:
              ;       STOP 123
000412 000000G        $SEQ,000016             ;TRACEBACK SEQUENCE 14
                      .GLOBL  $STOP
000416 000000G        $STOP                   ;TERMINATE PROGRAM
000420    061         .ASCII  123
000423    000         .BYTE 0
                      .EVEN
              ;       END
```

```
000424 000000'$A0001: +X                             ;ADR FOR ARRAY X
000426 054004         054004
000430 000012         000012
000432 000000'$A0002: +Y                             ;ADR FOR ARRAY Y
000434 054004         054004
000436 000012         000012
000440 012700 $P0003: MOV      A+4,%0                ;PUSH ROUTINE FOR A
000444 000467         BR      $F0006
000446 016746 $P0004: MOV     I,-(%6)                ;PUSH ROUTINE FOR I
000452 000134         JMP     @(%4)+
000454 016500 $P0010: MOV     000002(%5),%0          ;PUSH ASF PARAMETER
000460 000457         BR      $F0006-4
000462 012700 $P0011: MOV      $R0001+4,%0           ;PUSH ROUTINE FOR 1.0
000466 000456         BR      $F0006
000470 040200 $R0001: 040200
000472 000000         000000
000474 012700 $P0012: MOV      D+4,%0                ;PUSH ROUTINE FOR D
000500 000451         BR      $F0006
000502 000000 D:      0,0
000506 012746 $P0013: MOV      000001,-(%6)          ;PUSH ROUTINE FOR 1
000512 000134         JMP     @(%4)+
000514 000001 $I0002: 000001
000516 012700 $P0014: MOV      B+4,%0                ;PUSH ROUTINE FOR B
000522 000440         BR      $F0006
000524 000000 B:      0,0
000530 012700 $P0015: MOV      C+4,%0                ;PUSH ROUTINE FOR C
000534 000433         BR      $F0006
000536 000000 C:      0,0
000542 000000 X1:     0,0
000546 012746 $P0020: MOV      000002,-(%6)          ;PUSH ROUTINE FOR 2
000552 000134         JMP     @(%4)+
000554 000002 $I0003: 000002
000556 012700 $P0021: MOV      $R0003+4,%0           ;PUSH ROUTINE FOR 4
000562 000420         BR      $F0006
000564 040600 $R0003: 040600
000566 000000         000000
000570 012700 $P0022: MOV      $R0005+4,%0           ;PUSH ROUTINE FOR 2
000574 000413         BR      $F0006
000576 040400 $R0005: 040400
000600 000000         000000
000602 000000 X2:     0,0
000606 012746 $P0024: MOV      000003,-(%6)          ;PUSH ROUTINE FOR 3
000612 000134         JMP     @(%4)+
000614 000003 $I0005: 000003
000616 000010 $I0006: 000010
000620 062700         ADD      4,%0                  ;PUSH SERVICE ROUTINE
000624 014046 $F0006: MOV     -(%0),-(%6)
000626 014046         MOV     -(%0),-(%6)
000630 000134         JMP     @(%4)+
                      .GLOBL  $WRITE                 ;I/O
                      .GLOBL  $DCO                   ;LINKAGE
                      .GLOBL  $OTSV                  ;GLOBALS
       000000'        .END    MAIN.
```

```
A     = 000022R             ASF      000042R            B        000524R
C        000536R            D        000502R            I     = 000026R
J     = 000032R             MAIN.    000000RG           SIN   = 000000 G
X     = 000000R      002    X1       000542R            X2       000602R
Y     = 000000R      002    $ADR  = 000000 G            $A0001   000424R
$A0002   000432R            $CMR  = 000000 G            $DCO  = 000000 G
$DVR  = 000000 G            $ENDDO= 000000 G            $EQ   = 000000 G
$F0001   000036R            $F0002   000062R            $F0003   000314R
$F0004   000366R            $F0005   000412R            $F0006   000624R
$IOF  = 000000 G            $IOR  = 000000 G            $IR   = 000000 G
$I0002   000514R            $I0003   000554R            $I0005   000614R
$I0006   000616R            $MLR  = 000000 G            $NAM  = 000000 G
$OTSV = 000000 G            $OUTFI= 000000 G            $POLSH= 000000 G
$POPR3= 000000 G            $POP2 = 000000 G            $POP3 = 000000 G
$PSH  = 000000 G            $PSHR1= 000000 G            $PSHR3= 000000 G
$PWRI = 000000 G            $P0003   000440R            $P0004   000446R
$P0010   000454R            $P0011   000462R            $P0012   000474R
$P0013   000506R            $P0014   000516R            $P0015   000530R
$P0020   000546R            $P0021   000556R            $P0022   000570R
$P0024   000606R            $R0001   000470R            $R0003   000564R
$R0005   000576R            $SBR  = 000000 G            $SBS1 = 000000 G
$SEQ  = 000000 G            $STOP = 000000 G            $TR   = 000000 G
$TRTST= 000000 G            $WRITE= 000000 G            $100     000356R
.10      000150R            .100     000346R

        **COMPILER ----- CORE**
           PHASE        USED  FREE
        DECLARATIVES 00366 17853
        EXECUTABLES  00620 17599
        ASSEMBLY     01114 19968
```

8.0 COMMAND LANGUAGE AND STRUCTURE

The command input to the compiler happens as follows:

Upon typing the command RU FORTRN, the compiler is loaded and the compiler will type its name followed by two spaces followed by the compiler version number. On the next line a # is typed to signify that FORTRAN is ready to accept command input.

The command input typed must be of the form:

