The JOSHUA(J80) System Programmer's Manual (U)

Scientific Computations Section

August 1993

9-10-93
DERIVATIVE CLASSIFIER & REVIEWING OFFICIAL

UNCLASSIFIED

DOES NOT CONTAIN
UNCLASSIFIED CONTROLLED
NUCLEAR INFORMATION

ADD &
REVIEWING
ORIGIN:

DATE: 9-10-93

Savannah River Technology Center
Westinghouse Savannah River Company
Savannah River Site
Aiken, SC 29808

MASTER

SAVANNAH RIVER SITE

PREPARED FOR THE U.S. DEPARTMENT OF ENERGY UNDER CONTRACT NO. DE-AC09-88SR18035

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

DISCLAIMER

This report was prepared by Westinghouse Savannah River Company (WSRC) for the United States Department of Energy under Contract DE-AC09-88SR18035 and is an account of work performed under that Contract. Neither the United States, the United States Department of Energy nor WSRC, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed herein, or represents that its use will not infringe privately owned rights. Reference herein to any specific commercial product, process or service by trade name, mark, manufacturer, or otherwise does not necessarily constitute or imply endorsement, recommendation, or favoring of same by WSRC or by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

Acknowledgments

The development of the J80 system has spanned a period of several years, survived numerous organizational transitions, several managers and a rapidly changing computer environment. The following people are responsible:

JOSHUA System Programming.....	B. W. Westmoreland A. O. Smetana J. T. McCort
JOSHUA Precompiler	H. C. Honeck
Technical & Design Assistance	J. E. Suich R. N. Sims
Project Management - The early years	W. H. Reed R. R. Beckmeyer R. J. Pryor
Project Management - The later years.....	J. P. Church R. R. Beckmeyer M. R. Buckner

TABLE OF CONTENTS

1	INTRODUCTION	1-1
2	USING JS ROUTINES FOR JOSHUA OPERATIONS	2-1
2.1	INITIALIZING A MODULE	2-1
2.2	EXECUTING A MODULE	2-2
2.3	TERMINATING A MODULE	2-3
2.4	SETTING AND CHANGING THE READ HIERARCHY	2-3
2.5	READING A RECORD	2-4
2.6	WRITING A RECORD	2-7
2.7	PRINTING A RECORD OR SUBTREE OF RECORDS	2-8
2.8	COPYING A RECORD OR SUBTREE OF RECORDS	2-8
2.9	DELETING A RECORD OR SUBTREE	2-9
2.10	DELETING THE USER SCRATCH DATA SET	2-9
2.11	COMPRESSING A JOSHUA DATA SET	2-9
2.12	RENAMING A QUALIFIER	2-9
2.13	LISTING THE RECORDS IN A SUBTREE	2-10
2.14	DISPLAYING THE LIST OF RECORDS IN A SUBTREE	2-11
2.15	LISTING THE CHILDREN OF A SUBTREE	2-12
2.16	DISPLAYING THE LIST OF CHILDREN OF A SUBTREE	2-13
2.17	SETTING THE WRITE DATA SET	2-13
2.18	SETTING THE NONSTANDARD PATH	2-14
2.19	SETTING THE LOCATION OF A NONSTANDARD DATA SET	2-15
2.20	PLACING A COMMENT IN A DATA SET	2-16
2.21	GETTING DATA SET INFORMATION	2-16
2.22	DISPLAYING DATA SET INFORMATION	2-16
2.23	CREATING ERROR, WARNING, AND MESSAGE DISPLAYS	2-17
2.24	CONVERTING A LIST OF NAMES	2-19
3	DESCRIPTION OF THE JOSHUA DATA BASE	3-1
3.1	JOSHUA RECORD AND SUBTREE NAMES	3-1
3.2	JOSHUA DATA SET FILES	3-1
3.2.1	TREE FILES	3-2
3.2.1.1	Tree Header Records	3-2
3.2.1.2	Tree Node Records	3-2
3.2.2	DATA FILES	3-3
3.2.3	NAMING AND STORING DATA SET FILES	3-5
3.2.3.1	Nonstandard Data Set Files	3-5
3.2.3.2	Standard Data Set Files	3-6
3.3	CREATION OF JOSHUA DATA SETS	3-6
3.3.1	VAX/VMS	3-6
3.3.2	IBM/MVS	3-6
3.3.3	SIZE LIMITATIONS	3-7
3.4	THE JOSHUA DATA SET TABLE	3-7
3.4.1	CONTENTS OF THE DATA SET TABLE	3-7
3.4.2	STORAGE AND USE OF DATA SET TABLE	3-8
3.5	SPECIAL DATA SETS	3-8
3.6	THE READ HIERARCHY	3-8
3.6.1	MANAGEMENT OF READ HIERARCHY INFORMATION	3-9

