fortran iv
software support manual
OS/8 FORTRAN IV
SOFTWARE SUPPORT MANUAL

For additional copies, order No. DEC-68-LFSSA-A-D
from Software Distribution Center, Digital Equipment
Corporation, Maynard, Mass.

digital equipment corporation • maynard, massachusetts
The following are trademarks of Digital Equipment Corporation,
Maynard, Massachusetts:

CDP
COMPUTER LAB
COMTEX
COMSYST
DPT
DEC
DEC COMM
DECTAPE
DIBOL

DIGITAL
DNC
EDGRIN
EDUSYSTEM
FLIP CHIP
FOCAL
GLC-8
IDAC
IDACS
INDAC

KA10
LAB-8
LAB-8/e
LAB-K
OMNIBUS
OS/8
PDF
PHA

PS/8
QUICKPOINT
RAD-8
RSTS
RSX
RTM
SABR
TYPESET 8
UNIBUS
CONTENTS

CHAPTER 1 THE F4 COMPILER 1-1
CHAPTER 2 THE RALF ASSEMBLER 2-1
CHAPTER 3 THE FORTRAN IV LOADER 3-1
CHAPTER 4 THE FORTRAN IV RUN-TIME SYSTEM 4-1
CHAPTER 5 LIBRA AND FORLIB 5-1
APPENDIX A RALF Assembler Permanent Symbol Table A-1
APPENDIX B Assembly Instructions B-1
CHAPTER 1
THE F4 COMPILER

The OS/8 F4 compiler runs in 8K on either a PDP-8 or a PDP-12. It operates in three passes to transform FORTRAN IV source programs into RALF assembly language. The function of each of the three passes is:

1. Analyze statements, check syntax and convert to a polish notation.
2. Convert output of PASS1 to RALF assembly language making extensive use of code skeleton tables.
3. Produce a listing of the FORTRAN source program and/or chain to the assembler.

The following is a more complete description of each of the three passes.

PASS1 OPERATION

After opening the source language input file(s) and an intermediate output file, PASS1 processes statements in the following fashion:

1. Assemble a statement into the statement buffer by reading characters from the OS/8 input file. This section eliminates comments and handles continuations so that the statement buffer contains the entire statement as if it had been written on one long line.

2. The statement is first assumed to be an arithmetic assignment and an attempt is made to compile it as such. This is done with a special switch (NOCODE) set so that in the event the statement is not arithmetic, no erroneous output is produced. Thus, with this switch set, the expression analyzer subroutine is used merely as a syntax checker.

3. If the statement is indeed an arithmetic assignment statement (or arithmetic statement function) the switch is set off and the statement is then recompiled, this time producing output.
4. If not an arithmetic assignment, the statement might be one of the keyword defined statements. The compiler now checks the first symbol on the line to see if it is a legal keyword (REAL, GOTO, etc.) and jumps to the appropriate subroutine if so. Any statement that is not now classified is considered to be in error.

5. The compilation of each statement takes place. Some statements produce only symbol table entries (e.g., DIMENSION) which will be processed by PASS2. Others use the arithmetic expression analyzer (EXPR) and also output special purpose operators which will tell PASS2 what to do with the value represented by the arithmetic expression (e.g., IF, DO).

6. After the statement has been processed, control passes to the end-of-statement routine which handles DO-loop terminations and then outputs the end-of-statement code.

7. Statements containing some kind of error cause a special error code to be output.

8. The entire process is now repeated for the next statement.

9. When the END statement is encountered, PASS1 chains to PASS2.

PASS1 SYMBOL TABLE
A significant portion of the PASS1 processing involves the production of symbol table entries. These entries contain all storage related information, i.e., variable name, type, dimensions, etc.

The symbol table is organized as a set of linked lists. The first 26 such lists are for variables, with the first letter of the variable name corresponding to the ordinal number of the list. There are also separate lists for statement numbers and literals (integer, real, complex, double, and Hollerith). In addition to list elements, there are special entries for holding DIMENSION and EQUIVALENCE information.
A detailed description of each type of entry follows. (NOTE: All symbol table entries are in Field 1.)

1. VARIABLE - The first word of each entry is a pointer to the next entry, with a zero pointer signaling end of list. The second word contains type information. The third word points to the dimension and/or equivalence information blocks. The next one to three words contain the remainder of the name (the first character is implied by which list the entry is in) in stripped six-bit ASCII terminated by a zero character. Thus, shorter variables take less symbol table space. The entries are (as for all lists in the symbol table) arranged in order of increasing magnitude, or alphabetically.

```
POINTER
TYPE
DIMENSION/EQUIVALENCE
NAME 2-3
   N A
NAME 4-5
   M E
NAME 6
   X Ø
```

**TYPE WORD FORMAT**

```
0 1 2 3 4 5 6 7 8 9 10 11
| CM | DM | ET | SF | EUV | PLC | LIT | RG | T | Y | P | E |
```

**BIT**

Ø - Variable is in common.
1 - Variable is dimensioned.
2 - External symbol or subroutine/function name.
3 - Symbol is the name of an arithmetic statement function.
4 - Variable is an equivalence slave.
5 - Variable is explicitly typed.
6 - Entry is a literal.
7 - Variable is a formal parameter.
2. STATEMENT NUMBER - The first two words are the standard pointer/type. The next three words are the statement number, with leading zeros deleted, in stripped six-bit ASCII, filled to the right with blanks.

3. INTEGER OR REAL LITERALS - The first two words are the pointer and type. The next three words are the value in standard floating-point format (12-bit exponent, 24-bit signed 2's complement mantissa). Since the type of the literal must be preserved, there are two lists; hence use of 1 and 1.0 in the same program will cause one entry in each of the integer and real literal lists.

4. COMPLEX LITERALS - The first two words are standard. The next three are the real part in standard floating-point format. The next three are the imaginary part.
5. DOUBLE PRECISION LITERALS - The first two words are standard. The next six are the literal in FPP extended format (72-bit exponent, 60-bit mantissa).

6. HOLLERITH (quoted) LITERALS - The first two words are standard. The next \( N \) words are the characters of the literal in stripped six-bit ASCII, ending in a zero character.

7. DIMENSION INFORMATION BLOCK - If a variable is DIMENSIONed, the third word of its symbol table entry will point to its dimension information block (may be indirectly, see section 8 below). The first word of this block is the number of dimensions. The second word is the total size of the array in elements; thus the size in PDP-8 words may be 3 or 6 times
this number. The third word contains the "magic number" which is computed as follows:

\[ \text{MN} = 1 + \sum_{i=1}^{n-1} d_i \]

where \( d_j \) is the \( j \)th dimension and \( n \) is the number of dimensions.

For a 3-dimensional variable this number becomes:

\[ \text{MN} + 1 + d_1 + d_1 d_2 \]

The magic number must be subtracted from any computed index, since indexing starts at one and not zero. The fourth word will (in PASS2) contain the displacement from $\#LIT$ of a literal which will contain either the magic number in un-normalized form (for dimensioned variables which are subroutine arguments) or the address of the variable minus the magic number (for local or COMMON dimensioned variables). This literal is necessary for calling subroutines where a subscripted variable is an argument. The next \( N \) words are the dimensions of the variable. If the variable is a formal parameter of the subroutine, it may have one or more dimensions which are also formal parameters. In this case, the magic number is zero, and the dimension(s) is a pointer to the symbol table entry for the variable(s) used as a dimension.

<table>
<thead>
<tr>
<th>NUMBER OF DIMENSIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL NUMBER OF ELEMENTS</td>
</tr>
<tr>
<td>MAGIC NUMBER</td>
</tr>
<tr>
<td>RESERVED</td>
</tr>
<tr>
<td>DIMENSION 1</td>
</tr>
<tr>
<td>DIMENSION 2</td>
</tr>
<tr>
<td>............</td>
</tr>
<tr>
<td>DIMENSION ( n )</td>
</tr>
</tbody>
</table>

1-6
8. EQUIVALENCE INFORMATION BLOCK - If a variable is an EQUIVALENCE slave variable, the third word of its symbol table entry points to the equivalence information block. The first word of this block points to the dimension information (if any) of the variable. The second word points to the symbol table entry of the EQUIVALENCE master variable. The third word is the linearized subscript of the master variable from the EQUIVALENCE statement. The fourth word is the linearized subscript of the slave variable.

<table>
<thead>
<tr>
<th>POINTER TO DIMENSIONS</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTER TO MASTER</td>
<td>→</td>
</tr>
<tr>
<td>MASTER SUBSCRIPT</td>
<td>SSM</td>
</tr>
<tr>
<td>SLAVE SUBSCRIPT</td>
<td>SSM</td>
</tr>
</tbody>
</table>

9. COMMON INFORMATION BLOCK - If a symbol is defined as the name of a COMMON section, the third word of its symbol table entry points to a list of common information blocks. The first word of each such block points to the next block. The second word is the number of entries in the list that follows. The rest of the block is a set of pointers to the symbol table entries of the variables in the COMMON section.

<table>
<thead>
<tr>
<th>POINTER TO NEXT CIB</th>
<th>→</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER OF ENTRIES</td>
<td>#</td>
</tr>
<tr>
<td>POINTER TO VARIABLES IN THIS COMMON</td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
<tr>
<td></td>
<td>→</td>
</tr>
</tbody>
</table>

PASS1 OUTPUT
The output of PASS1 is a stream of polish with many special operators. Whenever an operand is to be output, the address of its symbol table entry is used. The following is a list of the output codes (in their mnemonic form, obtain numeric values from listing of PASS1) and the operation they are conveying to PASS2:
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUSH</td>
<td>The next word in the output file is an operand (symbol table pointer) to be put onto the stack.</td>
</tr>
<tr>
<td>ADD</td>
<td>Add the operands represented by the top two stack entries (actually this causes PASS2 to generate the RALF coding which will do the desired add).</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract top from next-to-top.</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiply top two.</td>
</tr>
<tr>
<td>DIV</td>
<td>Divide top into next-to-top.</td>
</tr>
<tr>
<td>EXP</td>
<td>Raise next-to-top to power of top.</td>
</tr>
<tr>
<td>NOT</td>
<td>Logical .NOT. of top of stack.</td>
</tr>
<tr>
<td>NEG</td>
<td>Negate top of stack.</td>
</tr>
<tr>
<td>GE</td>
<td>Compare top two for greater than or equal to, this has TRUE value if the next-to-top is .GE. the top.</td>
</tr>
<tr>
<td>GT</td>
<td>Compare for greater than.</td>
</tr>
<tr>
<td>LE</td>
<td>Compare for less than or equal.</td>
</tr>
<tr>
<td>LT</td>
<td>Compare for less than.</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND of top two entries.</td>
</tr>
<tr>
<td>OR</td>
<td>Logical inclusive OR of top two.</td>
</tr>
<tr>
<td>EQ</td>
<td>Compare top two for equality.</td>
</tr>
<tr>
<td>NE</td>
<td>Compare top two for inequality.</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR of top two.</td>
</tr>
<tr>
<td>EQV</td>
<td>EQUIVALENCE of top two.</td>
</tr>
<tr>
<td>PAUSOP</td>
<td>Use top of stack as PAUSE number.</td>
</tr>
<tr>
<td>DPUSH</td>
<td>The next two words are a symbol table pointer and a displacement; put them onto the stack (used for DATA statements).</td>
</tr>
<tr>
<td>BINRD1</td>
<td>Take the top of stack as the unit number and compile an unformatted READ-open.</td>
</tr>
<tr>
<td>FMTRD1</td>
<td>The top two stack elements are the unit and format, take them and compile a formatted READ-open.</td>
</tr>
</tbody>
</table>
RCLOSE
Compile a READ-close.

DARD1
Take the top two stack elements as a unit number and a block number and compile a direct access unformatted READ-open.

BINWR1

FMTWR1

WCLOSE
Same as for the corresponding READ case, except substitute the word "WRITE".

DAWR1

DEFFIL
Take the top four stack entries as the unit, number of records, record size, and index variable and compile a DEFINE FILE call.

ASFDEF
Set the PASS2 switch which says that the following statement is an arithmetic statement function.

ARGSOP
The next word is a count, call it n; take the previous n stack entries as subscripts (or arguments) and the N+1st entry from the top as the array (or function) name; now compile this as an array reference (or function/subroutine call).

EOLCOD
The current statement is completed, reset stacks and do other housekeeping.

ERRCOD
The following word contains an error code, write it on the TTY together with the current line number, and put the error code and line number into the error list for possible PASS3.

RETOPR
Compile a subroutine RETURN.

REWOPR
Take the top of stack as a unit and compile a rewind.

STOROP
Compile a store of the top of stack into the next-to-top.

ENDOPR
Compile a RETURN if a function or subroutine or a CALL EXIT if a main program.

DEFLBL
The following word is a symbol table pointer to a statement number, compile this as the tag for the current RALF line.

DOFINI
The following word is a symbol table pointer for the DO-loop index, compile the corresponding DO-ending code.

ARTHIF
The following one, two, or three words are symbol table pointers to statement numbers for the less than zero, zero, and greater than zero conditions with the comparison to be made on the top of stack.

LIFBGN
The top of stack is taken as a logical expression PASS 2 should compile a jump-around-on-false; this implies that some statement is to follow.

1-9
DOBEGIN
The top two stack entries represent the final
value and increment of the DO-loop, process them
in hopes of finding a matching DFINI.

ENDPOP
The top of stack is a unit, compile an END FILE.

STOPOP
Compile a CALL EXIT.

ASNOPR
The next word is the address of the symbol table
entry for a statement number; compile an ASSIGN
of this statement number to the variable represented
by the top of stack.

BAKOPR
Take the top of stack as the unit and compile
a BACKSPACE.

FMTOPR
The following word is a count N; the next N words
after that are the image of the FORMAT statement.

GO2OPR
The following word is the symbol table entry for
the statement number which is to be executed next.

CGO2OP
The following word is a count N; the next N words
are symbol table pointers for the statement
numbers of a computed GO TO list; use the value
represented by the top of stack to compile a
computed GO TO into this list.

AGO2OP
Compile an assigned GO TO with the top of stack.

IOLMNT
Take the top of stack as a list element for an
I/O statement and compile read or write; PASS2
knows if it is a READ or WRITE by remembering
previous FMTRD1, FMTWR1, etc.

DATELM
The next word is a count N; the next N words are
a data element.

DREPTC
The next word is a repetition count for the set
of DATELMs up until the next ENDELM.

ENDELM
Signals the end of a data element group.

PRGSTK
Tells PASS2 to purge the top stack entry.

DOSTOR
Performs the same function as STOROP after
checking the top two stack elements for legal
DO-parameter type (integer or real).

PASS 1 SUBROUTINES
The following is a brief description of the function of each of the
major PASS1 subroutines:

RDWR
Compiles everything in a READ or WRITE statement
starting at the first left parenthesis.
RESTCP  Restore character pointer and count for the statement buffer from the stack.

OUTWRD  Output a word (the AC on entering) to the PASS1 output file.

COMARP  Test for comma or right parenthesis; skip one instruction if a comma, two if a right parenthesis, and none if neither.

BACK1  Backup the statement buffer character pointer.

GETSS  Scans a variable reference, or subscripted variable reference with numeric subscripts and returns the linearized subscript.

MUL12  Perform a 12-bit unsigned integer multiply.

DOSTUF  Handles compilation of DO-loop setup.

TYPLST  Process a type declaration, DIMENSION, or COMMON statement; sets up type bits and/or dimension information.

LOOKUP  Perform a symbol table search for variables and Hollerith literals.

LUKUP2  Perform a symbol table search for integer, real, complex, and double precision literals or statement numbers.

EXPR  Analyze and process an arithmetic expression.

LETTER  Get next character from the statement buffer and skip if it is a letter, otherwise put the character back and don't skip.

CHECKC  The first word after the JMS is the negative of the ASCII character to test for; if this is the next character, skip.

GETCWB  Get the next character from the statement buffer preserving blanks.

SAVECP  Save the character pointer and count on the stack.

GETC  Get the next character ignoring blanks.

ERMSG  Output an error code to PASS1 output file.

POP  Pop the stack into the AC.

PUSH  Push the AC onto the stack.

LEXPR  Analyze and process an arithmetic expression, legal to the left of the equal sign in an assignment statement.

GET2C  Get the next two character into one word.

1-11
STMNUM  
Scan off a statement number and do the symbol table search.

DIGIT  
Same as letter, except checks for a digit.

NUMBER  
Scans off an integer, real, or double precision literal.

GETNAM  
Scan off a variable name.

ICHAR  
Get the next character from the input file.

PASS2 OPERATION
The first part of PASS2 generates the storage for variables, arguments, arrays, literals and temporaries by processing the symbol table built by PASS1, which is kept in core. The next step is to generate the code for subroutine entry and exit including argument pickup and restore. After all such prolog code is generated, PASS20 is loaded into core, overlaying most of the prolog-generating functions. The main loop of the compiler is now entered. This consists simply of reading a PASS1 output code from the intermediate file and using this number as an index into a jump table. The sections of code entered in this way then perform the correct generation of RALF code.

Example:
The statement:  
A=B+C*D  
would produce the following PASS1 output:
(assuming A,B,C,D are REAL)

1)  PUSH
   →A  (symbol table address of A)

2)  PUSH
   →B

3)  PUSH
   →C

4)  PUSH
   →D

5)  MUL

6)  ADD

7)  STOROP

8)  EOLCOD
The corresponding operations performed by PASS2 are:

1) Make a 3-word entry on the stack corresponding to the variable A consisting of a pointer to the symbol table entry, a word containing the type, and one reserved word.

2) Repeat above for B.

3) Repeat above for C.

4) Repeat above for D.

5) The multiply operator is handled like any of the binary operators by the subroutine CODE. This routine is called with the address of the multiply skeleton table. The top two stack entries are taken as the operands, with their types used to index into the skeleton tables. (See description of binary operator skeleton tables below.) The correct skeleton for this combination is chosen based on the whereabouts of each of the operands (AC or memory) at the corresponding point in the code which is being compiled. There are three possible cases: Memory,AC; Memory,Memory; AC,Memory. In this example, both operands are in memory so the code generated would be:

   FLDA C
   FMUL D

The CODE subroutine then makes a new stack entry to replace the entries for C and D. This entry has a $ in place of the symbol table pointer, signifying that the operand is in the AC. Other special case operand codes are:

   $ - AC (Already mentioned)
   1 - 51 Temporaries
   52 - $ Array reference, the subscript of which is in an index register (1-7).
   61 - A variable, the address of which is in base location $.
   62 - A variable, the address of which is in base location 3.
   63-6777 - Symbol table entry (can be variable or literal).
   7000 - Special temporary

6) The add operator is handled in the same way as for multiply, except that in this case the add skeleton table is used. When the correct row is found, the memory,AC case is chosen since the result of C*D is now in the AC. This skeleton simply generates:

   FADD B

The new top of stack entry is a $, since the result is in the AC.

7) The store operation works in a similar manner using a special skeleton table to determine whether the value to be stored is
already in the AC and whether it must be converted from one
type to another. In this case, no conversion need be performed
and the code generated is:

FSTA A

8) The end of statement has been reached and any necessary
bookkeeping is performed.

PASS2 SYMBOL TABLE
PASS2 modifies the symbol table entries corresponding to variables
by replacing the first word of the entry with the first character of
the name, this character being derived from the list in which the name
is located.

PASS2 ERROR LIST
PASS2 creates a list (in field 1) of error codes and line numbers
corresponding to the errors printed on the Teletype during PASS2.
This list works downward starting just below the skeleton table area,
working towards the symbol table area. PASS3 uses this list to
write out extended error messages on the listing.

PASS2 SKELETON TABLES
All binary operators have associated with them a skeleton table
having 24 entries arranged in 8 rows and 3 columns. The rows
correspond to the following eight possibilities:

1) Both operands integer or real.
2) Both operands complex.
3) Both operands double precision.
4) First operand integer or real, second complex.
5) First operand integer or real, second double precision.
6) First operand complex, second integer or real.
7) First operand double precision, second integer or real.
8) Both operands logical.
The columns correspond to the following three possibilities:

1) First operand in memory, second in AC.
2) Both operands in memory.
3) First operand in the AC, second in memory.

Each entry of the skeleton tables is either zero (illegal operator-type combination) or points to a code skeleton (minus one). Code skeletons are composed of combinations of the following types of elements:

1) OPCODES - If an element has a non-negative value, it is taken as the address of a text string for the desired opcode. This works since all such text strings are stored below location 4000 (in field 9). In this case, the next word of the skeleton is taken as a designator for the address field, the possibilities are:

   a. A non-negative values means the address field is a literal text string, with the value being the address of the string. (Same restriction as for opcode text strings.)

   b. A zero indicates that this instruction should have no address field.

   c. A minus one indicates that the address field is the operand defined by the three variables ARG1, TYPE1, and BASE1.

   d. A minus two indicates that the address field is the operand defined by the three variables ARG2, TYPE2, and BASE2.

2) MODE CHANGE - An element value of minus one means generate a STARTF if currently in extended mode. A value of minus two means generate a STARTE if currently in single mode.