```
OBJECT-FILE,LIST-FILE<INPUT-FILE
```

The object file is the main compiler output file. The format of the file may be of one of two different types. If the /AS switch is specified as part of the file specification, the output will be an ASCII file suitable for assembly by the assembler. If the switch is not specified, the output will be object output suitable for linking. The default extension is .OBJ, unless the /AS switch is specified which has a .PAL default extension. If more than one FORTRAN routine was in the input file, the object file will contain an equivalent number of object routines and must be linked using the linker's /CC switch. Note that if /AS is specified, only the first FORTRAN source routine in the file will be compiled, any additional routines will be ignored. This is done because the assembler will not take concatenated sources as input.

The list file is used for the source listing, object listing, symbol table listing, and error diagnostics if any. The listing content may be made as comprehensive as desired by use of the /LI switch (12K compiler only). The switch should be specified with a value from 0 to 3. Specifying /LI:0 will give the minimal listing which consists only of any error diagnostics which occur and the block descriptor (section 4.11) which describes the program and common block sizes. The /LI:1 switch is the default option. This supplies a source listing with error diagnostics and the block descriptor. The /LI:2 switch is used if the assembly listing is desired in addition. Specifying /LI:3 gives a listing containing everything above plus the complete assembly symbol table listing.

The input file may contain one or more FORTRAN programs which are to be compiled. Note that if the /AS switch is set on the object file (or if the compiler is the 8K version) that all programs after the first will be ignored. Under normal circumstances, however, all routines in the file will be compiled. See section 3.2.1 for a more complete description of switch handling.

Upon the completion of compiling a file, FORTRAN will, if there are any errors, type the error count, and then return to the command handler and again type the # sign to signify that it is ready to start another compile.

When using the /AS and /LI switches, it is necessary to specify the switch each time the user types in a command. Be warned though, that all input switches, once set, stay set until the compiler is either REstarted or reloaded. This occurs in this fashion because it is assumed that the user will probably compile all programs in a given run with the same code generation features selected.

9.0 OPERATING PROCEDURES

This is described in detail in the "Getting on the Air with FORTRAN" document.

10.0 PHYSICAL DESCRIPTION AND ORGANIZATION

10.1 Compiler module descriptions

ASC1 - This routine is the control routine for CALL statements and arithmetic assignment statements. If it is a CALL statement, entry is made at location CALL. The first thing done is to generate the label, if any is needed. Then the routine name is obtained via a call to GET. Once the routine name is obtained and checked for legality, a call is made to FUN000 (in EVALU) and the remainder of the statement is evaluated as if it were a function call (which it is, sort of). Upon return from this evaluation, EXPGEN (also in EVALU) is called to generate the actual code from the Polish string which FUN000 placed on the stack. When EXPGEN is finished, the stack is cleared, end of line termination is checked, and a return is made to the calling routine.

On the other hand, if an entry is made to ASGN, it is assumed that an assignment statement is in the offing. This routine first checks for the existence of a left part terminated by a "=" sign. If this exists SUBEXP (in EVALU) is called to convert the right part of the expression to Polish. Note that if it is ever desired to allow multiple assignments, the code for the preliminary part should be placed immediately preceding this call. Upon return from this routine, EXPGEN is called to generate the actual Polish code itself and then the stack is cleared as usual. If at this point, the line terminator is zero, the processor resets the line pointer to the start and proceeds to generate the necessary code for the left part of the expression, making special exceptions of course for subscripted references. After the left part is complete, a normal exit is taken. Note that the assignment processor never takes the nonrecognition exit that the other processors are allowed to use since if it isn't an assignment, it can't be anything else!

ASC2 - This routine is used by ASC1 and DO. It is used only to generate the common POP code which is required in the simple case assignment and the DO setup. This routine does all necessary checking for parameter forms in subroutines.

ASF - This routine handles all Arithmetic Statement Functions. Upon completion of other declaratives, this routine will be called and will retain control until all ASF's have been processed. When ASF is entered, the first function it performs is to determine the validity of the line as an ASF, in other words, does the entity on the left side of the = sign consist of a previously undeclared name with one or more parameters. If this criterion is met, the ASF processing is invoked. The routine first sets up dummy parameters for the use of the function which will not conflict with later usages in the compiler. Once all arguments have been collected, the expression is compiled in the same form as a FORTRAN compiled function, except that when it is referenced in the program, no .GLOBL will be generated, thus making it a strictly local function. The normal expression handling routines SUBEXP and EXPGEN are used as part of this procedure. When the function is complete, the temporary entries in the symbol table are deleted to remove any possibility of later conflicts.

COMMON - Handles the declaratives COMMON and EQUIVALENCE. It also contains the subroutine ALOCAT which is called after the last declarative to allocate any declared variables, etc. When COMMON is entered, the first action which occurs is to see if the data allocations have already occurred. If so, the statement has been placed incorectly and the process is immediately terminated with a diagnostic. Otherwise, the processor proceeds to check for an (optional) block name followed by one or more variable or array names. Note that little conflict checking is done here and won't happen until the allocation itself is attempted. For each entity found, an entry is made in the COMMON/EQUIVALENCE table (see section 5). After each "," terminator a check is made for a new block name, and if found this whole procedure is started over. Termination occurs when the end of line terminator is encountered.

Another routine in this module is ALOCOM which is called by ALOCAT to do the actual code generation associated with the common declarations. Most conflict situations will be caught here which could not be caught by the COMMON processor itself.

The EQUIVALENCE processor, when entered, tries to collect as many discrete groups of Equivalences as possible and places them in the COMMON/EQUIVALENCE table. The ALOCAT routine can be considered to be part of this process and is called after the last non-executable statement has been encountered. It flags any EQUIVALENCE inconsistencies, generates the necessary equates, and calls ALOCOM.

CORET - The CONTINUE and RETURN statements are processed here. Since CONTINUE is a null operation, all it does is check for a line terminator and immediately exit! The RETURN processor checks to see whether the statement was issued from a main program, subprogram, or function. If a main program, a diagnostic is issued. If a subroutine, a $RET is generated. If a function, the necessary pop code is generated followed by the $RET.

DATA - This routine processes DATA statements and generates the required code. When DATA is entered, a call to ALOCAT is immediately made because no data allocations can occur until all other declared allocations have been made. Then the data name list is scanned until a "/" is encountered. This gives the compiler two string pointers to work with, one to point to the variable names and the other to point to the value list which should match the names in characteristics. As each variable is encountered, the corresponding data value is assigned to it in the object code. There are several cases which are checked while the allocation is occurring. These are BLOCK DATA, previously allocated non-COMMON, yet to be allocated non-COMMON. Note that in BLOCK DATA that only Named Common can be initialized and that its allocation has already occurred in ALOCOM. The processing of the statement continues as long as there is at least a comma after the last encountered variable, a comma after the last encountered constant, a non-zero repeat count for a variable (array), or a non-zero repeat count for a constant. All standard forms are allowed as data values, including those preceded by a unary minus.

DECLAR - This is the syntax recognizer for declarative statements. Upon finding a form it cannot recognize, it calls ASF and then jumps to EXECUT. This routine is part of the main control loop of the compiler. It calls the I/O routines to get a line and then dispatches to the proper statement handler via a JSR PC,xxx. Each handler when entered, can assume that R1 is pointing to the correct position to begin its recognition scan (for most statements this is immediately following the verb). Each processor is, in general, responsible for the complete processing of the input line. Return from the handler is as follows: A normal return will come back with an RTS PC with the V-bit of the status word clear. A return with the V-bit set may only occur if the line in question could not be of the assumed type.

DEFINE - DEFINE FILE statements are processed here. The form DEFINE FILF A(M,L,U,V) is directly converted to a call to the OTS routine $DEFIL with the equivalent parameters.

DO - This routine does the DO statement processing and places the required entries in the DO table. All parameters are checked for having the proper characteristics of simple integer variables or constants as required. Then ASC2 is called to produce the necessary initialization code to set up the loop.

DOFIN - This routine is called whenever a labelled statement is completed to scan for possible DO terminations. It is also used by IOSTMT for implied DO handling in I/O statements. When called it searches the DO table for statement matches which may be either normal statement numbers as in the case of a DO termination or "special" form labels which are used as dummy labels in implied DO situations. In either case when a match is encountered, an $ENDDO or $ENDDP is generated depending on whether parameters are being used in the DO loop, the more efficient form being that which does not use parameters. After generating the required code, the DO entry which was completed is removed from the table.

ELOC - This is a dummy routine which is used to mark the end of an overlay in the compiler. It in combination with one of the header blocks establishes the range of the particular overlay in question.

ENDPRO - This routine handles all of the processing required upon completion of a compile. It generates the various arrays, variables, etc. used by the program which had not been previously set up by DATA, COMMON, or EQUIVALENCE. It also generates the .GLOBL references which are required by the OTS for proper I/O linkages.

ENDSTM - This is the END and ENDFILE statement recoginizer. If an ENDFILE is found, routine MSSIO is called. If an END is found, ENDPRO is called and the overlay stack is restored.

ERRLOC - Logs the errors occurring during a statement. Up to 10 errors will be stored in the error table for processing at the completion of the statement. Note that a special case check is done for error 43 which is the error associated with COMMON/EQUIVALENCE table overflow. If this error should be logged, a flag is set to inhibit any attempt at generating the table structures using ALOCAT and/or ALOCOM since catastrophic errors could occur with the tables incomplete as they will be in such case.

ERRPRT - This routine is called at the completion of any statement having errors. It prints the diagnostic numbers and diagnostic text on the source and object devices as specified in the table created by ERRLOC. It obtains the diagnostic text from the file FORCOM.DGN. The diagnostic number defines a character position within the file where the proper text may be found, and the proper 64 characters worth of information are extracted therefrom. Note that this routine assumes that at least 256 words of stack are available for temporary buffer usage while it is printing the diagnostic in question. Once all diagnostics in the list have been printed, the stack is cleared and the compilation is permitted to resume.

EVALU - Contains two routines. The first SUBEXP, takes an ASCII expression and converts it into an internal POLISH string. This POLISH string is stored on the stack and is modified as required as the expression in question is evaluated (the structure of individual items in this stack is described in section 5.9). Upon completion of the parse, SUBEXP returns with the complete expression form converted to POLISH mode and stored on the stack with a zero terminator. All operations have been defined to correspond to accepted FORTRAN usage. Note that this is the routine where the mixed mode conversions are defined and inserted into the string of operations. The second routine, EXPGEN, takes the internal POLISH string and uses it to generate code. All actual assembler labels and special flags are assigned during this phase of evaluation. Minimal error checking is done since the bulk of the error handling is done by SUBEXP. When EXPGEN is finished, a return is made to the calling routine. Be amply warned that the calling routine is held totally responsible for cleaning up the stack after EXPGEN and SUBEXP have messed it up.

EXECUT - This is the executable statement syntax recognizer. The job of this routine is to recognize and dispatch to all forms of executable statements (FORMATS included). Note that when this routine is first entered, care must be taken to not irrevocably divorce the processor from the non-executables since one of the reasons for entry may be only a badly typed non-executable statement. Thus, if a statement is found wanting at this level and the EXEC flag has not yet been set, it is necessary to issue the normal diagnostic and then return to the non-executable processor. If EXEC is set, though, the statement is discarded upon non-recognition and the next statement read and processed.

EXTERN - EXTERNAL statement processor. This processor takes the list of names supplied after the EXTERNAL declaration and sets them as externally defined in the symbol table. If the symbol is not a formal parameter, the name is also specified in a .GLOBL in the assembler output. Errors are given for names which cannot be used as external, like local variables, function names which have already been declared, etc.

FORMAT - This routine generates the ASCII strings for FORMAT statements. This processor makes no attempt to do syntax checking of the majority of the FORMAT. It is only capable of checking nesting correctness and Hollerith counts. The only function of note that is out of the ordinary is that flags are set for each conversion type which is used in the FORMAT. This is done so the routine ENDPRO may generate the proper globals for loading the desired portions of the OTS I/O processors.

FUNNAM - This routine, when called by SUBEXP, checks for special type requirements for function calls. For example, when called with the name

DABS, it will flag the name as double precision.

GENOVL - This routine is used to output a compiler overlay to the disk overlay file in image form. When this routine is used in the overlay 0 builder, it also generates the correct length overlay file. This routine is used only for overlay building, not for the normal compiler operation. It, as well as STRTUP can be considered to be only temporary routines which are used to get the show on the air.

GOTO - All forms of GOTO statements, as well as STOP, PAUSE, and ASSIGN, are handled here. The GOTO processor does all necessary syntax checking and then generates the necessary POLISH ops to handle Unconditional GOTO's, Computed, and Assigned GOTO's as well. Note that when a variable is used in an assigned form, it must have been previously used in an ASSIGN statement.

The ASSIGN processor checks the statement syntax and then flags the variable specified as having been blessed by an ASSIGN thus making the Assigned GOTO an eligible receiver for the particular variable.

The STOP and PAUSE statements are processed identically and expect only a Hex or Octal constant as a parameter.

HDR00 - 8K overlay 0 header block. The discussion here also applies without change to the routines HDR01 - HDR04 as mentioned below. The purpose of these headers is to preserve sufficient information to allow the overlay handler to load and execute a proper routine in an overlay. Information is also present which is used only at the overlay build time. The various items in the headers consist of a descriptor list which has the addresses of all of the entry points in the overlay and a byte table describing the overlay number, entry location and return characteristics of the entry it describes. The return characteristic, in short, specifies whether the routine described may be called with a JSR. If a JSR call is allowed, the subroutine called MUST return normally with an RTS otherwise the overlay handler's internal table will get fouled up. The last part of the routine consists of a series of dummy entries which correspond to the byte table and create the proper linkages for the calling routine. Note especially that the order of the dummy entries is exactly the reverse of the physical order of the byte table.

HDR01 - 8K overlay 1 header block.

HDR02 - 8K overlay 2 header block.

HDR03 - 8K overlay 3 header block.

HDR04 - 8K overlay 4 header block.

HDRGEN - This routine generates the start up header information in the assembler output file.

HEAD00 - 12K overlay 0 header block. The information in this and the three following blocks is organized similarly to that described by HDR00 except that the exact number and kind of entries are set up for the 12K structures. All of the above description is accurate in this case.

HEAD01 - 12K overlay 1 header block.

HEAD02 - 12K overlay 2 header block.

HEAD03 - 12K overlay 3 header block.

IF - All forms of IF statements are processed here. When IF gains control, it acts essentially as a sub-processor within the compiler. This is done because in the case of Logical IF forms, other statements may be used as part of the IF statement itself. Because of this, the processor is allowed to call most other statement processors as if it was the normal syntax scanner itself. This specialized control mode is only invoked in the case of Logical forms. Normal arithmetic forms expect the standard list of statememt labels following the ")" in the statement. IF requires both SUBEXP and EXPGEN which it uses to evaluate the expression. Arithmetic forms will call the expression handler only once while logical forms require it twice.

IMPLIC - The IMPLICIT statement processor. This processor resets the type definition table to reflect the arguments specified to the statement. It is also required to scan the symbol table which already exists, and re-type any items in it. Since the IMPLICIT statement must precede all statements except FUNCTION or SUBROUTINE statements, the only retyping necessary will be done on formal parameters which have been defined in the FUNCTION or SUBROUTINE statements themselves.

INIT - This routine is used to initialize a compile. It prints the heading, accepts the command input, presets the data area initializes the devices and then jumps to DECLAR. After the END statement is processed, it outputs the core summary and switch summary and then re-initializes the compiler.

IOPACK - Handles the compiler I/O interface to the monitor. This does all the dog work involved in continuation lines, input buffering while checking continuations, comment cards are bypassed, and the other tasks associated with input. For output it is used to produce the source and object listings as well as output the object file if any. The 12K version also has a routine which counts lines to place fancy headings on every page of listed output. The strings of characters fed to the routine for assembler input are blocked and output when a buffer load of them is obtained. In other words, this routine does all the good things that an I/O processor would be expected to handle.

IOSTMT - This routine processes all I/O statements except FORMAT, STOP, and PAUSE. This routine is broken down into three parts, that which processes the parenthesized part of the statement (as well as the special cases for READ and PRINT), that which processes simple I/O lists, and that which processes DO implied lists. The DO list processor is called from the simple list processor whenever a left parenthesis is encountered in the list. The DO list processor then checks to see if the section of the statement under scan can possibly be an implied DO. If it cannot, then a return is made to the simple list processor defining the parenthesis as a simple delimiting parenthesis. If it is an implied DO then the whole section is processed at this point. Note that the implied DO processor calls the simple list processor if necessary to process embedded simple lists inside an implied DO.

MACFTN - restricted version of the MACRO assembler which is used to assemble the compiler output in the 12K compiler. If it is desired to change the class of output that the compiler generates, it will be necessary to update the assembler and its symbol table (PST) to match any new items which are required.

MSSIO - The REWIND, BACKSPACE, and ENDFILE statements are processed here. The processing for all three is identical except as detailed in sections 7.2.11, 7.2.12, and 7.2.13.

OUTSL - This outputs a statement label to the assembler output file.

OVLAY - This routine is the overlay controller for the compiler. It uses the various header blocks for overlay transfer addresses, etc. The controller uses the various entries in the headers to determine the overlay to be called, the location in the overlay to be entered, and whether the call is a JMP form which is non-returning, or a JSR form which will return to the calling overlay. If a returning call is used, the return MUST be taken or else a compiler failure will eventually occur. This will occur because the controller saves information on an internal list to determine where the return must be made. If the return is not made, the table will eventually fill. A side note concerns the placement of temporary variables. No temporaries should be placed in an overlay which may call using the returning call. If a temporary should violate this rule, it is guaranteed not to have the same information after the call that it had before, since the compiler makes no attempt to save the old core image. It instead brings in an entirely new copy from the overlay file, and justly assumes that the new copy is identical to the one previously destroyed.

PATCH - this routine consists only of 100 decimal bytes of space which may be used for a patching area.

PST - symbol table used by MACFTN. As described under MACFTN, this routine should be changed only if changes are made to the compiler output which requires op-codes on operations not currently required by the compiler.

RDCI - This is a conversion routine for converting ASCII to real or double precision. It is used only by SYMTAB. This routine is identical to the OTS routine of the same function. This is done to assure compatibility in the way numbers are converted at compile and run time.

SPCLST - This routine has the DIMENSION and TYPE processors. It also contains various subroutines used by the other declaratives.

STRTUP - This is the temporary part of the compiler start up code. It initializes the overlay handler and then jumps to INIT.

SUBFUN - This is the processor for SUBROUTINE and FUNCTION statements. This routine has the main function of getting the routine name and generating object code to specify this name. It also places in the symbol table any formal parameters which are declared in the statement. The FUNCTION processor also has a special submode used for typing the function name itself.

SYMBOL - This is a dummy routine, used only in the 8K compiler which has all the bit assignments described at the beginning of SYMTAB. This allows SYMTAB to be in only one overlay. WARNING - any changes made in the assignments in SYMTAB must be reflected identically in this routine, since its only purpose is to substitute for SYMTAB in the various overlays not containing the symbol table handler.

SYMTAB - Is the overall symbol table handler for the compiler. In the 12K compiler it is permanently resident, in the 8K compiler it is a separate overlay. It consists of several sections. The first allows one to look up a symbol according to its name (ASCII string). This type of lookup is generally done only on the first occurence of the particular variable or constant on the line. When this lookup is made, the attributes of the variable are returned to the caller. These are the type, class (constant, variable, array, or function), and serial number. The serial number is a 12 bit value which allows the secondary part of the symbol table handler to access an entry without a complicated search. The majority of references within the compiler after the first are done using the serial number of an entry rather than the name. This is expecially evident in the expression analyzer, which may need to scan a string several times. The other sections involve subsidiary routines used by the two main lookup routines to perform routine maintenance duties.

TABLES - This contains all of the impure area for the compiler. All changeable tables should be placed here. No code should be placed here.

UTILTY - This is a package of miscellaneous utility routines used by the compiler.

11.0 FUNCTIONAL DESCRIPTION AND OPERATION

11.1 Global Flow of Control

The overall operation of the compiler can be described as follows:

After having built the compiler and its overlays (see section 13.0) the system consists of the files FORTRN, FORTRN.OVR, and FORCOM.DGN. The first is the compiler proper, the second is the overlay file, and the last is the diagnostic text file.

When a RU FORTRN command is given, the compiler main program is loaded and execution starts at the entry point of routine STRTUP. This routine determines the machine size, determines if the diagnostic file is present, and then finds the absolute disk address of each overlay in the overlay file. Upon completion a jump to location START in routine INIT is taken. Note that STRTUP is a once only routine and is overlayed by the compiler symbol table.

Upon entering routine INIT, the keyboard is INITed and the program name is typed followed by a # sign. The user then types a command string having the format as required by the standards for command input. When the command has been typed, each file as specified is OPENed and made ready for use. If either of the two output files already exists, a DELETE followed by a reopen is done.

At this point, if the assembly output has not been requested with the /AS switch, the file FORTRN.TMP is opened for output. This file is used to retain the compiler output for the automatic assembly pass. In the 8K compiler, it is not possible to get this interface, and it operates as if the /AS switch is always specified.

After completion of these tasks the Implicit table is set up to the default values, setting letters I-N to integer and the rest of the alphabet to real. The expendable entries in TABLES are cleared. At this point, the compiler is now completely initialized and ready to compile a program.

The routine DECLAR is now entered through location SCANNR. DECLAR is used to recognize the verbs of all declarative statements. As each declarative is recognized, a jump is made to the corresponding handler. If the respective handler cannot handle the input text as required (e.g. - COMMON=1) it will return to the recognizer with the V (overflow) bit set. A return to the recognizer with the V bit set will only be made if the statement might be allowed as another type of statement. It would be futile to return in this manner from the equivalence processor because the word EQUIVALENCE has already been found and there is no other form which could start with that immense name.

A return to the recognizer without the V bit set implies that the statement was processed, so the recognizer will read another input line and transfer as required.

When the V bit return is ultimately taken (as it well must) the recognizer then calls the ASF processor. If the input line was a legitimate ASF, a normal return occurs and the recognizer reads another line and calls ASF again. If the return from ASF has the V bit set, it was not recognized and the routine EXECUT is called.

EXECUT is the executable statement recognizer. At the time it is called it has been determined that there are probably no more declaratives to be processed.

EXECUT works similarly to DECLAR except that there still is a possibility that the initial lines found are not executable and that a return may have to be made to DECLAR. At the point where the first executable statement is definitely found, the EXEC flag is set to one and return to non-executables becomes impossible.

EXECUT proceeds along, getting each line and dispatching to the various routines needed to process the statements. If a line cannot be recognized, or if the statement processor called returns with a V bit flag, then it is possibly an assignment statement. The assignment statement processor (residing, in part, in ASC1) is the last resort for redeeming grace. If after being rejected everywhere else, a statement cannot be parsed by the assignment processor, an error diagnostic is given to the effect that the statement in question is undefined, unrecognized, or otherwise undecipherable.

One side note, at the time of the issuance of any diagnostic message by a compiler routine, the error message word, and the text pointer are saved in a list of diagnostic statements. At the conclusion of each statement, if this list is non-null, the error print routine (ERRPRT) is called. This routine prints a portion of the input text for recognition, the error number, and if the diagnostic file is present it also prints the text of the error message which matches the error number.

Upon encountering an END statement or an end-of-file, which forces an implied END, the routine ENDPRO is entered. This routine goes through the symbol table and creates the various data items, push routines, and array entries as required. After this is completed, the FORMAT flags are checked and .GLOBLs are issued for each I/O type and format type encountered during the compile.

Now the file FORTRN.TMP is closed, reopened for input, and the assembler pass is called. The assembler is a highly modified version of the MACRO-11 assembler with all features which are not needed by FORTRAN stripped from it. At the end of the first phase of the assembly pass, the data block descriptor summary as described in section 4.11 is generated. During the second phase, the binary is generated as well as any listings desired. Upon completion the binary file generated is NOT closed, but the file FORTRN.TMP is closed and deleted since it is no longer needed. The assembler returns to INIT which checks to see if any more input is to be processed from the input file. If so, the compiler is re-initialized and the next routine is processed. When an end-of-file has occured on the input file, the binary and list files are closed and released, the keyboard is INITed, the error count is printed on the keyboard (no error count is printed if no errors occurred, obviously), and the compiler is restarted. At this point it types a # sign and the whole show is ready to start over again.

11.2 Individual Statement Flow of Control

See section 10 and the source listings for detailed statement flow.

12.0 PROGRAMMING CONSIDERATIONS

12.1 Code and Data storage

Because of the possibility of organizing the overlays in such a way as allowing a routine to call a subroutine in a different overlay, all data (impure) storage should be separated from all code. This is currently accomplished by placing the impure areas in the routine TABLES which must always be part of the resident root. Note that this also implies that self-modifying code should be avoided if possible to avoid placing unnecessary restrictions on which compiler routines may or may not be overlaid.

12.2 Adding Statement Classes

Though adding statement classes is comparitively easy, several potential problems exist. In the routines DECLAR and EXECUT there exists a prototype list which describes the ASCII representation of each of the statement verbs. This list is a two part list, the first part of which consists of a string of pointers, the second of which consists of a string of ASCII prototypes. When an insertion is desired in the prototype list, a new pointer must be placed in the first part, and an ASCII prototype must be placed in the corresponding position in the second part. If confusion results, consult the listing as an example.

A. In the routines DECLAR and EXECUT are placed the ASCII prototype names for recognizing declarative and executable statements respectively. If it is desired to add a new name to the list, it must be realized that order is somewhat important, for instance the name INTEGER*4 must occur before the name INTEGER. This is required because the scanner searches for the first occurrence of a completely matching prototype to the string being scanned, thus if the largest occurrences do not come first in the list, success may be incorrectly reported on a subset of the full name desired.

B. In DECLAR, the pointer list NEXTBL is logically broken up into several sections. From NEXTBL to the end of the list is scanned for the first statement of a routine. From HDRN to the end of the list is scanned for all other statements in the declaratives, except when a BLOCKDATA has been found, in which case the scan starts at BDATA. This makes it easy to make statements like EXTERNAL and DEFINEFILE illegal in BLOCKDATA subroutines.

The entries from NXTBL1 to the end of the list are data type entries like INTEGER, REAL*2, etc. The order of these entries matches the order of the byte table MODE which is used to assign a numerical value (the data type) to the entry. The table MODE must have exactly the same number of entries as there are prototype entries to recognize.

C. The entries in the two prototype tables match in order entries in the various overlay headers, allowing a quick transfer to the proper routine for handling the particular name. The only exception occurs in the data type entries as described above, which all transfer to either the IMPLICIT or the TYPE handler with the proper data type as retrieved from the MODE table. The IMPLICIT and TYPE handling is done specially to minimize the effort required by the individual processors.

D. In EXECUT the pointer list EXTBL is logically broken down into two parts. For normal statement handling, the whole list is scanned. For logical IF processing, the scan is started at IFTAB to eliminate illegal combinations of statement types within the statement by default. This structure is handled in a similar manner to that in section B above.

13.0 PREPARATION AND/OR SYSTEM BUILD

13.1 Building the Compiler

Building a compiler from scratch requires assembling the 48 separate modules and linking then as described in the following steps. Note that when a file name is mentioned below, the extensions are not mentioned. Source files all have the .MAC extension, object files are .OBJ, and load modules have the .LDA extension.

Assembling must be done using MACRO (make sure that the system macro file SYSMAC.SML is present while assembling). Linking is done using LINK-11 (note that a compiler cannot be linked on a machine smaller than 12K).

A. Assemble the following modules.

ASC1
ASC2
ASF
COMMON
CORET
DATA
DECLAR
DEFINE
DO
DOFIN
ELOC
ENDSTM
ERRLOC
EVALU
EXECUT
EXTERN
FORMAT
FUNNAM
GCMPLX
GOTO
HDRGEN
IF
IMPLIC
IOSTMT
MSSIO
OUTSL
OVLAY
PATCH
PST
RDCI
RUNLNK
SPCLST
SUBFUN

SYMTAB
UTILTY

Assemble the routine MACFTN using the /NL:CND switch.

If building an 8K compiler go to step B, otherwise go to step C.

B. Assemble the following routines with the file 8K.MAC as their headers.

ENDPRO
ERRPRT
GENOVL
INIT
IOPACK
STRTUP
SYMBOL
TABLES

An example of a MACRO command string used here might be:

```
#INIT,LP:<8K,INIT
```

Now assemble the following routines without the file 8K.MAC.

HDR00
HDR01
HDR02
HDR03
HDR04

Now go to step D.

C. Assemble the following modules:

ENDPRO
ERRPRT
GENOVL
INIT
IOPACK
STRTUP
TABLES
HEAD00
HEAD01
HEAD02
HEAD03
HEAD04

D. At this point all of the assemblies are complete. The next phase consists of linking the compiler main program and each of the overlay builders. If building an 8K compiler go to step E, otherwise use step F.

E. Do the following links. This will cause the overlay builders (OV0-OV4) to be built as well as the compiler main program.

```
$RUN LINK

#OV0,LP:<GENOVL,STRTUP,ELOC,SYMBOL,SUBFUN,DATA
#GCMPLX,OUTSL,HDRGEN,MSSIO,FUNNAM,ERRPRT,INIT
#HDR00,OVLAY,UTILTY,ERRLOC,TABLES,IOPACK
#PATCH/T:37460/E

#OV1,LP:<GENOVL,STRTUP,ELOC,ENDPRO,ENDSTM,RDCI
#SYMTAB,HDR01,OVLAY,UTILTY,ERRLOC,TABLES,IOPACK
#PATCH/T:37460/E

#OV2,LP:<GENOVL,STRTUP,ELOC,SYMBOL,COMMON,EXTERN
#DEFINE,IMPLIC,SPCLST,ASF,GCMPLX,DECLAR,HDR02
#OVLAY,UTILTY,ERRLOC,TABLES,IOPACK,PATCH/T:37460/E

#OV3,LP:<GENOVL,STRTUP,ELOC,SYMBOL,CORET,DOFIN,DO
#ASC2,ASC1,EVALU,GCMPLX,IF,EXECUT,HDR03,OVLAY
#UTILTY,ERRLOC,TABLES,IOPACK,PATCH/T:37460/E

#OV4,LP:<GENOVL,STRTUP,ELOC,SYMBOL,FORMAT,DEFINE
#IOSTMT,GOTO,DOFIN,ASC2,IF,FXECUT,HDR04,OVLAY
#UTILTY,ERRLOC,TABLES,IOPACK,PATCH/T:37460/E

#FORTRN,LP:<STRTUP,GENOVL,ELOC,SYMBOL,SUBFUN,DATA
#GCMPLX,OUTSL,HDRGEN,MSSIO,FUNNAM,ERRPRT,INIT
#HDR00,OVLAY,UTILTY,ERRLOC,TABLES,IOPACK
#PATCH/T:37460/E
```

This completes all of the linking for the 8K compiler. Now run the files OV0 through OV4 inclusive as follows:

```
        $RUN OV0
        $RUN OV1
        $RUN OV2
        $RUN OV3
        $RUN OV4
```

The files OV0 through OV4 may now be discarded. The file FORTRN.OVR which was just created by running these routines is the master overlay file. The file FORTRN.LDA linked previously is the compiler main program.

To build the diagnostic file go to section 13.2.

F. Do the following links. This will cause the overlay builders (OV0-OV4) to be built as well as the compiler main program. Note that the /T switch specified below should have the top address specified here for the varrious machine configurations.

/T value	machine size
57460	12K
77460	16K
117460	20K
137460	24K
157460	28K

The particular example shown below is a 16K link.