4	JOSHUA DATA BASE OPERATIONS	4-1
4.1	SEARCHING THE TREE CATALOGS	4-1
4.1.1	SETTING UP A NAME STRING	4-1
4.1.1.1	JOSHUA name table	4-2
4.1.2	CHOOSING DATA SETS TO BE SEARCHED	4-2
4.1.3	FINDING A MATCH FOR A NAMESTRING	4-3
4.1.3.1	tree and path TABLES	4-4
4.2	CONTINUING TO TRAVERSE THE CATALOG	4-5
4.2.1	LISTING THE RECORDS IN A SUBTREE	4-5
4.2.2	LISTING THE CHILDREN	4-5
4.3	REVISING THE CATALOG	4-6
4.3.1	ADDING NAMES	4-6
4.3.2	DELETING	4-6
4.3.3	RENAMING	4-7
4.4	READING FROM AND WRITING TO DATA FILES	4-7
4.4.1	COPYING	4-8
4.4.2	PRINTING	4-8
5	THE MONITOR PROGRAM, JSMON	5-1
5.1	INVOKING JSMON	5-1
5.2	SPECIFYING JOB PARAMETERS	5-1
5.2.1	THE JSIN FILE	5-1
5.2.2	THE JSIN TABLE	5-1
5.2.2.1	INITIALIZING THE JSIN table	5-2
5.2.2.2	PRINTING the JSIN table	5-2
5.2.2.3	USING the jsin table	5-2
5.3	CREATING SCRATCH DATA SETS	5-2
5.3.1	THE SYSTEM SCRATCH DATA SET	5-2
5.3.2	THE USER SCRATCH DATA SET	5-2
5.4	INITIALIZING THE DATA SET TABLE	5-3
5.5	INITIALIZING SEQUENTIAL FILES	5-3
5.6	WRITING THE BOOT RECORD	5-3
5.7	TERMINATING THE JOB	5-4
5.7.1	PRINTING THE STATISTICS TABLE	5-4
6	MODULE EXECUTION	6-1
6.1	PREPARING FOR EXECUTION	6-1
6.1.1	PASSING ARGUMENTS	6-1
6.1.2	SENDING THE BOOT MESSAGE	6-1
6.1.2.1	VAX/VMS	6-2
6.1.2.2	IBM/MVS	6-2
6.1.3	FINDING THE EXECUTABLE FILE	6-2
6.1.3.1	module path	6-2
6.1.3.1.1	VAX/VMS	6-2
6.1.3.1.2	IBM/MVS	6-3
6.1.3.2	module table	6-3
6.1.4	MAINTAINING FILE INTEGRITY	6-3
6.1.5	CREATING A MODULE ID	6-3
6.1.6	SUSPENDING A PARENT AND ACTIVATING A CHILD	6-3
6.1.6.1	VAX/VMS	6-3
6.1.6.2	IBM/MVS	6-4
6.2	TIMING EXECUTION	6-4
6.2.1	VAX/VMS	6-5