3) MACRO - Any other negative value is taken as the address (minus 3) of a sub-skeleton. This sub-skeleton may contain anything except another sub-skeleton reference. When the end of the sub-skeleton is encountered, the main skeleton is re-entered.

4) END-OF-SKELETON - A zero indicates the end of the skeleton.

PASS2 SUBROUTINES

The following is a list of the major PASS 2 subroutines together with a brief functional description.
ERMSG  Output a 2-character error code together with the line number on the Teletype; also put the code and line number into the error list for PASS3.

UCODE  Generate the code for unary operators, given the skeleton table address.

CODE   Generate code for binary operators, given the skeleton table address.

INWORD Read a word from the PASS1 output file.

FATAL  Output a fatal error message and exit to OS/8.

ONUMBER Output the AC as a 4-digit octal number.

SAVEAC Generate an FSTA #TMP+XXXX if necessary.

GENCOD Generate the code specified by the given code skeleton.

OPCOD  Output a TAB followed by the specified opcode field.

OPCODE  Same as OPCOD, except output a second TAB after the opcode field.

OADDR  Generate the address field specified by the argument.

GENSTF Generate STARTF if in E mode.

GENSTE Generate STARTE if in F mode.

OSNUM  Output a statement number preceded by a "#".

CRLF   Output a carriage return/line feed.

OTAB   Output a TAB.

OUTSYM Output a text string.

GARG   Pop the top entry of the stack into ARG1, TYPE1, and BASE1.

GARGS  Pop the top two stack entries into ARG1, TYPE1, BASE1 and ARG2, TYPE2, BASE2.

OUTNAM Output a variable name.

OLABEL Output a generated label.

GETSS  Find the address of the dimension information block given the symbol table address.

SKPIRL Skip if integer, real, or logical.

GENCAL Generate the code for a subroutine call from the information contained on the stack.

MUL12  Do a 12-bit unsigned multiply.
OINS       Output a literal opcode and address field.
OCHAR      Output a character
NUMBRO     Output a 5-digit octal number.

PASS3 OPERATION
PASS3 first initializes the listing header line with the version number, date, and page number. It then processes lines, much like PASS1, handling continuations and comments and outputs their image to the listing file together with the line number. A constant check is made on the error message list for line numbers that correspond to the current line number. When such a correspondence occurs, the error code is used to find the associated detailed error message, which is then printed out.
CHAPTER 2

THE RALF ASSEMBLER

RALF and FLAP are essentially the same program, with differences controlled by the conditional assembly parameter RALF, which must be non-zero to assemble RALF, or zero to assemble FLAP. The source may be assembled by either PAL8 or FLAP; although FLAP flags one error (a US on a FIELD statement), this may safely be ignored. The remainder of this chapter applies to RALF only. The following definitions are prerequisite to discussion of the operation of this assembler.

| **MODULE** | The relocatable binary output of an assembly. A module is physically an OS/8 file or sub-file in a library, and is made up of an external symbol dictionary and related text. Logically, it consists of one or more program sections and COMMON sections. |
| **LIBRARY** | An OS/8 file on a directory device containing a catalog and one or more modules as sub-files. Used solely by the loader, as a source of modules with which to satisfy unresolved symbols in a program being loaded. |
| **CATALOG** | A list of entry points defined in modules contained in a library, with an indication of the locations of the modules which define them. |
| **EXTERNAL SYMBOL DICTIONARY** | A list of the global symbols defined in and/or used by a module. Usually called ESD table. |
| **TEXT** | That part of the assembler's binary output which contains the binary data to be loaded into memory, along with sufficient information for the loader to associate the output with specific memory locations through references to the ESD table. |
| **SECTION** | A unit of binary data output by the assembler as part of a module to be loaded into a contiguous area of memory. COMMON sections are a special case in that they may be defined with the same name in each of many modules. In this case, all the definitions are combined to create a single section in memory whose size is that of the largest COMMON section with the given name. Program sections, the only other type of section, must have unique names. Sections are listed in the ESD table by name, type and size. |
| **ENTRY POINT** | An address within a section which is named and defined to be global, so that it may be used for the resolution of external references in other sections. Entry points are listed in the ESD table by name, type and address within the section in which they occur. |
EXTERNAL SYMBOL

A symbol which is specified at assembly time to be defined in another module as an entry point. External symbols are listed in the ESD table by name and type. A complete program must include entry point names equivalent to every external symbol defined in every module in the program. There need not, however, be an external symbol for every entry point, nor is there any limit on the number of modules which may contain external symbols referencing one entry point. From a functional viewpoint, entry points correspond to tags within a program and external symbols correspond to references to those tags. Every section is considered to have an entry point at location zero of the section. The name of this entry point is the section name.

When RALF is called from the monitor, execution begins at the tag BEGIN. Unless entry is via CHAIN, the OS/8 command decoder is called to obtain input and output file designations. If entry is by way of CHAIN, it is assumed that the command decoder area has already been set up by the caller. In either case, it is always assumed that the USR is already in core. A check is made to determine that the first output file is a directory device file and, if no first output file was specified, the default file SYS:FORTRN.RL is set up.

Default output file extensions are defined if none were specified to the command decoder, using .RL for the first output file and .LS for the second output file. The first output file is then opened, and the handler for the first input file is FETCHed. If /L or /G was specified, the loader is looked up on SYS so that chaining will be possible. The symbol table, which is loader above 12000 in order to preserve the USR, is now moved down to 10000. Finally, the system date word is converted to character form and stored in the title buffer. This completes the initialization procedure, and control is passed to NEWLIN to collect the first line in the buffer.

At NEXTST, tests are made to determine whether the line just assembled needs to be listed, and whether there are any remaining significant characters in the line which have not been assembled. If a semicolon
terminated the statement, the character pointers are bumped to skip over it, and control passes to ASMBL to process the next statement on the line. If the assembler is currently in a REPEAT line and the count is not exhausted, the current line is re-assembled. Otherwise, a new line is obtained in the line buffer by collecting input characters until a carriage return is found. If the line is longer than 128 characters, all characters after the 128th are ignored and the LT message is printed. The line length is calculated and saved.

At ASMBL, ASMOF is tested to determine whether the assembly is currently inside a conditional. If so, the line is scanned for angle brackets but not assembled. If not, and the first character is not a slash, leading blanks are thrown away and control passes to LUNAME. If there is a name, it is collected. If it is followed by a comma, the symbol is looked up in the user symbol table. If the symbol is undefined, it is defined as a label. If it was already defined, the current location counter is compared with it to check for a possible MD error. Control then returns to ASMBL.

If the symbol found by LUNAME was followed by an equal sign, it is looked up and defined according to the expression to the right of the equal sign. If it was followed by a space, either of the characters ' or #, or the character % and then a space, it is looked up in the op-code table. If it is found, control passes to the appropriate op-code handler. Otherwise, control is dispatched to GETEXP which restores the character pointers saved by LUNAME, processes the rest of the line as a single-word expression, and returns to NEXTST for the next statement.
Expressions are processed on a strict left-to-right basis by the routine EXPR. A symbol is looked up, and its value is stored in WORD1 and WORD2. It is then combined with the accumulated expressions in EXPVAL according to the operator in LASTOP. A new operator (if any) is then located, and the loop begins again. When no operator is found after some symbol, the expression is considered complete and control returns to the calling routine. Undefined symbols appearing in an expression cause output of a US message, and the value zero is used in their place. COMMON and section names in the symbol table have special values (namely their lengths), but they always refer to the starting location of the sections they define, and their values are taken to be zero of the section so named. If GETNAM is not able to find a symbol in the expression, three possibilities are checked before flagging the expression as invalid:

1. It may be a number, rather than a symbol.
2. It may be one of the characters period (representing the current value of the location counter) or double quote (representing the binary value of the next ASCII character).
3. The last operator may have been a plus sign in an indexed FPP instruction.

At the end of expression evaluation, the console keyboard flag is checked to ensure that the user has not typed CTRL/C to stop the assembly.

There are six expression operator routines, one each for the operations add, subtract, AND, OR, multiply and divide. Except for add and subtract, these routines must operate on absolute addresses because the loader does not have facilities for non-additive resolution of address constants.
The symbol table is the sole occupant of field 1, except for the OS/8 field 1 resident. The symbol table is loaded at location 12000 to prevent an unnecessary swap of the USR, but moved down, to start at location 10000, during initialization. Subsequent calls to the USR do require a swap. The symbol table is a set of linked lists, or, more properly, two sets; one for user-defined symbols and one for op-codes and pseudo-ops. Each set contains a list corresponding to every letter of the alphabet, and each list consists of the symbols which start with that same letter. Every time a symbol is encountered in the source, the list corresponding to its first letter is searched until a match is found, or until the end of the list or a symbol of higher alphabetical order is found. In the latter cases, the new symbol is inserted into the user symbol table by changing the list pointers so that the new symbol appears in the list in correct alphabetical order. The pre-defined symbol table is never changed, because the user is not permitted to define op-codes or pseudo-ops.

A RALF output file of relocatable binary data consists of two parts; the ESD table and the text. The ESD table contains all information required by LIBRA or the loader, and is generated between the first and second passes of assembly. It serves as a partial symbol table for the loader (the full symbol table is built up from the ESD tables of all the modules in a program) and provides the name, attributes, and value of every global symbol used by any module, as well as an ESD code by which the symbol may be referred to within the text. Every entry in the ESD table is six words long. The first three words are the symbol itself, packed in stripped ASCII, with two characters per word. The next word contains type information in the following format:
A VALUE OF  

0  
Last entry in the ESD table.

1  
The symbol is defined as external to this module. 
The value of the symbol must be resolved by a 
symbol of the same name appearing in the ESD 
table of another module. The ESD code which 
follows the type code is the code by which 
references to this symbol will be identified 
in the text.

2  
The symbol is defined as an entry point in this 
module. It is therefore suitable for the 
resolution of external references in other 
modules. The ESD code which follows the type 
word identifies the program section in which 
this entry point appears, and the value of 
the symbol is relative to that section.

3  
The symbol is defined as a COMMON section whose 
size is at least as large as specified by the 
value of the symbol. If several modules contain 
ESD entries referring to COMMON sections with 
the same name, a single COMMON block having the 
size of the largest symbol is allocated for all 
of them. A name consisting of blanks is treated 
in the same manner as any other name.

4  
The symbol is defined as a section of location 
independent (that is, fully word-relocatable) 
code of a size equal to the value of the symbol. 
The ESD code for this section allows text from 
the module to be included in this section, and 
relocated with respect to it.

5-17  
Undefined

The text portion of a relocatable binary file consists of the binary 
data to be loaded into memory, along with information directing the 
loader on how to modify that data to correct the addresses for program 
relocation. The first word of text is a control word, which is made 
up of a 4-bit type code and an 8-bit indicator. Following the control 
word, and depending on the type code, are a number of data words to 
be loaded as directed by the type code and the indicator. The control 
word type codes are:

<table>
<thead>
<tr>
<th>CODE</th>
<th>FUNCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End of text, if the indicator is zero, or no operation otherwise.</td>
</tr>
</tbody>
</table>
1 Copy the number of words given by the indicator from text directly into memory without modification.

2 Re-origin to the section identified by the indicator, with a relative location defined by bits 9-23 of the following doubleword. Thus, the next two words define a new origin for the following text, in the program section identified by the indicator.

3 Relocate the following doubleword bits 9-23 by the value of the symbol whose ESD code is identified by the indicator. The following doubleword is usually a two-word PP instruction, the low-order 15 bits of which are to be relocated by the value of the symbol identified by the indicator.

WRITING PDP-8 CODE UNDER OS/8 FORTRAN IV

RALF contains the normal set of PDP-8 instructions (TAD, DCA, CDF, KSF, etc.), however RALF does not allow literals, the PAGE pseudo-op, or the use of I to specify indirect addressing. PDP-8 code generated by RALF is not relocatable; therefore, operations such as the following are illegal:

```
EXTERN SWAP /Illegal
TAD (SWAP /Under
CDF SWAP /RALF
```

The character % appended to the end of a memory reference instruction indicates indirect addressing, and the character Z indicates a page 0 reference:

<table>
<thead>
<tr>
<th>CURRENT PAGE</th>
<th>PAGE ZERO</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECT</td>
<td>INDIRECT</td>
</tr>
<tr>
<td>TAD A</td>
<td>TAD% A</td>
</tr>
<tr>
<td>DCA B</td>
<td>DCA% B</td>
</tr>
</tbody>
</table>

Spaces are not allowed between memory reference instructions and either the Z or the % characters. The Z must precede the % when both are used. I.e., do not write "DCA%Z".

Three pseudo-ops have been added to RALF: SECT8, COMMZ, and FIELD1. All three define sections of code and are handled in the same manner.
as SECT; however, these new sections have special meaning for the loader. The address pseudo-op (ADDR) which generates a two word re-locatable 15 bit address (i.e., JA TAG without use of JA) might prove useful in 8-mode routines. The following example demonstrates a way in which an 8-mode routine in one RALF module calls an 8-mode routine in another module:

```
EXTERN SUB
.
  RIF     /Set DF to current
  TAD ACDF /IF for return
  DCA .+1  /CDF X
  0       /Make a CIF from
  TAD KSUB /Field bits
  RTL CLL  /Field bits
  RAL
  TAD ACIF
  DCA .+1  /CIF to field
  /Containing SUB
  JMS% KSUB+1

KSUB, ADDR SUB /Psuedo-op to
     /Generate 15 bit
     /ADDR of subroutine
     /SUB
ACDF, CDF
ACIF, CIF
```

In general the address pseudo-op can be used to supply an 8-mode section with an argument or pointer external to the section.

FPP and 8-mode code may be intermixed in any RALF section. PDP-8 mode routines must be called in FPP mode by either:

```
TRAP3 SUB
```

or

```
TRAP4 SUB
```

A TRAP3 SUB causes FRTS to generate a JMP SUB with interrupts on and the FPP hardware (if any) halted. TRAP4 generates a JMS SUB under the same conditions. The return from TRAP4 is:

```
CDF CIF 0
JMP% SUB
```

The return from TRAP3 is:

```
CDF CIF 0
JMP% RETURN+1
```
Communication between FPP and 8-mode routines is best done at the FPP level because of greater flexibility in both addressing and relocation in FPP mode. The following routine demonstrates how to pass an argument to, and retrieve an argument from, an 8-mode routine:

```
EXTERN SUB
EXTERN SUBIN
EXTERN SUBOUT
.
.
FLDA X     /Arg for SUB
FSTA SUBIN /Call SUB
FLDA SUBOUT /Get result
FSTA Y
```

If the 8-mode routine SUB were in the same module as the FPP routine, the externs would not be necessary. In practice it is common for FPP and 8-mode routines that communicate with one another to be in the same section. A number of techniques can be used to pass arguments. For example, an FPP routine could move the index registers to an 8-mode section and pass single precision arguments via ATX.

Because 8-mode routines are commonly used in conjunction with FPP code (generated by the compiler), the 8-mode programmer should be familiar with OS/8 FORTRAN IV subroutine calling conventions. The general code for a subroutine call is a JSR, followed by a JA around a list of arguments, followed by a list of pointers to the arguments. The FPP code for the statement:

```
CALL SUB (X,Y,Z)
```

would be

```
EXTERN SUB
JSR SUB
JA BYARG
JA X
```
JA Y
JA Z

BYARG, .
.
.
The general format of every subroutine obeys the following scheme:

<table>
<thead>
<tr>
<th>SECT</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>#ST</td>
</tr>
</tbody>
</table>

/Jump to start of Routine

<table>
<thead>
<tr>
<th>TEXT +SUB+</th>
</tr>
</thead>
</table>

/Needed for Trace back

<table>
<thead>
<tr>
<th>RTN, SETX XSUB</th>
</tr>
</thead>
</table>

/Reset SUB's index

<table>
<thead>
<tr>
<th>SETB BSUB</th>
</tr>
</thead>
</table>

/And base page

<table>
<thead>
<tr>
<th>BSUB, FNOP</th>
</tr>
</thead>
</table>

/Start of base page

<table>
<thead>
<tr>
<th>JA</th>
</tr>
</thead>
</table>

. .

<table>
<thead>
<tr>
<th>ORG BSUB+30</th>
</tr>
</thead>
</table>

/Restart for SUB

<table>
<thead>
<tr>
<th>FNOP:JA RTN</th>
</tr>
</thead>
</table>

/Calling program

GOBAK, FNOP:JA . /Return to

Location 00000 of the calling routine's base page points to the list of arguments, if any, and may be used by the called subroutine provided that it is not modified. Location 0003 of the calling routine's base page is free for use by the called subroutine.

Location 0030 of the calling routine's base page contains the address where execution is to continue upon exit from the subroutine, so that a subroutine should not return from a JSR call via location 0 of the calling routine:

<table>
<thead>
<tr>
<th>CORRECT</th>
<th>INCORRECT</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLDA 30</td>
<td>FLDA 0</td>
</tr>
<tr>
<td>JAC</td>
<td>JAC</td>
</tr>
</tbody>
</table>

The "non-standard" return allows the calling routine to reset its own index registers and base page before continuing in-line execution.

General initialization code for a subroutine would be:

<table>
<thead>
<tr>
<th>SECT</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>JA</td>
<td>#ST</td>
</tr>
</tbody>
</table>

. .

<table>
<thead>
<tr>
<th>BASE</th>
</tr>
</thead>
</table>

0
The above code can be optimized for routines that do not require full
generality. The JA #ST around the base page code is a convenience
which may be omitted. The three words of text are necessary only for
error traceback and may also be omitted. If the subroutine is not
going to call any general subroutines, the SETX and SETB instructions
at location RTN and the JA RTN at location 0030 are not necessary. If
the subroutine does not require a base page, the SETB instruction is
not necessary in subroutine initialization; similar remarks apply to
index registers. If neither base page nor index registers are modified
by the subroutine, the return sequence:

FLDA 0
JAC

is also legal. In a subroutine call, the JA around the list of argu-
ments is unnecessary when there are no arguments. A RALF listing of
a FORTRAN source will provide a good reference of general FPP coding
conventions.

In order to generate good 8-mode code, one must be aware of the manner
in which the loader links and relocates RALF code. The loader handles
three 8-mode section types: COMMZ, FIELDL, and SECTB. All three
types of section are forced to begin and end on page boundaries and to
be a part of level MAIN; 8-mode sections never reside in overlays.
COMMZ and FIELDL sections are forced to reside in field 1; SECT
sections may be in any field. The first COMMZ section encountered is forced to begin at location 10000, thus enabling a page 0 in field 1. COMMZ sections of the same name are handled like COMMON sections of the same name (i.e., they are combined into one common section). This feature allows 8-mode code in different modules to share page 0, provided that the modules do not destroy each other's page 0 allocations. Suppose two modules were to share page 0, with the first using location 0-17 and the second using locations 20-37:

```
/Module A
COMMZ SHARE
P1, 1
P2, 2
KSUBA1, SUBA1
KSUBA2, SUBA2
.
.
LASTA, -1  /Should not go over
/20 locations
FIELD1 A

TADZ P1
JMSZ% KSUBA1
.
.
/Module B
COMMZ SHARE
ORG .+20  /ORG past module A's

P3, 3
P4, 4
KSUBB, SUBB
.
.
.
LASTB, -2
FIELD1 B
TADZ P3
.
.
```

The two COMMZ sections will be put on top of one another, however, because of the ORG .+20 in module B, they will effectively reside back to back. When the image is loaded, the COMMZ sections will look as follows:
<table>
<thead>
<tr>
<th>LOC</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>SUBA1</td>
</tr>
<tr>
<td>3</td>
<td>SUBA2</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0017</td>
<td>-1</td>
</tr>
<tr>
<td>1 0020</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>SUBB</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>-2</td>
</tr>
</tbody>
</table>

If module A is to reference module B's page 0, the procedure is:

```
P3=20
TADZ P3
```

Alternately, a duplicate of the source code for COMMZ SHARE may be included in module B. Modules that are using the same COMMZ section must be aware of how it is divided up. Although COMMZ SHARE takes only 40 locations, the loader allocates a full 200 locations to it. All 8-mode section core allocations are always rounded up so that they terminate on a page boundary. If COMMZ sections of different names exist, they are accepted by the loader and inserted into field 1, but only one COMMZ is the real page 0. In general, it is unwise to have more than 1 COMMZ section name.

FIELD1 sections are identical to COMMZ sections in most respects. Memory allocation for FIELD1 sections is assigned after COMMZ sections, however, and FIELD1 sections are combined with FORTRAN COMMON sections of the same name as well as other FIELD1 sections of the same name. The first difference ensures that COMMZ will be allocated page 0 storage even in the presence of FIELD1 sections. The second allows PDP-8 code to be loaded into COMMON, making it possible to load initialization code into data buffers. Two FIELD1 sections with the same name may be combined in the same manner as two COMMZ, sections.
The primary purpose of COMMZ is to provide a PDP-8 page 0; the primary purpose of FIELD1 is to ensure that 8-mode code will be loaded into field 1 and that generating CIF CDF instructions in-line is not necessary. SECT8 sections may not be combined in the manner of a COMMON and are not ensured of being placed into field 1.