```
$RUN LINK

#OV0,LP:<GENOVL,STRTUP,ELOC,MSSIO,FUNNAM,ERRPRT
#ENDPRO,ENDSTM,INIT,RUNLNK,OUTSL,HDRGEN
#UTILTY,ERRLOC,RDCI,SYMTAB,HEAD00,OVLAY
#TABLES,IOPACK,PATCH/T:77460/E

#OV1,LP:<GENOVL,STRTUP,ELOC,COMMON,EXTERN,SUBFUN
#DEFINE,IMPLIC,GCMPLX,DATA,SPCLST,ASF,DECLAR
#OUTSL,HDRGEN,UTILTY,ERRLOC,RDCI
#SYMTAB,HEAD01,OVLAY,TABLES,IOPACK,PATCH/T:77460/E

#OV2,LP:<GENOVL,STRTUP,ELOC,CORET,IOSTMT,DOFIN,DO
#ASC2,ASC1,IF,GCMPLX,EVALU,EXECUT,OUTSL,HDRGEN
#UTILTY,ERRLOC,RDCI,SYMTAB,HEAD02,OVLAY,TABLES,IOPACK
#PATCH/T:77460/E

#OV3,LP:<GENOVL,STRTUP,ELOC,GOTO,DEFINE,FORMAT,DOFIN
#IF,GCMPLX,EVALU,EXECUT,OUTSL,HDRGEN,UTILTY,ERRLOC
#RDCI,SYMTAB,HEAD03,OVLAY,TABLES,IOPACK
#PATCH/T:77460/E

#OV4,LP:<GENOVL,STRTUP,ELOC,PST,MACFTN,HEAD04,OVLAY
#TABLES,IOPACK,PATCH/T:77460/E

#FORTRN[1,1]<STRTUP,GENOVL,ELOC,MSSIO,FUNNAM
#ERRPRT,ENDPRO,ENDSTM,INIT,RUNLNK,OUTSL,HDRGEN
#UTILTY,ERRLOC,RDCI,SYMTAB,HEAD00,OVLAY,TABLES
#IOPACK,PATCH/T:77460/E
```

This completes the linking of the compiler. Now run the files OV0 through OV4 inclusive as follows:

```
$RU OV0
$RU OV1
$RU OV2
```

```
$RU OV3
$RU OV4
```

The files OV0 through OV4 may now be discarded. The file FORTRN.OVR which was just created by running these routines is the master overlay file. The file FORTRN.LDA linked previously is the compiler main program.

To build the diagnostic file go to section 13.2.

13.3 Building the Diagnostic file

See chapter 4 of the "Getting on the air with FORTRAN" document.

14.0 TERMINOLOGY

General register - one of the eight fast processor registers 0-7 on the PDP-11.
ANSI - American National Standards Institute
ASCII - American Standard Code for Information Interchange.
CREF - Cross REFerence (listing).
OTS - "Object Time System", that portion of the FORTRAN which interfaces the compiled program to the world.

15.0 Timing Analysis

The following document is not directly related to the operation of the compiler. But, due to the fact that it may be useful to anyone desiring to improve the compiler efficiency, it is included in the specification as an appendix. The analysis was done in September 1971, using V001B of the compiler and V005.2 of the DOS monitor.

Though this timing analysis was done with an early version of the compiler, the basic conclusions can still be considered to be correct, even though the exact times may have changed.

A FORTRAN COMPILER TIMING ANALYSIS

D. Knight

INTRODUCTION

The following information is the result of several runs of the PDP-11 FORTRAN compiler running under the supervision of a statistical sampling package as implemented under PDP-11 DOS Version 5.1. This package is designed to allow a user program to be run under the monitor while collecting data about the frequency of execution of all or part of the user program and/or the monitor. The result of a run consists of a file of information containing a large number of addresses and pointers pertaining to the flow of execution as sampled statistically at approximately 16 millisecond intervals. For more information about the program in question see Appendix I.

PROGRAM CONFIGURATION

The compiler version in question has 4 overlays, 3 of which are used to convert FORTRAN IV source code into assembler acceptable input, the fourth overlay being used to assemble that input and create an object file suitable for linking by the relocatable linker. The fourth overlay is a highly modified version of PAL-11R which has been restructured to remove all assembly features not needed by the FORTRAN system as well as to improve speed where possible. Previous to this test, the assembler phase was run stand-alone with the sampling package to effect these improvements. Currently, the assembler phase runs approximately 3 times faster than the standard assembler for identical input and output requirements. The remainder of the compiler is as described in the compiler specification 130-309-001.

TIMING RESULTS

Due to the various compiler runs being made on different files of varying lengths, all of the data given here have been consolidated. The timing figures given here are all in terms of a "percent of total" rather than an absolute number of seconds or minutes. These figures are based on a total compile time of approximately 40 minutes. Approximately 60 different routines of varying length and requirements were compiled, some very short and some very long. About half of the routines compiled came from the Fortran Scientific Subroutines Package (SSP). The average real compile speeds tended from 50 to 100 statements per minute on both RF-11 and RK-11 based systems.

TOTAL TIMES

Considering the total time used, the percentage actually spent in the compiler amounted to 36.2% while the time spent in the monitor was 63.8%.

COMPILER BREAKDOWN

In this and following sections, times in parentheses refer to the percentage of the TOTAL clock time.

The compiler will be referred to as two separate sections, the compile phase which generates assembly code, and the assembly phase which converts the assembly code to binary. The compile phase accounts for 20 (7.3) percent of the time while the assembler accounts for 80 (28.9) percent. Within the compile phase there are three areas which account for 92 (5.2) percent of the compile phase timing. The symbol table handling, being the major part of this time takes 41 (2.9) percent of the compile time phase. The I/O and character manipulation routines (especially the general string search routine) account for the remainder with the I/O handler taking 32.0 (1.8) percent of this phase.

The assembler phase is dominated by 5 routines which take up

43.8 (15.8) percent of the total compiler time.

These routines are:

BLKBUF -which is used to blank a listing buffer requires 7.8 (2.8) percent.

ITEM - which is used for part of the syntax scan requires 12.4 (4.49) percent.

SERCHB - which is used to search the symbol table uses 6.8 (2.46) percent.

The routines to save and restore registers account for 9.0 (3.26) percent.

A routine to search for a line terminator (TERMIN) requires 7.8 (2.8) percent.

Appendix II contains the detailed table from which these timings were abstracted.

CONCLUSIONS

If it is desired to increase the speed of the compiler as it now exists, several options are available. First, the technique likely to show the most immediate gain in speed would be to try to reduce the percentage of time spent in the monitor proper. The time (63.8%) spent in the monitor currently dominates the entire compiler so that changes to the compiler system are highly likely to be masked by the monitor overhead. I believe, but have been unable to substantiate yet, that an appreciable portion of the monitor overhead is being spent deleting the temporary file required by the compiler. I have no concrete figures on this yet, but the various attempts at estimating and/or measuring this time tend to account for between 15 to 30 percent of the total time used by the compiler. If this is true, this is an obvious place to start to improve compiler speed.

After the monitor is improved, or if it is determined that the monitor cannot be improved (!) the next option would be to modify the assembly phase in the routines which are currently taking the most time. One obvious candidate is to try to find a way around the current need to blank the listing buffer before placing anything in it. Even so, this or any other individual change, if done without changing the monitor overhead will show no more than a 3 to 8 percent improvement in the total throughput.

The least fruitful option is to make improvements in the compiler (except with respect to overlay handling), Any change here would likely account for only a small increase in total speed. If core requirements are such that the compiler could be made into fewer overlays or in the unlikely possibility of being able to make it entirely core resident, a total improvement of 5 - 15 percent is likely.

The statistical sampling package as used here consist of three routines. The first, SAMPLE, is linked with a user program when it is desired to obtain timing information about that program or the system it runs with. This program gains control of the KW-11 line clock and uses the clock interrupts to find out where the execution is taking place at the time of a clock interrupt. Thus, if the program is run long enough to get a few thousand samples, a statistical picture may be built showing the relative percentages of time spent in various portions of a program.

This technique is very useful, but not without problems. Any code which is dependent on the clock, such as interrupt handlers which get service once for every clock interrupt are not likely to show up in the data. Also if a program is timing dependent, the additional overhead inserted in the clock interrupt loop may be sufficient to distort the information. Another problem occurs because the sampling routine needs the monitor itself to output the data collected. The normal user program generally will not be affected by these problems, though it must be recognized that there is a reasonable level of uncertainty which makes it necessary to examine each usage of the technique carefully.

In the DOS implementation here, the samples collected are stored in a buffer until 170 have been collected. At this point the user program is suspended for a short time while the data is output to a file which is reserved for accumulating the data. Upon the completion of the run, this file is used as data for a program which generates histograms of core utilization with respect to time.

The program to evaluate the data consists of two parts. The first, written in assembly code is used only to read the file and pass the information along to a calling program. The second, written in FORTRAN evaluates the raw data obtained from the first and generates the histograms on the line printer. The reason for this routine being written in FORTRAN is that depending on the program being timed and the data collected, it may be desirable to modify the report generator in various places to "tailor" it to the task. Thus information which is deemed pertinent by the user of the system may be collected that was not recognized as being important by earlier users.

This system will be described in more detail shortly when a complete document is finished describing it and its parts.

ITEM	% COMPILE PHASE	% ASSEM. PHASE	% WHOLE COMPILE	% TOTAL TIME
COMPILE			20.	7.3
ASSEMBLE			80.	28.9
MONITOR				63.8
COMPILE BREAKDOWN				
IOPACK	32.		5.2	1.8
SYMTAB	15.1		2.0	.72
RDCI	16.7		2.9	1.04
UTILTY	28.3		4.7	1.70
IOPACK BREAKDOWN				
CHECK INPUT LINE	12.0		2.4	.87
BUILD INTERNAL LINE	4.3		.8	.29
OUTLN,OUTLN1	10.8		1.8	.65
SYMTAB BREAKDOWN				
GETSYM	4.1		.7	.25
SERATR	5.3		1.	.36
RDCI BREAKDOWN				
DIVIDE	9.3		1.7	.61
MUL54	6.0		1.1	.39
UTILTY BREAKDOWN				
PACK	3.3		.6	.22
NXTCH,CNXC	4.6		.8	.29
SCAN2A	9.4		1.6	.58
UNPACK	5.9		1.	.36

ASSEMBLE BREAKDOWN

BLKBUF	9.4	7.8	2.8	
ITEM	14.9	12.4	4.5	
SERCHB	8.3	6.8	2.46	
REGISTER SAVE AND RESTORE	10.9	9.0	3.26	
ENTER3	.8	.7	.25	3 WORD LOOP
BINSRCH	5.0	4.1	1.5	
TERMIN	9.6	7.8	2.8	
NUMBER	3.5	3.1	1.1	
NCHAR	4.5	3.7	1.3	
SETUP1	5.1	4.2	1.5	
BINASC	4.8	4.0	1.4	
ENDLINE ERROR LOOP	3.5	3.0	1.0	

MONITOR BREAKDOWN

Of the time spent in the monitor (63.8%), approximately 60% (40%) of the time is spent in the I/O wait loop of which about half of the wait time can be attributed to the file delete which occurs and the other half occurs during the compiler I/O. The remainder of the monitor time, 30% (24%) is taken up by the read/write processor.

Figure 1 - Array Descriptor Block

Address				
N	Address of first data element			
n + 2	# Dims	Data Type		Data element size in bytes
n + 4	max. dimension - first subscript			
n + 6	max. dimension - second subscript			
n + 10	max. dimension - third subscript			

Words 3 and/or 4 appear only for two and three dimensional arrays respectively. The ADB as stored in the complier is similar except that the first word is used to point to the next ADB entry.

Figure 2 - Integer Format

Address	
n	value
n + 2	filler

The filler is used only as a placeholder. The /ON causes the removal of this filler word at compile time.

Figure 3 - Real and Double Precision Format

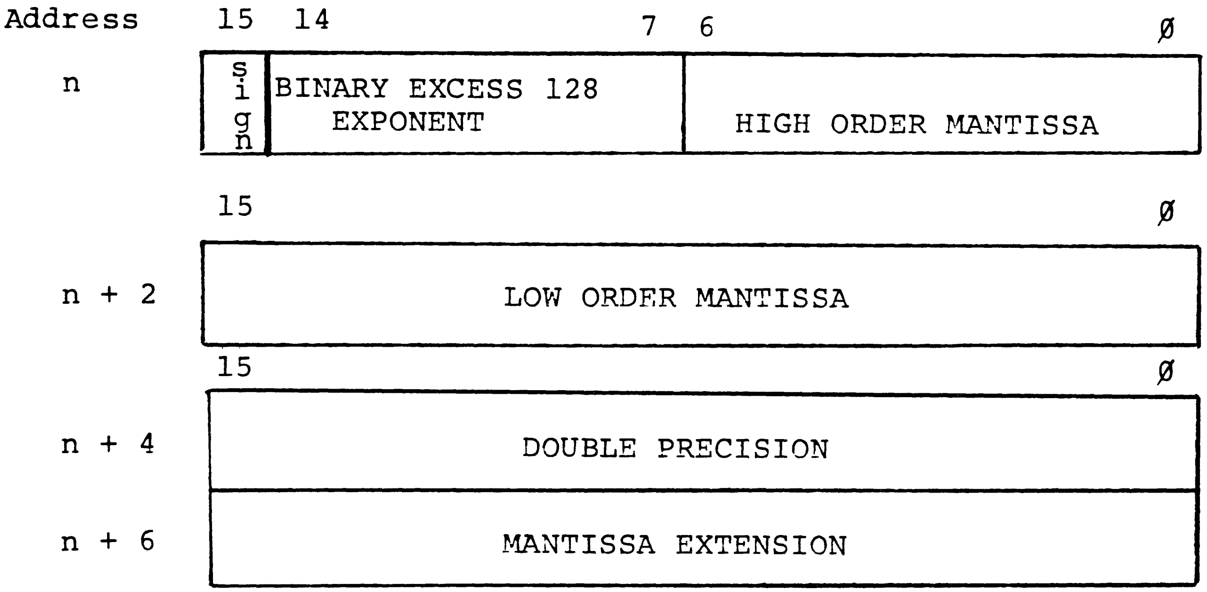


Figure 4 - Complex Format

Address	
N	Real Part (format same as in figure 3)
N + 4	Imaginary Part (format same as in figure 3)

Figure 5 - Main Symbol Table Format

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N	Entry Type		Data Type			ADJ Array	Prog Name	Constant	Length of data item (if constant = 1) or parameter list index (if constant = 0 and parameter = 1)							
N + 2	Common	ADJ Array	Equi-valence	Para-meter	Entry Serial Number											
N + 4	Address of next entry in table (-1 if last entry)															
N + 6	Symbol Name in RADIX 50 (two words)															
N + 12	Single Ref.	Assign		Explicity Type		Used in Expr.	Gener. PUSH	# of Dimensions		Unused	Common Block Sequence (0 = not in common, 1 = blank common)					
N + 14	ADB pointer for subscripted items															
N + 16	Any additional words will contain the entry value if the CONSTANT bit is set.															

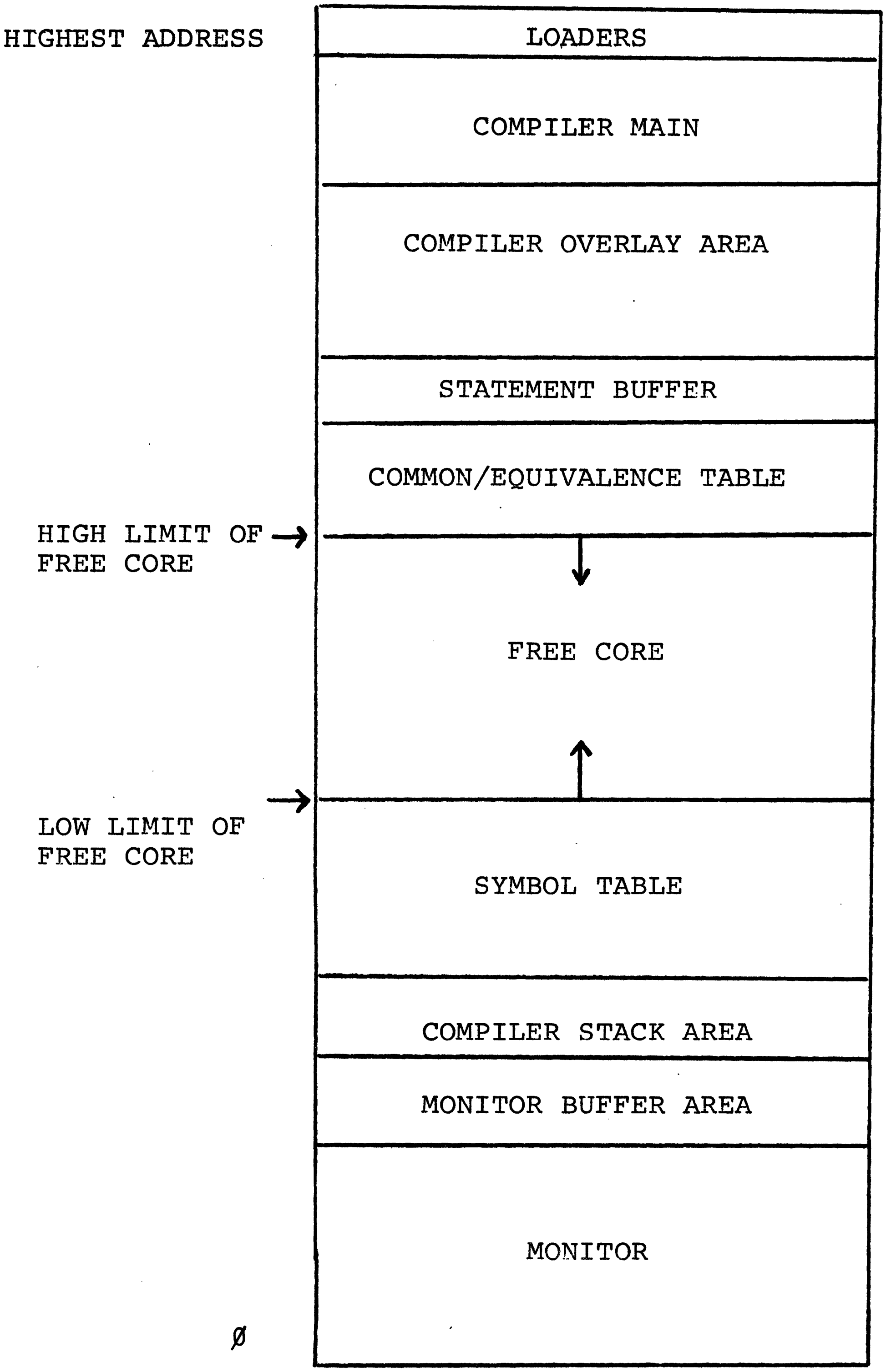
Figure 6 - Common Block Header

N	Link to next block header
N + 2	First characters of block name (ASCII)
N + 4	Second 2 characters of block name (ASCII)
N + 6	Last 2 characters of block name (ASCII)
N + 8	0
N + 10	Link variables in this block

Figure 7 - DO Table

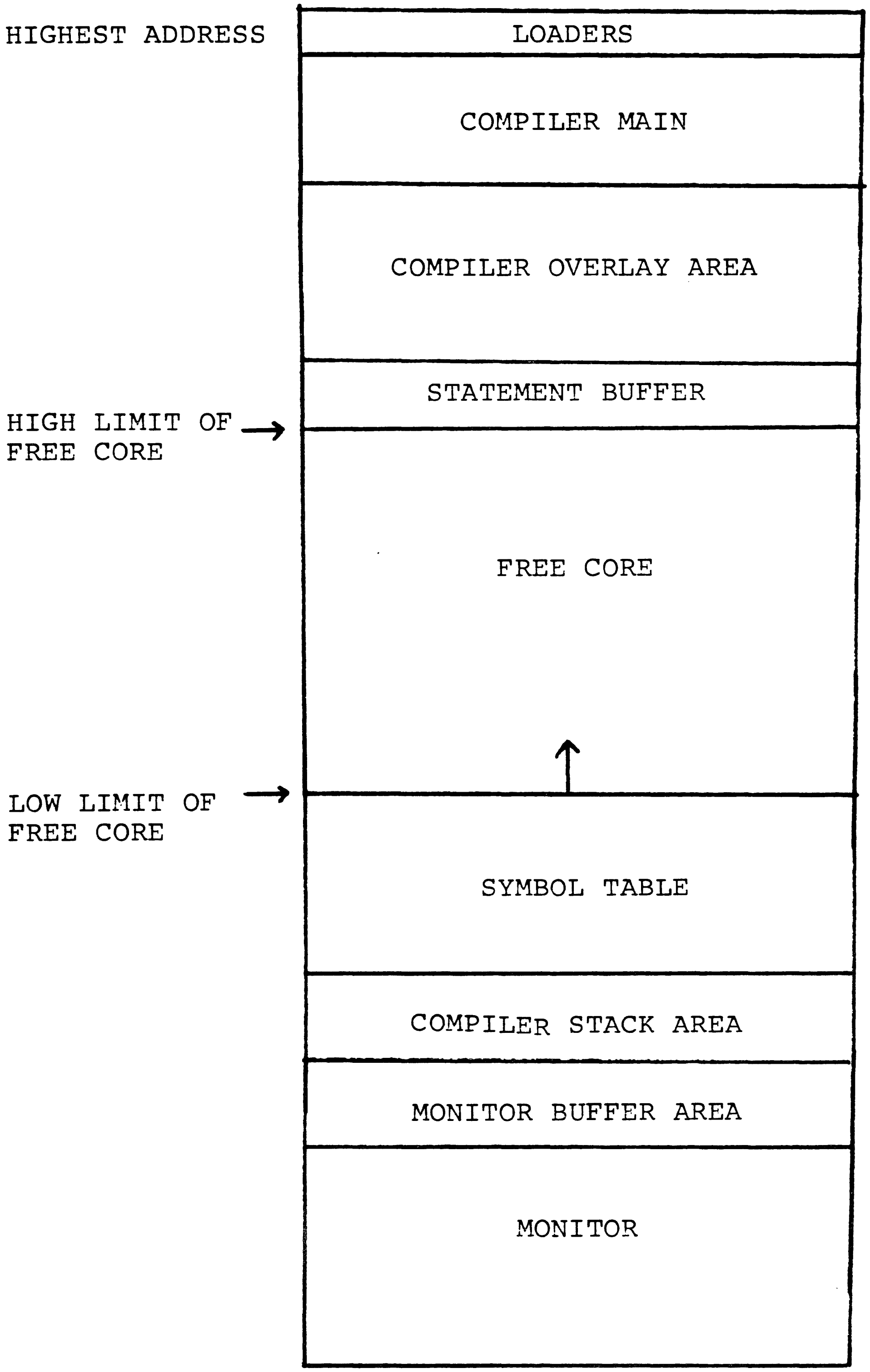
N	Statement number of terminal statement in RADIX 50
N + 4	Serial number of destination return label
N + 6	Pointer to control variable symbol table entry
N + 10	Pointer to initial parameter symbol table entry
N + 12	Pointer to terminal parameter symbol table entry
N + 14	Pointer to step value symbol table entry

Figure 8 - Compiler Memory Layout - Declarative Phase

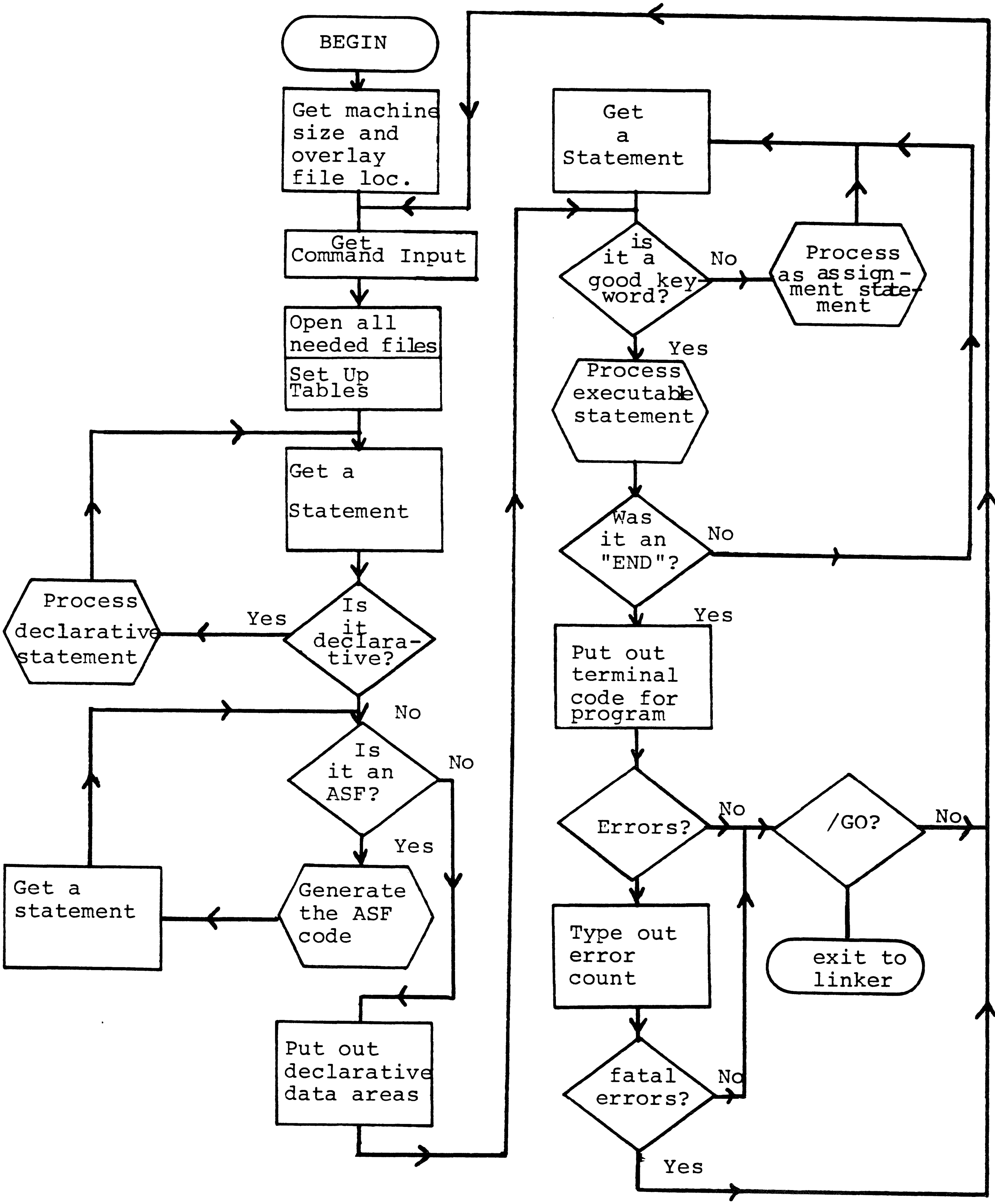


The symbol table is allowed to expand upwards. The COMMON/Equivalence table is allowed to expand downwards.

Figure 9 - Compiler Memory Layout - Executable Phase



FORTRAN Compiler Overall Flow



PART II

PDP-11

FORTRAN IV OBJECT TIME SYSTEM

VERSION 20A

Table of Contents

1.0 OVERALL DESCRIPTION

1.1 Usage

The Object Time System (OTS) is a library of assembly language programs to be used as a complement to compiled code in running Fortran on the PDP11. The OTS is divided into four principal parts:

A) The I/O processing routines.

B) The mathematical subroutines and function generators.

C) Miscellaneous service routines.

D) I/O device tables, buffers and run 'switches'

The OTS will handle certain unit interface problems connected with I/O but will not include any load, link or monitor section. For a description of the Fortran functions to be supported see 130-309-001.

1.1.1 I/O Processing Routines

The I/O section will handle building of user input and output records (according to any Fortran format specifications) and accomplish file munipulations through the system monitor. The format processor consists of three sections:

A) A routine which associates items in the format with items in the I/O list and I/O record.

B) A set of routines which perform format specified conversions to and from character strings.

C) A set of monitor interface routines will act as device drivers.

1.1.2 Mathematical Subroutines and Functions

The math routines section will handle two types of tasks:

A) Arithmetic operations not supported by the 11/20 hardware - eg floating point, double precision, etc.

B) Standard mathematical functions supported by Fortran - eg, SIN,ATAN, EXP, etc.

1.1.3 Miscellaneous Service Routines

The miscellaneous service routines will perform such functions as array index arithmetic, exit, and error processing.

1.1.4 I/O Device Tables and Buffers

I/O device tables, buffers and run switches represent the non- reentrant portion of the OTS. It contains the link blocks, file blocks, device status switches, etc. for Fortran I/O and any buffers which are necessary to perform I/O. It also contains any global values or switches which are necessary to execute the users program(e.g. last encountered statement number, subroutine name chain).

1.2 Design Philosophy

In coding The Object Time Package emphasis will be placed on speed efficiency in the arithmetic routines, and economy of core at the possible expense of speed in the I/O routines.

In general, routines will be segmented so that only code needed by a particular Fortran run will need to be loaded.

1.3 References

The mathematical function algorithms will be drawn from DEC-11-GGPA-D, IBM-C28-6596-2, IBM-C26-5929-4, Hastings - Approximations for Digital Computers, Hart - Computer Approximations, standard reference texts, etc.

2.0 HARDWARE ENVIRONMENT

2.1 Minimum Requirements

PDP11 Fortran OTS will run under the Disk Monitor, which requires a minimum of 8K of core, RF-11 Disk, High Speed Reader/Punch or Dectape, and an ASR-33 TELETYPE.

2.2 Options

If more than 8K of core is available, OTS can be linked to use all available space. OTS will support all standard product line options which are supported by the Disk Monitor (DOS).

2.3 Future Considerations

The OTS is a highly modular system; extensions to support new hardware options within the DOS environment can be made with minimal effort.

3.0 SOFTWARE ENVIRONMENT

3.1 Minimum Requirements

The Object Time System will require the services of the linking loader and the PDP-11 Disk Monitor.

3.2 Options

The Object Time Package will support array overflow diagnostics optionally set up by the compiler. (see 130-309-001 section 3.1.2.6)

3.3 Future Considerations

All Object Time Code will be re-entrant to facilitate future development; however, certain OTS routines require addresses of areas which are task specific. The RSX-11B-C monitor provides a facility by which an address within a task area can be obtained by the OTS when that task is in execution. This address is the address of a vector of pointers to the specific impure area used by the OTS ($OTSV, see 5.5)

4.0 CONVENTIONS AND STANDARDS

4.1 Registers

The general registers will be referred to as follows:

Reg 0-5 = R0-R5
Reg 6 = SP
Reg 7 = PC

The 11/45 floating point accumulators will be referred to as:

Accumulator 0-5 = F0-F5

4.2 Calling Conventions

4.2.1 Standard Calling Conventions

All standard calling conventions are discussed in 130-309-001.

4.2.2 Internal Calling Conventions

OTS routines which do not interact directly with fortran compiled code may use calling conventions different from any of the above. These will be explained in the routine descriptions.

4.3 Documentation

Documentation will consist of :

a) The latest revision of this document
b) General system block diagrams
c) Heavily annotated listings. Almost all lines of code will have comments. Each routine will have heading text describing its function and use.
d) Discussion of algorithms used in the math package.

4.4 Data Formats

4.4.1 Integer Format

An integer is a sixteen bit, two's complement, word aligned quantity. Only the high order (lower-addressed) word (or its address) of integers stored in the two word format will be presented to OTS computational routines. Similarly, all integer results from such routines will be one word only.

4.4.2 Real Format

Real numbers consist of two words of data as follows:

```
WORD n bit 15 = sign of fraction
       bits 14-7 = binary exponent excess 128
       bits 6-0 = high order fraction magnitude bits

WORD n+2 bits 15-0 = low order fraction magnitude bits
```

This format is limited to normalized numbers. The high order bit of the fraction (always 1) Is omitted from its implied position (bit 7 of word n).

4.4.3 Double Precision Format

Same as 4.4.2 followed by two more words of extended low order fraction.

4.5 Re-entrancy

All OTS code will be re-entrant.(see 3.3)

4.6 Labeling Conventions

All OTS routines will have a title which begins with a '$' followed by three alpha's and two digits. The three alpha's will indicate the routine name while the two numerics will indicate the current version number.

Internal labels are freely chosen for mnemonic significance. All OTS .GLOBL's have '$' as their first character to avoid conflict with user GLOBL symbols; user-callable subroutine entry points do not begin with '$' (eg SIN, SETFIL).

5.0 DATA STRUCTURES

5.1 I/O Device Table ($DEVTB)

Resides in the non-reentrant portion of OTS. It is used to associate a real device and file with the user specified device number. It consists of two sections - a header for general control information and entries for specific information about I/O status.

5.1.1 Device Table Header

	WORD-1	vector entry for device -3 (logical data set name CMO) used for error logging device
	WORD 0	addr of error msg file link, file and block block
$DEVTB:	WORD 1	number of entries in device table (default 8)
	WORD 2	device num of print file (error msg logging device) default device=-3
	WORDS 3-10	address of $DEVTB entry for device number (1-8) (if any word=0 device number is not allowed)

NOTE: The number of devices available may be changed by altering WORD 1 and appending to or deleting from last 8 words of header.

5.1.2 Device Table Entries

Each entry is 16 words long

WORD 1	DDB pointer (from link block after init)
WORD 2	physical device name in RAD50 (See Section 5.1.4 for defaults)
WORD 3	unit num (default=0) (NOTE 1)
WORD 4-5	filename RAD50 (default 'FORnnn' where nnn is device num)
WORD 6	file extension RAD50 (default 'DAT')
WORD 7	switches (NOTE 2) /protect code (default 233)

WORD 8-14 for non-random I/O

WORD 8	status/mode from line buff header
WORD 9	count of I/O operations for this device
WORD 10-12	unused

WORD 13 num records to allocate (from SETFIL) Note 1
WORD 14 record length in bytes to allocate (from SETFIL) Note 1

WORDS 8-14 for random I/O

WORD 8 function word
WORD 9 block number
WORD 10 buffer address (from monitor)
WORD 11 buffer length (from monitor)
WORD 12 associated var addr (from DEFINE FILE)
WORD 13 maxnum records in file (from DEFINE FILE)
WORD 14 record length in bytes (from DEFINE FILE)

WORD 15 UIC(default 0, implies login UIC)
WORD 16 addr of error value var (from SETFIL)

NOTE 1 - The how open byte is normally 0 when the file is closed. However, if the user wishes to allocate a contiguous file at run time, SETFIL will set this value to 127, and if the file is non random will set words 13-14 of the entry

NOTE 2 - WORD 7 switches
- bits 0-1
  - 0=file closed
  - 1=FMTD file open
  - 2=UNFMTD file open
  - 3=RANDOM file open
- bit 2 1=DEFINE FILE done on this device
- bits 3-7 unused

5.1.3 Device Entry for Error Message File

Is a link block with a file block and a block block. It is 16 words long, the protect code is 322 the file name is 'FORRUN.DGN' and the logical dataset name is 'ERR'. The file itself is contigous with one sixty four character message per logical record.

5.1.4 Device Table Defaults

The physical device defaults in $DEVTB are as follows:

Unit	Physical Device
-3	KB (error logging)
1	SY
2	SY
3	SY
4	PR

5	LP
6	KB
7	SY
8	BI

The logical unit -3 (logical data set name CMO) is used for error-logging and PDUMP output. The FORTRAN user program may also write ASCII output to CMO; since the compiler does not permit negative integers as unit numbers in WRITE statements, the unit number must be an integer variable whose value is -3.

5.2 I/O Buffer ($IOBUF)

There is one sixty eight word I/O buffer prefixed by a five word prototype link block, a seven word prototype file block and a three word prototype line block header. All I/O uses this 'I/O buffer' with the exception of random I/O which does not use the last sixty seven words. The global name $IOBUF is the address of DDB pointer in the link block.

NOTE: Error processing also uses this 'I/O buffer' for message logging in such a way that any information in the link block or data portion is not destroyed.

5.3 I/O Processing Stack Usage

Each form of I/O requires information about the pending I/O request. This information is most easily transmitted between routines via the stack. Thus for each form of I/O there is a stack format which is required. The stack is divided into two logical sections. The fixed I/O information section(which will be addressed by R4 while in the I/O routines) and the specific I/O routine information section(which will be addressed by R5 while in the I/O routines).

5.3.1 General I/O Information Stack (IOPSTK 19 words addressed by R4)

SAVER5	callers R5
SAVER4	callers R4
SAVER3	callers R3
SAVER2	callers R2
SAVER1	callers R1
SAVER0	callers R0
ERRFLG	set to one if errors require flushing of the

	I/O list
ARGLEN	length of current I/O item 1,2,4,8 or 0 if none
ARGTYP	type of current I/O item 1=LOG, 2=I2, 4=I4, 5=R, 6=C, 8=D
ARGPTR	addr of current I/O item
RECEND	addr of I/O buffer end + 1
RECPTR	addr of current position in I/O buffer
RECADR	addr of start of I/O buffer
IOTADR	addr of I/O routine $INFR, $OUTFW, $INR, $OUTW, $INRR, or $OUTRW
IOTSW	type of I/O 0=FMTD, 1=UNFMTD, -1=RANDOM
IOADDR	addr of item processing routine $FIO, $UIO or $RIO
IOSW	I/O switch 0=output, 1=input
FMTADR/RNUMAD/UFNULL	FMTD I/O addr of format statement UNFMTD I/O = 0 RANDOM I/O addr of record number
UNITAD	address of I/O unit number

5.3.2 Formatted I/O Stack (IOFSTK 17 words addressed by R5)

PSCALE	current P scaling factor
DSCALE	current decimal width
FWIDTH	current field width
REPCNT	current conversion specification repeat count
SCNKAR	character addressed by FMTPTR
CVTRTN	current conversion routine address
CVTSW	current conversion type 0=D,-1=E F G,1=I O, 2=A, 3=L
INT	format processor accumulator
INTSW	state of INT 0= empty, 1= pos num, -1 = neg num
EXITSW	if items cvtd since last record I/O = 0
GRPCTS	highest nesting's unexhausted group rep count
GRPCTI	highest nesting's initial group rep count
GRPCT	current group repeat count
NPRN2	ptr to lowest nesting's (in fmt stmt
NPRN1	ptr to highest nesting's (in fmt stmt
NEST	current nesting level
FMTPTR	ptr to current position in fmt stmt

5.3.3 Unformatted I/O Stack (IOUSTK 1 word addressed by R5)

IOSTAT record status for current I/O

0 = not first or last segment
1 = first segment
2 = last segment
3 = first and last segment

5.3.4 Random I/O Stack (IORSTK 3 words addressed by R5)

RECMAX	number of records in file
LENMAX	number of bytes remaining in this block
AVADDR	address of associated variable

5.4 Error Processing Tables

For the purpose of minimizing core usage, errors will be divide into classes on the basis of functional similarity(e.g. Arith checks, function errors, recoverable I/O errors, unrecoverable I/O errors, etc) each class of error will have an associated maximum allowed occurrence before termination count which may be overridden by calling SETERR .The text for each error message will be in a contiguous file on a disk. If this file is not present, just a message number will be logged.

5.4.1 Error Class Table ($ERRC)

The global name $ERRC points to WORD 1 of the first error class. The preceeding word ($ERRC-2) Contains a count of the number of error classes. Each entry is of the form:

WORD 1	number of messages in this class/logical record number of first message of class in file
WORD 2	transfer address
WORD 3	max allowed occurrence count pos = log msg until max 0 = log msg and ignore -1 = ignore error(no log) -2 = exit without logging -3 = immediate run abort
WORD 4	actual count (if word 3 pos)

The error class table also contains a byte vector $ERRF,initialized to 0's. The I'th byte is set to 1 by the error routine $ERR on any occurrence of an error of class I, independently of any other error action taken. This byte may be tested and cleared by the user-callable subroutine TSTERR.

5.4.2 Error Transfer Address (entry word two)

Since it is desired that the sophisticated user be allowed to alter what occurs after message logging, a transfer address is provided. For each error message class WORD 2 is the address to which control will be passed after message logging. The default is to RTS R5. This implies that the routine which called error will be responsible for any further processing. Control is passed in such a way that it appears that error was not in the calling path.

5.5 Monitor Switches and Values ($OTSV) See 3.3

This section contains global run information which is not logically a part of any other section. In the RSX-11B-C version of the OTS, these task-specific addresses are obtained through the monitor call EMT 42.