6.2.2	IBM/MVS.....	6-5
6.3	TERMINATING EXECUTION	6-5
6.3.1	MAINTAINING FILE INTEGRITY	6-6
6.3.2	UPDATING THE TIME RECORD	6-6
6.3.3	WRITING THE STATISTICS RECORD	6-6
6.3.4	WRITING THE MODULE COMPLETION RECORD.....	6-6
6.4	RESUMING THE PARENT	6-6
7	THE TERMINAL SYSTEM.....	7-1
7.1	JSTERM ON VAX/VMS	7-1
7.1.1	THE TPU PROGRAM.....	7-2
7.2	JSTERM ON IBM/MVS.....	7-3
7.3	STORAGE FORMAT OF TEMPLATE RECORDS	7-4
7.4	THE FORTRAN PROGRAM	7-7
7.4.1	DATA STRUCTURES	7-7
7.5	JSTERM COMMAND FUNCTIONS	7-8
A	THE LITTLE INPUT LANGUAGE, LIL	A-1
A.1	THE LIL SYNTAX	A-1
A.2	THE LIL ROUTINES.....	A-2
A.3	SUMMARY OF THE JSLILX ROUTINES	A-4
A.4	ERROR NUMBERS AND MESSAGES	A-5
A.5	A SAMPLE PROGRAM	A-5
B	SUMMARY OF JS ROUTINES.....	B-1
B.1	GENERAL ROUTINES	B-1
B.1.1	INCLUDE FILES.....	B-1
B.1.2	FORTRAN FILES	B-2
B.2	TERMINAL SYSTEM ROUTINES	B-10
B.2.1	INCLUDE FILES.....	B-10
B.2.1	FORTRAN FILES	B-10
B.3	VAX/VMS ROUTINES	B-12
B.3.1	INCLUDE FILES.....	B-12
B.3.2	FORTRAN FILES	B-12
B.3.3	MISCELLANEOUS FILES	B-13
B.4	IBM/MVS ROUTINES	B-13
B.4.1	FORTRAN FILES	B-13
B.4.2	ASSEMBLER FILES.....	B-14
B.5	LITTLE INPUT LANGUAGE (LIL) ROUTINES	B-15
C	JS INTERNAL ERROR MESSAGES	C-1
D	REFERENCES	D-1

1 INTRODUCTION

The JOSHUA system routines (JS routines) can be used to manage a JOSHUA data base and execute JOSHUA modules on VAX/VMS and IBM/MVS computer systems. This manual provides instructions for using the JS routines and information about the internal data structures and logic used by the routines. It is intended for use primarily by JOSHUA systems programmers, however, advanced applications programmers may also find it useful.

The JS routines are, as far as possible, written in ANSI FORTRAN 77 [1] so that they are easily maintainable and easily portable to different computer systems. Nevertheless, the JOSHUA system provides features that are not available in ANSI FORTRAN 77, notably dynamic module execution and a data base of named, variable length, unformatted records, so some parts of the routines are coded in nonstandard FORTRAN or assembler (as a last resort). In most cases, the nonstandard sections of code are different for each computer system. To make it easy for programmers using the JS routines to avoid naming conflicts, the JS routines and common block all have six character names that begin with the characters "JS".

Before using this manual, you should be familiar with the JOSHUA system as described in *The JOSHUA Users' Manual* [2], ANSI FORTRAN 77 [1], and at least one of the computer systems for which the JS routines have been implemented [3-15].

2 USING JS ROUTINES FOR JOSHUA OPERATIONS

This chapter describes how to use JS routines to perform various JOSHUA operations from a JOSHUA module. A sequence of calls to JS routines is required for many operations. Typically, a first call begins the operation, intervening calls actually do it, and a final call ends the operation. For most operations, the work of coding calls to these routines can be avoided entirely by using the JOSHUA statements, described in *The JOSHUA Users' Manual* [2], and using the JOSHUA Precompiler to parse the statements and generate calls to the JS routines. There are, however, several operations, originally designed to be performed from the terminal system, that do not have corresponding JOSHUA statements.