An 8-mode section does not have to be less than a page in length; however, the programmer should be aware that a SECT8 section which exceeds one page may be loaded across a field boundary and could thereby produce disastrous results at execution time. For this reason, it is generally unwise to cross pages in SECT8 code. This situation will never occur on an 8K configuration. If the total amount of COMMZ and FIELD1 code exceeds 4K, the loader generates an OVER CORE message. The loader generates an MS error for any of the following:

1. A COMMZ section name is identical to some entry point or some non-COMMZ section name.
2. A FIELD1 section name is identical to some entry point or a SECT, SECT8 or COMMZ section name.
3. A SECT8 section name is identical to an entry point or some other section name.

COMMZ sections, like FORTRAN COMMONS, are never entered in the library catalog.

For users who intend to write 8-mode code that will execute in conjunction with certain 8-mode library routines, the layout of PDP-8 FIELD1 #PAGE 0 is:

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>USE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-1</td>
<td>Temps for any non-interrupt time routine.</td>
</tr>
<tr>
<td>2-13</td>
<td>User locations.</td>
</tr>
<tr>
<td>14-157</td>
<td>System locations.</td>
</tr>
<tr>
<td>160-177</td>
<td>User locations.</td>
</tr>
</tbody>
</table>

1. Do not define any COMMZ sections other than the system COMMZ which is #PAGE0.
2. If the system page 0 is desired, it will be pulled in from the library if EXTERN #DISP appears in the code.

3. Do not use any part of page 0 reserved for the system.

Special purpose PDP-8 mode subroutines may be written to perform idle jobs (refreshing a scope, checking sense lines) or to handle specific interrupts not serviced by FRTS.

The run-time system enters idle loops while waiting for the FPP to complete a task or for an I/O job to complete. It is possible to effect a JMS to a user routine during the idle loop.

RTS contains a set of instructions such as:

```
#IDLE, JMP .+4
0
CDF CIF
JMS I -.2
```

This sequence of instructions must be revised if an IDLE routine is to be called.

The location #IDLE must be changed to a SKP (7410). #IDLE+1 must be set to the address of the routine to be called. #IDLE+2 must be set to a CDF CIF to the field of the routine. This setup can be done in a routine that is called at the beginning of MAIN. For example:

```
CALL SETIDL
```

where SETIDL is a routine such as:

```
SECT8 SETIDL /*Must be an 8-mode section
JA #RET
TEXT +SETIDL+ /*Traceback information

SX R, SETX XR
SETB BP
BP, 0.0
XR, 0.0

ORG 10*3+BP
```

2-15
FNOP
JA SXR
  .
0
RET, JA .
  .
#RET,
STARTD
PLDA 10*3
PSTA RET
SETB BP
TRAP4 SET8
/For trace back
/Return address
/Just for traceback
/Go to the 8 mode
/Routine set 8
/Return to main
TAD IDLAD
CLL RTL
RAL
/TAD SCDF
DCA +3
/TAD IDLAD+1
DCA IDPTR
0
/TAD S7410
/SKP
DCA% IDPTR
/TAD JOB+1
ISZ IDPTR
/TAD JOB
DCA IDPTR
/TAD IDLAD
RAL
/TAD SFIELD
ISZ IDPTR
DCA% IDPTR
/CDF to #IDLE
/Address of #IDLE
/CDF goes here
/Store at #IDLE
/Address of IDLE top routine
/Store a #IDLE+1
/Field of routine
/Position
/Store at #IDLE+2
/Set to field 0
/Return to instruction
/Following "TRAP4 SET8"
EXTERN #IDLE

IDLAD, ADDR #IDLE
JOB, ADDR DOIT
/15 bit address of IDLE
/15 bit address of IDLE
/Routine "DOIT"

SCDF, 6201
SFIEL, 6203
IDPTR, 0
S7410, 7410
/CDF
/CDF CIF
/Skip
/The following routine performs the
/IDLE task
/Executed during IDLE loops
/Perform task
/Back to field 0
/And back

2-16
If the subroutine is checking for an illegal argument, an argument error message with traceback can be included in the subroutine by adding two lines somewhere on the base page:

```
EXTERN #ARGER
EXAMER, TRAP4 #ARGER
```

When the error is detected in the program, effect a jump to the TRAP4 instruction. For example,

```
FLD% EXTMP1
JEQ EXAMER /A value of 0 is illegal
```

or

```
FLDA EXTMP1
FNEG
FADD EXTMP2
JLT EXAMER /The value in EXTMP1 must be
greater than that in EXTMP2
```

Some points to note in the above example

1. Using a # as the first character in the name of the start of the program assumes that the name is not called from the FORTRAN level. This is because # is an illegal FORTRAN keyboard character.

2. If index registers 3-5 are not used by the subroutine, the space from XR3 to the ORG statement can be used for temporary storage, if needed.

3. The arguments passed from the FORTRAN level do not have to be picked up all at once at the start of the calculation (3-word) portion of the program. They can be picked up as required during the program, can be saved in temporary space, or accessed indirectly each time required, as best suits the subroutine.

If a call to this routine such as Z=EXAMPL(A,B,C,D) were encountered by the compiler, it would generate the following call to the routine:

```
JSR EXAMPL /go to the routine
JA .+10 /jump around arguments
JA A /pointer to 1st argument
JA B /pointer to 2nd argument
JA C /pointer to 3rd argument
JA D /pointer to 4th argument
```

The AMOD routine is listed below to illustrate an application of the formal calling sequence. It also includes an error condition check and picks up two arguments. When called from FORTRAN, the code is AMOD(X,Y).

2-17
AMOD
--

//SUBROUTINE AMOD(X,Y)
SECT AMOD
ENTRY MOD
JA #AMOD
TEXT +AMOD +
AMODXR, SIX XRAMOD
SETB BPAMOD
BPAMOD, F 0.0
XRAMOD, F 0.0
AMODX, F 0.0
ORG 10*3+BPAMOD
FNOP
JA AMODXR

AMDRIN, JA .
EXTERN #ARGER
AMODER, TRAPA #ARGER
FCLA
JA AMDRIN
BASE 0

/LONG ENOUGH TO GET RETURN ADDRESS
MOD, STARID
#AMOD, STARID
FLDA 10*3
FLSA AMDRIN
FLDA 0
SETX XRAMOD
SETB BPAMOD
BASE BPAMOD
LDX 1,1
FLSIA BPAMOD
FLDZA BPAMOD,1
FLSA AMODX
FLDZA BPAMOD,1
FLSIA BPAMOD
STARTF
FLDZA BPAMOD
JEQ AMODER
JGT .+3
FNEG
FLSIA BPAMOD
FLDZA AMODX
JGT .+9
FNEG
LDX 0,1
FLSIA AMODX
FDIV BPAMOD
JAL AMODER
ALN 0
FNORM
FMUL BPAMOD
FNEG
FADD AMODX
JXN AM,1
FNEG
AM. JA AMDRIN

2-18
RTS has its own interrupt skip chain in which all on-line device flags are checked and serviced. This chain may be extended to handle special interrupts. The external tag #INT marks the first of three locations on RTS which have to be modified to effect a JMS to the user's special interrupt handler. The three locations must be set up in exactly the same manner as that used to set up #IDLE, #IDLE1, #IDLE2 as described above. All the same conventions hold. Refer also to the library subroutines ONQI and ONQB.

Three pseudo-ops have been added to RALF to help the loader determine core allocation. Each is a more definitive case of the SECT pseudo-op and defines a chunk of code, thereby providing more control for the user. They are:

- **SECT8** - section starts at a page boundary
- **FIELD1** - section starts at a page boundary and is in field 1
- **COMMZ** - section starts at page 0 of field 1

If there is more than one SECT8 section in a module, those sections are not necessarily loaded in contiguous core. The loader considers core to be in two chunks - one block in field 0, and all of field 1 and above.

If there is more than one COMMZ pseudo-op in a module, they are stacked one behind the other, but there is no way of specifying which one starts at absolute location 0 of field 1. COMMZ sections are allocated by the loader before FIELD1 sections.

Modules can share a COMMZ section in the same way that FORTRAN COMMON sections can be shared. FIELD1 sections can also be shared by using the same FIELD1 section name in each module.

The first occurrence of a section name defines that section. For example,
SECT8 PARTA
  :
SECT8 PARTB
  :
SECT8 PARTA

The second mention of PARTA in the same module continues the source
where the first mention of PARTA ended at execution time. (There is
a location counter for each section.)

To save core, a RALF FIELD1 section and FORTRAN COMMON section of the
same name are mapped on top of each other, being allocated the length
of the longer and the same absolute address by the loader. This
feature is useful for initialization (once-only) code, which can later
be overlayed by a data area. Thus, the occurrence of FIELD1 AREA1 in
the RALF module and COMMON AREA1 in the FORTRAN program causes AREA1
to start the same location (in field 1) and have a length of at least
200 locations (depending on the length of the RALF FIELD1 section or
of the COMMON section in the FORTRAN).

If the subroutine is longer than one page and values are to be passed
across page boundaries, the address pseudo-op, ADDR, is required.
The format is:

AVAR1, ADDR VAR1

This generates a two-word reference to the proper location on another
page, here VAR1. For example, to pass a value to VAR1, possible code
is:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Oct</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00124</td>
<td>1244</td>
<td>TAD VAR2 /Value on this page</td>
</tr>
<tr>
<td>00125</td>
<td>3757</td>
<td>DCA% AVAR1+1 /Pass through 12-bit location</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00156</td>
<td>0000</td>
<td>AVAR1,ADDR VAR1 /Field and location of VAR1</td>
</tr>
<tr>
<td>00157</td>
<td>0322</td>
<td></td>
</tr>
</tbody>
</table>

Any reference to an absolute address can be effected by the ADDR
pseudo-op.
If it is doubtful that the effective address is in the current data field, it is necessary to create a CDF instruction to the proper field. In the above example, suitable code to add to specify the data field is:

```
TAD AVAR1  /Get field bits
RTL        /Rotate to bits 6-8
RAL
TAD (6201  /Add a CDF
DCA +1     /Deposit in line
0          /Execute CDFn
```

If the subroutine includes an off-page reference to another RALF module (e.g., in FORLIB), it can be addressed by using an EXTERN with an ADDR pseudo-op. For example, in the display program, a reference to the non-interrupt task subroutine ONQB is coded as

```
EXTERN ONQB
ONQBX, ADDR ONQB
```

and is called by

```
JMS% ONQBX+1
```

The next instruction in the program is ADDR DISPLY so that DISPLY will be added to the background list. Execution from ONQB returns after the ADDR pseudo-op.

It may be desirable to salvage the first (field) word allocated by ADDR pseudo-ops. If the address requires only twelve bits for proper execution, code such as

```
TMP, TEMP ADDR X
ARG, ADDR X or ARG= .-1
```

permits TMP to be used for temporary storage because ARG+1 in the left hand example or just ARG in the right hand example defines the 12-bit address.

RALF does not recognize LINC instruction or PDP-8 laboratory device instructions. Such instructions can be included in the subroutine by defining them by equate statements in the program.
For example, adding the statements:

\[
\begin{align*}
PDP &= 2 \\
LINC &= 6141 \\
DIS &= 140
\end{align*}
\]

takes care of all instructions for coding the PDP-12 display subroutine.

When writing a routine that is going to be longer than a page, it can
be useful to have a non-fixed origin in order not to waste core and to
facilitate modification of the code. A statement such as

\[\text{IFPOS .-SECNAM\&177-K<ORG .-SECNAM\&7600+200+SECNAM}>\]

will start a new page only if the value [current location less section
name] is greater than some \(K\) (start of section has a relative value of
0) where \(K<177\) and is the relative location on the current page before
which a new page should be started. The ORG statement includes an AND
mask of 7600 to preserve the current page. When added to 200 for the
next page and the section name, the new origin is set.

When calculating directly in a module, the following rules apply to
relative and absolute values.

\[
\begin{align*}
\text{relative} - \text{relative} &= \text{absolute} \\
\text{absolute} + \text{relative} &= \text{relative} \\
\text{OR} (\&), \text{AND} (\&\&) \text{ and ADD (+) of relative symbols} \\
\text{generate the RALF error message RE.}
\end{align*}
\]

When passing arguments (single precision) from FPP code to PDP code,
using the index registers is very efficient. For example,

\[
\begin{align*}
\ldots & \\
\text{PLDA} & \text{ARG1} & /\text{Get argument in FPP mode} \\
\text{SETX} & \text{MODE8} & /\text{Change index registers so XR0 is} \\
\text{ATX} & \text{MODE8} & /\text{At MODE8} \\
\ldots & \\
\text{TRAP4} & \text{SUB8} & /\text{Save argument} \\
\ldots & \\
\text{SUB8,} & 0 & /\text{Go to PDP-8 routine} \\
\ldots & \\
\text{TAD} & \text{MODE8} & /\text{PDP-8 routine} \\
\ldots & \\
\text{MODE8,} & 0 & /\text{Get argument} \\
\ldots & \\
\end{align*}
\]

\[\text{/Index registers set here}\]
CHAPTER 3

THE FORTRAN IV LOADER

The FORTRAN IV loader accepts a set of (up to 128) RALF modules as input, and links the modules, along with any necessary library components, to form a loader image file that may be read into memory and executed by the run-time system. The main task accomplished by the loader is program relocation, achieved by replacing the relative starting address of every section with an absolute core address. Absolute addresses are also assigned to all entry points, all relocatable binary text, and the externs.

The loader executes in three passes. Pass 0 begins by determining how much memory is available on the running hardware configuration, and then constructs tables from the OS/8 command decoder input for use by pass 1 and pass 2.

Pass 1 reads the relocatable binary input and creates the loader symbol table. The length of each input module is computed and stored, along with the relative values of entry points defined within the input modules. When an undefined symbol is encountered, pass 1 searches the catalog of the FORTRAN IV library specified to pass 0, or FORLIB.RL if no other library was explicitly specified, and loads the library routine corresponding to the undefined symbol.

Pass 1 also allocates absolute core addresses to all modules and, through them, to all symbols. Pass 1 execution concludes by computing the lengths of all overlay levels defined for the current FORTRAN IV job. Trap vectors are also set up at this time, and the tables required for pass 2 loading are initialized.

Pass 2 concludes loader execution by creating a loader image file from the relocated binary input and symbol values processed by pass 1.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>OS/8 Command Decoder</td>
</tr>
<tr>
<td>02000</td>
<td>Loader Pass 1 and Pass 2</td>
</tr>
<tr>
<td>04600</td>
<td>Core measuring routine and scratch area to save 00000-02000 during CD calls</td>
</tr>
<tr>
<td>06600</td>
<td>Unused</td>
</tr>
<tr>
<td>07600</td>
<td>OS/8 Field Ø resident</td>
</tr>
<tr>
<td>10000</td>
<td>OS/8 User Service Routine</td>
</tr>
<tr>
<td>12000</td>
<td>Symbol table, loader map titles</td>
</tr>
<tr>
<td>12400</td>
<td></td>
</tr>
<tr>
<td>13200</td>
<td>Pass Ø code</td>
</tr>
<tr>
<td>14000</td>
<td>Pass 1 initialization</td>
</tr>
<tr>
<td>16000</td>
<td>Module count and module tables</td>
</tr>
<tr>
<td>17000</td>
<td>Library catalog header read into this block</td>
</tr>
<tr>
<td>17600</td>
<td>OS/8 Field 1 resident</td>
</tr>
</tbody>
</table>

Pass 2 also produces the loader symbol map, if requested, and chains to the run-time system if /G was specified.

Pass 0 contains very few subroutines. The routine CORDSW checks for the presence of /U, /C or /O option specifications, as supplied to the command decoder, and processes these options if necessary. A routine called UPDMOD is called when input to each overlay has been concluded, to update the module counts in the module count table.
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>Pass 1 and Pass 2 utility routines</td>
</tr>
<tr>
<td>01400</td>
<td>Symbol map printer</td>
</tr>
<tr>
<td>02000</td>
<td>Pass 2</td>
</tr>
<tr>
<td>03200</td>
<td>Pass 1 symbol collection</td>
</tr>
<tr>
<td>04000</td>
<td>Inter-pass code allocates storage, builds and writes Loader Image Header Block.</td>
</tr>
<tr>
<td>04600</td>
<td>Library catalog loads here in 8K. Unused in 12K or more.</td>
</tr>
<tr>
<td>07200</td>
<td>Input device handlers</td>
</tr>
<tr>
<td>07600</td>
<td>OS/8 Field Ø resident</td>
</tr>
<tr>
<td>10000</td>
<td>ESD table</td>
</tr>
<tr>
<td>11400</td>
<td>Symbol table</td>
</tr>
<tr>
<td>12000</td>
<td>Overlay table</td>
</tr>
<tr>
<td>15400</td>
<td>Module count and module tables (MCITYL, MODTABL)</td>
</tr>
<tr>
<td>17200</td>
<td>Loader header</td>
</tr>
<tr>
<td>17400</td>
<td>ESD reference page</td>
</tr>
<tr>
<td>17600</td>
<td>OS/8 Field 1 resident</td>
</tr>
<tr>
<td>20000</td>
<td>Library catalog loads here in 12K or more.</td>
</tr>
<tr>
<td>25000</td>
<td>OS/8 BATCH processor if 12K or more and BATCH is running</td>
</tr>
</tbody>
</table>

CORMOV is a general core-moving subroutine, called by the instruction sequence:

```
JMS CORMOV
CDF FROMFIELD
FROMADDR - 1
CDF TOFIELD
TOADDR - 1
- COUNT
```

while ERROR is the local error processing routine, called with a pointer to the appropriate error message in the accumulator.

The major pass 1 and pass 2 subroutines, described below, operate on the loader internal tables, whose format is presented later in this
<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>Utility routines: Symbol table look-up, TTY message handler, OS/8 block I/O, MCTTBL processor.</td>
</tr>
<tr>
<td>01400</td>
<td>Routine to print symbol map.</td>
</tr>
<tr>
<td>02000</td>
<td>Pass 2</td>
</tr>
<tr>
<td>03200</td>
<td>Binary buffer #1</td>
</tr>
<tr>
<td>05200</td>
<td>Binary buffer #2</td>
</tr>
<tr>
<td>07200</td>
<td>I/O device handlers</td>
</tr>
<tr>
<td>07600</td>
<td>OS/8 Field 0 resident</td>
</tr>
<tr>
<td>10000</td>
<td>RALF module text loads here if 8K.</td>
</tr>
<tr>
<td>12000</td>
<td>Symbol table</td>
</tr>
<tr>
<td>15400</td>
<td>Overlay length table</td>
</tr>
<tr>
<td>16000</td>
<td>MCTTBL and MODTBL</td>
</tr>
<tr>
<td>17200</td>
<td>Binary section table and binary buffer (LDBUFS) table</td>
</tr>
<tr>
<td>17400</td>
<td>ESD reference page</td>
</tr>
<tr>
<td>17600</td>
<td>OS/8 Field 1 resident</td>
</tr>
<tr>
<td>20000</td>
<td>Binary buffer #3, if &gt;8K</td>
</tr>
<tr>
<td>22000</td>
<td>Binary buffer #4, if &gt;8K</td>
</tr>
<tr>
<td>24000</td>
<td>Binary buffer #5, if &gt;12K</td>
</tr>
<tr>
<td>26000</td>
<td>Unused</td>
</tr>
<tr>
<td>30000</td>
<td>RALF module text loads here if &gt;12K.</td>
</tr>
</tbody>
</table>

chapter. The subroutines are presented in approximately the order that they occur in the source listing.

SETBPT Sets words BPTR and BPT2 to contain AC and AC+1, respectively.

TTYHAN Subroutine to unpack and print a TEXT message on the console terminal. TTYHAN is called by:

CDF CURRENT
CIF 0
JMS TTYHAN
CDF MSGFIELD
MSG
RTNOS8
Prints a fatal error message and then returns to the OS/8 monitor. A pointer to the message must follow the JMS RTNOS8.

IOHAN
Used to execute all I/O under OS/8. The calling sequence is:

TAD (ACARG /Optional
CDF CURRENT
CIF 0
JMS IOHAN
ADDR
ARG1
ARG2
ARG3

where ARG1, ARG2 and ARG3 are standard OS/8 device handler arguments and ADDR points to a three-word block in field 1 which contains the OS/8 unit number in word 1, the file length in word 2, and the starting block number in word 3.

If ACARG is zero, the indicated I/O operation is executed after the handler has been FETCHed, if necessary. If ACARG=n (greater than zero), the handler for OS/8 unit n is FETCHed, no I/O is done, and the four arguments that conclude the calling sequence are not needed.

ADVOVR
Called to initialize the loader to accept a new input module. ADVOVR determines whether a new overlay or level is being started by accessing the module count table. If so, it sets various pointers and internal counters accordingly, rounds the previous overlay to terminate on a 200 word boundary, and updates the length of the previous level, if necessary, as the maximum of its constituent overlay lengths.