```
 address of $DEVTB
 address of $IOBUF
 address of $ERRC
 $NAMC     addr of end of subroutine traceback chain
 $SEQC     value of current internal sequence number
 $EXSW     value for EXIT routine
     0   = user call
     1   = errors have ocurred this run
     2   = an error has reached it max allowed count
     3   = an error whose max allow count word is -2 has
           called EXIT
```

$ERRWK 14 bytes of work area. Used by $ERR if the message file is not available. also used by $IOSET to create the logical dataset name for each device.

5.6 Fortran Listing Device Table

$FLDEV = is a user modifiable table which defines Fortran listing devices at the cost of one word per entry. The device names are packed in RAD50. The current table contains the devices LP, KB and TT. The list is terminitated by a zero entry.

It is used by $INFR to deterime if a device is a printing device.

6.0 INPUT/OUTPUT

6.1 I/O Packages

There Will Be Three I/O Packages:

a) Formatted
b) Unformatted
c) Random access

6.1.1 Formatted I/O

The formatted I/O routines will interact with the object code at three entry points:

1) Initialize I/O, transmit unit and format information
2) Transmit/receive data
3) Close out current record

6.1.2 Unformatted I/O

The unformatted I/O routines will interact with the object code at three entry points:

1) Initialize I/O, and transmit unit information.
2) Transmit/receive data items.
3) Close out current record

6.1.3 Random I/O

The Random access I/O routines will interact with the object code at four entry points.

1) Position the specified file at the indicated record.
2) Verify that the proper record of the file is positioned and initialize I/O.
3) Transmit/receive data items.
4) Close out the current record.

6.2 I/O Handlers

All input and output will be accomplished through monitor calls, but blocking and deblocking may be done in the object time routines .

All I/O will be single buffered (using $IOBUF) A wait will be issued after each I/O request and execution will be suspended until the I/O completes. Thus all errors which return through the ERR= return will be precise.

6.3 Nondata Operations

There will be a package to handle nondata operations - eg BACKSPACE, REWIND, ENDFILE.

6.4 Device Dependence

I/O will be device independent so that the user may do logical assignments at run time using monitor $ASSIGN commands or by calling the 'SETFIL' subroutine. However, in the absence of a $ASSIGN certain defaults are assumed (See Section 5.1.4).

The special device -3 is assumed to be the message logging device. Therefore it must be available for formatted ASCII output when required. The user may modify the default logging device number by re-assembling $DEVTB.

The number of Fortran unit numbers is not limited, but in order to use numbers greater than eight the Fortran device table must be expanded.

The logical dataset name used for Fortran devices will correspond to the unit number, except for the error logging data set (logical data set name = CMO)

6.5 File Structures

6.5.1 Formatted Input

Formatted input will read 'formatted ASCII' records using the .OPENI and the .READ monitor macros(all file structures which OPENI supports will be allowed). The maximum length input record allowed is 133 characters. Longer records will be truncated without diagnostic, shorter records will be padded with blanks after deleting the last character (LF, FF, VT) and the second to last character if it is a CR.

6.5.2 Formatted Output

Formatted output will write 'formatted ASCII' records using the .OPENO or .OPENC and the .WRITE monitor macros(all file structures which .OPENO or .OPENC support will be allowed). The maximum length output record allowed is 133 characters. If the output device is not a Fortran listing device , a CR and a LF will be appended to the end of the record. If the output device is a Fortran listing device a CR,VT is appended to the end of each record and the first character of each record is interpreted as a line spacing command.

6.5.3 Unformatted Input

Unformatted input will read 'formatted binary' records using the .OPENI and the .READ monitor macros. there is no maximum record size. Records will be input in segments of up to 63 words length where the first word of each segment is a control word(0=not first or last segment, 1=first segment, 2=last segment,3=first and last segment).

6.5.4 Unformatted Output

Unformatted output will write 'formatted binary' records using the .OPENO or .OPENC and .WRITE monitor macros. Records and segments same as for unformatted input.

6.5.5 Random Input/Output

Random input/output will read/write binary records using the .OPENU and the .BLOCK monitor macros(contiguous files only). Record length is limited to 32767 bytes. Random I/O will determine the block number and the displacement to the proper record from the users DEFINE FILE statement and the I/O statement's record number.

NOTE: it is required that the blocksize for a device containing a random access file be a power of two.

7.0 LANGUAGE

The language to be serviced has been specified in 130-309-001.

8.0 COMMAND LANGUAGE AND STRUCTURE

N/A

9.0 OPERATING PROCEDURES

9.1 Putting the Object Time Library on a System

The complete, current procedure for putting the OTS on a system is fully documented in current versions of Getting FORTRAN on the Air, DEC-11-LFOAA-A-D.

9.2 Linking Considerations

The Fortran compiler generates the necessary globals for the I/O conversion routines $DCI, $DCO, $ICI, $ICO, $LCI and $LCO. Any routines not needed for the specific Fortran program will be satisfied by the dummy routines which follow $FIO in the library.

When creating core libraries the conversion routines necessary for program execution must be included in the resident section. I.E. it is not possible to link a specific conversion routine to a non resident section.

10.0 PHYSICAL DESCRIPTION AND ORGANIZATION

In general, functions which can be added as entry points into other functions without significantly increasing space requirements will be so combined. It is assumed that the linking loader will be capable of resolving all explicit user function and subroutine requests, as well as requests implied by them, by a search of the OTS library. To facillitate dummy references to certain routines which may not be used in a particular run, it is further assumed that the library is searched in a predictable order and that the compiler will flag references to each type of format conversion used in the run.

10.1 Arithmetic Package

The routines in this section are called in accordance with the conventions listed in section 4.8 of 130-009-001. All routines are position independent except as noted. In general, OTS routines fail to be position-independent primarily because they contain Polish calls (address constants).

Type conversions among all data types are supported. The conversion routine names are of the form $XY; $XY converts an item of type X on top of the stack to an item of type Y on top of the stack.

This section consists of the following routines:

$IR float the integer on the top of the stack. (32 words)

$ID entry into $IR which first moves the argument down two words (filling in with zeros) before executing the $IR code.

$IC integer to complex conversion. Same entry point as $ID.

$DR put the high order words of the double precision quanitiy on the top of the stack rounding to real format. Call $ERR if exponent overflow. (21 words)

$RD append a double word of zeros to the real quantity on the top of the stack. (9 words)

$RI truncate and fix the real number of the top of the stack. Call $ERR if result is outside the range

-2**15-1 to 2**15-1. (45 words)

$DI entry into $RI which moves the argument up the stack two words (discarding the low order part) Before executing the $RI code.

$IB integer to byte conversion. clears the byte at 1(SP) and returns. (3 words)

$BI byte to integer conversion. sign-extends the byte on top of the stack to an integer quantity.

$BL same entry as $BI. does byte to logical conversion.

$LB same entry as $IB. does logical to byte conversion.

$ADR replace the two real numbers on the top of the stack with their sum. No codes will be set. Call $ERR on exponent overflow or underflow. (176 words)

$SBR entry in $ADR which negates the number on top of the stack before doing the add.

$ADD replace the two double precision numbers on the top of the stack with their sum. No codes will be set. Call $ERR on exponent overflow or underflow. (290 words)

$SBD entry in $ADD which negates the number on the top of the stack before doing the add.

$ADI replace the two integer nubers on the top of the stack with their sum. No codes will be set. Call $ERR on overflow. (6 words)

$SBI replace the two integer numbers on the top of the stack with their difference. (2 words)

$ADC replace the two complex numbers on the top of the stack with their sum. Uses $ADR to compute (A + BI) + (C + DI). This routine is position dependent. (44 words)

$SBC entry in $ADC which negates the number on the top of the stack before doing the add.

$ADB replace the two byte quantities on top of the stack with their sum. The addition is done using integer arithmetic; no byte overflow indications are given.

$SBB entry in $ADB which negates the byte on top of the stack before doing the addition.

$CMR compare corresponding words of the two items on the stack until a mismatch is found (if one exists). Clear the stack and return the Z and N codes defined in 130-309-001 section 3.1.2.2. (23 words)

$CMD this is the same as $CMR except that the items are double precision. (31 words)

$CMI this is the same as $CMR except that the items are integer. (2 words)

$CMB compares two 8-bit byte quantities (5 words).

$CMC compares two complex quantities. Only the Z bit return code is set, since complex quantities may only be compared for .EQ. and .NE.

$TSR test and flush the real number on top of the stack and return to @(R4) If it is negative, @(R4+2) If zero, and @(R4+4) If positive. (14 words)

$TSD entry in $TSR for double precision.

$TSI entry in $TSR for integer.

$MLB multiply two byte quantities on top of the stack. Sign-extends bytes to integers and jumps to $MLI; no error indications given on byte overflow.

$MLI replace the two integers on the top of the stack with their product. Call $ERR if the result is not in the range -2**15-1 to 2**15-1.(56 words)

$MLR replace the two real numbers on the top of the stack with their product. Call $ERR on exponent overflow or underflow. (115 words)

$MLD replace the two double precision numbers on the top of the stack with their product. Call $ERR on exponent overflow or underflow. (204 words)

$MLC replace the two complex numbers on the top of the stack with their product. Uses $MLR, $ADR, and $SBR. This routine is position dependent. (64 words)

$DVB divide two byte quantities on top of the stack. Sign-extends bytes to integers and jumps to $DVI; no error indications given on byte overflow.

$DVI replace the two integers on the top of the stack with the integer part of the quotient of the top stack item divided into the second item. A zero divisor will result in a call to $ERR. (43 words)

$DVR replace the two real numbers on the top of the stack with their quotient (second/top). A zero divisor, exponent overflow or underflow result in a call to $ERR. (139 words)

$DVD replace the two double precision numbers on the top of the stack with their quotient. A zero divisor, exponent overflow or underflow result in a call to $ERR. (222 words)

$DVC replace the two complex numbers on the top of the stack with their quotient. Uses $ADR, $SBR, $MLR and $DVR. This routine is position dependent. (130 words)

$NGI negate the integer item on the top of the stack. Call $ERR if result is 100000 octal. (12 words)

$NGR entry in $NGI for real.

$NGD entry in $NGI for double precision.

$NGB entry in $NGI for bytes.

$NGC entry in $NGI for complex.

10.2 Fortran Functions

10.2.1 Absolute Value

These routines compute the absolute value.

These routines are called in accordance with the conventions listed in section 4.7 of 130-009-001. all routines in this section with the exception of CABS are position independent.

IABS put the twos complement absolute value of the argument in R0. 100000 octal results in a call to $ERR. (10 words)

ABS put the real argument with sign bit set to zero in R0,R1. (8 words)

DABS put the double precision argument with the sign bit set to zero in R0-R3. (10 words)

CABS put the absolute value of the complex argument in R0-R3. Uses $DVR, $MLR, $ADR, $FCALL and SQRT. (74 words)

10.2.2 Sign

These routines transfer the sign of the second argument to the first argument.

These routines are called in accordance with the conventions listed in section 4.7 of 130-009-001. All routines in this section are position independent.

ISIGN put the twos complement absolute value of the first integer argument in R0 and complement it if the second argument is negative. Call $ERR if first argument is 100000 octal. (14 words)

SIGN put the first argument with its sign bit set to agree with the sign of the second argument in R0,R1. (11 words)

DSIGN put the first (double precision) Argument with its sign bit set to agree with the sign of the second argument in R0-R3. (15 words)

10.2.3 Type Conversion

These routines are the Fortran type conversion functions.

These routines are called in accordance with the conventions listed in section 4.7 of 130-009-001. All routines in this section with the exception of AINT, SNGL and DBLE are position dependent.

AINT truncate the real argument to an integer and put it in R0,R1. (39 words)

INT use $RI and return result in R0. (11 words)

IDINT entry in INT.

IFIX truncate the real argument using $RI. Equivalent to INT. Return result in R0. (11 words)

FLOAT use $IR and return the result in R0,R1. (8 words)

SNGL put the first two words of the argument in R0,R1. Call $ERR if exponent overflow on round. (15 words)

DBLE put the argument in R0,R1 and zero R2 and R3. (7 words)

10.2.4 Modulo

These routines perform remaindering.

These routines are called in accordance with the conventions listed in section 4.7 of 130-009-001. All routines in this section are position dependent.

MOD using $DVI and $MLI compute ARG1-(ARG1/ARG2)*ARG2 and return the results in R0. (16 words)

AMOD form a real integer Q=(ARG1/ARG2) using $DVR and AINT. Form a real, P=Q*ARG2, using $MLR. Form the result R=ARG1-P using $SBR and return the result in R0,R1. (23 words)

DMOD this routine is the same as AMOD except that the double precision routines $DVD, $MLD, $SBD, and $DINT are used. Results are returned in R0-R3. (31 words)

10.2.5 Positive Difference

These routines perform positive differencing for integer and real arguments.

These routines are called in accordance with the converntions listed in section 4.7 of 130-009-001. All routines in this section with the exception of IDIM are position dependent.

IDIM return ARG1-ARG2 in R0 if it is positive; otherwise return 0. Call $ERR if overflow occurs. (12 words)

DIM form ARG1-ARG2 using $SBR. Return it if it is positive; otherwise return zero. (21 words)

10.2.6 Minimum/Maximum

These routines search a variable length argument string and select the minimum or maximum value encountered.

These routines are called in accordance with the conventions listed in section 4.7 of 130-009-001. All routines in this section with the exception of MIN0 and MAX0 are position dependent.

MIN0 use n-1 comparisons to determine the minimal argument. (9 words)

AMIN0 use n-1 comparisons and $IR . Return the result in R0,R1. (22 words)

AMIN1 use n-1 calls to $CMR to determine the minimal argument and return it in R0,R1. (58 words)

MIN1 entry into AMIN1 which uses $RI to fix the final result and returns it in R0.

DMIN1 use n-1 calls to $CMD to determine the minimal argument and return it in R0-R3. (46 words)

MAX0 corresponds to MIN0 except that the comparisons are made to determine the maximal argument. (9 words)

AMAX0 entry in AMIN0. corresponds to AMIN0 except that the comparisons are made to determine the maximal argument.

AMAX1 entry in AMIN1. corresponds to AMIN1 except that the comparisons are made to determine the maximal argument.

MAX1 entry in AMIN1. corresponds to MIN1 except that the comparisons are made to determine the maximal argument.

DMAX1 entry in DMIN1. corresponds to DMIN1 except that the comparisons are made to determine the maximal argument.

10.2.7 Random Numbers

The Fortran random number generator, RAN

This function generates pseudo-random numbers uniformly distributed between 0.0 and 1.0; the subroutine uses a congruential random-number algorithm. It is called in accordance with the conventions listed in section 4.7 of 130-009-001 and is position independent.

It is called with two arguments:

RAN(I1,I2)

where I1, I2 is the integer generator base for this call and must be zero for the initial call. The real pseudo-random result will be returned in R0,R1.

I1 and I2 will be updated to a new generator base during each call and may be saved by the user at any point.

Any such saved base may be stored in I1 and I2 at any time to cause the pseudo-random sequence to begin at the point where I1 and I2 were saved.

These numbers have a special form however, and only zero or saved values of I1, I2 should be stored in I1, I2.

(45 words)

An alternate random number subroutine RANDU is provided. It is called with three arguments:

CALL RANDU(I1,I2,F)

I1 and I2 are as for RAN, and F is a real variable. The real pseudo-random result is returned in F; the algorithm used is the same as that for RAN.

(48 words)

10.3 Mathematical Function Package

This section will consist of the following routines:

10.3.1 Sine/Cosine

These routines compute real, double precision and complex sines and cosines.

The routines in this section are called in accordance with the conventions listed in setion 4.7 of 130-009-001. All routines in this section are position dependent.

SIN reduce the real argument to be in the range (-PI/2,+PI/2) radians and expand it in a power series to produce a real value for SIN(x). (see Hastings, page 140) Uses $ADR, $MLR, $SBR, $DVR and $INTR. (118 WORDS,FPU=66)

COS use SIN with the argument increased by PI/2 radians.

DSIN this is the same as SIN except that double precision is used. (see Hart, page 236) Uses $ADD, $MLD, $SBD, $DVD and $DINT. (184 WORDS,FPU=96)

DCOS this is the same as COS except that double precision is used.

CSIN complex sine. Uses SIN, COS, EXP, $FCALL, $ADR, $SBR, $DVR and $MLR.

(122 words)

CCOS entry in CSIN. Computes complex cosine.

10.3.2 Arctangent

These routines compute arctangents for real and double precision arguments.

The routines in this section are called in accordance with the conventions listed in setion 4.7 of 130-009-001. All routines in this section are position dependent.

ATAN reduce the real argument to be in the range (0,1) and expand in a power series to be transformed into ARCTAN(x) in real format. Uses $ADR, $MLR and $DVR. (215 words,FPU=115)

ATAN2 entry into ATAN. If the magnitude of ARG1/ARG2 will be > 2**24 (or ARG2 is 0) The magnitude of the result will be PI/2. Otherwise send ARG1/ARG2 to ATAN.

DATAN this is the same as ATAN except that double precision is used. Uses $ADD, $SBD, $MLD and $DVD. (361 WORDS,FPU=153)

DATAN2 entry into DATAN. If the magnitude of ARG1/ARG2 will be > 2**56 (or ARG2 is 0) The magnitude of the result will be PI/2. Otherwise send ARG1/ARG2 to DATAN.

10.3.3 Square Roots

These routines take real, double precision and complex square roots.

The routines in this section are called in accordance with the conventions listed in setion 4.7 of 130-009-001. All routines in this section are position dependent.

SQRT form an initial estimate based on a linear fit and use a Newton-Raphson iteration to develop square root result. Uses $ADR and $DVR. Call $ERR if the argument is less than zero. (47 words,FPU=34)

DSQRT this is the same as SQRT except that double precision is used. Uses $ADD and $DVD. Call $ERR if the argument is less than zero. (69 words,FPU=43)

CSQRT complex square root. Uses CABS, $FCALL, $ADR, SQRT and $DVR. Call $ERR on underflow.

(74 words)

10.3.4 Powers

These routines handle all exponentiation.

The routines EXP, DEXP and CEXP are called in accordance with the conventions listed in section 4.7 of 130-009-001. All other routines are called in accordance with the conventions listed in section 4.8 of 130-009-001.

All routines in this section are position dependent.

EXP use e**x=2**i(x*log2(e))*e**y. Expand EXP(y) In a continued fraction, scale by i(x*log2(e) And return result in R0,R1. Use $ADR, $DVR, $IR, $MLR, $RI and $SBR. Call $ERR if exponent is not in the range -88.7 to 88.7. (117 words,FPU=72)

DEXP same as EXP except that the double precision routines $ADD, $DVD, $ID, $MLD, $DI and $SBD are used. (210 words,FPU=110)

CEXP complex expotential. Uses SIN, $FCALL, $MLR, $SBR, COS and $MLC.

(36 words)

$PWII use $MLI exponent times. Integer exponent (E) is @SP. Integer base (B) is at 2(SP). Replaces the with integer result of B**E. Call $ERR if base equals zero and exponent is less than or equal to zero. Return 0 if exponent is less than 0. (64 words,EIS=48)

$PWRI use $MLR to do successive squarings and multiplications by the base. Use $DVR if exponent is less than zero. Call $ERR if base equals zero and exponent is less than or equal to zero. (89 words,FPU=36)

$PWDI same as $PWRI except that $MLD and $DVD are used. (129 words,FPU=36)

$PWCI same as $PWDI except that $MLC and $DVC are used. (129 words)

$PWRR to raise a real to a real power call ALOG, $MLR, and EXP. Use $FCALL to call ALOG and EXP. Call $ERR if base is less than or equal to zero and the expontent is not equal to zero. (57 words)

$PWDD to raise a double to a double power use DLOG, $MLD, and DEXP. Use $FCALL to call DLOG and DEXP. Call $ERR if base is less than or equal to zero and the exponent is not equal to zero. (86 words)

$PWRD entry into $PWDD. Append a double word of 0 to the base before going to $PWDD.

$PWDR entry into $PWDD. Append a double word of 0 to the exponent before going to $PWDD.

10.3.5 Logarithms

These routines compute natural and common logarithms of real, double precision and complex quantities.

The routines in this section are called in accordance with the conventions listed in setion 4.7 of

130-309-001. All routines in this section are position dependent.

ALOG expand fractional part in argument in chebyshev polynomial and add ln(2)*exponent part of argument to form result. Call $ERR if argument is less than or equal to zero. Uses $ADR, $DVR, $SBR, $MLR and $IR. (120 words,FPU=59)

DLOG same as ALOG except that double precision is used. Uses $ADD, $DVD, $SBD, $MLD and $ID. (192 words,FPU=89)

ALOG10 entry in ALOG. Returns ln(10).

DLOG10 entry in DLOG. Returns ln(10).

CLOG the complex logarithm function. Uses ATAN2, CABS, $FCALL and ALOG.

(38 words)

10.4 INPUT/OUTPUT CONVERSION ROUTINES

10.4.1 I/O Initialization Routines

10.4.1.1 Formatted I/O

$IOFI has two entry points.(27 words)

$INFI initialize formatted input

Set IOSW for input. Set IOADDR to $FIO. Set IOTADR to $INFR. Then call $IOFI1 to complete the allocation of IOPSTK. Upon return allocate IOFSTK and call $IOFI3 to read the first record, set input buffer pointers and return to caller.

$OUTFI initialize formatted output

Set IOSW for output. Set IOADDR to $FIO. Set IOTADR to $OUTFW. Then call $IOFI1 to complete the allocation of IOPSTK. Upon return allocate IOFSTK and call $IOFI3 to initialize output, set output buffer pointers and return to caller.

10.4.1.2 Unformatted I/O

$IOUI has two entry points.(32 words)

$INI- initialize unformatted input

Same as $INFI except IOADDR is set to $UIO, IOTADR is set to $INR and IOUSTK is allocated.

$OUTI - initialize unformatted output

Same as $OUTFI except IOADDR is set to $UIO, IOTADR is set to $OUTW and IOUSTK is allocated.

10.4.1.3 Random I/O

$IORI has two entry points.(28 words)

$INRI - initialize random input

Same as $INFI except IOADDR is set to $RIO, IOTADR is set to $INRR and IORSTK is allocated.

$OUTRI - initialize random output

Same as $OUTFI except IOADDR is set to $RIO, IOTADR is set to $OUTRW and IORSTK is allocated.

10.4.2 I/O Finalize Routines

$IOF - called to do I/O finalization saves R4 and address I/O stacks then jumps to IOFD, $IOUD or $IORD (71 words)

10.4.2.1 Formatted I/O

IOFD - entry in $IOF. calls $FIO with no more items to process upon return free stack space and return via JMP @(R4)+

10.4.2.2 Unformatted I/O

$IOUD - if input, call $INR to flush the rest of the segments of the record. if output, finalize and call $OUTW to output the last segment. jump to

$IOFX to free stack space and return. (17 words)

10.4.2.3 Random I/O

$IORD - if output, pad rest of segment(s) with zero. call $OUTRW to do output. if input, ignore the rest of the segment(s), set associated variable and jump to $IOFX to free stack space and return. (25 words)

10.4.3 Item Processing Routines

10.4.3.1 Formatted I/O - entry $FIO (670 words)

$FIO if initial call insure that first character is (. initialize scan of format and go to LPARN; else go to place of last exit (REENT).

NOTE: formats must be character strings beginning with a (and ending with a). Therefore, if a read in format is to be used, the user must take care to insure that each character is read into the next byte of the array with no gaps between (eg reading a format into array I when I is a two word integer array will not work properly.)

NOTE also: it is the format specification which governs conversions. The item type is not checked for compatibility except in the case of A conversion. (see $ICA, $OCA)

SCAN accumulate any signed number and return to caller.

FCONT insure that next character is , T /) or ' . if so go to COMMA, TSPEC, SLASH, RPARN, QUOTE. otherwise go to BADSYN.

FCONT1 - insure that next format character is ,) (/ ' A D E F G H I L O P T or X. if so go to COMMA, RPARN, LPARN, SLASH, QUOTE, ASPEC, DSPEC, ESPEC, FSPEC, GSPED, HSPEC, ISPEC, LSEPC, OSPEC, PFACT, TSPEC, XSPEC. otherwise go to BADSYN.

FCONT2 - same as FCONT1 except , and (are not allowed.

COMMA - insure that no numeric preceeded. Call SCAN and go to FCONT2.

PFACT - insure that numeric preceeded. Save number as P scale. Call SCAN and go to FCONT1.

TSPEC - Insure that no numeric preceeded. Call SCAN. Set record pointer to position specified, checking that it is in record bounds. Go to FCONT.

XSPEC - if no preceeding numeric, set to one. If preceeding numeric is less than one call error. set record pointer, check record bounds, call SCAN and go to FCONT1.

HSPEC - Insure that preceeding numeric was present and greater than zero. Check if specification will overflow record (if so , error). Otherwise move specified number of characters to record from format (if output) or vice versa (if input). Call SCAN and go to FCONT1. Any character which the monitor allows to be read in formatted ASCII mode is valid.

SLASH - Insure that no numeric preceeded. Call $INFR or $OUTFW to read next or write last record. Call SCAN and go to FCONT1.

QUOTE - Insure that no numeric preceeded. if input move characters to format from record until an unpaired quote is found. If output, move characters to record from format until an unpaired quote is found moving to record one quote of each pair encountered . Check record overflow while moving is occuring. Call SCAN and go to FCONT1 any character which the monitor allows to be read in formatted ASCII mode is valid.

LPARN - If no preceeding numeric set to one. If less than zero call error. If this is an initializing call (no nesting) Set no cvts done switch, set init switch off. If this is highest level nest save format pointer location, save group count, set no cvts done switch, set current group count and set new nesting. If this is lowest (second) Level nest save unexhausted higher level group count, set current group count, set new nesting. Call SCAN and go to FCONT2.

RPARN - Insure that no numeric preceeds. If this is outer) (no nesting) check if more I/O list items. If not call $OUTFW (if output) and exit to main. If more, test cvts done switch. If off, call error. Otherwise call $INFR or $OUTFW. Determine if there was any nesting. If not, set up to go repeat entire format. If it was, set up to repeat last nesting group. In either case reset cvts done switch. Call SCAN and go to FCONT1. If this is highest level nesting) check if group count is exhausted. If not, set up to repeat group. If so, pop a nesting level. In either case call nscan and go to FCONT1. If this is lowest level nesting) Check if group count is exhausted. If not set up to repeat group. If so pop a level reset current group count with highest level nesting's remaining group count. In either case call SCAN and go to FCONT1.

DSPEC,ESPEC,FSPEC,GSPEC if no numeric preceeds set to one. Insure that preceeding numeric is greater than zero. Save it as repetition count. Call SCAN. Must return positive non zero number to save as field width. Insure that next character is "." call SCAN. Must be non-negative returned to save as D specification Set address of appropriate conversion routine. Go to FMT.

ASPEC, ISPEC, LSPEC, OSPEC if no numeric preceeds set to one. Save as repeat count. Call SCAN. Must return positive nonzero number for field width. Set address of appropriate conversion routine. Go to FMT.

FMT check if more I/O list items. If not call $OUTFW (if output) and return to main. If so push required values onto stack depending upon conversion specification and I/O type. Call conversion routine. Pop required values from stack depending upon conversion specification and I/O type. Check if any conversion error. Return to main to get next item or finalize call.

REENT reenter here check if repeat count is exhausted. If not go to FMT. If so set cvt done switch. Go to FCONT.

ICA do input A conversion.

OCA do output A conversion.

NOTE: For both input and output A conversion, the bytes transferred is determined by the item size and not the format specification Otherwise a conversion will occur as described in the Fortran standard(i.e. Left justification into item with trailing blanks and left most characterss of item into record if item size greater that specified width. Right most characters from record into item and right justification into record from item if specified width greater than item size.)

10.4.3.2 Unformatted I/O - entry $UIO.(58 words)

Input if end of record call error(short record). If end of segment call $INR. Else move number of bytes required from record into item and return to caller.

Output if item will not fit in segment call $OUTW. Move item to record and return to caller.

10.4.3.3 Random I/O - entry $RIO.(52 words)

If end of record call error(short record). If end of segment call $INRR/$OUTRW. Else transfer bytes from item to record(or vice versa) one at a time, checking for end of segment or end of record. Upon completion return to caller.

10.4.4 Item Transmission Routines (for all I/O types)

$IOARG - has 8 entry points. (181 words)

$IOB accept list of addresses for one byte logical variables and call $FIO, $UIO, or $RIO for each item depending on I/O type.

$IOI same as $IOB except items are one word integer variable pointers.

$IOL same entry as $IOI except items are one word logical variable pointers.

$IOJ same as $IOB except items are two word integer variable pointers.

$IOR same as $IOB except that items are two word real variable pointers.

$IOC same as $IOB except items are four word complex variable pointers.

$IOD same as $IOB except items are four word double precision variable pointers.

$IOA accept list of array pointers determine type of each array and call $FIO, $UIO, or $RIO for each element of each array.

Upon completion of item transmission remove list of items from stack and return via JMP @(R4)+

10.4.5 Logical and Physical I/O Routines

Note each entry pair is the same core location

10.4.5.1 Formatted Logical I/O Routine

Entry $INFR, $OUTFW - (240 words)

$INFR - if first call of read statement, determine

1) if device number is valid
2) if file is open (if not call $OPEN)
3) if file is open is it opened for formatted I/O. if not error. if formatted output call $CLOSE, and $OPEN.

on any call

1) call $READ
2) evaluate error returns
3) set buffer pointers on stack
4) remove CR, LF, VT, FF, if present and pad buffer with blanks to standard size
5) return via RTS PC

$OUTFW - if first call of write statement

make determinations of file status similar to those made for input however, do not write a record just set buffer pointers on stack and return.

on non-first call

1) determine if print file. if so generate proper line spacing.
2) call $WRITE after appending CRLF, or CRVT
3) evaluate error returns
4) set buffer pointers on stack
5) return via RTS PC

10.4.5.2 Unformatted Logical I/O Routine

entry $INR, $OUTW - (127 words)

$INR - if first segment call make same determinations as for formatted input

On any call

1) call $READ
2) evaluate error returns
3) set buffer pointers and record control word on stack
4) return via RTS PC

$OUTW - if first segment call make device status determinations

on first call

do not call $WRITE, just set buffer pointers on stack and return.

on non-first call

1) put record control word at start of record
2) call $WRITE
3) evaluate error returns
4) set buffer pointers on stack
5) return via RTS PC

10.4.5.3 Random Logical I/O Routine

entry $INRR, $OUTRW - (168 words)

On first call, does $DEVTB checks and calls $OPEN,if the file is not already open. If the $OPEN fails because the file does not exist, another $OPEN is attempted with the DVHOPN byte set to request allocation of a contiguous file. If this $OPEN fails, $ERR is called.

$INRR - if first segment call

1) make device status determinations
2) calculate number of block desired
3) check if block in core. If not, call $READ. $READ is not called if a full-block WRITE is being done. For a full-block WRITE, $GET is instead called to obtain a monitor core buffer; no physical I/O is done.

4) evaluate error returns
5) set buffer pointers, associated variable address, and length of record (not segment) on stack.
6) return via RTS PC

on non-first call

1) increment block number
2) call $READ, unless full-block WRITE
3) evaluate error returns
4) set buffer pointers on stack
5) return via RTS PC

$OUTRW - if first segment call

same as $INRR

on non-first call

1) call $WRITE
2) if not last call same as $INRR
3) if last call just return

10.4.5.4 $FIND (for random I/O)

This routine does not position the disk mechanism, but it does logically FIND the record by updating the associated variable. FIND is provided only for compatibility purposes; it's use is not recommended. (33 words)

10.4.5.5 Physical I/O Routine $OPEN

$OPEN - does .INIT and .OPEN (I, O, U or C) of files for all I/O types and does error checking. In addition if SETFIL or $INRR/$OUTRW set indications that a contiguous file is to be allocated it is done by $OPEN.

Returns via RTS PC. (145 words)

10.4.5.6 Physical I/O Routine $CLOSE

$CLOSE - does .CLOSE and .RLSE of files for all I/O types;clears various device status words and pointers.

Returns via RTS PC. (34 words)

10.4.5.7 Physical I/O Routine $READ

$READ - performs input reading and error checking for all I/O modes.

1) for formatted I/O does formatted ASCII normal mode reading.
2) for unformatted I/O does formatted binary normal mode reading.
3) for random I/O does block transfers.

Returns via RTS PC. (95 words)

10.4.5.8 Physical I/O Routine $WRITE

$WRITE - entry in $READ. Pushes a flag to indicate a write before executing the $READ code.

10.4.5.9 Physical I/O Routine $GET

$GET - entry in $READ. Used by random-access I/O to get a monitor buffer for full-block WRITEs, for which no prior $READ is required.

10.4.6 Conversion Routines

10.4.6.1 Real and Double Precision

These routines accept field pointers and conversion type codes from $FIO and convert a floating point data item from a character string

$DCI convert an input item according to D format. (384 words)

Calling Sequence:

```
push address of field start
push field width
push D part of W.D
push P format scale factor
```

```
JSR PC,$DCI
```

Will return a four word floating point result on top of the stack. This and all following I/O conversion routines return with the C bit clear unless there was a conversion error, in which case it is set.

$RCI entry in $DCI for E, F or G format. Returns a two word result on top of the stack.

$DCO convert an output item according to D format. (456 words)

Calling Sequence:

```
push address of field start
push field width
push D part of W.D
push P format scale.
Push value to be converted.
JSR PC,$DCO
```

In the event of an output conversion error $ECO, $FCO, $GCO, and $DCO will return a field of ******.......***** as well as the C bit mentioned above.

$FCO convert an output item according to F format. This is an entry point in $DCO and has the same calling sequence.

$GCO convert an output item according to the G format rules. (see ANSI standard x3.9-1966) This is an entry point in $DCO and has the same calling sequence.

$ECO convert an output item according to E format. This is an entry point in $DCO and has the same calling sequence.

10.4.6.2 Integer

These routines act on I and O format requests.

$ICI accept field pointers from $FIO and convert an I format input item to an integer. (98 words)

Calling Sequence:

```
push address of field start
push field width
```

```
JSR PC,$ICI
```

Will return a one word integer on top of the stack.

$OCI this is an entry in $ICI which treats the external field as octal data;the data item is treated as a 16-bit unsigned quantity.

$ICO convert an output item according to I format. (93 words)

Calling Sequence:

```
push address of field start
push field width
push data item
JSR PC,$ICO
```

In the event of an output conversion error $ICO and $OCO will return a field of *****....**** as well as the C bit mentioned above.

$OCO this is an entry in $ICO which converts the internal dataum into an octal character string.

10.4.6.3 Logical

This routine codes and decodes logical true/false I/O requests.

$LCI accept field pointers from $FIO and convert a logical field to a binary value. (31 words)

Calling Sequence is the same as $ICI.

.TRUE. = -1, .FALSE. = 0.

$LCO covert a data item according to L format. (31 words)

Calling sequence is the same as for $ICO.

10.5 Miscellaneous Routines

10.5.1 Subscript Calculations ($SBS and $SBX)

This routine computes and verifies the address of an array element. Called in the polish mode it receives the address of the ADB for the array as the next item in the operator list. (see 130-309-001 3.1.2.6 for a description of the ADB) (49 words,EIS=35)

$SBS1 single subscript. Return A+(I-1)*S in R0.

$SBS2 double subscript. Return A+(I-1+(J-1)*D1)*S in R0.

$SBS3 triple subscript. Return A+(I-1+(J-1)*D1+(K-1)*D1*D2)*S in R0.

A check is always made that subscripts are positive integer quantities. If the /CK switch is used during the compilation , the compiler generates Polish calls $SBX1, $SBX2,$SBX3. The $SBXn entry points check that the actual subscripts are less than or equal to the maximum given in the ADB. $ERR is called if the subscript is out of bounds(65 words, EIS=51).

10.5.2 Deleted

10.5.3 Object Time Error Routine ($ERR/$ERRA 274 words)

$ERR - calling sequence

```
JSR R5,$ERR
BR ANY
.BYTE error class number
.BYTE error number within class
```

NOTE: There is BR after the JSR R5, but it need NOT be to the location following the argument list; the call is not a standard Fortran call.

$ERRA - calling sequence

```
     MOV X,R0
     JSR R5,$ERRA

  X: .BYTE error class number
     .BYTE error number within class
```

Determines if the message number and class number are valid. If not changes them into class zero message zero (invalid message number). Sets byte in error vector $ERRF for the class to 1.

Then determines if this is an immediate abort error (maximum log count=-3). If so issues a fatal monitor diagnostic (F030) with the error class and error number (3 octal characters each) to be displayed.

If this is a no log and exit error (maximum log count =-2) $EXSW is set to three and $EXIT is called.

If this is a no log and ignore error (maximum log count=-1) control is transferred to the location specified in the transfer address word.

Otherwise, the device table ($DEVTB) Is checked to determine if a logging device exists and is available for formatted ASCII output (i.e. Is already open for formatted ASCII output or is closed and no define file has been done; and is not an actual or potential contiguous file)

Monitor diagnostic I353 is issued if the logging device is not available. The additional information is the class and error number in octal ASCII.

Otherwise the existence of the message contiguous file is determined. If the file exists the appropriate block is read and the proper message within that block is written (formatted, ASCII, dump mode) on the logging device.

Monitor diagnostic I352 is issued if an end of file is detected on the logging device.

NOTE: The format of the message file is one sixty four character ASCII message per logical record. Each message of the form

```
FORTcccnnn - message text
```

Where ccc is the class number in decimal ASCII and nnn is the message number in decimal ASCII. The entire message is sixty four characters or less including two characters at the beginning and end of the text which will be overlayed with a CR,LF.

NOTE also: $RANDM (10.2.12) will be used to determine the physical block and displacement based on the block size of the device, the record number of the start of the error class (from $ERRC 5.4.1) and the error number.

If no message file exists

'FORTcccnnn' is written on the logging device. Using the $ERRWK area to format the message. After writing, the routine $TRCBK is called to write a subroutine trace back (if available).

Error routine end up

The error class maximum log count word is checked for zero (log message and ignore). If non zero the log count is incremented by one. If it is then equal to the maximum log count word, $EXSW is set to two and $EXIT is called, else $EXSW is set to one. The error routine now exits via the transfer address word in such a way that it appears that the routine which called error actually called the routine whose location is in the transfer address word. The normal contents of the transfer address word is the global $RTS which is the address in the error routine of an RTS R5 instruction. Thus the error routine will normally return to the calling program.

10.5.3.1 Traceback (95 words)

$TRCBK - calling sequence

R0-R1,R4 unused
R2 = address of DDB pointer followed by a four word line buffer header
R3 = set to one if EOF on writing, else cleared
R5 = address of fourteen byte buffer ($ERRWK)

```
JSR PC,$TRCBK
```

```
NAME                SEQ
NNNNNN              SSSSSS
NNNNNN              SSSSSS
   .                 .
   .                 .
   .                 .
MAIN.               SSSSSS
```

Writing is formatted, ASCII, dump mode. No abnormal conditions are checked except end of file in which case a one is returned in R3.

10.5.3.2 Name Option ($NAM 24 words,FPU=35)

$NAM - is a routine invoked at MAIN program and subroutine entrance. It adds the current subroutines name entry to the end of the subroutine name chain (in $NAMC).

In the 11/45 FPULIB version of the OTS, $NAM checks for entry to the MAIN program. On entry to MAIN, the Floating Point status register is initialized for floating point interrupts and an interrupt vector for FPP interrupts is established through a monitor call.

$RET - is the complement to $NAM. It removes the current subroutine entry from the end of the subroutine name chain at subroutine return.

10.5.3.3 Sequence Option ($SEQ 3 words)

$SEQ - saves the current statements sequence number in $SEQC.

10.5.3.4 11/45 Floating Point Error Routine $FPERR

$FPERR - entry point for any Floating Point Processor interrupts. Checks FPP error code and calls $ERRA with appropriate parameter. In the case of FPP underflow interrupts, picks up the instruction causing the underflow, extracts accumulator field, and clears the appropriate floating accumulator to 0.0 (FPU=43 words).

10.5.4 Exit Routine (14 words)

$EXIT (EXIT) - this routine is called by the user or by the STOP statement or by error processing routines to do job termination. $EXIT calls the subroutine $CLSUP to cycle through $DEVTB and close all files opened by OTS. If $EXIT was not called by error processing, it exits to the monitor via EMT 60. Otherwise, it examines $EXSW and uses the monitor diagnostic facility to log a message before the EMT 60. The diagnostic is informational (I351) and the value displayed with it is -

000001 if $EXSW = 2 (error reached maximum count)
000002 if $EXSW = 3 (error was no log and call exit)

Calling Sequence

```
     JSR R5,$EXIT (EXIT)
```

Return to monitor

```
     EMT 60
```

10.5.5 Rewind/End File ($RWIND/$ENDFL 33 words)

$RWIND/$ENDFL - determine if device is open. If not return, else call $CLOSE then return. Called in the polish mode with the device number on the top of the stack.

Return via JMP @(R4)+

10.5.6 Backspace ($BCKSP 83 words)

$BCKSP - will call $CLOSE, $OPEN and $READ. It will read n-1 records processed before the close. If the device is closed or if open for random I/O no action is taken. Backspace is undefined on a non-file structured device.

$BCKSP is called in the polish mode with the device number on the top of the stack.

Return via JMP @(R4)+

10.5.7 Define File ($DEFIL 39 words)

$DEFIL - does random I/O device table ($DEVTB) entry initialization. If a previous DEFINE FILE has occured for this device number and no $CLOSE has occurred, this DEFINE FILE is ignored.

Calling Sequence

```
     MOV  X,R4
     JMP @(R4)+

  X: $DEFIL
     address of device number
     address of maximum number of records
     address of record length
     address of associated variable
```

Return via JMP @(R4)+

10.5.8 SETFIL Subroutine (140 words)

SETFIL - is a user callable subroutine which allows the user to override (at run time) the default or previously specified values in the device table entry for a specific device number.

Calling Sequence

```
JSR R5,SETFIL
BR NEXT
address of device number (integer)
address of file name (ten character ASCII string)
address of error value variable (integer)
address of physical device mnemonic  (three  char-
acter ASCII string)
address of unit number (integer)
address of UIC (integer)
address of protect code (integer)
address of allocate file switch (integer)
address of record length to allocate (integer)
address of number of records to allocate (integer)
NEXT = .
```

NOTE:1) Any trailing sequence of arguments may be omitted.

2) If the device is open or the device number is invalid no action is performed.

3)If the file name argument is less than 10 characters long, it must be in .ASCIZ format; the compiler generates .ASCIZ for hollerith literals appearing as subroutine arguments.

The error value variable is used for two purposes. 1) To indicate errors in SETFIL processing and 2) when an ERR= exit is taken from I/O processing to indicate what type of I/O error occurred. The only error so indicated by SETFIL is error value variable = -1 which means that the allocate file switch was one and two arguments did not follow.

The allocate file switch is supplied to allow the user to indicate that he wishes a contiguous file to be allocated at file open time.

Switch values:

2 = allocate a contiguous file for random I/O in which case the DEFINE FILE statement will set the record length and the number of records in the file.

1 = allocate a contiguous file for non-random I/O. In which case the following two arguments are used to set the record length and the number of records in the file.

(see 5.1.2 note 1)

SETFIL returns via RTS R5

10.5.9 SETERR Subroutine (26 words)

SETERR - is a user callable subroutine which allows the user to override (at run time) the default maximum count for any class of error (except classes 0,8).

Calling Sequence

```
JSR R5,SETERR
BR NEXT
address of error class number
address of override maximum count value
NEXT = .
```

If the class number is not valid no action is performed.

All values of the maximum count are allowed, however values less than minus three are set to minus three. (see 5.4.1)

10.5.10 Stop (10 words)

$STOP - generates a monitor diagnostic informational message (I350) along with the octal value supplied on the STOP statement and then calls $EXIT.

$STOP is called in the polish mode. Upon entry R4 points to the address of the stop octal value.

Returns via JMP @(R4)+.

10.5.11 Pause (15 words)

$PAUSE - generates a monitor diagnostic action required message (A005) along with the octal value supplied on the PAUSE statement. The user must respond to the request for action. If he replies CONTINUE, $PAUSE returns via JMP @(R4)+

$PAUSE is called in the polish mode. Upon entry R4 points to the address of the pause octal value.

10.5.12 OTS I/O Support Function Routines

$FNDEV - calling sequence

```
R0 = number of device
R1 = address of $DEVTB entry (returned)
R2-R5 = unused
JSR PC,$FNDEV
```

Uses the device number in R0 to obtain the address of the $DEVTB entry for that device and returns the address in R1. If invalid device number return zero in R1.

(16 words)

$IOSET - calling sequence

```
R0 = number of device
R1 = address of $DEVTB entry for device num in R0
R2 = address of $IOBUF (or equivalent)
R3 is destroyed accross the call
R4-R5 unused
JSR PC,$IOSET
```

Sets up the file block and link block addressed by R2 with the device table entry information addressed by R1 and the device number in R0.

(55 words)

$RANDM - calling sequence

```
push record length
push record num-1
push block size
JSR PC,$RANDM
```

Uses double precision multiplication to calculate the block number and block displacement for direct access I/O or contiguous file allocation on return the stack is modified. Record length is replaced by block num, record num-1 is replaced by block displacement, and block size is left unmodified.

NOTE: This routine requires that block size be a power of two.

(48 words)

$CLSUP - calling sequence

```
JSR   PC,$CLSUP
```

Cycles through all entries of $DEVTB closing any files which are open. Called by $EXIT at program end and by the RUN subroutine.

(29 words)

10.5.13 TSTERR Subroutine (16 words)

TSTERR is a user callable subroutine which allows the user to test for occurrence of run-time errors.

```
CALL TSTERR(I,J)
```

tests and resets the error byte for class I in the error class table ($ERRF(I)).Returns J=1 if an error in error class I has occurred; returns J=2 otherwise. The error flag byte for class I is reset to 0.

10.5.14 SSWTCH Subroutine (20 words)

SSWTCH is a user callable subroutine which allows the user program to read the console switch register.

```
CALL SSWTCH(I,J)
```

tests the I'th bit of the console switch register and returns

```
J = 1 if bit I = 1 (UP)
J = 2 if bit I = 0 (DOWN)
```

J is not modified if I is out of range; no error indications are given.

10.5.15 DATE and TIME Subroutines (30 words)

DATE and TIME are entry points in the module $TIMnn.

DATE is a user callable subroutine which obtains the date as an ASCII string from the DOS monitor.

```
     CALL DATE(A)
```

returns the date as a 9-byte ASCII string in A, in the form

```
     19-JUN-72
```

Uses the DOS DATE/TIME conversion, EMT 66.

TIME is a user callable subroutine which obtains the time in either ASCII or binary form.

```
     CALL TIME(A)
```

returns an 8-byte ASCII string in A representing the current time in the form HH:MM:SS . Uses DOS EMT 66.

```
     CALL TIME(I1,I2)
```

returns the current time in clock ticks in I1,I2. High order 15 bits in I1, low order 15 bits in I2. Uses DOS EMT 41 to obtain time from the System Vector Table.

```
     CALL TIME(A,I1,I2)
```

converts the clock tick time in I1, I2 to an 8-byte ASCII string in A. Uses DOS EMT 66.

10.5.16 Program Overlays: LINK, RETURN (150 words)

The entry points LINK and RETURN in the LINK subroutine support the PDP-11 FORTRAN program overlay facility. The call forms are:

```
     CALL LINK('dev:file.ext[uic]')
```

and

```
     CALL RETURN
```

The argument to LINK specifies the name of an overlay file to be loaded; DOS defaults apply if any portions of the argument string are omitted. The argument string may contain a maximum of 25 characters, and it must be in .ASCIZ format (a 0 byte terminating the string).

If the entry to LINK occurs in the resident section, R0-R5 are saved in local storage in LINK. A subsequent CALL RETURN restores these saved registers.

The head of the subroutine name chain $NAMC is reset to 0. LINK uses EMT's .CSI1 and .CSI2 to scan the argument and set up the file block. It then calls the DOS RUN EMT to unconditionally move the stack and load the requested overlay. If any errors occur, DOS fatal error messages are output using IOT calls. After successful loading, control is transferred to the transfer address of the loaded module (first executable statement of the MAIN program for a Fortran-compiled program).

RETURN, called from a non-resident section, restores R0-R5 and does an RTS R5. This returns control to the statement following the last CALL LINK issued from the resident section. No checking of the validity of the call is done.

NOTE: The stack pointer SP and all elements on the stack (eg subroutine return addresses) must remain undisturbed over a CALL LINK - CALL RETURN sequence. Destruction of values on the stack will corrupt the overlay system in unpredictable ways.

10.5.17 RUN Subroutine

RUN is a user callable subroutine, actually an alternate entry point in the LINK subroutine. The RUN subroutine deletes all items on the processor stack, calls $CLSUP to close files opened by $OPEN, and uses the LINK code to invoke the DOS RUN EMT. The calling form is:

```
CALL RUN('dev:file.ext[uic]')
```

The argument is interpreted as in CALL LINK.

11.0 FUNCTIONAL DESCRIPTION AND OPERATION

N/A

12.0 PROGRAMMING CONSIDERATIONS

N/A

13.0 PREPARATION AND/OR SYSTEM BUILD

13.1 Assembly Instructions

Included in the source files supplied for the Fortran OTS are files named ASMOTS.nnA and BLDOTS.nnA. These are the BATCH-11 control files used in assembling the Version nnA OTS. These files assume a system configuration with two RK03/RK05 disk drives, but they may be editted for other configurations.

Using PIP the following routines should be combined into a single file 'SYMBOL.PAL'.

```
IOPSTK.PAL
IOFSTK.PAL
IOUSTK.PAL
IORSTK.PAL
IOBUF.MAP
DEVTB.MAP
END.PAL
```

Using PIP the following routines should be combined into a single file 'MAP.PAL'.

```
IOBUF.MAP
DEVTB.MAP
END.PAL
```

NOTE:In the source tapes supplied by the DEC Program Library, the files SYMBOL.PAL and MAP.PAL are supplied as complete files. The above PIP steps may be omitted.

Using MACRO-11 the following routines should be assembled with 'SYMBOL.PAL' as a trailer (I.E. FILE,LP:<FILE,SYMBOL.PAL)