Many of the JS routines use similar arguments. A JOSHUA record or subtree name must be passed to the JS routines in string form, described in Section 3.1. The same convention is used to pass a list of read hierarchy elements. Most routines provide an optional alternate return to a specified label if an error is detected during their execution. If you want the alternate return to be taken, set the `a_flag` argument to 1, otherwise, the error will be handled by the JS routines and execution of your module will be terminated. For routines with an `a_flag` argument, regardless of the `a_flag` value, you must always code an existing label in the argument list.

The usage for JS routines is given in this chapter in the following format:

```
CALL JSXXXX (arg1,arg2,*label)
```

When using this format as a guide for calling JS routines, code characters in uppercase as they appear and substitute characters in lower case appropriately.

The following format is used to describe arguments:

arg1	I	i	Input argument
arg2	C*(*)	o	Output argument

The first field corresponds to an argument name in the the preceding usage description. The next field gives the type of the argument, I for INTEGER, C*n for CHARACTER*n, and C*(*) for CHARACTER*(*). The third field designates the argument as either input to the routine (i) or output from the routine (o). The final field provides a brief description of the argument.

2.1 INITIALIZING A MODULE

The first executable statement in a JOSHUA module should be a call to the Module Begin routine, JSMDDBG. If the module expects a list of arguments from its parent, the `list_flag` argument must be set. When `list_flag` is set, JSMDDBG initiates a read operation for the argument record that was written by the parent in the system SCRATCH data set. However, you must follow JSMDDBG by calls to JSRDxx and/or JSRlxx routines to obtain the values from the record. End the read by a call to JSRDEN.

Usage and argument description:

```
CALL JSMDBG (module_name,id_string,list_flag,a_flag,*label)
```

```
module_name  C*(*) i  Module name, <= C*6
id_string    C*(*) i  String to identify module, <= C*64
list_flag    I      i  For argument list from parent 1, otherwise 0
a_flag       I      i  For error alternate return 1, otherwise 0
```

Example 1:

```
CALL JSMDBG ('ORANGE','Version 3A by SUNKIST',0,0,*10)
```

No argument list is expected from the parent module. Any errors will be processed by the JS routines.

Example 2:

```
CALL JSMDBG ('GRAPE','SEEDLESS',1,0,*10)
CALL JSRDIN (N,1)
CALL JSRDCH (NAMES,N)
CALL JSRDEN (0,*10)
```

In this example, arguments are expected from the parent module that can be described with the FORTRAN io-list: N,(NAMES(I),I=1,N).

2.2 EXECUTING A MODULE

You specify the first module to be executed in your job in your JSIN file, and JSMON executes it for you. That module or any subsequent modules in your job can call JS routines to execute a module. First, call the Execution Begin routine, JSEXBG, to begin preparation for module execution. If you want to pass arguments to the child module, set the list_flag. JSEXBG will begin the write operation for the arguments record in the system SCRATCH data set, but you must make calls to JSWRxx and/or JSWlxx routines to write the record. End the write by a call to JSWREN. After the arguments, if any, have been written call JSEXEC to execute the module.

Usage and argument description:

```
CALL JSEXBG (module_name,list_flag,a_flag,*label)
CALL JSEXEC (module_name,list_flag,a_flag,*label)
```

```
module_name  C*(*) i  Module name, <= C*6
list_flag    I      i  For argument list for child 1, otherwise 0
a_flag       I      i  For error alternate return 1, otherwise 0
```

The list_flag argument should have the same value in the calls to both JSEXBG and JSEXEC.

Example 1:

```
CALL JSEXBG ('SEARCH',0,0,*100)
CALL JSEXEC ('SEARCH',0,0,*100)
```

In this example, no arguments will be sent. Also, no alternate return will be made in case of error.