NXTOVR
Called by ADVOVR when the next input module will be the first module in a new overlay.

SETCNT
Initializes the pointers and counters used by ADVOVR. SETCNT is called once at the beginning of each pass.

LOOK
Executes a symbol look-up in the loader symbol table. LOOK is called by:

TAD (Pointer to symbol name in RALF ESD format
JMS LOOK
RETURN here if not found
RETURN here if found
GPTR points to word following entry name

If the symbol is not found, it is inserted into the loader symbol table and GPTR is set to point to the word following the symbol name.

SYMMAP
Produces the symbol map.
PUTSYM

Enters an ESD symbol in the loader symbol table. PUTSYM calls LOOK to determine whether the symbol is already present in the symbol table and, if so, verifies that the symbol is not multiply defined. Otherwise, it copies the ESD data words into the symbol table entry, updates the length of the current overlay by the length associated with the symbol, and links the symbol to its parent symbol, if any.

FIT

Fits a section into core by subtracting its length from the amount of core still available and substituting its load address for its length in the symbol table.

DO8S, FIT8S

Fits an 8-mode section into core by calling FIT and then checking for field overflow.

SETREF

Extracts data from the ESD table of the current module and initializes the ESD reference page at 17400.

BLDTV

Builds the transfer vector. A transfer vector entry is created for each subroutine in an overlay. This entry provides the information that the run-time system will require in order to load the overlay containing the referenced subroutine.

NEWORG

Called whenever an origin is found in an input module, to map the location referenced by the origin into a block of the loader image file and an address within that block.

NEWBB

Called whenever a new binary buffer is needed during loader image file construction. NEWBB scans a list of available buffers and dumps the content of the least recently accessed buffer to free up space for new data.

MERGE

Relocates an input word pair and outputs it to the loader image file.

GETCTL

Gets a control byte from the input module and increments its return address by the content of the control byte.

PUTBIN

Inserts words, sequentially, into the current binary buffer. When the buffer is full, PUTBIN calls NEWBB to execute output to the loader image file and supply a new buffer.

TXTSCN

Called once for each input module. TXTSCN reads and relocates an entire input module, executing calls to MERGE, PUTBIN and NEWORG as needed.
SYMBOL TABLE

The loader symbol table begins at location 12000 and contains room for 26 (decimal) permanent system symbol entries and 218 (decimal) user entries. Each entry is 7 words long, and provides the name and definition of a symbol. The table is organized in buckets according to the first character of the symbol, which must be A to Z, #, or blank (for blank COMMON). The table of bucket pointers begins at location 12000 with the pointer to bucket A, and consists of one word per bucket. This word contains a value of zero, if there are no symbols in the corresponding bucket, or else the address of the first symbol in the bucket.

Symbols within a bucket are arranged in alphabetical order, with each symbol entry pointing to the following entry, and the last entry pointing to zero. Thus, the symbol table appears as a set of threaded lists in core. The format of a symbol table entry is:

<table>
<thead>
<tr>
<th>WORD 1</th>
<th>WORD 2</th>
<th>WORD 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer to next symbol in bucket (zero if none).</td>
<td>S</td>
<td>Y</td>
</tr>
</tbody>
</table>

1-bit trap vector flag during pass 1. Error flag during pass 2.

4-bit type code
0- undefined
1- entry point
2- extern
3- common sect
4- program sect
5- multiple entry point
6- multiple sect
7- SECT8 sect
10-COMZ
11- FIELD1
12 to 17- undefined

9-bit pointer to parent symbol during pass 1 (zero if none).
Trap vector displacement during pass 2.

Field bits (Length during pass 1)
Several special symbols are created by the loader. The symbol \#YLVLn, where n is an octal digit, describes overlay level n. This symbol table entry contains the length of level n during pass 1 and the starting address of level n during pass 2.

The symbol \#YTRAP describes the trap vector, a method by which the run-time system controls automatic overlaying of user subroutines. Four words are allocated in the trap vector for each entry point in every overlay except overlay \#MAIN. The symbol table entry for \#YTRAP contains the accumulated length of the trap vector during pass 1 and the trap vector starting address during pass 2.

ESD CORRESPONDENCE TABLE (ESDPG)

The ESD correspondence table begins at location 17400 and contains 128 (decimal) 1-word entries. This table establishes the correspondence between the local ESD reference numbers used to reference a symbol inside a RALF module, and the address of that symbol in the loader symbol table. The n\text{th} entry in the ESD correspondence table points to the address of ESD symbol n.

BINARY BUFFER TABLE (LDBUFS)

The binary buffer table begins at location 17247 and contains from two to ten entries, depending upon the amount of memory available. Each entry is 4 words in length. The binary buffers function as windows into the loader image file, through which the loaded program is written onto mass storage. Each binary buffer is 8 pages (4 OS/8 blocks) in length. The loader tries to minimize the amount of "window turning" necessary to buffer the binary data by keeping a record of the last time each buffer was referenced. In this way,
when the content of a binary buffer must be dumped to make room for new data, the loader empties that buffer which was least recently used.

In addition, program loading is overlay oriented such that only one overlay is loaded at a time and while any specific overlay is being loaded, only origins inside that overlay are legal.

The format of a binary buffer table entry is:

| Pointer to the binary buffer of "next earliest reference", i.e., the youngest buffer older than this buffer. Contains zero if this buffer is oldest. | WORD 1 |
| Loader image block #. Contains zero if buffer has not been used. | WORD 2 |
| Blocks left in current overlay. If <4, only part of buffer will be dumped. | WORD 3 |
| Page address of buffer. | Buffer field | Unused |
| | | | WORD 4 |

The number of binary buffers used varies with the amount of memory available as follows:

<table>
<thead>
<tr>
<th>MEMORY AVAIL</th>
<th>NO. OF BUFFERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>8K</td>
<td>2</td>
</tr>
<tr>
<td>12K</td>
<td>4</td>
</tr>
<tr>
<td>16K</td>
<td>5</td>
</tr>
<tr>
<td>20K</td>
<td>7</td>
</tr>
<tr>
<td>24K</td>
<td>10 (decimal)</td>
</tr>
<tr>
<td>28K</td>
<td>10 (decimal)</td>
</tr>
<tr>
<td>32K</td>
<td>10 (decimal)</td>
</tr>
</tbody>
</table>
BINARY SECTION TABLE

The binary section table overlays the loader image header block (described under FRTS) after the latter has been written into the loader image file at the beginning of pass 2. Thus, the binary section table begins at location 17200 and contains eight 4-word entries. Each entry relates the core origin of one of the eight overlay levels to that level’s position in the loader image file. The format of a binary section table entry is:

<table>
<thead>
<tr>
<th>Unused</th>
<th>Field of level</th>
<th>WORD 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of level</td>
<td>WORD 2</td>
<td></td>
</tr>
<tr>
<td>Relative block #</td>
<td>WORD 3</td>
<td></td>
</tr>
<tr>
<td>Length (in blocks)</td>
<td>WORD 4</td>
<td></td>
</tr>
</tbody>
</table>

OVERLAY TABLE (OVLTBL)

The overlay table begins at location 15435 and contains room for 113 (decimal) 2-word entries. There is one entry for each overlay defined, including overlay MAIN, with each entry designating the length in words, of the corresponding overlay. The format of an overlay table entry is:

OVLTBL

| LEVEL MAIN |
| LEVEL 1 OVERLAY 1 |
| . . . |
| LEVEL m OVERLAY n-1 |
| LEVEL m OVERLAY n |
| OVLTBL format |

Negated to indicate last table entry

| HIGH-order bits of length | WORD 1 |
| LOW-order bits of length | WORD 2 |
| individual entry (2 words) |
MODULE DESCRIPTOR TABLE (MODTBL)

The module descriptor table begins at location 16172 and contains room for 172 (decimal) 3-word entries. Each entry provides the information needed to locate an input module. The first MODTBL entry corresponds to the library file to be used in building the current loader image. Successive entries correspond to input modules and appear in the order that the modules were specified by the user, (i.e., in ascending order by level, and ascending by overlay within any given level.) At the end of pass 1, entries corresponding to individual library modules are appended to the end of the table, even though the library modules load into level MAIN. The table format is:

MODTBL

<table>
<thead>
<tr>
<th>FORLIB.RL or user-specified library</th>
<th>Level MAIN module #1</th>
<th>OS/8 I/O unit #</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Level MAIN module #2</td>
<td>File length (positive)</td>
</tr>
<tr>
<td></td>
<td>Level MAIN module #3</td>
<td>Starting block #</td>
</tr>
<tr>
<td>Level MAIN module n</td>
<td>Level 1 Overlay 1 module #1</td>
<td>MODTBL format of</td>
</tr>
<tr>
<td></td>
<td>Level 1 Overlay 1 module #2</td>
<td>individual entry (3 words)</td>
</tr>
<tr>
<td></td>
<td>Level 1 Overlay 1 module #n</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Level 1 Overlay 2 module #1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>:</td>
</tr>
<tr>
<td></td>
<td>Level m Overlay n module #p</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Library module #1</td>
<td>MODTBL format</td>
</tr>
<tr>
<td></td>
<td>Library module #2</td>
<td></td>
</tr>
</tbody>
</table>
MODULE COUNT TABLE (MCTTBL)

The module count table begins at location 16000 and contains room for 122 (decimal) 1-word entries that give the (two's complement) module count for each overlay level. The table format is:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LEVEL</td>
<td>MAIN</td>
</tr>
<tr>
<td></td>
<td>Ø</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY 1</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY 2</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY 3</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY n</td>
</tr>
<tr>
<td></td>
<td>Ø</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY 1</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY 2</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY n</td>
</tr>
<tr>
<td></td>
<td>Ø</td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>LEVEL</td>
<td>OVERLAY n</td>
</tr>
<tr>
<td></td>
<td>Ø</td>
</tr>
</tbody>
</table>

If an overlay or level is not defined for a specific program, there is no module count table entry corresponding to that overlay or level.

The loader image file, produced by the loader and read as input by the run-time system, consists of a header block followed by a binary image of each level defined in the FORTRAN IV job.
The loader image file header block contains information in the following format:

<table>
<thead>
<tr>
<th>LOCATION</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2 -- Identifies the file as a loader image file.</td>
</tr>
<tr>
<td>1-2</td>
<td>Initial SWAP arguments to load level MAIN.</td>
</tr>
<tr>
<td>3-4</td>
<td>Highest address used by core load, including overlays but not including OS/8 device handlers.</td>
</tr>
<tr>
<td>5</td>
<td>Loader version number.</td>
</tr>
<tr>
<td>6</td>
<td>Double-precision flag.</td>
</tr>
<tr>
<td>7-46</td>
<td>User overlay information table containing one 4-word entry per overlay level (the level MAIN entry is ignored) in the following format:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unused until SWAP time. Must be positive or zero.</th>
<th>WORD 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load address -&gt;</td>
<td>Page bits</td>
</tr>
<tr>
<td>Block number of this level, relative to header block.</td>
<td>WORD 3</td>
</tr>
<tr>
<td>Length of overlays in this level, in blocks.</td>
<td>WORD 4</td>
</tr>
</tbody>
</table>
CHAPTER 4
THE FORTRAN IV RUN-TIME SYSTEM

The FORTRAN IV run-time system supervises execution of a FORTRAN job and provides an I/O interface between the running program and the OS/8 operating system. FRTS includes its own loader, which should not be confused with LOAD, the system loader. It executes with only one overlay, used to restore the resident monitor and effect program termination. The run-time system was designed to permit convenient modification or enhancement, and it is well documented in the assembly language source, available from the Software Distribution Center, which includes extensive comments.

One of the most valuable modifications to FRTS provides for the inclusion of background (or idle) jobs. When FORTRAN is waiting for I/O operations or the FPP to complete execution, the PDP-8 or PDP-12 processor is sitting in an idle loop. An idle job may be executed by the PDP-8 or PDP-12 CPU during this time, perhaps for the purpose of refreshing a CRT display, for example, or monitoring a controlled process. To indicate such a job, the idle wait loop must be modified to include a reference to the user's PDP-8 routing. The routine #IDLE in FRTS must be changed as part of the user's subroutine from

```
#IDLE, JMP .+4 to #IDLE, SKP
  0 ADDUSR
  CDF CIF FLDUSR
  JMS I -.2
```

Devices issuing interrupts may be added to the interrupt skip chain so that FORTRAN checks the user's device as well as system devices.

The original code is:

```
#INT, JMP .+4
  0 CDF CIF
  JMS I -.2
```
and must be changed, as above, to:

```
#INT, SKP
ADDUSR
FLDUSR
JMS I .-2
```

In both cases, ADDUSR should be the address of the user's routine, and FLDUSR should be the memory field of the user's routine.

The idle job is initiated by the subroutine HANG in the run-time system. Hang should only be called when the FORTRAN program must wait for an I/O device flag. The calling sequence is:

```
EXTERN #HANG

IOF          /Important.
CDF n        /Where n is current field.
CIF 0
JMS% HANG+1
ADDRSS      /Return here with interrupts OFF
             /When device flag is raised.
```

**HANG, ADDR #HANG**

The word ADDRSS must point to a location in page 400 of the run-time system which must normally contain a JMP DISMIS. Three such locations have been provided for the user at #DISMS, #DISMS+1, and #DISMS+2. The selected location must be the location via which the interrupt caused by the desired flag is dismissed. No two flag routines should use the same dismiss location. The following program example illustrates these calling conventions. This routine may be used to drive a Teletype terminal via the PT08 option.
EXTERN #ONQI
EXTERN #DISMS
FIELD1 GETCH /JMS GETCH GEIS A CHAR
Ø /GETCH RUNS IN FIELD 1 ONLY
ISZ FIRST
JMP NOTFST
JMSZ ONQI+1
KSF1
ADDR KSFSUB
TAD DISMIS+1 /SET UP TO CALL HANG
DCA HNGLOC
NOTFST, IOF
TAD INCHR
SZA CLA
JMP GOT1
CIF Ø
JMSZ HANG+1 /NO CHAR READY: HANG
HNGLOC, Ø /HANG RETURNS W/ IOF
GOT1, TAD INCHR
DCA FIRST
DCA INCHR
TAD FIRST
ION
JMP% GETCH /INTERRUPT ROUTINE
KSFSUB, Ø /CALLED AS SUBROUTINE
KRBI
DCA INCHR
CDF CIF Ø
JMP% DISMIS+1 /RETURN TO SYSTEM LOCATION
/CONTAINING "JMP DISMIS"
INCHR, Ø
ONQI, ADDR #ONQI
HANG, ADDR #HANG
DISMIS, ADDR #DISMS
FIRST, -1

In most cases, it is easier to include references to the FORLIB module ONQI for adding a handler to the interrupt skip chain and ONQB for adding a job to the idle chain, instead of trying to modify #IDLE and #INT. ONQB provides slots for up to 9 idle jobs to be executed round-robin, and ONQI provides for up to 9 user flags to be tested on program interrupts.

FRTS entry points are listed, along with the core map, on the following pages. The FRTS calling sequence must be observed in any user subroutine. The formal calling sequence is illustrated below. In general, it can be used exactly as illustrated, changing only the section, entry, base page, index register and return location names.
FRTS CALLING SEQUENCE

SECT EXAMPL
   /Section name. Your module may
   /require another section pseudo-op
   /such as FIELD1 or SECT8.
JA #EXSRT
   /Jump to start of subroutine
TEXT +EXAMPL+
   /Use # for first character
EXAMXR, SETX XREXAM
   /Set up index registers
   /for this subroutine
SETB BPEXAM
   /and its base page.
BPEXAM, F 0.0
   /Base page
XREXAM, F 0.0
   /Index registers 0-2
   /Index registers 3-5 (optional)
EXTMP1, F 0.0
   /Space between index registers
EXTMP2, F 0.0
   /and the ORG for temporary
EXTMP3, F 0.0
   /storage (optional)
ORG 10*3+BPEXAM
   /Location 30 of base page
FNOP
   /Force a two-word instruction
JA EXAMXR
   /Jump to base page for
   /return to calling program
0
   /Force a two-word instruction
EXMRTN, JA .
   /Will be replaced by return jump
BASE 0
   /Caller's base page
#EXSRT, STARTD
   /Start of subroutine
FLDA 10*3
   /Get return jump from caller's
   /base page
FSTA EXMRTN
   /Save in return location for
   /this routine
FLDA 0
   /Location 0 of caller's routine
   /is a pointer to the argument list
SETX XREXAM
   /Change to EXAMPL's index registers
SETB BPEXAM
   /Change to EXAMPL's base page
BASE BPEXAM
FSTA BPEXAM
   /Save the pointer
LDX l,l
   /Set up index register l
FDLA% BPEXAM, l
   /Get address of argument list
FSTA EXTMP1
   /Save the addresses
FDLA% BPEXAM, l+
   /of all passed arguments
FSTA EXTMP2
   /Continue for all arguments
FSTA EXTMP3
   /to be picked up
STARTF
   /Continue for all arguments
FLDA% EXTMP1
   /Continue to get arguments
   /as required in routine
JA EXMRTN
   /Exit when done
<table>
<thead>
<tr>
<th>RTS ENTRY POINT</th>
<th>USEAGE AND COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>#UE TRAP3 #UE</td>
<td>/Produces USER ERROR error message.</td>
</tr>
<tr>
<td>#ARGER or</td>
<td>TRAP4 #ARGER /Produces BAD ARG error message.</td>
</tr>
<tr>
<td>#ARGERR</td>
<td>TRAP3 #READO /Initializes</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /formatted</td>
</tr>
<tr>
<td></td>
<td>JA FORMAT /read operation.</td>
</tr>
<tr>
<td>#READO</td>
<td>TRAP3 #WITO /Initializes</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /formatted</td>
</tr>
<tr>
<td></td>
<td>JA FORMAT /write operation.</td>
</tr>
<tr>
<td>#WITO</td>
<td>TRAP3 #RUO /Initializes unformatted</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /read operation.</td>
</tr>
<tr>
<td>#RUO</td>
<td>TRAP3 #WUO /Initializes unformatted</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /write operation.</td>
</tr>
<tr>
<td>#WUO</td>
<td>TRAP3 #RDAO /Initializes</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /read operation.</td>
</tr>
<tr>
<td>#RDAO</td>
<td>TRAP3 #WDAO /Initializes</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /read operation.</td>
</tr>
<tr>
<td>#WDAO</td>
<td>TRAP3 #RFSV /Passes a variable to or from the read/write processors via the floating AC.</td>
</tr>
<tr>
<td>#RFSV</td>
<td>TRAP3 #REND0 /Terminates a read/write operation.</td>
</tr>
<tr>
<td>#REND0</td>
<td>FLDA UNITNO /Executes an</td>
</tr>
<tr>
<td></td>
<td>TRAP3 #ENDF /end file,</td>
</tr>
<tr>
<td>#ENDF</td>
<td>or TRAP3 #REW /rewind,</td>
</tr>
<tr>
<td>#REW</td>
<td>or TRAP3 #BAK /backspace (depending upon the entry used)</td>
</tr>
<tr>
<td>#BAK</td>
<td>on the referenced I/O unit.</td>
</tr>
<tr>
<td>#DEF</td>
<td>TRAP3 #DEF /Opens a file</td>
</tr>
<tr>
<td></td>
<td>JA UNITNO /for direct access I/O.</td>
</tr>
<tr>
<td></td>
<td>JA RECORDS</td>
</tr>
<tr>
<td></td>
<td>JA FPNUM /(FPP numbers per record)</td>
</tr>
<tr>
<td></td>
<td>JA VARIABLE /Refer to DEFINE FILE statement</td>
</tr>
<tr>
<td>#EXIT</td>
<td>JSR #EXIT /Terminates current FORTRAN IV job.</td>
</tr>
<tr>
<td>#SWAP</td>
<td>TRAP3 #SWAP /Reads overlay OVLY into level LVL and</td>
</tr>
<tr>
<td></td>
<td>ADDR /jumps to ADR. ADDR is given by:</td>
</tr>
<tr>
<td></td>
<td>ADDR=4000000<em>OVLY+100000</em>LVL+ADR</td>
</tr>
<tr>
<td>#8OR12</td>
<td>/=00000001 if the CPU is a PDP-12.</td>
</tr>
<tr>
<td>#IDLE</td>
<td>Address of background job, used by ONQB. Contains:</td>
</tr>
<tr>
<td></td>
<td>JMP I (NULJOB /Replace by SKP</td>
</tr>
<tr>
<td></td>
<td>0 /Replace by addr of background job</td>
</tr>
<tr>
<td></td>
<td>CDF CIF 0 /Replace by field of background job</td>
</tr>
<tr>
<td></td>
<td>JMS I .-2</td>
</tr>
<tr>
<td></td>
<td>JMP .-4</td>
</tr>
</tbody>
</table>
## Core Layout of FRTS

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>Page zero (0120-0134 free)</td>
</tr>
<tr>
<td>0200</td>
<td>Most entry points, character I/O handlers, interrupt service, and HANG routine</td>
</tr>
<tr>
<td>0600</td>
<td>Format decoder; A, H, and ' format processors, and EXIT</td>
</tr>
<tr>
<td>1400</td>
<td>REWIND, ENDFILE, BACKSPACE and general unit initialization. DATABL table (3 wds/unit)</td>
</tr>
<tr>
<td>2000</td>
<td>I, E, F and G output</td>
</tr>
<tr>
<td>2400</td>
<td>I, E, F and G input</td>
</tr>
<tr>
<td>2600</td>
<td>X, L and T formats and GETHND routine</td>
</tr>
<tr>
<td>3000</td>
<td>Char in and char out routines including OS/8 packing, editing and forms control</td>
</tr>
<tr>
<td>3400</td>
<td>Binary and D. A. I/O, and DEFINE FILE processor</td>
</tr>
<tr>
<td>3600</td>
<td>Overlay loader</td>
</tr>
<tr>
<td>4000</td>
<td>Input line buffer, overlay and DSRN tables, FORMAT parenth pushdown list, /P processor and init flag clear</td>
</tr>
<tr>
<td>4400</td>
<td>Floating-point utilities (shift, add, etc.) used even w/FPP</td>
</tr>
<tr>
<td>4600</td>
<td>Error routine and messages</td>
</tr>
<tr>
<td>5200</td>
<td>OS/8 handler area and part of FRTS loader initialization</td>
</tr>
<tr>
<td>5600</td>
<td>FPP simulator</td>
</tr>
<tr>
<td>6000</td>
<td>FPP start-up and trap routines</td>
</tr>
<tr>
<td>6600</td>
<td>B and D format I/O</td>
</tr>
<tr>
<td>6600</td>
<td>Floating-point package and part of LPT ring buffer</td>
</tr>
<tr>
<td>7400</td>
<td>Floating-point package (never used) and part of LPT ring buffer</td>
</tr>
<tr>
<td>7600</td>
<td>Most of LPT ring buffer</td>
</tr>
<tr>
<td>7600</td>
<td>OS/8 handler and field 0 resident</td>
</tr>
<tr>
<td>10000</td>
<td>OS/8 User Service Routine</td>
</tr>
</tbody>
</table>
12000 | FRTS loader tables, IONTBL | Locations 12000 to 17400 are overlaid at execution time
12200 | FRTS loader: main flow |
12400 | program start-up\(^1\) |
12600 | initialize and configure system |
13000 | Load OS/8 handlers and assign unit numbers to OS/8 files |
13400 | Utility and error routines, error messages |
14000 |
15600 | FPP start-up and trap routines | Locations 14000 to 16777 are used to save lower field 0 during loading of device handlers and file specifications |
16000 | B and D format I/O |
16600 | EAE Floating-point package |
17400 | Termination routine | Locations 17400 to 17777 are written on SYS block 37 before program load and restored on termination |
17600 | OS/8 field 1 resident |

\(^1\)Program start-up moves OS/8 handler to top of core, writes field 1 resident onto SYS, and termination routine goes to FRTS to load program.