```
ECDnn.PAL
IEDnn.PAL
BSPnn.PAL
FIOnn.PAL
NFRnn.PAL
INRnn.PAL
NRRnn.PAL
ARGnn.PAL
IOCnn.PAL
```

IOFnn.PAL
IFInn.PAL
IRDnn.PAL
IRInn.PAL
IUDnn.PAL
IUInn.PAL
OPNnn.PAL
RADnn.PAL
RIOnn.PAL
UIOnn.PAL

Using MACRO-11 the following routines should be assembled with 'MAP.PAL' as a trailer.

CLSnn.PAL
CLPnn.PAL
DFLnn.PAL
FNDnn.PAL
ERRnn.PAL
ISTnn.PAL
RWDnn.PAL
SFLnn.PAL

Using MACRO-11 all other routines are assembled individually.

NOTE: 'nn' in the above file names is the current version number.

N.B. - No assembly errors are expected for any file.

13.1.1 EAE Conditional Assembly

The following routines contain conditional code for the PDP-11/20 Extendend Arithmetic Element (KE11-A).

SBSnn.PAL
SBXnn.PAL
MLInn.PAL
MLRnn.PAL
MLDnn.PAL
IRnn.PAL
DVRnn.PAL
DVInn.PAL
ANTnn.PAL
ADDnn.PAL
ADRnn.PAL

RInn.PAL
DNTnn.PAL
POLnn.PAL

Using MACRO-11 assemble the above routines with EAE.PAL as a header (I.E. FILE,LP:<EAE,FILE) to obtain their EAE versions.

13.1.2 FPU Conditional Assembly

The following routines contain conditional code for the Floating Point Processor on the PDP-11/45.

NAMnn.PAL
SINnn.PAL
DSNnn.PAL
PRInn.PAL
PDInn.PAL
DSQnn.PAL
SQTnn.PAL
ATNnn.PAL
DTNnn.PAL
EXPnn.PAL
DXPnn.PAL
ALGnn.PAL
DLGnn.PAL
RDnn.PAL
TSInn.PAL
MLCnn.PAL
DVCnn.PAL
MLRnn.PAL
MLDnn.PAL
IRnn.PAL
DRnn.PAL
DVRnn.PAL
DVDnn.PAL
CMRnn.PAL
CMDnn.PAL
ANTnn.PAL
ADDnn.PAL
ADCnn.PAL
ADRnn.PAL
RInn.PAL
DNTnn.PAL
SERnn.PAL
FPRnn.PAL
POLnn.PAL

Using MACRO-11 assemble the above routines with FPU.PAL as a header to obtain their FPU versions.

In addition, using MACRO-11 assemble the following routines with EIS.PAL as a header.

PIInn.PAL
SBSnn.PAL
SBXnn.PAL
MLInn.PAL
DVInn.PAL

13.1.3 EIS(MULDIV) Conditional Assembly

The following routines contain conditional code for the multiply/divide and multi-bit shift features of the PDP-11/45, a standard part of the 11/45 instruction set. The symbols EIS and MULDIV are used interchangeably in the conditionalized code for this assembly option; the resulting OTS is referred to as EISLIB in DEC documents.

PIInn.PAL
SBSnn.PAL
SBXnn.PAL
MLInn.PAL
MLRnn.PAL
MLDnn.PAL
DVRnn.PAL
DVInn.PAL
ANTnn.PAL
ADDnn.PAL
ADRnn.PAL
RInn.PAL
DNTnn.PAL
POLnn.PAL

Using MACRO-11 assemble the above routines with EIS.PAL to obtain their EIS versions.

13.1.4 RSX Conditional Assembly

The following routines contain conditional code for the RSX11C and RSX11B operating systems.

PDUnn.PAL
BSPnn.PAL
NFRnn.PAL
INRnn.PAL
NRRnn.PAL
RWDnn.PAL
NAMnn.PAL
RSTnn.PAL
FDVnn.PAL
SFLnn.PAL
OPNnn.PAL
SERnn.PAL
SEQnn.PAL
ERRnn.PAL
TRCnn.PAL
ERCnn.PAL
EXTnn.PAL
ISTnn.PAL
DVBnn.PAL

Using MACRO-11 assemble the above routines with RSX.PAL as a header to obtain their RSX versions.

13.2 Library Ordering Constraints

The current library ordering is considered optimal. This ordering is given in the library listing obtainable from LIBR-11.

New modules may be inserted at any point in the library after module one without affecting the current ordering.

If it is desired to move a module within the library programming department memo 130-311-006 lists inter module dependencies.

Any module required by another should physically follow that module within the library.

13.3 Building the Diagnostic File

The procedure for constructing a diagnostic file is documented in the "Getting FORTRAN on the Air" document.

FORMATTED I/O

$IOFI $INFI, $OUTFI	INITIALIZE FORMATTED I/O REQUIRES $INFR, $IOC, $FIO	27
$FIO	FORMAT SCANNING AND CONVERSION REQUIRES $ERR, $IOARG, $INFR, $DCO, $ICO, $LCO, $DCI, $ICI, $LCI	670
$IOF $IOERR, $IOFX	END OF I/O LIST PROCESSING FOR FORMATTED, UNFORMATTED AND RANDOM I/O REQUIRES $ERR, $EXIT, $IOUD, $IORD	71
$INFR $OUTFW	CONTROLS FORMATTED INPUT/OUTPUT REQUIRES $OPEN, $CLOSE, $READ $FNDEV, $IOSET, $IOF $FLDEV	240
$FLDEV	DEFINES FORTRAN LISTING DEVICES FOR $INFR	4
$DCO $ECO, $FCO, $GCO	OUTPUT CONVERSIONS D,E,F,G	456
$ICO $OCO	OUTPUT CONVERSIONS I,O	93
$LCO	OUTPUT CONVERSIONS L	31
$DCI $RCI	INPUT CONVERSIONS D,E,F,G	384
$ICI $OCI	INPUT CONVERSIONS I,O	98
$LCI	INPUT CONVERSIONS L	31

UNFORMATTED I/O

$IOUI $INI $OUTI	- INITIALIZE UNFORMATTED I/O REQUIRES $UIO, $INR, $IOC, $IOUD	32
$UIO	- ITEM TRANSFERS TO AND FROM UNFORMATTED I/O REQUIRES $ERR, $IOARG	58
$IOUD	- END OF I/O LIST PROCESSING FOR UNFORMATTED I/O REQUIRES $IOF, $INR	17
$INR $OUTW	- CONTROLS UNFORMATTED I/O REQUIRES $OPEN, $CLOSE, $READ, $FNDEV, $IOSET, $IOF	127

RANDOM I/O

$IORI $INRI, $OUTRI	- INITIALIZE RANDOM I/O REQUIRES $INRR, $IORD, $RIO, $IOC	28
$RIO	- TRANSFERS TO AND FROM RANDOM I/O RECORDS REQUIRES $ERR, $IOARG, $INRR	52
$IORD	- END OF I/O LIST PROCESSING FOR RANDOM I/O REQUIRES $IOF, $INRR	25
$INRR $OUTRW	- CONTROLS RANDOM I/O REQUIRES $OPEN, $CLOSE, $READ, $IOSET, $FNDEV, $RANDM, $IOF	168
$DEFIL	- OBJECT TIME DEFINF FILE FOR RANDOM I/O REQUIRES $FNDEV, $ERR, $EXIT	39
$FIND	- OBJECT TIME FIND ROUTINE FOR RANDOM I/O	33
$RANDM	- BLOCK DISPLACEMENT FOR RANDOM I/O	48

INTERNAL COMMON I/O ROUTINES

$IOARG - ITEM TRANSMISSION ROUTINES FOR ALL I/O TYPES REQUIRES $MLI $IOA, $IOB, $IOL, $IOI, $IOJ, $IOR, $IOD, $IOC, $IOELM	181
$IOC - MISCELLANEOUS COMMON SUBROUTINES USED BY I/O	30
$OPEN - DOES .INIT AND .OPEN OF PHYSICAL DEVICE	150
$CLOSE - DOES .CLOSE AND .RLSE OF PHYSICAL DEVICE	34
$FNDEV - GET ADDRESS OF DEVICE TABLE ENTRY	16
$CLSUP - CLOSE FILES AT PROGRAM END	29
$IOSET - SET UP FILE AND LINK BLOCK FROM DEVICE TABLE ENTRY	55
$READ/$WRITE - READ OR WRITE A RECORD	95

THESE ROUTINES ARE REQUIRED FOR FORMATTED, UNFORMATTED AND RANDOM I/O.

ERROR HANDLING ROUTINES

$ERR - OBJECT TIME ERROR HANDLER REQUIRES $EXIT, $TRCBK	274
$FPERR - 11/45 FPP INTERRUPT ROUTINE	43
$TRCBK - NAME, SEQUENCE TRACE OF SUBROUTINE CHAIN	95
$SEQ - SAVES SEQUENCE NUMBER OF CURRENT FORTRAN STATEMENT	3
$NAM - CHAINS SUBROUTINE NAMES	24
$RET SAVES END OF CHAIN POINTER	

NON-REENTRANT AREAS

$OTSV - CONTAINS POINTERS TO $DEVTB, $IOBUF, $ERRC $ERRWK, AND STORAGE FOR $NAMC, $SEQC, $EXSW, $ERRWK $EXSW, $NAMC, $SEQC	13
$DEVTB - OBJECT TIME DEVICE TABLE (ENTRIES FOR 8 DEVICES)	172
$IOBUF - I/O BUFFER, INCLUDES LINK AND FILE BLOCKS	83
$ERRC - ERROR CLASS TABLE USED BY $ERR	42

MISC. OBJECT TIME SUPPORT

$STOP - OBJECT TIME STOP ROUTINE	10
$PAUSE - OBJECT TIME PAUSE ROUTINE	15
EXIT - OBJECT TIME EXIT ROUTINE	14
$BCKSP - OBJECT TIME BACKSPACE ROUTINE REQUIRES $ERR, $CLOSE, $OPEN $READ, $IOSET, $FNDEV	89
$RWIND/$ENDFL - OBJECT TIME REWIND/ENDFILE ROUTINE REQUIRES $CLOSE, $FNDEV, $ERR, $EXIT	36

GENERAL POLISH ROUTINES

```
----------------------
$POLSH - ENTER POLISH MODE                                   2
$PSH - PUSH AN ADDRESS OR                                    2

       VALUE ON THE STACK
$POP2 - POP AND INTEGER OR LOGICAL ITEM                      2

$POP1
$POP 5 - POP A DOUBLE, COMPLEX, OR REAL ITEM                 8
$POP4,
$POP3,
$POP4A,
$POP4B
$PUT - PUTS ONE, TWO OR FOUR WORDS                           5
$PUT5, FROM THE STACK AT ADDRESS
$PUT3, SPECIFIED IN R0. R0 PREVIOUSLY
$PUT2, SET BY THE SUBSCRIPT ROUTINE
$PUT1
$GET - GET AN ITEM FROM ADDRESS SPECIFIED                    8
$GET5, IN R0 AND PLACE IT ON THE
$GET4, STACK. R0 PREVIOUSLY SET BY THE
$GET3, SUBSCRIPT ROUTINE.
$GET2,
$GET1
$PHR  - PLACES ONE, TWO OR FOUR ITEMS                        5
$PSHR5, ON THE STACK FROM R0-R3.
$PSHR4, USED AFTER FUNCTION CALLS.
$PSHR3,
$PSHR2,
$PSHR1
$SVSP - SAVES STACK ADDRESS                                  2
           AT ADDRESS SPECIFIED.
           USED FOR ADDRESS SUBSTITUTION
           IN SUBROUTINE CALLS.

$SBS - SUBSCRIPT CALCULATION ROUTINE                        55
$SBS1, SETS R0 TO POINT TO ARRAY ELEMENT.
$SBS2,
$SBS3
$NEG - NEGATION FOR INTEGER, REAL AND                       20
$NGD, DOUBLE PRECISION ITEMS.
$NGI,
$NGR
$NGB
$NGC
```

SUBROUTINE RELATED POLISH ROUTINES

$ADJ	INITIALIZE ADJUSTABLE DIMENSION	16
$PSHP	PUSH A PARAMETER ADDRESS OR VALUE ON THE STACK	4
$SVA	GET AN ARRAY ADDRESS FROM THE ARRAY DESCRIPTOR BLOCK.	2
$SVE	SAVE THE ADDRESS OF A SINGLE ARRAY ELEMENT	2
$SVP	SAVE THE ADDRESS OF A PARAMETER	4
$POPP3	POP A REAL PARAMETER REQUIRES $POP5	5
$POPR5 $POPR4, $POPR3	POP TWO OR FOUR ITEMS FROM THE STACK, PLACE IN R0-R3. USED BY EXTERNAL FUNCTIONS	8
$POPP5 $POPP4	POP A FOUR WORD PARAMETER REQUIRES $POP5	5

BYTE MODE RELATED POLISH ROUTINES

$POPP0 $POPR0	POP A BYTE PARAMETER USED BY SUBROUTINES AND EXTERNAL SUNCTIONS	7
$ANB	LOGICAL AND OF BYTE OPERANDS	5
$ORB	LOGICAL OR OF BYTE OPERANDS	4
$NTB	LOGICAL NOT OF BYTE OPERANDS	4
$CMB	BYTE COMPARE ROUTINE USED BY LOGICAL IF	5
$BYTE $GET0, $PUT0, $PSHR0, $POP0	COMMON BYTE POLISH ROUTINES	8

LOGICAL IF SUPPORT

$CMD	DOUBLE COMPARE ROUTINE USED BY LOGICAL IF	31
$CMI	INTEGER COMPARE ROUTINE USED BY LOGICAL IF	2
$CMR	REAL COMPARE ROUTINE USED BY LOGICAL IF	23
$GLE $LE, $LT, $EQ, $NE, $GE, $GT	TESTS FOR LOGICAL OPERATORS LT, EQ, GT, ETC.	13
$TRTST	CONTROLS TRANSFER OF LOGICAL IF	6
$ANI	LOGICAL AND OF INTEGER OPERANDS	3
$NTI	LOGICAL NOT OF INTERGER OPERANDS	2
$ORI	LOGICAL OR OF INTEGER OPERANDS	2

GO TO SUPPORT

$TR	UNCONDITIONAL GO TO	2
$TRX	COMPUTED GO TO REQUIRES $ERR	18
$TRA $AS	ASSIGN AND ASSIGNED GO TO WITH NO LIST	4
$TRAL	ASSIGNED GO TO WITH LIST REQUIRES $ERR	12
$ASP $POPP2, $POPP1, $POPR2, $POPR1	ASSIGN TO DUMMY PARAMETER INCLUDES POP ROUTINE FOR INTEGER OR LOGICAL PARAMTERS	9

ARITHMETIC IF

$TSI $TSR, $TSD	CONTROLS TRANSFER FOR ARITHMETIC IF	14

DO STATEMENTS

$ENDDO	DO STATEMENT END PROCESSING	14
$ENDDP	DO STATEMENT END PROCESSING DO PARAMETERS PASSED TO SUBPROGRAMS.	27

TYPE CONVERSIONS

$RD	REAL TO DOUBLE	9
$RC	REAL TO COMPLEX	
$IR	INTEGER TO REAL	32
$ID	INTEGER TO DOUBLE	
$IC	INTEGER TO COMPLEX	
$DR	DOUBLE TO REAL REQUIRES $ERR	21
$RI	REAL TO INTEGER	45
$DI	DOUBLE TO INTEGER	
$BI,	BYTE TO INTEGER	5
$IB,	INTEGER TO BYTE	
$BL,	BYTE TO LOGICAL	
$LB	LOGICAL TO BYTE	
$CI	COMPLEX TO INTEGER	20
$CR	COMPLEX TO REAL	
$DC	DOUBLE TO COMPLEX	
$CD	COMPLEX TO DOUBLE	
$BC	BYTE TO COMPLEX	8
$BD	BYTE TO DOUBLE	
$BR	BYTE TO REAL	
$CB	COMPLEX TO BYTE	21
$DB	DOUBLE TO BYTE	
$RB	REAL TO BYTE	

MULTIPLICATION

$MLI	MULTIPLE INTEGER REQUIRES $ERR	56
$MLR	MULTIPLY REAL REQUIRES $ERR	117
$MLD	MULTIPLE DOUBLE REQUIRES $ERR	206
$MLC	MULTIPLY COMPLEX REQUIRES $MLR, $ADR	64

DIVISION

$DVI	DIVIDE INTEGER REQUIRES $ERR	43
$DVR	DIVIDE REAL REQUIRES $ERR	139
$DVD	DIVIDE DOUBLE REQUIRES $ERR	230
$DVC	DIVIDE COMPLEX REQUIRES $ADR, $SBR, $MLR, $DVR, $ERR	130

EXPONENTIATION

$PWRI -	POWER REAL TO INTEGER REQUIRES $MLR, $DVR, $ERR, $POLSH	89
$PWDI -	POWER DOUBLE TO INTEGER REQUIRES $MLD, $DVD, $ERR, $POLSH	129
$PWCI -	POWER COMPLX TO INTEGER REQUIRES $MLC, $DVC, $ERR, $POLSH	129
$PWII -	POWER INTEGER TO INTEGER REQUIRES $MLI, $ERR, $POLSH	64
$PWRR -	POWER REAL TO REAL REQUIRES EXP, ALOG, $MLR $FCALL, $ERR, $POLSH	57
$PWDD, $PWDR, $PWRD	POWER DOUBLE TO DOUBLE POWER DOUBLE TO REAL POWER REAL TO DOUBLE REQUIRES DEXP, DLOG, $MLD $FCALL, $ERR, $POLSH	86

ADDITION AND SUBTRACTION

$ADR,	REAL ADDITION	176
$SBR	REAL SUBTRACTION	
	REQUIRES $ERR	
$ADD,	DOUBLE ADDITION	290
$SBD	DOUBLE SUBTRACTION	
	REQUIRES $ERR	
$ADI	INTEGER ADDITION	6
$SBI	INTEGER SUBTRACTION	2
$ADC,	COMPLEX ADDITION	44
$SBC	COMPLEX SUBTRACTION	
	REQUIRES $ADR	

INTERNAL MATH ROUTINES

$FCALL - USED FOR CALLING SINGLE ARGUMENT FORTRAN FUNCTIONS FROM WITHIN FORTRAN FUNCTIONS	12
$DINT - FINDS THE INTEGER PART OF A DOUBLE PRECISION NUMBER	53

$FCALL IS REQUIRED BY $PWRR, $PWDD, CABS, CEXP, CLOG, CSIN, CSQRT AND TANH.

$DINT IS REQUIRED BY DSIN AND DMOD.

FUNCTIONS AND SUBROUTINES	
DATE, TIME SUBROUTINES	30
LINK, RETURN, RUN : OVERLAYS	150
PDUMP - FORTRAN PDUMP ROUTINE REQUIRES $OUTFI, $IOF, $ICO, $DCO	87
SETFIL - PROVIDES A MEANS TO OVERRIDE THE DEVICE TABLE DEFAULTS	140
SETERR - PROVIDES A MEANS TO OVERRIDE ERROR CLASS TABLE DEFAULTS	26
SSWTCH - READ CONSOLE SWITCHES	20
TSTERR - TEST RUNTIME ERROR SWITCHES	16

STANDARD FORTRAN FUNCTIONS

ABS		8
AIMAG		6
AMIN0, AMAX0	REQUIRES $IR, $POLSH, $POPR3	22
ALOG, ALOG10	REQUIRES $ADR, $DVR, $SBR, $MLR, $IR, $POLSH	120
AMOD	REQUIRES $DVR, $INTR, $MLR, $SBR, $POLSH, $POPR3	23
ATAN, ATAN2	REQUIRES $ADR, $MLR, $DVR, $POLSH, $POPR3	215
CABS	REQUIRES $DVR, $MLR, $ADR, $FCALL, SQRT, $POLSH	74
CEXP	REQUIRES SIN, $FCALL, $MLR $SBR, COS, $MLC, $POLSH,$PSHR3,$POPR4	36
CLOG	REQUIRES ATAN2, CABS, $FCALL, ALOG	38
CMPLX		9
CONJG		11
CSIN, CCOS	REQUIRES SIN, COS, EXP, $FCALL, $ADR, $SBR, $DVR, $MLR	122
CSQRT	REQUIRES CABS, $FCALL, $ADR, SQRT, $DVR, $ERR	74
DABS		10
DATAN, DATAN2	REQUIRES $ADD, $SBD, $MLD, $DVD $POLSH,$POPR4	361
DBLE		7
DEXP	REQUIRES $ADD, $DVD, $SBD, $MLD, $ID, $DI, $ERR, $POLSH, $POPR4	210
DIM	REQUIRES $SBR, $POLSH	21
DLOG, DLOG10,	REQUIRES $SBD, $ADD, $DVD, $MLD, $ID, $ERR, $POLSH, $POPR4	192
DMIN1,	REQUIRES $CMD, $POLSH	46
DSIGN		15
DSIN, DCOS	REQUIRES $ADD, $MLD, $SBD, $DVD $DINT, $POLSH, $POPR4	181
DSQRT	REQUIRES $ADD, $DVD, $ERR, $POLSH	69
EXP	REQUIRES $ADR, $DVR, $IR, $MLR $RI, $SBR, $ERR, $POLSH	117
FLOAT	REQUIRES $IR, $POLSH, $POPR3	8
DMOD	REQUIRES $DVD, $MLD, $SBD, $DINT $POLSH,$POPR4	31
IABS	REQUIRES $ERR	10
IDIM	REQUIRES $ERR	12
INT, IDINT	REQUIRES $RI, $POLSH	11
IFIX	REQUIRES $ADR, $RI, $POLSH	11
ISIGN	REQUIRES $ERR	14
MAX0		9
AMAX1, MAX1, AMIN1,	REQUIRES $CMR, $RI, $POLSH	58

MIN1 MIN0		9
MOD	REQUIRES $DVI, $MLI, $POLSH	16
RAN -	RANDOM NUMBER GENERATOR	45
REAL		5
SIGN		11
SIN, COS	REQUIRES $ADR, $MLR, $SBR, $DVR, $INTR, $POLSH	118
SNGL	REQUIRES $ERR	15
SQRT	REQUIRES $ADR, $DVR, $ERR, $POLSH	47
TANH	REQUIRES EXP, $ADR, $SBR, $DVR, $FCALL, $POLSH, $PSHR3	121
AINT, $INTR		39

END OF THE OTS DOCUMENT