Example 2:

```
CALL JSEXBG ('TEST',1,1,*99)
CALL JSWRCH ('PROBLEM 1',1)
CALL JSWRIN (N,1)
CALL JSWRIN (IDAT(1),N)
CALL JSWRRL (DAT(1),N)
CALL JSWRRL (CORE(1,1),2*N)
CALL JSWREN (1,*99)
CALL JSEEXEC ('TEST',1,1,*99)
```

This module is passing arguments to its child that, depending on the definition of the CORE array, might be described with the io-list:

```
'PROBLEM 1',N,(IDAT(I),I=1,N),(DAT(I),I=1,N),((CORE(I,J),I=1,2),J=1,N)
```

2.3 TERMINATING A MODULE

Termination of a module should always be through a JS routine, see Section 6.3. The Return routine JSRETN can be used to terminate execution of a module and return a completion code and message to the parent.

Usage and argument description:

```
CALL JSRETN (return_code,message)
```

```
return_code  I      i  Module completion code
message      C*(*) i  Module completion message
```

Example 1:

```
CALL JSRETN (0, ' ')
```

This is a typical normal completion:

Example 2:

```
CALL JSRETN (1,'BAD NAME IN NAME LIST: '//NAME(J))
```

2.4 SETTING AND CHANGING THE READ HIERARCHY

To set the default read hierarchy, use JSSEDR. To change the current read hierarchy use JSSERH. The default and current read hierarchies are described in Section 3.6.

Usage and argument description:

```
CALL JSSEDR (name_string,a_flag,*label)
CALL JSSERH (name_string)
```

```
name_string  C*(*) i  Blank or a list of read hierarchy elements
a_flag       I      i  For error alternate return 1, otherwise 0
```

Example 1:

```
CALL JSSEDR (' ',1,*10)
```

The default read hierarchy will be set to be the WRITE data set, if any, the USER data set if any, and 'STD'. If an error occurs, an alternate return will be made to label 10.

Example 2:

```
CALL JSSEDR ('USER1 STD ',0,*10)
```

Example 3:

```
CALL JSSERH (' ')
```

This call will set the current read hierarchy to be the same as the default read hierarchy.

Example 4:

```
CALL JSSERH (JSIQWR()// 'STD ' // DATSET(K) // DATSET(K+1))
```

The inquiry function JSIQWR is used to place the WRITE data set first in the read hierarchy. The DATSET array would most reasonably be CHARACTER*8, but it could be CHARACTER*16 or CHARACTER*24, as long as the string passed to JSSERH can be divided into at most 8 valid read hierarchy elements, each 8 characters long.

2.5 READING A RECORD

You initiate a read operation with a call to a Read Begin routine:

JSRSBG	to read from the system SCRATCH data set
JSRTBG	to read a TEMPLATE record that matches a given record name
JSRDBG	to read any other record

Usage and argument description:

```
CALL JSRSBG (name_string,list_flag,a_flag,*label)
CALL JSRTBG (name_string,list_flag,a_flag,*label)
CALL JSRDBG (name_string,list_flag,a_flag,*label)
```

name_string	C*(*)	i	Record name in string form
list_flag	I	i	To actually read data 1, otherwise 0
a_flag	I	i	For error alternate return 1, otherwise 0

If the list_flag is not set, the Read Begin essentially serves to inquire about the existence of a given JOSHUA record. No further calls to JS routines are needed.

To actually read data from the record, set the list_flag argument. Then make calls to Read Value routines, depending on the data types of the values:

JSRTxx, JSR2xx	for TEMPLATE records
JSRDxx, JSR1xx	for other records

where xx is:

IN	for INTEGER
RL	for REAL
DP	for DOUBLE PRECISION
LG	for LOGICAL
CM	for COMPLEX
CH	for CHARACTER

Usage and argument description:

```
CALL JSRTxx (xxval,nxx)
CALL JSRDxx (xxval,nxx)
CALL JSR2xx (xxval,nxx,a_flag,*label)
CALL JSR1xx (xxval,nxx,a_flag,*label)
```

xxval	xx	i	Variable of type xx to receive values read
nxx	I	i	Number of xxvals to read
a_flag	I	i	For error alternate return 1, otherwise 0