#INT
   /Address of user interrupt location, used by ONQI:
   JMP .+4 /Replace with SKP
   0 /Replace with address of interrupt processor
   CDF CIF 0 /Replace with field of interrupt processor
   JMS I -.2

#DISMS
   /Addresses first of three JMP DISMIS instructions for use by specialized I/O routines.

#HANG
   /Addresses I/O dismiss routine.

#RETRN
   /Provides return from TRAP3.
The DSRN table controls files and I/O devices used under OS/8 FORTRAN IV ASCII, binary and direct access I/O operations, including BACKSPACE, REWIND, and END FILE operations. The exact meaning of the initials DSRN is one of the great, unanswered questions of FORTRAN IV development and, as such, has considerable historical interest. The DSRN table provides room for 9 entries; each entry is 9 words in length, and contains the following data:

**WORD 1:** (HAND) Handler entry point. If this value is positive, the I/O device handler is a FORTRAN internal (character-oriented) handler, and the remainder of the DSRN table entry is ignored. If the value is negative, the handler is an OS/8 device handler whose entry point is the two's complement of the value. Entry points always fall in the range [7607, 7777] for resident handlers or [5200, 5377] for non-resident handlers. Space for non-resident handlers is allocated downward from the top of memory, and the handlers are moved into locations 5200 to 5577 before being called.

**WORD 2:** (HCODEW) Handler code word. Bits 0-4 of this word specify the page into which the device handler was loaded, while bits 6-8 specify the memory field. If all of bits 0-8 are zero, the handler is permanently resident. When any of these bits are non-zero, the data is used to determine which handler, if any, currently occupies locations 5200-5577. This eliminates unnecessarily moving the content of memory. Bit 10 is set if forms control has been inhibited on the I/O unit. Bit 11 is set if the device handler can execute with the interrupt system enabled. The data in bits 10 and 11 is obtained from the IOWTBL table in the FRTS loader.

**WORD 3:** (BADFDL) Buffer address and field. Bits 0-4 address the memory page at which the I/O buffer for this unit begins, while bits 6-8 specify the memory field. Unlike the FORTRAN internal I/O unit buffers, OS/8 device handler buffers always occupy two full pages of memory. Buffer space is allocated upward from the top of the FORTRAN program.

**WORD 4:** (CHRPTER) Character pointer.

**WORD 5:** (CHRCTTR) Character counter. Words 4 and 5 of each DSRN table entry define the current character/position in the I/O buffer as follows:
<table>
<thead>
<tr>
<th>Value of CHRCTR</th>
<th>Character position</th>
<th>Next value of CHRCTR</th>
<th>Next value of CHRPRTR</th>
<th>Special Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>Bits 4-11 of word addressed by CHRPRTR</td>
<td>-2</td>
<td>CHRPRTR + 1</td>
<td>Refresh buffer if input operation and CHRPRTR mod 256=0</td>
</tr>
<tr>
<td>-2</td>
<td>&quot;</td>
<td>-1</td>
<td>&quot;</td>
<td>none</td>
</tr>
<tr>
<td>-1</td>
<td>Bits 0-3 of words addressed by CHRPRTR-2 and CHRPRTR-1</td>
<td>-3</td>
<td>CHRPRTR</td>
<td>Dump buffer if output operation and CHRPRTR mod 256=0</td>
</tr>
</tbody>
</table>

WORD 6: (STBLK) Starting block of file.

WORD 7: (RELBLIC) Current relative block of file. That is, block to be accessed next.

WORD 8: (TOTBLK) Length of file in blocks.

WORD 9: (FFLAGS) Status flags:

- Bit 0 - Has been written flag. Set to 1 if unit has received output since last REWIND.
- Bit 1 - Formatted I/O flag. Set to 1 if an ASCII I/O operation has occurred since last REWIND.
- Bit 2 - Unformatted I/O flag. Set to 1 if a binary or direct access I/O operation has occurred since last REWIND. Bits 1 and 2 are never set simultaneously.
- Bit 11 - END FILEd flag. Set to 1 if unit has been END FILEd. Bit 11 is not cleared by a REWIND.

When any active unit is selected for an I/O operation, the DSRN table entry for that unit is moved into 9 words on page 0. These 9 words are tagged with the labels cited above. Upon completion of the I/O operation, the 9 words are moved from page 0 back into the DSRN table.
/FORTRAN 4 RUNTIME SYSTEM - R.L. PALE-V8

/PAGE ZERO FOR FORTRAN IV RTS

0000      *Ø        /INTERRUPT STUFF
00000     00000     0
00001     5402      JMP I	+1
00002     0400      INTRPT
00003     5165      LPGET, LPBUFR   /LINE PRINTER RING BUFFER FETCH
00004     0000      TOCHR, 0     /TELETYPWR STATUS WORD
00005     0000      KBDCHR, 0     /KEYBOARD INPUT CHARACTER
00006     0000      POCCHR, 0     /P.T. PUNCH COMPLETION FLAG
00007     0000      RDRCHR, 0     /P.T. READER STATUS
00010     0000      FMTXPXR, 0    /XR USED TO INDEX FORMAT PARENTHESES
00011     3777      INXR, INBUFR-1 /XR USED TO GET CHARs FROM INPUT
00012     0000      XR, 0
00013     0000      XRI, 0

0016      *16
00016     0000     VEOFSW, 0     /USED BY "EOFCHK" TO STORE VARIABLE ADDRESS
00017     0000     0           /** MUST BE IN AUTO - XR
00020     0000     T, 0        /TEMPORARY
00021     0000     DFLG, 0     /Ø = F.P., 1 = D.P.
00022     0000     INST, 0     /CURRENT INSTRUCTION WORD

/IOH PAGE ZERO LOCATIONS

00023     0000     RWFLAG, 0     /READ/WRITE FLAG
00024     0000     FMITYP, 0     /TYPE OF CONVERSION BEING DONE
00025     0000     EOLS, 0      /EOL SW ON INPUT - CHAR POS ON OUT
00026     0000     N, 0        /REPEAT FACTOR
00027     0000     W, 0        /FIELD WIDTH
00030     0000     D, 0        /NUMBER OF PLACES AFTER DECIMAL
00031     0000     DATCDF, 0     /SUBROUTINE TO CHANGE DATA FIELD
00032     0000     DATAP, 0     /CONTAINS VARIOUS CDF'S
00033     5431      JMP I     DATCDF /RETURN
00034     5013      ERR, ERROR   /POINTER TO ERROR ROUTINE
00035     0000      FATAL, 0    /FATAL ERROR FLAG - Ø=FATAL
00036     5000      MCFDF, MAXCDF

/FPP PARAMETER TABLE LOCATIONS:

00037     0000     APT, 0     /VARIOUS FIELD BITS FOR FPP
00040     5313      PC, DPIEST  /FPP PROGRAM COUNTER
00041     0000     XRBASE, 0   /FPP INDEX REGISTER ARRAY ADDRESS
00042     0000     BASADR, 0   /FPP BASE PAGE ADDRESS
00043     0000     ADR, 0      /ADDRESS TEMPORARY
00044     0000     ACK, 0
00045     0000     ACH, 0     /*** FLOATING ACCUMULATOR ***
00046     0000     ACL, 0
00047     0000     EAC1, 0    /** FOR EXTENDED PRECISION OPTION **
00050     0000     EAC2, 0
00051     0000     EAC3, 0
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8

/FLOATING POINT PACKAGE LOCATIONS

00052 0000 AC0, 0
00053 0000 AC1, 0          /FLOATING AC OVERFLOW WORD
00054 0000 AC2, 0          /OPERAND OVFLOW WORD
00055 0000 OPH, 0          /*** FLOATING OPERAND REGISTER ***
00056 0000 OPL, 0

/RIS I/O SYSTEM LOCATIONS

00060 0000 FMITBYT, 0       /FORMAT BYTE POINTER
00061 0000 IFLG, 0          /I FORMAT FLAG
00062 0000 GFLG, 0          /G FORMAT FLAG
00063 0000 EFLG, 0          /E FORMAT FLAG - SOMETIMES ON FOR
00064 0000 OD, 0
00065 0000 SCALE, 0
00066 0000 PFACT, 0         /P-SCALE FACTOR
00067 0000 PFACTX, 0        /TEMP FOR PFACT
00070 0000 INESW, 0         /EXponent Switch
00071 0000 CHCH, 0
00072 0000 FMINUM, 0        /CONTAINS ACCUMULATED NUMERIC VALUE
00073 0000 CCTCNH, 0        /C INHIBIT FLAG
00074 0320 PTTY, ITY        /POINTER TO ITY HANDLER - USED BY
00075 0000 0                /SO FORMS CONTROL WILL WORK ON
00076 6001 FPXNT, ICYCLE    /USED AS INTERPRETER ADDRESS IF

/DSRN IMAGE

00077 0000 HAND, 0          /HANDLER ENTRY POINT
00100 0000 HCODEW, 0        /HANDLER LOAD ADDR & FIELD + IOFFL
00101 0000 BADFIL, 0        /BUFFER ADDRESS AND FIELD
00102 0000 CHRPTR, 0        /ACTUALLY A WORD POINTER
00103 0000 CHRCTR, 0        /COUNTER - RANGES FROM -3 TO -1
00104 0000 STBLK, 0         /STARTING BLOCK OF FILE
00105 0000 RELBLK, 0        /CURRENT RELATIVE BLOCK NUMBER
00106 0000 TOTBLK, 0        /LENGTH OF FILE
00107 0000 FFLAGS, 0        /FILE FLAGS:
                            /BIT 0 - "HAS BEEN WRITTEN" FLAG
                            /BITS 1-2 - FORMATTED/UNFORMATTED
                            /BIT 11 - "END-FILE" FLAG

00110 0000 BUFDL, 0         /ROUTINE TO SET DF TO BUFFER FIELD
00111 7402 BUFCDF, HLIT
00112 5510 JMP I BUFDL

00113 0000 FGPBF, 0         /THESE THREE WORDS ARE USED
00114 0000 BIPTR, 0         /TO FETCH AND STORE FLOATING POINT
00115 0000 FEXIT
0200  PAGE

4-11
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V6  PAGE 5

/STARTUP CODE

00200  2203  FTEMP2, ISZ  +3  /ALSO USED AS I/O F.P. TEMPORARY
00201  6213  CDF CIF 10
00202  5683  JMP I  +1
00203  2200  VDATE, RTSLDR  /USED TO STORE OS/8 DATE

/RTS ENTRY POINTS - "VERSION INDEPENDENT"

00204  5777  VUERR, JMP I  (USERR /USER ERROR

00205  4434  VARGER, JMS I  ERR  /LIBRARY ARGUMENT ERROR

00206  2023  VREND0, ISZ  IFLAG  /END OF I/O LIST

00207  5634  VRFSV, JMP I  GELMN  /I/O LIST ARG ENTRY - COROUTINE

00208  5776  VBAK, JMP I  (BKSPC "BACKSPACE" ROUTINE

00209  5775  VENDP, JMP I  (ENDFS "END FILE" ROUTINE

00210  5774  VREW, JMP I  (HWIN "REWIND ROUTINE

00211  5773  VDEP, JMP I  (DEFINE "DEFINE FILE" ROUTINE

00212  7330  VWO, AC4000  /UNFORMATTED WRITE

00213  5772  VROI, JMP I  (RWUNF /UNFORMATTED READ

00214  7330  WDAO, AC4000  /DIRECT ACCESS WRITE

00215  5771  VRDAO, JMP I  (RWDACC /DIRECT ACCESS READ

00216  7330  WRIOT, AC4000  /FORMATTED (ASCII) WRITE

00217  5770  VREAD0, JMP I  (RWASC1 /FORMATTED (ASCII) READ

00218  5767  VSWAP, JMP I  (SWAP /OVERLAY PROCESSOR

00219  3000  VEXIT, TRAP5;  CALX11 /"STOP" ROUTINE - ENTERED IN FPP

00220  1317  VBG0R, 0;01  /0;1 IF CPU IS A PDP-12

00221  0000  VB0R12, 0;01

00222  5766  VBACKG, JMP I  (NULLJB /BACKGROUND JOB DISPATCHER

00223  0000  0

00224  0203  CDF CIF 0  /USED BY ROUTINE "ONQB" IN LIBRARY

00225  4530  JMS I  -2

00226  5227  JMP VBACKG

/IOH GET VARIABLE ROUTINE.
/THIS ROUTINE MAKES THE FORMATTED I/O PROCESSOR AND THE
/PROGRAM CO-RUTINES (DEF(COROUTINE)= 2 ROUTINES EACH
/IS A SUBROUTINE). ON ENTRY FAC=INPUT NUMBER
/IF I/O IS A READ, ON RETURN FAC=OUTPUT NUMBER IF I/O

00234  0000  GELMN, 0

00235  5777  VRETNR, JMP I  (RETURN
All FORTRAN IV mass storage I/O is performed in terms of OS/8 blocks, including direct access I/O. Hence, all FORTRAN IV files conform to OS/8 standard ASCII file format. When a formatted READ or WRITE is requested, the data is converted to or from 8-bit binary representation according to the FORMAT statement associated with the READ or WRITE. Standard OS/8 file format packs three 8-bit characters into two 12-bit words as follows:

<table>
<thead>
<tr>
<th>MASS STORAGE</th>
<th>CORE</th>
</tr>
</thead>
<tbody>
<tr>
<td>WORD 3 bits 0-3</td>
<td>WORD 1</td>
</tr>
<tr>
<td>WORD 3 bits 4-7</td>
<td>WORD 2</td>
</tr>
<tr>
<td></td>
<td>WORD 3</td>
</tr>
</tbody>
</table>

Unformatted (i.e., direct access) READ and WRITE operations also operate on standard OS/8 format files, with each statement causing one FORTRAN IV record to be read or written. A FORTRAN IV record must contain at least one OS/8 block, and always contains an integral number of blocks. The number of variables contained in a 1-block record depends upon the content and format of the I/O list, as follows:

<table>
<thead>
<tr>
<th>Format type</th>
<th>Number of 12-bit Words/Variable</th>
<th>Number of Variables/Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>3</td>
<td>85</td>
</tr>
<tr>
<td>Real</td>
<td>3</td>
<td>85</td>
</tr>
<tr>
<td>Double precision</td>
<td>6</td>
<td>42 1/2</td>
</tr>
<tr>
<td>Complex</td>
<td>6</td>
<td>42 1/2</td>
</tr>
</tbody>
</table>

It is possible to mix any types of data in an I/O list; however, no more than 85 variables may be stored in one OS/8 block. The number of blocks required for a FORTRAN IV record depends, therefore, upon the number of variables in the I/O list, and may be minimized by supplying every direct access WRITE with sufficient data to nearly fill an integral number of blocks without overflowing the last block.
The last word in every file block contains a block count sequence number and is not available for data storage. FRTS assigns block count numbers sequentially, beginning with 1, whenever a file is written. Block count numbers must be maintained by the user when FORTRAN IV files are created outside of an OS/8 FORTRAN IV environment. While reading a binary file, FRTS checks the block count sequence numbers on input blocks and ignores any block whose sequence number is larger than expected. Sequence number checking is disabled during direct access READ operations.

When FRTS is loaded and started, the initialization routines determine what optional hardware, such as FPP-12 Floating Point Processor or KESE Extended Arithmetic Element, is present in the running hardware configuration. The initialization routines then modify FRTS to use the optional hardware, if available. When an FPP is present in the system and it becomes desirable to disable the FPP under FRTS, this may be accomplished by changing the content of location 12621 from 6555 to 7200. The extended arithmetic element may be disabled in the same manner by changing the content of FRTS location 12623 from 7413 to 7200. These changes must be made before FRTS is started. The OS/8 monitor GET and ODT commands provide an excellent mechanism for changes of this type.

The FRTS internal line printer handler uses a linked ring buffer for maximum I/O buffering efficiency. The buffer consists of several contiguous sections of memory, linked together by pointers. All of these buffer segments are located above 04000, so that the pointers are readily distinguishable from buffereded characters. The entire 07400 page is included in the line printer ring buffer. If it becomes desirable to modify FRTS by patching or reassembly, most of the 07400 page may be reclaimed from the buffer by changing the
content of location 07402 from 7577 to 5164. This frees up locations 
07403 to 07577 for new code and still leaves about eighty character 
positions in the LPT ring buffer.

Because FRTS executes with the processor interrupt system enabled, 
it may hang up on hardware configurations that include equipment 
capable of generating spurious program interrupts. In addition, 
any OS/8 I/O device handler that exits without clearing all device 
flags may cause troublesome interrupts when it is assigned as a 
FORTRAN I/O unit under FRTS. To counteract these potential 
problems, FRTS provides certain areas that are reserved for 
ineclusion of user-generated code designed to clear device flags 
and/or inhibit spurious interrupts.

A string of NOP instructions beginning at location 04020 is 
executed during FRTS initialization, just before the interrupt 
system is enabled. When the \h option is specified to FRTS, the 
system halts after these NOPs have been executed and the interrupt 
system has been enabled. Another string of NOPs occupying the 
eight locations from 03746 to 03755 is executed after every call 
to an OS/8 device handler. Any of these NOP instructions may be 
replaced by flag-handling or interrupt-servicing code. If 
additional memory locations are required, they may be obtained by 
replacing some of the code from locations 04007 to 04017 with 
flag-handling code. Locations 04007-17 are used to clear flags 
associated with LAB-8/E peripheral devices.

Due to memory limitations, it is not possible to add internal I/O 
device handlers to the four internal handlers supplied with the 
system. However, FORTRAN I/O unit 0, which is not defined by the ANSI 
standard, may be specified for terminal I/O via the internal console 
terminal handler. I/O unit 0 is not re-assignable.
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 6

/INTERRUPT DRIVEN I/O HANDLERS

00236 0000 LPI, 0 /RING-BUFFERED - LP08 OR LS6E
00237 0176 AND 377 /JUST IN CASE
00240 7450 LPTSN A, SNA
00241 5765 JMP I (IOERR /CANNOT BE USED FOR INPUT
00242 6002 IOF
00243 3667 DCA I LPPUT
00244 1003 TAD LPGET
00245 7041 CIA
00246 1267 TAD LPPUT
00247 7640 SZA CLA /IS LPI QUIET?
00250 5253 JMP +3 /NO
00251 1667 TAD I LPPUT
00252 6666 LLS /YES - START 'ER UP
00253 7201 CLA IAC
00254 6665 LIE /ENABLE LPT INTERRUPTS
00255 1267 TAD LPPUT /IN AC, REMEMBER?
00256 3267 DCA LPPUT
00257 1667 TAD I LPPUT
00260 7510 SPA
00261 5256 JMP -3 /NEGATIVE NUMBERS ARE BUFFER LINKS
00262 7640 SZA CLA /ANY ROOM LEFT IN BUFFER?
00263 4764 JMS I (HANG
00264 0436 LPUHNG /WAIT FOR LINE PRINTER
00265 6001 ION /TURN INTERRUPTS BACK ON
00266 5636 JMP I LPT /RETURN

00267 5165 LPPUT, LPBUFR

00270 0000 PIP, 0 /PAPER TAPE PUNCH HANDLER
00271 7450 SNA
00272 5765 JMP I (IOERR /INPUT IS ERROR
00273 3236 DCA LPT /SAVE CHAR
00274 6002 IOF
00275 1006 TAD POCHR /IF PUNCH IS NOT IDLE,
00276 7640 SZA CLA /WE DISMISS JOB
00277 4764 JMS I (HANG
00300 0502 PPUHNG /WAIT FOR PUNCH INTERRUPT
00301 1236 TAD LPT
00302 6026 PLS /OUTPUT CHAR
00303 3006 DCA POCHR /SET FLAG NON-ZERO
00304 6001 ION
00305 5670 JMP I PIP

/*K* THE FOLLOWING ADDRESSES GET FALLEN INTO & MUST BE SMALL

IFNZRO PPUHNG&7000 <<<ERROR>>>
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 7

/INTERRUPT-DRIVEN PTR AND TELETYPETO HANDLE

00306 0000 PTR, 0 /CRUDE READER HANDLER
00307 7640 SZA CLA
00310 5765 JMP I COERR /OUTPUT ILLEGAL TO PTR
00311 6062 IOF
00312 6014 RFC /START READER
00313 4764 JMS I CHANG
00314 0510 RDUHNG /HANG UNTIL COMPLETE
00315 1007 TAD RDRCHR /GET CHARACTER
00316 6001 ION
00317 5706 JMP I PTR /RETURN

00320 0000 ITY, 0 /BUFFERS 2 CHARMS ON OUTPUT, I ON
00321 6002 IOF /DELECTIVE CODE AHEAD
00322 7450 SNA /INPUT OR OUTPUT?
00323 5342 JMP KBD /INPUT
00324 3836 DCA LPT /OUTPUT - SAVE CHAR
00325 1004 TAD TOCHR /GET ITY STATUS
00326 7740 SNA SZA CLA /G.1. 0 MEANS A CHAR IS BACKED UP
00327 4764 JMS I CHANG
00330 0451 ITUHNG /WAIT FOR LOG JAM TO CLEAR
00331 1004 TAD TOCHR /NO CHAR BACKED UP - SEE IF ITY
00332 7104 CLL RAL /*BUSY" FLAG IN LINK - INTERRUPTS
00333 7230 CLA CML RAR /*COMPLEMENT OF BUSY IN SIGN
00334 1236 TAD LPI /GET CHAR
00335 7510 SPA /IF ITY NOT BUSY,
00336 6046 TLS /OUTPUT CHAR
00337 3004 DCA TOCHR /STORE POS OR NEG, BACKED UP
00340 6001 ITYREI, ION /TURN INTERRUPTS BACK ON
00341 5720 JMP I ITY /AND LEAVE

/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 8

00342 1005 KBD, TAD KBDCMR /HAS A CHARACTER BEEN INPUT?
00343 7650 SNA CLA
00344 4764 JMS I CHANG
00345 0465 KBUHNG /NO - RUN BACKGROUND UNTIL ONE IS
00346 1005 TAD KBDCMR /GET CHARACTER
00347 3236 DCA LPI
00350 3005 DCA KBDCHR /CLEAR CHARACTER BUFFER
00351 1236 TAD LPT
00352 5340 JMP ITYREI /RETURN WITH INTERRUPTS ON

00353 6554 KILFPP, FPFLIT /*BRING FPP TO A SCREECHING HALT
00354 2353 ISZ .-1
00355 5354 JMP .-1 /WAIT FOR IT TO STOP
00356 6552 FPICL /CLEAN UP MESS HALT HAS MADE IN FPP
00357 7430 SZL /TC OR TB?
00360 5763 JMP I (7600 /%C - HIYO SILVER, AWAY!
00361 6032 KCC /CLEAR KBD FLAG ON TB
00362 4434 CTLBER, JMS I ERR /*** THIS MAY BE DANGEROUS! **
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8

/INTERUPT SERVICE ROUTINES

00400  3322  INTRPT, DCA  INTAC
00401  7010  RAR
00402  3323  DCA  INILNK
00403  5207  VINT, JMP  +4  /** MUST BE AT 403 **
           IFNZRO VINT-403  <=== CHANGE LOADER!!!>
00404  0000  0
00405  6203  CDF CIF 0  /USER INTERRUPT ROUTINE GOES HERE
00406  4604  JMS I  .-2
00407  6551  FPINT  /CHECK FOR FPP DONE
00410  5215  JMP  LPTEST
00411  5314  FPUHNG, JMP  DISMIS  /ALWAYS GOES TO RESTRT
00412  5314  VDISMS, JMP  DISMIS  /FOR USE BY USERS
00413  5314  JMP  DISMIS
00414  5314  JMP  DISMIS
00415  6661  LPTEST, LSF
00416  5240  JMP  NOTLPI
00417  6662  LPTLCF, LCF  NOTLPI  /CLEAR FLAG
00420  1403  TAD I  LPGET
00421  7650  SNA CLA  /CHECK FOR SPURIOUS INTERRUPT
00422  5314  JMPDIS, JMP  DISMIS  /GO AWAY IF SO
00423  3403  DCA I  LPGET  /ZERO CHAR JUST OUTPUT
00424  2003  ISZ  LPGET
00425  1403  TAD I  LPGET
00426  7510  SPA
00427  3033  DCA  LPGET  /TAKE CARE OF BUFFER LINKS
00430  7450  SNA
00431  1403  TAD I  LPGET  /MAKE SURE CHAR IS IN AC
00432  7440  SZA  /IS THERE A CHARACTER?
00433  6666  LLS  /YES - PRINT IT
00434  7200  CLA
00435  6661  LSF  /CHECK FOR IMMEDIATE FLAG
00436  5314  LPUHNG, JMP  DISMIS  /NO - MAYBE RESTART PROGRAM
00437  5217  JMP  LPTLCF  /YES - LOOP
00440  6041  NOTLPI, ISF  /CHECK ITY
00441  5252  JMP  NOTITY
00442  6042  TCF  /CLEAR FLAG
00443  1004  TAD TOCHR  /GET ITY STATUS
00444  7540  SMA SZA  /IF THERE IS A CHARACTER WAITING,
00445  6046  TLS  /OUTPUT IT.
00446  7740  SMA SZA CLA  /CHANGE "WAITING" TO "BUSY",
00447  7150  STI RAR  /"BUSY" TO "IDLE".
00450  3004  DCA TOCHR
00451  5314  TIUHNG, JMP  DISMIS
/FORTRAN 4 RUNTIME SYSTEM - R.L. PALB-V8

/KBD AND PIP INTERRUPTS

00452 6031 NOTITY, KSF
00453 5276 JMP NOTIKBD
00454 1175 TAD [200]
00455 6034 KRS /USE KRS TO FORCE PARITY BIT
00456 3005 DCA KBDCHR /AND ALSO SO THAT *C WILL STILL
00457 1005 TAD KBDCHR
00458 1377 TAD <-202 /CHECK FOR *C OR *B
00461 7120 CLL RAR
00462 7650 SNA CLA
00463 5266 JMP CTCCIB /YUP - TAKE SOME DRASIC ACTION
00464 6032 KCC /DATA CHARACTER - CLEAR FLAG
00465 5314 KBUHNG, JMP DISMIS

00466 1073 CTCCIB, TAD CTICNH
00467 7650 SNA CLA /ARE WE IN A HANDLER?
00470 5366 JMP NOTINH /NO
00471 1323 TAD INTLNK
00472 7104 CLL RAL /YES - RETURN WITH INTERRUPTS OFF
00473 1322 TAD INTAC /TRUST IN GOD AND RIS
00474 6244 RMF
00475 5400 JMP I 0

00476 6021 NOTIKBD, PSF
00477 5303 JMP NOTIPIP
00500 6022 PCF /P.T. PUNCH INTERRUPT - CLEAR FLAG
00501 3006 DCA POCHR /CLEAR SOFTWARE FLAG
00502 5314 PPUHNG, JMP DISMIS

00503 6011 NOTIPIP, RSF
00504 5311 JMP LPIERR
00505 1175 TAD [200
00506 6012 RRB /GET RDR CHAR
00507 2007 DCA RDCRCH
00510 5314 RDUHNG, JMP DISMIS

00511 6663 LPIERR, LSE /TEST FOR LP08 ERROR FLAG
00512 7410 SKP
00513 6667 LIF /DISABLE LP08 INTERRUPTS IF ERROR
00514 1323 DISMIS, TAD INTLNK
00515 7124 CLL RAL
00516 1322 TAD INTAC /RESTORE AC AND LINK
00517 6244 RMF
00520 6001 ION
00521 5400 JMP I 0 /RETURN FROM THE INTERRUPT

00522 0000 INTAC, 0
00523 0000 INTLNK, 0
/FORTRAN 4 RUNTIME SYSTEM - R.L  PALB-V8

/BACKGROUND INITIATE/TERMINATE ROUTINE

00524 0000 HANG, 0 /ALWAYS CALLED WITH INTERRUPTS OFF!
00525 1724 TAD I HANG /GET POINTER TO UNHANGING LOCATION
00526 3371 DCA UNHANG
00527 6214 RDF /GET FIELD CALLED FROM
00530 1332 TAD HCIDFØ
00531 3364 DCA HNGCDF /SAVE FOR RETURN
00532 6203 HCIDFØ, CDF CIF Ø
00533 1376 TAD (JMP RESTRT) /CHANGE THE "JMP DISMIS"
00534 3771 DCA I UNHANG /TO A "JMP RESTRT"
00535 1373 TAD BACKLK
00536 7104 CLL RAL
00537 1372 TAD BACKAC /SET UP BACKGROUND AC AND LINK
00540 6202 BAKCIF, CIF Ø
00541 6201 BAKCDF, CDF Ø
00542 6001 ION
00543 5774 JMP I BACKPC /INITIATE BACKGROUND

/ COME HERE WHEN THE HANG CONDITION HAS GONE AWAY

00544 1222 RESTRT, TAD JMPDIS /RESTORE THE UNHANG LOCATION
00545 3771 DCA I UNHANG
00546 1322 TAD INTAC /SUSPEND THE BACKGROUND
00547 3372 DCA BACKAC
00548 1323 TAD INTLWK
00551 3373 DCA BACKLK
00552 1000 TAD Ø
00553 3374 DCA BACKPC
00554 6234 RIB
00555 0174 AND [IØ
00556 1332 TAD HCIDFØ
00557 3340 DCA BAKCIF
00560 6234 RIB
00561 4436 JMS I MCFD /*K*/ OK SINCE BACKGROUND DOESN'T
00562 3341 DCA BAKCDF
00563 2324 ISZ HANG
00564 7402 HNGCDF, HLT
00565 5724 JMP I HANG /INTERRUPTS ARE OFF - RETURN

00566 1222 NOTINH, TAD JMPDIS /IN CASE WE WERE HUNG, WE DON'T
00567 3771 DCA I UNHANG /TO GET "UNHANG" OUT OF THE ERROR
00570 5775 JMP I (KILFPP /KILL FPP AND GO TO EXIT OR ERROR

00571 0000 UNHANG, Ø
00572 0000 BACKAC, Ø
00573 0000 BACKLK, Ø
00574 0227 BACKPC, VBACKG
0524 VHANG= HANG
        IFNZRO VHANG=0524  <--- CHANGE LOADER!>

00575 0353
00576 5344
00577 7576
0600  PAGE
The FRTS /P option provides a mechanism whereby the core image generated from a FORTRAN program may be punched onto paper tape in binary loader format. This permits the loader image to be executed on a hardware configuration that does not include mass-storage devices.

To use the /P option, specify /P to FRTS and assign a device or file as FORTRAN I/O unit 9. Assigning the paper tape punch as unit 9 causes the image to be punched out directly; however, it may be desirable to direct the binary output to an intermediate file for later transfer to paper tape via OS/9 PIP. In any event, FRTS returns to the monitor once the core image has been transferred.

The output file is a binary image of memory locations 00000 to 77577 and 10000 up to the highest location used by the FORTRAN load. The content of each field is punched separately with its own checksum and leader/trailer.

With the BIN loader resident in field $, load the binary tape produced under the /P option by reading each segment separately and verifying the checksum as each memory field is loaded. When all segments have been read into memory, start execution at location $2000. The following restrictions apply:

1. OS/8 device handlers which have been assigned FORTRAN I/O unit numbers are not necessarily punched out. For this reason, I/O unit assignments other than in the form /n=m should be avoided.

2. With respect to the presence of an FPP and/or EAE, the configuration on which the image is punched must be identical to the configuration on which it is to be run. If the punching configuration contains hardware that is absent from the target configuration, this hardware must be disabled under FRTS. If the target configuration contains hardware that is absent from the punching configuration, the extraneous hardware will not be used.

3. The statements STOP and CALL EXIT cause a core load produced under the /P option to halt. Any fatal error flagged during punching or execution causes error traceback followed by a halt. Do not press CONTinue in response to either of these machine halts.
A FORTRAN IV program is terminated in one of three ways:

1. A fatal error condition is flagged (CTRL/B) is processed as a fatal error.
2. CTRL/C is recognized, or the CPU is halted and re-started in 07600.
3. A STOP, CALL EXIT, or (under RALF) JSR #EXIT statement is executed.

The sequence of events that results in program termination proceeds as follows:

At point A, FRTS executes the following operations.

1. Read termination routine into memory.
2. Read OS/8 field 0 resident from block 37 of SYS.
3. Jump into termination routine at location 17400.
4. Restore normal content of locations 07600 and 07605 (in OS/8 resident).
5. If configuration is an in-core TD8E DECTape system, restore second part of TD8E handler from n7600 to 27600.
6. Wait for TTY to finish all pending I/O. If BATCH is running, print LF on TTY and LPT.
7. If normal termination flag is set, close any output files that were opened by the FRTS loader.
8. Return to OS/8 monitor via location 07605.
/FORTRAN 4 RUNTIME SYSTEM - R.L  PALS-V6

6600 FPPKG= .  /FOR EAE OVERLAY

/23-BIT FLOATING PT INTERPRETER
/W.J. CLOGHER, MODIFIED BY R.LARY FOR FORTRAN

06600 0000 LPBUF2, ZBLOCK 16
06616 7160 LPBUF3

06617 0000 ALIBM, 0  /**K* UTILITY SUBROUTINE
06620 7240 SIA
06621 1044 TAD ACX
06622 3044 DCA ACX
06623 4542 JMS I [AL]
06624 5617 JMP I ALIBM

/FLOATING MULTIPLY-DOES 2 24x12 BIT MULTIPLIES
06625 4777 DDMPY, JMS I [DARGET
06626 7410 SKP
06627 4776 FFM PY, JMS I [TARGET /GET OPERAND
06630 4304 JMS MDSET /SET UP FOR MPY-OPX IN AC ON REIN.
06631 1044 TAD ACX /DO EXPONENT ADDITION
06632 3044 DCA ACX /STORE FINAL EXPONENT
06633 3304 DCA MDSET /ZERO TEMP STORAGE FOR MPY ROUTINE
06634 3054 DCA AC2
06635 1045 TAD ACH /IS FAC=0?
06636 7650 SNA CLA
06637 3044 DCA ACX /YES-ZERO EXPONENT
06640 4334 JMS MP24 /NO-MULTIPLY FAC BY LOW ORDER OPR.
06641 1056 TAD OPH /NOW MULTIPLY FAC BY HI ORDER MULT
06642 3057 DCA OPL
06643 4334 JMS MP24
06644 1054 TAD AC2 /STORE RESULT BACK IN FAC
06645 3046 DCA ACL /LOW ORDER
06646 1304 TAD MDSET /HIGH ORDER
06647 3045 DCA ACH
06650 1045 TAD ACH /DO WE NEED TO NORMALIZE?
06651 7004 RAL
06652 7710 SPA CLA
06653 4217 JMS ALIBM /YES-DO IT FAST
06654 1053 TAD AC1
06655 7710 SPA CLA /CHECK OVERFLOW WORD
06656 2046 ISZ ACL /HIGH BIT ON - ROUND RESULT
06657 5265 JMP MDONE
06660 2045 ISZ ACH /LOW ORDER OVERFLOWED - INCREMENT
06661 1045 TAD ACH
06662 7510 SPA /CHECK FOR OVERFLOW TO 4000 0000
06663 5775 JMP I [SHR1 /WE HANDLE A SIMILAR CASE IN
06664 7200 CLA

4-23
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 79

06665 3053 MDONE, DCA ACI /ZERO OVERFLOW WD(DO I NEED THIS???
06666 2333 ISZ MSIGN /SHOULD RESULT BE NEGATIVE?
06667 7410 SKP /NO
06670 4543 JMS I [FFNEG /YES-NEGATE IT
06671 1045 TAD ACH
06672 7650 SNA CLA /A ZERO AC MEANS A ZERO EXPONENT
06673 3044 DCA ACX
06674 1021 TAD DFLG
06675 7740 SMA SZA CLA /D.P. INTEGER MODE?
06676 1044 TAD ACX /WITH ACX LESS THAN 0?
06677 7450 SNA
06700 5476 JMP I FPNXT /NO - RETURN
06701 7040 CLA
06702 4541 JMS I [ACSR /UN-NORMALIZE RESULT
06703 5476 JMP I FPNXT /RETURN

/MDSEI-SEIS UP SIGNS FOR MULTIPLY AND DIVIDE
/ALSO SHIFTS OPERAND ONE BIT TO THE LEFT.
/EXIT WITH EXPONENT OF OPERAND IN AC FOR EXPONENT
/CALCULATION-CALLED WITH ADDRESS OF OPERAND IN AC AND
/DATA FIELD SET PROPERLY FOR OPERAND.