For character data (xx=CH), the length of the character string as well as the number of values to be read determines the amount of data that will be read from the JOSHUA record. The number of characters read by a call to a Read Character Value routine will be:

$LEN(xxval) * nxx$

If the list_flag argument was set in the Begin Read call, terminate a read operation by calling one of the Read End routines:

JSRTEN	for TEMPLATE records
JSRDEN	for other records

Usage and argument description:

```
CALL JSRTEN (a_flag,*label)
CALL JSRDEN (a_flag,*label)
```

a_flag	I	i	For error alternate return 1, otherwise 0
--------	---	---	---

Note that the JSRTxx and JSRDxx routines do not have alternate returns. If an error occurs during a call to one of these routines, the error code will be saved internally and a normal return will be made from the Read Value routines. The Read End routine will, however, recognize that an error condition exists, and make an alternate return to the specified label, depending on the value of a_flag. Thus, the overall result for the read operation is an alternate return. Alternate returns are provided with the JSR1xx and JSR2xx routines.

Example 1:

```
10      CALL JSRDBG ('1      2      3      ',0,1,*10)
      CONTINUE
      I = JSIQSE()
      IF (I .EQ. 0) THEN
        (actions to take when the record exists)
      ELSE IF (I .EQ. 1130) THEN
```

```

        (actions to take when the record does not exist)
ELSE
        (actions to take when some other error occurs)
ENDIF

```

This code fragment uses a Read Begin routine to determine if the record 1.2.3 exists, but does not actually read the record. The inquiry function JSIQSE returns the completion code for the most recent operation, in this case the Read Begin. A 0 means success, the record was found. The code for "RECORD OR SUBTREE NOT FOUND" is 1130.

Example 2:

```

CALL JSRDBG (JSCVIN(1)//JSCVIN(2)//JSCVIN(K),1,1,*10)
CALL JSR1IN (N,1,1,*10)
DO 5 I = 1,N
    CALL JSR1CH (STR(I),1,1,*10)
    CALL JSR1IN (IVAL(I),1,1,*10)
5  CONTINUE
CALL JSRDEN (1,*10)

```

Assuming K is an integer variable, this example illustrate use of the conversion function JSCVIN to pass integer qualifiers to the JS routines. The data that will be read can be described with the io-list: N,(STR(I),IVAL(I),I=1,N).

Example 3:

```

CALL JSRDBG (NAME(1)//NAME(2)//NAME(3),1,1,*30)
N = 1
10  CONTINUE
CALL JSR1IN (IDAT(N),1,1,*30)
IF (IDAT(N) .GE. 0) THEN
    N = N + 1
    GOTO 10
ENDIF
CALL JSRDEN (1,*30)
N = N - 1

```

This sequence of calls reads an arbitrarily long list of nonnegative integers into the IDAT array. The array NAME would typically be a CHARACTER*8 array containing valid JOSHUA qualifiers as elements. An alternate return will be made to label 30 if an error occurs.

Example 4:

```

CALL JSRTBG ('TEMPLATE'// 'INPUT' '//' 'GLASS' '//'CASE,1,1,*5)

        (calls to JSRTxx routines to read the TEMPLATE record)

CALL JSRTEN (1,*5)

```

Assume CASE is a CHARACTER*8 variable. This code fragment finds and reads a template record that matches the data record INPUT.GLASS.case. Typically, a matching template will have a wildcard qualifier to match the variable qualifier in the name, for example:

TEMPLATE.INPUT.GLASS.?CASE

2.6 WRITING A RECORD

Writing a record involves steps similar to those used in reading a record. First, call a Write Begin routine:

JSWSBG to write to the system SCRATCH data set
JSWRBG to write to any other data set

Usage and argument description:

```
CALL JSWSBG (name_string,list_flag,a_flag,*label)
CALL JSWRBG (name_string,list_flag,a_flag,*label)

name_string  C*(*)  i  Record name in string form
list_flag    I      i  To actually write data 1, otherwise 0
a_flag       I      i  For error alternate return 1, otherwise 0
```

If the list_flag argument is not set for a Write Begin operation, no values will be written and no other calls should be made. However, if the record does not exist, the return code will be zero since this is acceptable for a write operation. Nonexistent records are added to the catalog if and only if data will be placed in them.