06704 0000 MDSET, 0
06705 7344 CLA CLL CMA RAL /SET SIGN CHECK TO -2
06706 3333 DCA MSIGN
06707 1056 TAD OPH /IS OPERAND NEGATIVE?
06710 7700 SMA CLA
06711 5314 JMP +3 /NO
06712 4774 JMS I [OPNEG /YES-NEGATE IT
06713 2333 ISZ MSIGN /BUMP SIGN CHECK
06714 1057 TAD OPL /AND SHIFT OPERAND LEFT ONE BIT
06715 7104 CLL RAL
06716 3057 DCA OPL
06717 1056 TAD OPH
06720 7084 RAL
06721 3056 DCA OPH
06722 3053 DCA ACI /CLR OVERFLOW WORF OF FAC
06723 1045 TAD ACH /IS FAC NEGATIVE
06724 7700 SMA CLA
06725 5331 JMP LEV /NO-GO ON
06726 4543 JMS I [FFNEG /YES-NEGATE IT
06727 2333 ISZ MSIGN /BUMP SIGN CHECK
06730 7000 NOP /MAY SKIP
06731 1055 LEV, TAD OPL /EXIT WITH OPERAND EXPONENT IN AC
06732 5704 JMP I MDSET
06733 0000 MSIGN, 0

4-24
/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 81

/24 BIT BY 12 BIT MULTIPLY, MULTIPLIER IS IN OPL
/MULTIPlicAND IS IN ACH AND ACL
/RESULT LEFT IN MDSET,AC2, AND AC1

06734  0000  MP24,  0
06735  1373  TAD    <-14 /SET UP 12 BIT COUNTER
06736  3055  DCA    OPX
06737  1057  TAD    OPL /IS MULTIPLIER=0?
06740  7440  SZA
06741  5345  JMP    MPLP1 /NO-GO ON
06742  3053  DCA    AC1 /YES-INSURE RESULT=0
06743  5784  JMP    MP24 /RETURN
06744  1057  MPLP,  TAD    OPL /SHIFT A BIT OUT OF LOW ORDER
06745  7010  MPLP1,  RAR /OF MULTIPLIER AND INTO LINK
06746  3057  DCA    OPL
06747  7420  SNL    /WAS IT A 1?
06750  5356  JMP    MPLP2 /NO - 0 - JUST SHIFT PARTIAL PROD
06751  1054  TAD    AC2 /YES-ADD MULTIPLICAND TO PARTIAL
06752  1046  TAD    ACL /LOW ORDER
06753  3054  DCA    AC2
06754  7024  CML    RAL    /*K* NOTE THE "SNL" 5 WORDS BACK!
06755  1045  TAD    AC2    /HI ORDER
06756  1304  MPLP2,  TAD    MDSET
06757  7010  RAR    /NOW SHIFT PARTIAL PROD. RIGHT 1
06760  3304  DCA    MDSET
06761  1054  TAD    AC2
06762  7010  RAR
06763  3054  DCA    AC2
06764  1053  TAD    AC1
06765  7010  RAR    /OVERFLOW TO AC1
06766  3053  DCA    AC1
06767  2055  ISZ    OPX    /DONE ALL 12 MULTIPLIER BITS?
06770  5344  JMP    MPLP /NO-GO ON
06771  5784  JMP    MP24 /YES-RETURN
06773  7764
06774  7203
06775  7110
06776  6514
06777  6460
    7000    PAGE

/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 82

/DIVIDE-BY-ZERO ROUTINE - MUST BE AT BEGINNING OF PAGE

07000  2035  DBAD,  ISZ    FATAL /DIVIDE BY 0 NON-FATAL
07001  4344  JMS    I    ERR    /GIVE ERROR MSG
07002  1220  TAD    DBAD
07003  3044  DCA    ACX    /RETURN A VERY LARGE POSITIVE NUM
07004  7332  AC2000
07005  5325  JMP    FD

4-25
/FLOATING DIVIDE - USES DIVIDE-AND-CORRECT METHOD

07006  4777  DDDIV, JMS I  (DARGET
07007  7410  SKP
07010  4778  FFDIV, JMS I  (DARGET  /GET OPERAND
07011  4775  JMS I  (MDSET  /GO SET UP FOR DIVIDE-OPX IN AC
07012  7841  CMA  IAC  /NEGATE EXP. OF OPERAND
07013  1844  TAD  ACK  /ADD EXP OF FAC
07014  3244  DCA  ACH  /STORE AS FINAL EXPONENT
07015  1056  TAD  OPH  /NEGATE HI ORDER OP. FOR USE
07016  7141  CLL  CMA  IAC  /AS DIVISOR
07017  3056  DCA  OPH
07020  4231  JMS  DV24  /CALL DIV.--(ACH+ACL)/OPH
07021  1046  TAD  ACL  /SAVE QUOT. FOR LATER
07022  3053  DCA  AC1
07023  1057  TAD  OPL
07024  7650  SNA  CLA
07025  5327  JMP  DVL2  /AVOID MULTIPLYING BY 0
07026  1374  TAD  (-15  /SET COUNTER FOR 12 BIT MULTIPLY
07027  3231  DCA  DV24  /TO MULTIPLY QUOT. OF DIV. BY
07030  5267  JMP  DVL1  /LOW ORDER OF OPERAND (OPL)

/DIVIDE ROUTINE - (ACH,ACL)/OPH = ACL REMAINDER REM

07031  0000  DV24,  0
07032  1045  TAD  ACH  /CHECK THAT DIVISOR IS .GT.
07033  1056  TAD  OPH  /DIVISOR IN OPH (NEGATIVE)
07034  7630  SCL  CLA  /IS IT?
07035  5200  JMP  DBAD  /NO-DIVIDE OVERFLOW
07036  1374  TAD  (-15  /YES-SET UP 12 BIT LOOP
07037  3054  DCA  AC2
07040  5251  JMP  DV1  /GO BEGIN DIVIDE
07041  1045  DV2,  TAD  ACH  /CONTINUE SHIFT OF FAC LEFT
07042  7004  RAL
07043  3045  DCA  ACH  /RESTORE HI ORDER
07044  1045  TAD  ACH  /NOW SUBTRACT DIVISOR FROM HI ORDER
07045  1056  TAD  OPH  /DIVIDEND
07046  7430  SCL  /GOOD SUBTRACT?
07047  3045  DCA  ACH  /YES-RESTORE HI DIVIDEND
07050  7200  CLA  /NO-DON'T RESTORE--OPH,GT,ACH
07051  1846  DV1,  TAD  ACL  /SHIFT FAC LEFT 1 BIT-ALSO SHIFT
07052  7004  RAL  /1 BIT OF QUOT. INTO LOW ORD OF ACL
07053  3046  DCA  ACL
07054  2054  ISZ  AC2  /DONE 12 BITS OF QUOT?
07055  5241  JMP  DV2  /NO-GO ON
07056  5631  JMP  I  DV24  /YES-RETN W/AC2=0

/FORTRAN 4 RUNTIME SYSTEM - R.L  PAL8-V8  PAGE 83

/DIVIDE ROUTINE CONTINUED

07057  3057  MP12L,  DCA  OPL  /STORE BACK MULTIPLICAT
07058  1054  TAD  AC2  /GET PRODUCT SO FAR
07059  7420  SNL  /WAS MULTIPLIER BIT A 1?
07062  5265  JMP  +3  /NO-JUST SHIFT THE PARTIAL PRODUCT
07063  7190  CLL  /YES-CLEAR LINK AND ADD MULTIPLICATOR
07064  1046  TAD  ACL  /TO PARTIAL PRODUCT
07065  7010  RAR  /SHIFT PARTIAL PRODUCT-THIS IS HI
07066  3054  DCA  AC2  /RESULT-STORE BACK
07067  1057  DVL1,  TAD  OPL  /SHIFT A BIT OUT OF MULTIPLIER
07070  7010  RAR  /AND A BIT OR RESLT. INTO IT (LO
07071  2231  ISZ  DV24  /DONE ALL BITS?
07072 5257 JMP MP12L /NO-LOOP BACK
07073 7141 CLL CIA /YES-LOW ORDER PROD. OF QUOT. X
07074 3046 DCA ACL /NEGATE AND STORE
07075 7024 CML RAL /PROPAGATE CARRY
07076 1054 TAD AC2 /NEGATE HI ORDER PRODUCT
07077 7161 SIL CIA
07100 1045 TAD ACH /COMPARE WITH REMAINDER OF FIRST
07101 7430 SZL /WELL?
07102 5331 JMP DVOPS /GREATER THAN REM.-ADJUST QUOT OF
07103 3045 DCA ACH /OK-DO (REM- Q*OPL)/ OPH
07104 4231 DVL3, JMS DV24 /DIVIDE BY OPH (HI ORDER OPERAND)
07105 1053 DVL1, TAD AC1 /GET QUOT. OF FIRST DIV.
07106 7500 SMA /IF HI ORDER BIT SET-MUST SHIFT 1
07107 5325 JMP FD /NO-ITS NORMALIZED-DONE
07110 7100 SHR1, CLL
07111 2046 ISZ ACL /ROUND AND SHIFT RIGHT ONE
07112 7410 SKP
07113 7001 IAC /DOUBLE PRECISION INCREMENT
07114 7010 RAR
07115 3045 DCA ACH /STORE IN FAC
07116 1046 TAD ACL /SHIFT LOW ORDER RIGHT
07117 7010 RAR
07120 5306 DCA ACL /STORE BACK
07121 2044 ISZ ACX /BUMP EXPONENT
07122 7000 NOP
07123 1045 TAD ACH
07124 5306 JMP DVL1+1 /IF FRACT WAS 77777777 MUST
07125 3045 FD, DCA ACH /STORE HIGH ORDER RESULT
07126 5775 JMP I (MDONE /GO LEAVE DIVIDE
07127 3046 DVL2, DCA ACL /COME HERE IF LOW-ORDER QUO=0
07130 5304 JMP DVL3 /SAVE SOME TIME

/FORTRAN 4 RUNTIME SYSTEM - R.L PAL8-V8 PAGE 84

/Routine to Adjust Quotient of First Divide (Maybe) When
/Remainder of the First Divide is Less Than Quot*OPL

07131 7041 DVOPS, CMA IAC /NEGATE AND STORE REVISED REMAINDER
07132 3045 DCA ACH
07133 7100 CLL
07134 1045 TAD OPH
07135 1045 TAD ACH /WATCH FOR OVERFLOW
07136 7420 SNL
07137 5344 JMP DVOP1 /OVERFLOW-DON'T ADJUST QUOT. OF 1
07140 3045 DCA ACH /NO OVERFLOW-STORE NEW REM.
07141 7040 CMA ACH /SUBTRACT 1 FROM QUOT OF
07142 1053 TAD AC1 /FIRST DIVIDE
07143 3045 DCA AC1
07144 7040 DVL1, CMA ACL /GET HI ORD OF REMAINDER
07145 1045 TAD ACH /IS IT ZERO?
07146 7450 SNA
07147 3046 DVOP2, DCA ACL /YES-MAKE WHOLE THING ZERO
07150 3045 DCA ACH
07151 4231 JMS DV24 /DIVIDE EXTENDED REM. BY HI DIVISOR
07152 1046 TAD ACL /NEGATE THE RESULT /
07153 7141 CLL CMA IAC
07154 3045 DCA ACL
07155 7420 SNA /IF QUOT. IS NON-ZERO, SUBTRACT
07156 7040 CMA /ONE FROM HIGH ORDER QUOT.
07157 5304 JMP DVL1 /GO TO IT
/FORTRAN 4 RUNTIME SYSTEM - R.L PALB-V8  PAGE 85

"NRMFAC" AND "OPNEG" MUST BE AT 0 AND 3 ON PAGE

07200 3053 NRMFAC, DCA AC1 /KILL OVERFLOW BIT
07201 4271 JMS FFNOR
07202 5476 JMP I FPNXT

07203 0000 OPNEG, 0 /ROUTINE TO NEGATE OPERAND
07204 1057 TAD OPL /GET LOW ORDER
07205 7141 CML CMA IAC /NEGATE AND STORE BACK
07206 3057 DCA OPL
07207 7024 CML RAL /PROPAGATE CARRY
07210 1056 TAD OPH /GET HI ORDER
07211 7141 CLL CMA IAC /NEGATE AND STORE BACK
07212 3056 DCA OPH
07213 5603 JMP I OPNEG

/ FLOATING SUBTRACT AND ADD /

07214 4777 FFSUB, JMS I (TARGET /PICK UP THE OP.
07215 4203 JMS OPNEG /NEGATE OPERAND
07216 7140 SKP
07217 4777 FFADD, JMS I (TARGET /PICK UP OPERAND
07220 1056 TAD OPH /IS OPERAND = 0
07221 7650 SNA CLA
07222 5476 JMP I FPNXT /YES-DONE
07223 1045 TAD ACH /NO-IS FAC=0?
07224 7650 SNA CLA
07225 5236 JMP DOADD /YES-DO ADD
07226 1044 TAD ACX /NO-DO EXPONENT CALCULATION
07227 7141 CLL CMA IAC
07230 1055 TAD OPX
07231 7540 SMA SZA /WHICH EXP. GREATER?
07232 5243 JMP FACR /OPEHANDS-SHIFT FAC
07233 7041 CMA IAC /FAC'S-SHIFT OPERAND=DIFFRNCE+1
07234 4246 JMS OPSR
07235 4541 JMS I IACSR /SHIFT FAC ONE PLACE RIGHT
07236 1055 DOADD, TAD OPX /SET EXPONENT OF RESULT
07237 3044 DCA ACX
07240 4537 JMS I IOADD /DO THE ADDITION
07241 4271 JMS FFNOR /NORMALIZE RESULT
07242 5476 JMP I FPNXT /RETURN
07243 4541 FACR, JMS I IACSR /SHIFT FAC = DIFF.+1
07244 4246 JMS OPSR /SHIFT OPR. 1 PLACE
07245 5236 JMP DOADD /DO ADDITION

4-28
07246  0000  OPSR, 0
07247  7040  CMA   /* (COUNT+1) TO SHIFT COUNTER
07250  3052  DCA   AC0
07251  1056  LOP2, TAD   OPH  /*GET SIGN BIT
07252  7100  CLL   /* TO LINK
07253  7510  SPA
07254  7020  CML   /*WITH HI MANTISSA IN AC
07255  7010  RAR   /*SHIFT IT RIGHT, PROPAGATING SIGN
07256  3056  DCA   OPH  /*STORE BACK
07257  1057  TAD   OPL
07260  7010  RAR
07261  3057  DCA   OPL  /*STORE LO ORDER BACK
07262  2055  ISZ   OPX  /*INCREMENT EXPONENT
07263  7000  NOP
07264  2052  ISZ   AC0  /*DONE ALL SHIFTS?
07265  5251  JMP   LOP2  /*NO-LOOP
07266  7010  RAR   /*SAVE 1 BIT OF OVERFLOW
07267  3054  DCA   AC2  /*IN AC2
07270  5646  JMP I  OPSR  /*YES-RET.

07271  0000  FFNOR, 0  /*ROUTINE TO NORMALIZE THE FAC
07272  1045  TAD   ACH  /*GET THE HI ORDER MANTISSA
07273  7450  SNA   /*ZERO?
07274  1046  TAD   ACL  /*YES-HOW ABOUT LOW?
07275  7450  SNA
07276  1053  TAD   AC1  /*LOW=0, IS OVRFLO BIT ON?
07277  7650  SNA   CLA
07280  5313  JMP   ZEXP  /*#=0-ZERO EXPONENT
07301  7332  NORMLP, CLA  CLL  CML  RTR  /*NOT 0-MAKE A 2000 IN AC
07302  1045  TAD   ACH  /*ADD HI ORDER MANTISSA
07303  7440  SZA   /*HI ORDER = 6000
07304  5307  JMP   +3  /*NO-CHECK LEFT MOST DIGIT
07305  1046  TAD   ACL  /*YES-6000 OK IF LOW=0
07306  7640  SZA   CLA
07307  7710  SPA   CLA  /2,3,4,5,ARE LEGAL LEFT MOST DIGS.
07310  5314  JMP   FFNORR  /*FOR NORMALIZED #-(+2000=4,5,6,7)
07311  4034  JMS I  IAL1BMP  /*SHIFT AC LEFT AND BUMP ACX DOWN
07312  5301  JMP   NORMLP  /*GO BACK AND SEE IF NORMALIZED
07313  3044  ZEXP, DCA   ACX
07314  3053  FFNORR, DCA   AC1  /*DONE W/NORMALIZE - CLEAR AC1
07315  5671  JMP I  FFNOR  /*RETURN

07316  0000  LPBUF4, ZBLOCK 60
07317  7400  LPBUF
07317  6514  7400  PAGE
CHAPTER 5

LIBRA AND FORLIB

The binary output of an assembly under RALF is called a RALF module. Every RALF module consists of an External Symbol Dictionary (or ESD) and associated text. The ESD lists all global symbols defined in the assembly, while the text contains the actual binary output along with relocation data.

There are three major classes of global symbols. Entry points are global symbols defined in a module and referenced by code in other modules. Thus, entry points include the names of all modules and the names of all globally callable subroutines within modules. Externs are global symbols that are referenced in a module but not defined in that module. For example, the entry point of module A would appear as an extern if referenced in module B. The COMMON area comprises a third class of global symbols including all global symbols which define COMMON.

A FORTRAN IV library is a specially formatted file, created with LIBRA, consisting of a library catalog (which lists section names and entry points of library modules) and a set of RALF modules, perhaps interspersed with empty subfiles. The loader uses one such library, specified by the user, to resolve externs while building a loader image file. The general structure of a FORTRAN IV library is:

<table>
<thead>
<tr>
<th>CATALOG</th>
<th>MODULE</th>
<th>FREE AREA</th>
<th>MODULE</th>
<th>MODULE</th>
<th>etc.</th>
</tr>
</thead>
</table>

5-1
LIBRA is a very simple program, basically a file-to-file copy inside several nested loops. The outer loop begins at START, and calls the command decoder for specification of the library and input files. If no library is specified, the previous library name is used (initially this is SYS:FORLIB. RL). If a new name is given, but no extension is specified, .RL is forced. A check is made to verify that the specified library is on a file-structured device, and the handler is FETCHed.

At ZTEST, the /Z switch is tested. If it was set, control passes to NEWLIB to create a new library. Otherwise, an attempt is made to find an old library of the specified name on the device. If it fails, control passes to NEWLIB. Otherwise, the catalog of the old library is read and scanned to determine the starting block of available space. This is stored at LAVAIL. Control then passes to GETINF to begin reading input files.

If /Z was set, or the specified library isn't found, a new library is entered at NEWLIB, and an empty catalog is written. Control passes to GETINF. There, a check is made to determine whether input is presently coming from another library. If it is, control passes to INLIB to obtain the next module from the library. Otherwise, the next input file is obtained from the command decoder area in field 1, and if one exists, control passes to FCHIN to load the handler. If there is none, the /C switch is tested. If it is not set, control is passed to LCLOSE to close the library. If it is set, however, the command decoder is recalled to obtain a continuation of the preceding input line, and control returns to NXTINF to look in the command decoder area.
At FETCHIN, the unit, starting block, and length of the next input file are obtained from the command decoder area, the appropriate device handler is fetched, and at LUKMOD, the input file is read to ensure that it is either a module or a library. If a library, control passes to GOTLIB, which sets INLSW and goes to INLIB to obtain the first module from the library. Otherwise, the length is checked against the available length in the library, to ensure that this module can be fit in, and control goes to NXTEBK to read the ESD.

At INLIB, the catalog of the library being input is read, and scanned until a module is found with a starting block greater than the starting block of the last input module (in the case of the first module in a library, MODBLK, which normally contains the starting block of a module, contains the starting block of the library, so this scan yields the starting block of the first module in the library). When the next module has been found, control returns to LUKMOD to check the length of the module against the available length in the library.

At NXTEBK, the end of the input module is scanned for entry point and section names. Whenever one is found, the catalog of the output library is scanned for a matching name. If a match is found, control passes to GOTMAT, which prints the duplicated name, and if the /I switch is set, asks the operator which name to keep. If he types N, for new, control passes to DLETO to delete the old name. Otherwise, control is passed to ESDLND to find the next entry point or section name in the input. If /I is not set, /R is tested. If it is not set, control is passed to ESDLND. If it is, control flows into DELTO, where the old name is cleared, and the rest of the catalog is scanned to find the first available name slot. Control then passes to INSERT.
If no match was found, the /I switch is tested. If it was set, the operator is asked whether to include the name. If he types, N, for no, control is passed to ESDLND. Otherwise, or if /I was not set, a pointer is set up for the new name, and control passes to INSERT, where the new name is added to the catalog.

When the entire ESD has been scanned, INCLUD is tested to determine whether any name has been included in the catalog, and assuming at least one has, the module is copied into the library, and LAVAIL is updated to indicate the next available block in the library. Control returns to GETINF for another module.

LCLOSE receives control whenever the end of the input file string is reached and /C is not set. Here, any remaining changes in the library catalog are written, and if a new library was entered, it is closed. Control passes to CATLST, to create a catalog listing. The second output file, if any was specified, is opened, a title is output to it, and at PRCAT, the entire contents of the catalog are listed. When this process is complete, the output file is closed, and control returns to start for more command decoder input.
User-coded modules may be added to the system library or incorporated in a new library provided that entry points, variable storage allocations, calling sequences, error conditions and the like are handled with care.

Every library module must have a unique section (and entry) name(s). The library supplied by DEC uses the character # before names where duplication in the FORTRAN program may be possible. Note that this character is acceptable to RALF, but is illegal in a FORTRAN source. If more than one entry is required to the routine, they should be listed as such using the pseudo-op ENTRY before they are encountered as tags in the code. Thus, if a double precision tangent routine is being written, it may be helpful to have an entry for a double precision co-tangent calculation also. Appropriate code would be:

```
SECT DTAN
JA #DTAN
ENTRY DCOT
JA #DCOT
:
#DCOT,
:
#DTAN,
```

When routines will handle double precision or complex values, allocate six words for their storage. Such routines can switch between the STARTF (3 word format) and STARTE (6 word format) pseudo-ops as required, being careful to define variables of the proper length to keep track of temporary locations.
All user-written library routines are called by a JSR in STARTF mode. Depending on the type of function, the routine must be coded to exit as follows in order to return the result to the program:

Single precision (integer, real and logical)

- FLDA ANSWER /In STARTF mode
- JA RETURN /3 word result

Double precision:

- FLDA ANSWER /In STARTE mode
- JA RETURN /6 word result

Complex:

- EXTERN #CAC /Real part in first 3 words
- STARTE /Imaginary in last 3 words
- FLDA ANSWER /Exit in STARTE mode
- FSTA #CAC /6 word result
- JA RETURN

Routines should conform to the FPP FORTRAN calling sequence. An example of that sequence follows:

- SECT DTAN /Sector name
- JA #DTAN /Jump to Start of Function
- TEXT +DTAN + /6 characters for trace
- /back feature must be
- /immediately before index
- /register assignment.

- DTANXR, SETX XRDTAN /This tag referenced when
- /returning to reset base
- /page and index registers
- /If this routine is called.

- SETB BPDPTAN

- BPDPTAN, F ø ø /3 words each
- XRDTAN, F ø ø /These locations may be
- /used for temporary storage or
- ORG 10*3+BPDPTAN /If this routine is called,
- /will set up return to it.