To write data call Write Value routines, JSWRxx and/or JSWlxx, that correspond to the data types of the values to be written, where xx is:

IN for INTEGER
RL for REAL
DP for DOUBLE PRECISION
LG for LOGICAL
CM for COMPLEX
CH for CHARACTER

Usage and argument description:

```
CALL JSWRxx (xxval,nxx)
CALL JSWlxx (xxval,nxx,a_flag,*label)

xxval        xx      i  Variable of type xx values to be written
nxx          I      i  Number of xxvals to be written
a_flag       I      i  For error alternate return 1, otherwise 0
```

For character data (xx=CH), the length of the character string as well as the number of values to be written determines the amount of data that will be output to the JOSHUA record. The total number of characters written by a call to JSWRCH or JSW1CH will be:

$LEN(xxval) * nx$

If the list_flag argument was set for the Write Begin, complete the operation by calling the Write End routine, JSWREN.

Much like the Read Value routines, the JSWlxx routines provide an error alternate return option, but the JSWRxx routines continue with a normal returns, with any error condition being remembered and processed in JSWREN.

Usage and argument description:

```
CALL JSWREN (a_flag,*label)
```

```
a_flag      I      i  For error alternate return 1, otherwise 0
```

Example 1:

```
CALL JSWSBG ('DATA    RECORD  '//MODID,1,1,*10)
CALL JSWRCH (MODNM,1)
CALL JSWRDP (CPUTIM,1)
CALL JSWREN (1,*10)
```

Assuming that MODID is a CHARACTER*8 variable, this example writes a record DATA.RECORD.modid to the system SCRATCH data set.

Example 2:

```
CALL JSWRBG ('SAVE    DATA    99      ',1,1,*99)
CALL JSWIIN (N,1,1,*99)
CALL JSWIRL (A,N,1,*99)
CALL JSWREN (1,*99)
```

This sequence of calls writes the JOSHUA record SAVE.DATA.99 using the io-list: N,(A(I),I=1,N).

2.7 PRINTING A RECORD OR SUBTREE OF RECORDS

To print a record or subtree of records in the JSOUT file using templates for formatting information, use JSPRNT.

Usage and argument description:

```
CALL JSPRNT (name_string,a_flag,*label)
```

```
name_string  C*(*) i  Record or subtree name, in string form
a_flag       I      i  For error alternate return 1, otherwise 0
```

2.8 COPYING A RECORD OR SUBTREE OF RECORDS

You can copy a record or subtree of records using JSCOPY.

Usage and argument description:

```
CALL JSCOPY (name_string1,name_string2,a_flag,*label)
```

```
name_string1 C*(*) i  Record or subtree to copy from, in string
                    form
name_string2 C*(*) i  Record or subtree to copy to,  in string form
a_flag       I      i  For error alternate return 1, otherwise 0
```

Example:

```
CALL JSCOPY ('SCRATCH OUTPUT  TEMP  ',
*           'OUTPUT  SAVE    ',1,*999)
```

This call will copy all the records in the subtree OUTPUT.TEMP in the user SCRATCH data set to the subtree OUTPUT.SAVE in the WRITE data set.

2.9 DELETING A RECORD OR SUBTREE

To delete a record or subtree, use JSDLTE.

Usage and argument description:

```
CALL JSDLTE (name_string,a_flag,*label)

name_string  C*(*) i  Record or subtree to delete, in string form
a_flag       I       i  For error alternate return 1, otherwise 0
```

2.10 DELETING THE USER SCRATCH DATA SET

To delete the data in the user SCRATCH data set and make the space available for reuse, use JSDLTE.