- FNOP
- JA DTANXR ø

- DTNRTN, JA . /Return to calling program
- #DTAN, BASE ø /Still on caller's base page
- STARTD /Start of subroutine
- FLD 10*3 /Get jump to caller's return jump
- FSTA DTNRTN /Save for return from this routine
FLDA Ø
/Get next location in caller's
/routine (pointer to argument list)
SETX XRDTAN
/Change index registers to this
/routine's
SETB BPDTAN
/Change base page to this routine's
BASE BPDTAN
/Change base page to this routine's
FSTA TEMP
/Save pointer
LDX 1,1
/Save it
FLDA% TEMP,1
/Get address of argument list
FSTA TEMP
/A double precision routine
STARTE
FSTA TEMP
/Save variable
FLDA% TEMP
/Get variable
FSTA TEMP
/Save variable
/
/Calculate result
.
.
.
.
.
FLDA ANSWER
/Load answer
JA DTNRTN
/Exit

The following conventions must be observed to return to the calling
program at the correct location, to permit the error trace back
feature to function properly, and to preserve index registers and
base page integrity.

Locations Ø and 3Ø of the called (user-coded) program are determined
by a statement in the form ORG 10*3+BPAGE which must be followed by
a two-word jump to the index register and base page assignment in-
suctions JA BPXR. In the above example, the code is:

ORG 10*3+BPDATN
FNOP
JA DTANXR

By saving the contents of location 30 of the calling program (FLDA
10*3,FSTA RETURN) for the return exit, the called program executes
(when control is returned to it) a JA BPXR to its base page and index
register assignment statement. In the calling program this resets
the index registers and base page and then returns to execute the
instruction in the calling program. In the tangent example above,
the code is:

5-7
FLDA 10*3
FSTA DTNRTN

which creates the instruction

JA xxx

at the tag DTNRTN, where xxx is the location in the calling routine whose function corresponds to DTANXR in DTAN.

When called, the routine must assign its own base page and index registers (SETX XROWN, SETB BROWN). If arguments are to be passed to the called routine, a scheme such as illustrated above permits any number of arguments to be passed from the calling program and saved on the base page of the called program, in this case just two arguments.

The corresponding code for the calling program (as created by the compiler) is:

EXTERN DTAN
JSR DTAN
JA .+4  /Jump past all arguments
JA A    /Argument
:
FSTA Q  /Save result in some variable

The FORTRAN for such code is:

Q = DTAN (A)

The calling sequence is also discussed in Chapter 2.

To permit the error trace back feature to function properly, a TEXT statement followed by a six alphanumeric character name is required immediately before the index register and base page assignment statements. Thus, if the cotangent routine includes a JSR TAN and an
unacceptable argument is passed to the tangent function, the trace
back indicates the location of the problem by a sequence such as:

    DIV0 MAIN
    ARGUMENT
    7777 SIN
    0000 TAN
    0000 COT
    0007 MAIN

(Line numbers are not relevant in RALF modules such as TAN and SIN:
they are meaningful only in FORTRAN source programs.)

A new library routine may call other new or existing library routines
as part of its function, as well as the error handling function of
the run-time system. To invoke the error message program, code such
as the following is required:

    EXTERN #ARGER
    MERROR, TRAP4 #ARGER

Then any condition encountered in the program that is an error should
jump to MERROR. For example, if an argument of ≤Ø is illegal, it
could be examined and handled as follows:

    FLDA# ARG2
    JLE MERROR /≤Ø error
    FSTA NEXT / Save non-zero value

In this case, the TRAP4 #ARGER at MERROR will produce the message
BAD ARG DTAN nnnn followed by traceback and program termination.
If a new library routine would like to use an existing library routine,
a JSR to that routine is required. The sequence for passing arguments
is:

    EXTERN ATAN2
    JSR ATAN2
    JA .+6 /Execute upon exit from
    JA A /1st arg
    JA B /2nd arg
    FSTA ANSWER /Save answer
The arguments must be referenced in the order expected by the called routine and must agree in number and type. The following routines can be used in this manner:

<table>
<thead>
<tr>
<th>ROUTINE</th>
<th>ARGUMENTS PASSED</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMOD</td>
<td>Address of X then Y</td>
</tr>
<tr>
<td>SQRT</td>
<td>Address of X</td>
</tr>
<tr>
<td>ALOG10</td>
<td>Address of X</td>
</tr>
<tr>
<td>EXP</td>
<td>Address of X</td>
</tr>
<tr>
<td>SIN</td>
<td>Address of X</td>
</tr>
<tr>
<td>COS</td>
<td>Address of X</td>
</tr>
<tr>
<td>TAN</td>
<td>Address of X</td>
</tr>
<tr>
<td>SIND</td>
<td>Address of X</td>
</tr>
<tr>
<td>COSD</td>
<td>Address of X</td>
</tr>
<tr>
<td>TAND</td>
<td>Address of X</td>
</tr>
<tr>
<td>ASIN</td>
<td>Address of X</td>
</tr>
<tr>
<td>ACOS</td>
<td>Address of X</td>
</tr>
<tr>
<td>ATAN</td>
<td>Address of X</td>
</tr>
<tr>
<td>ATAN2</td>
<td>Address of X then Y</td>
</tr>
<tr>
<td>SINH</td>
<td>Address of X</td>
</tr>
<tr>
<td>COSH</td>
<td>Address of X</td>
</tr>
<tr>
<td>TANH</td>
<td>Address of X</td>
</tr>
<tr>
<td>DMD</td>
<td>Address of X then Y</td>
</tr>
<tr>
<td>DSIGN</td>
<td>Address of X then Y</td>
</tr>
<tr>
<td>DSIN</td>
<td>Address of X</td>
</tr>
<tr>
<td>DLOG</td>
<td>Address of X</td>
</tr>
<tr>
<td>DSQRT</td>
<td>Address of X</td>
</tr>
<tr>
<td>DCOS</td>
<td>Address of X</td>
</tr>
<tr>
<td>DLOG10</td>
<td>Address of X</td>
</tr>
<tr>
<td>DATAN2</td>
<td>Address of X then Y</td>
</tr>
<tr>
<td>DATAN</td>
<td>Address of X</td>
</tr>
<tr>
<td>DEXP</td>
<td>Address of X</td>
</tr>
<tr>
<td>CMPLX</td>
<td>Address of X</td>
</tr>
<tr>
<td>CSIN</td>
<td>Address of X</td>
</tr>
<tr>
<td>CCOS</td>
<td>Address of X</td>
</tr>
<tr>
<td>REAL</td>
<td>Address of X</td>
</tr>
<tr>
<td>AIMAG</td>
<td>Address of X</td>
</tr>
<tr>
<td>CONJG</td>
<td>Address of X</td>
</tr>
<tr>
<td>CEXP</td>
<td>Address of X</td>
</tr>
<tr>
<td>CLOG</td>
<td>Address of X</td>
</tr>
<tr>
<td>CABS</td>
<td>Address of X</td>
</tr>
<tr>
<td>CSQRT</td>
<td>Address of X</td>
</tr>
</tbody>
</table>

For real and double precision routines, the result is returned via the FAC (3 or 6 words, respectively). For complex routines, the result is returned in #CAC (6 words).
The TAN function from FORLIB is included here as an example of the requirements just discussed. The TAN function calls two external functions, has the standard calling sequence, and contains an error condition exit.

/ TAN /
/ SUBROUTINE TAN(X) /
SECT TAN /SECTION NAME
JA #TAN /JUMP AROUND BASE PAGE
EXTERN #ANGER /EXIT TO ERROR MESSAGE HANDLER
TRAP #ANGER
TEXT +TAN +
/ FOR ERROR TRACE BACK
TANXR, SETX XRIAN /START OF FORMAL CALLING SEQUENCE
SEIB BPTAN
BTAN, FNOP /START OF BASE PAGE
0 0
XRIAN, F 0.0 /INDEX REGISTERS
TAN1, F 0.0 /LOCATIONS 21-42 OCTAL AVAILABLE
TAN2, F 0.0 /FOR USER STORAGE
ORG 10+3+BPTAN /SET UP FOR A RETURN
/TO THIS ROUTINE
FNOP
JA TANXR /JUMP TO XR + RP ASSIGNMENT
0
TANRIN, JA .
BASE 0
#TAN, STARTD
FLDA 10*3 /SAVE RETURN JUMP
FLSTA TANRIN
FLDA 0 /GET NEXT LOCATION
/IN CALLING PROGRAM
SEIT XRIAN /SET UP FOR TAN'S INDEX REGS
SEIB BPTAN /SET UP FOR TAN'S BP
BASE BPTAN
LDX 1,1
FLSTA BPTAN
FLDAZ BPTAN,1 /GET ADDRESS OF X
FLSTA BPTAN
STARTF
FLDAZ BPTAN /GET X
JEQ TANRIN /IF 0 RETURN NOW
FLSTA TAN1 /SAVE FOR A SECOND
EXTERN COS
JSR COS /TAKE COS(X)
JA .+4 /JUMP AROUND ARGUMENT LIST
JA TAN1 /REFERENCE TO PASSED ARGUMENT
JEQ TANER /COS=0. A NO-NO
FLSTA TAN2 /SAVE IT
EXTERN SIN
JSR SIN /NOW TAKE SJN(X)
JA .+4 /JUMP AROUND ARGUMENT LIST
JA TAN1 /REFERENCE TO ARGUMENT
FDIV TAN2 /DIV BY COS(X)
JA TANRIN /EXIT

5-11
The library routine ONQI illustrates many of the same conventions.
This listing may also prove valuable as a guide to interfacing with
the run-time system.

FIELDI ONQI /ROUTINE TO ADD A
/HANDLER TO INTERRUPT SKIP CHAIN
/PUT THIS CODE IN FIELD I
Ø
JMP SETINIT /SET UP INT INITIALLY
ISZ ONQI /BUMP ARGUMENT POINTER
ISZ INTQ+1 /BUMP INTERRUPT Q POINTER
DCAZ INTQ+1 /STICK IOT ONTO INT Q
TAD XSKP /FOLLOWED BY A SKIP
ISZ INTQ+1
DCAZ INTQ+1 /ONTO INT Q
ISZ ONQI /SKIP FIRST WORD OF ADDR
ISZ INTQ+1
ONQISW, TADZ ONQI /GET INT HANDLER ADDRESS
ISZ ONQI
DCAZ INTADR+1 /ONTO ADDRESS STACK
TAD INTADR+1 /NOW MAKE JMSZ
AND L177
TAD L4600
DCAZ INTQ+1 /ONTO INT Q
ISZ INTADR+1
ISZ IQSIZE /ROOM FOR MORE?
JMPZ ONQI /YES
TAD -1 /NO, CLOSE OUT THE SUBR
DCA ONQI+1
JMPZ ONQI
SEINIT, TAD ONQISW /DO THIS PART ONLY ONCE
DCA ONQI+1
CDF
TAD XSKP /FIX UP #INT
DCAZ XINT+1 /PUT SKIP INST. FIRST
ISZ XINT+1
TAD INTQ+1
DCAZ XINT+1 /GET ADDR. OF USER'S ROUTINE
ISZ XINT+1 /ADD TO INTERRUPT CALL
TAD CICDF /GET FIELD INSTRUCTION

/FIELDI SECTION INSURES ITS IN FIELD I
DCAZ XINT+1
CICDF, CDF CIF 10
JMP ONQI+1 /BACK TO ONQI
EXTERN #INI
XINT, ADDR #INT /POINTS TO INT RIN IN COMMON
INTQ, ADDR IHANDL /MUST USE 15 BIT ADDRESS
INTADR, ADDR IADRS /

IQSIZE, -5
XSKP, SKP
L177, 177
L4600, 4600
CDF CIF
JMPZ IHANDL
IHANDL, Ø
REPEAT 16
JMP IHANDL-2
IADRS, Ø; Ø; Ø; Ø; Ø; Ø; /CAN SET UP 1-5 DEVICES

S-12
ENTRY ONQB /USE "ENTRY" TO PERMIT
/ACCESS FROM OUTSIDE OF SECTION
/Routine TO SET UP AN IDLE JOB
ONQB, 0
JMP SETBAK /SETUP #IDLE
TADZ ONQB /GET ADDRESS OF IDLE JOB
ONQB3W, ISZ ONQB
DCAZ BAKADR+1 /STORE INTO BACKGROUND JOB Q
TAD BAKADR+1 /MAKE A JMSZ
ISZ BAKADR+1
AND L177
TAD L4600
ISZ BAKQ+1
DCAZ BAKQ+1
ISZ BQSIZE /MORE ROOM?
JMPZ ONQB /YES
TAD .-1 /NO, CLOSE THE DOOR
DCA ONQB+1
JMPZ ONQB
SEITBAK, TAD ONQB3W /CLOSE OFF #IDLE INITIALIZATION
DCA ONQB+1
CDF
TAD XSKP /FIX UP #IDLE
DCAZ XIDLE+1 /ADD SKIP TO IDLE CALL
TAD BAKQ+1 /GET ADDRESS OF ROUTINE
ISZ XIDLE+1
DCAZ XIDLE+1
ISZ XIDLE+1
TAD CIFCDF /GET FIELD INSTR.
DCAZ XIDLE+1
CIF CDF 10
JMP ONQB+1
EXTERN #IDLE /EXTERNAL REFERENCE
XIDLE, ADDR #IDLE

BAKQ, ADDR BAKRND
BAKADR, ADDR BHADRS
BQSIZE, -5
CDF CIF
JMPZ BAKRND
BAKRND, 0
REPEAT 6
JMP BAKRND-2
BHADRS, 0;0;0;0;0;0 /1-5 JOBS
APPENDIX A
RALF Assembler Permanent Symbol Table

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FADD</td>
<td>1000</td>
</tr>
<tr>
<td>FADDM</td>
<td>5000</td>
</tr>
<tr>
<td>FDIV</td>
<td>3000</td>
</tr>
<tr>
<td>FLDA</td>
<td>0000</td>
</tr>
<tr>
<td>FMUL</td>
<td>4000</td>
</tr>
<tr>
<td>FMULM</td>
<td>7000</td>
</tr>
<tr>
<td>FSTA</td>
<td>6000</td>
</tr>
<tr>
<td>FSUB</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>IOT'S</td>
<td></td>
</tr>
<tr>
<td>FPINT</td>
<td>6551</td>
</tr>
<tr>
<td>FPICL</td>
<td>6552</td>
</tr>
<tr>
<td>FPCOM</td>
<td>6553</td>
</tr>
<tr>
<td>FPHLT</td>
<td>6554</td>
</tr>
<tr>
<td>FPST</td>
<td>6555</td>
</tr>
<tr>
<td>FPRST</td>
<td>6556</td>
</tr>
<tr>
<td>FPIST</td>
<td>6557</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>ADDX</td>
<td>0110</td>
</tr>
<tr>
<td>ALN</td>
<td>0010</td>
</tr>
<tr>
<td>ATX</td>
<td>0020</td>
</tr>
<tr>
<td>FCLA</td>
<td>0002</td>
</tr>
<tr>
<td>FEXTT</td>
<td>0</td>
</tr>
<tr>
<td>FNEG</td>
<td>0003</td>
</tr>
<tr>
<td>FNOP</td>
<td>0040</td>
</tr>
<tr>
<td>FPNORM</td>
<td>0044</td>
</tr>
<tr>
<td>FPAUSE</td>
<td>0001</td>
</tr>
<tr>
<td>JA</td>
<td>1030</td>
</tr>
<tr>
<td>JAC</td>
<td>0007</td>
</tr>
<tr>
<td>JAL</td>
<td>1070</td>
</tr>
<tr>
<td>JEQ</td>
<td>1000</td>
</tr>
<tr>
<td>JGE</td>
<td>1010</td>
</tr>
<tr>
<td>JGT</td>
<td>1060</td>
</tr>
<tr>
<td>JLE</td>
<td>1020</td>
</tr>
<tr>
<td>JLT</td>
<td>1050</td>
</tr>
<tr>
<td>JNE</td>
<td>1040</td>
</tr>
<tr>
<td>JSA</td>
<td>1120</td>
</tr>
<tr>
<td>JSR</td>
<td>1130</td>
</tr>
<tr>
<td>JXN</td>
<td>2000</td>
</tr>
</tbody>
</table>
APPENDIX B

ASSEMBLY INSTRUCTIONS

The following sequence of commands may be used to assemble the OS/8 FORTRAN IV system programs. It is assumed that all PAL language sources reside on DSK. In this example, DTAl is shown as the target device, however any other device could be used via the appropriate ASSIGN command. Note that PASS20.SV is produced by conditional assembly of PASS20.PA and that the "O" in PASS20 is an oh, not a zero. The initial dot and asterisk characters on every command line shown are printed by the monitor. All other characters (except carriage return, in some cases) are typed by the user.

Type CTRL/Z after each of the three system pauses at point 1, to continue assembly of PASS20. Type ALT MODE to produce the "$" character.

ASSIGN DTAl DEV
.R PAL8
*.F4.BN,LIST.LS<F4$
.R ABSDLR
*.F4$
.SAVE DEV F4=0;12200$
.R PAL8
*.PASS2.BN,LIST.LS<PASS2$
.R ABSDLR
*.PASS2$
.SAVE DEV PASS2=0;5000$
.R PAL8
*.PASS20.BN,LIST.LS<TTY:,DSK:PASS2$OVERLY=1

.R ABSDLR
.PASS20$
.SAVE DEV PASS20=0;7605$
.R PAL8
*.PASS3.BN,LIST.LS<PASS3$
.R ABSDLR
*.PASS3$
.SAVE DEV PASS3=0;400$
.R PAL8
*.RALF.BN,LIST.LS<RALF$
.R ABSDLR
*.RALF$
.SAVE DEV RALF=0;200$
.R PAL8
*.LOAD.BN,LIST.LS<LOAD$
.R ABSDLR
*.LOAD$
.SAVE DEV LOAD=0;200$
.R PAL8
*.FRTS.BN,LIST.LS<RTS,RTL$
.R ABSDLR
*.FRTS$
.SAVE DEV FRTS=0;200$
.R PAL8
*.LIBRA.BN,LIST.LS<LIBRA$
.R ABSDLR
*.LIBRA$
.SAVE DEV LIBRA=0;200
.
INDEX

Argument passing, 2-10
Arithmetic expression
   analyser, 1-2
Background jobs, 4-1
Binary buffer table, 3-8
Binary section table, 3-10
Block count sequence number, 4-14
COMMON information block, 1-7
Communication, 2-9
COMMS sections, 2-12
Compilation, 1-1
Compiler symbol table, 1-2
Device handlers, 4-16
Define flag handlers, 4-2
Dimension information block, 1-5
DSRN table, 4-8
Entry point, 2-1
EQUIVALENCE information table, 1-7
ESD, 2-1, 2-5, 2-6
ESD correspondence table, 3-8
External symbol, 2-1
External symbol dictionary, 2-1, 2-5, 2-6
FIELD1 sections, 2-13
Files, 4-13
Formatted I/O, 4-13
FRTS
   Calling sequence, 4-4
   Core maps, 4-6
   Entry points, 4-5
   Initialization, 4-14
   Page zero, 4-10
Header block, 3-13
Idle jobs, 4-1
Indirect addressing, 2-7
Interrupts,
   Servicing, 4-1, 4-15
   Spurious, 4-15
Keyword, 1-2
LIBRA, 5-2
Library, 2-1
   Format, 5-1
Line printer handler, 4-14
Literals, 1-4, 1-5
Loader, 3-1
   Core maps, 3-2 to 3-4
   Image file, 3-12
   Subroutines, 3-3
Loader symbol table, 3-1, 3-7
Magic number, 1-6
Mixing code, 2-8
Module, 2-1
Module count table, 3-12
Module descriptor table, 3-11
Off-page references, 2-19
Optimized code, 2-11
Output codes, 1-7
Overlay table, 3-10
/P option, 4-21
Page boundaries, 2-18
PASS1, 1-1
   Output, 1-7
   Subroutines, 1-10
PASS2, 1-12
   Error list, 1-14
   Skeleton tables, 1-14
   Symbol table, 1-14
   Subroutines, 1-15
PASS3, 1-17
   PDP-8 code, 2-7
   Program loading, 3-9
   Program termination, 4-2
   Pseudo-ops, 2-7
RALF, 2-1
   Expressions, 2-4
   Symbol table, 2-5
   RALF output file, 2-6
Section, 2-1
Section types, 2-11
Statement number, 1-4
Subroutine calls, 2-9
Subroutine return sequence, 2-10
Symbol table,
   Compiler, 1-14
   Loader, 3-7
   RALF, 2-5
Termination, program, 4-22
Text, 2-1
TRAP3 and TRAP4, 2-8
Variable type word, 1-3
8-mode sections, 2-14

X-1
Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback—your critical evaluation of this document.

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

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

How can this document be improved?

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

How does this document compare with other technical documents you have read?

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

Job Title____________________________________Date:_____________________

Name:____________________________________Organization:_____________

Street:____________________________________Department:_______________

City:______________________State:__________Zip or Country__________