Usage and argument description:

```
CALL JSDLTE ('SCRATCH ',a_flag,*label)

a_flag       I       i  For error alternate return 1, otherwise 0
```

2.11 COMPRESSING A JOSHUA DATA SET

When records are deleted in a JOSHUA data set, the space is not automatically reused. To recover the space used by deleted records, the data set must be compressed or compacted. To recover the spaced and make it available for reuse, use JSJTCP.

Usage and argument description:

```
CALL JSJTCP (ds_name,*label)

ds_name      C*(*) i  Name of data set to be compressed
```

2.12 RENAMING A QUALIFIER

To rename a qualifier in a record or subtree, use JSRNME.

Usage and argument description:

```
CALL JSRNME (name_string,new_qual,a_flag,*label)

name_string  C*(*) i  Record or subtree name that ends with the
                    qualifier to be renamed, in string form
new_qual     C*(*) i  New qualifier, <= C*8
a_flag       I       i  For error alternate return 1, otherwise 0
```

Example:

```
CALL JSRNME ('INPUT  APPLES  ', 'ORANGES', 1, *99)
```

The qualifier APPLES will be changed to ORANGES.

2.13 LISTING THE RECORDS IN A SUBTREE

To obtain a list of the records in a subtree in preorder initiate the operation by call a List Begin routine:

JLSDBG	to list a subtree in the system SCRATCH data set
JSLIBG	to list a subtree in any other data set

Usage and argument description:

```
CALL JLSDBG (name_string, a_flag, *label)
CALL JSLIBG (name_string, a_flag, *label)
```

name_string	C*(*)	i	Record or subtree name, in string form
a_flag	I	i	For error alternate return 1, otherwise 0

Make calls to JSLINX to perform the traversal of the tree in preorder. Each call to JSLINX will find the next record, if any. A call to JSLINX after there are no more records in the subtree is an error.

If you set the list_flag argument for JSLINX, in addition to finding the next record, JSLINX will prepare the record for reading. Thus, you can call JSRDxx and/or JSRlxx routines after each JSLINX to read each record in the subtree. Do not, however, call JSRDEN; JSLINX will terminate the read operations for you. Both COPY and PRINT operations make use of this feature of the list subtree operation to process all the records in a subtree.

Usage and argument description:

```
CALL JSLINX (more_flag, list_flag, a_flag, *label)
```

more_flag	L	o	T if another record has been found, F otherwise
list_flag	I	i	To read the records found 1, otherwise 0
a_flag	I	i	For error alternate return 1, otherwise 0

You can obtain the name of the most recently found record by calling the Inquiry routine JSIQLN.

Usage and argument description:

```
CALL JSIQLN (str, n, a_flag, *label)
```

str	C*(*)	o	Record name, returned in string form
n	I	o	Number of characters returned in str
a_flag	I	i	For error alternate return 1, otherwise 0

Example 1:

```

      CALL JSSERH ('STD      ')
      CALL JSLIBG ('ASSEMBLY',0,0,*99)
      N = 0
10    CONTINUE
      CALL JSLINX (MORE,0,0,*99)
      IF (MORE) THEN
        N = N + 1
        CALL JSIQLN (STR,L,0,*99)
        WRITE (6,15) N, (STR(I:I+7),I=1,L,8)
15    FORMAT (1X,I6,3X,13(A8:'. '))
        GOTO 10
      ENDIF

```

A numbered list of the records in the ASSEMBLY standard data set will be printed by this sequence of calls.

Example 2:

```

      CALL JSLSBG ('DATA    RECORD ',1,1,*99)
10    CONTINUE
      CALL JSLINX (MORE,1,1,*99)
      IF (MORE) THEN
        CALL JSRDCH (NAME,1)
        CALL JSRDDP (TIME,1)
        WRITE (6,*) NAME,TIME
        GOTO 10
      ENDIF

```

This code fragment could be used to read and print the contents of the subtree of records that would be created by executing the calls in Section 2.6, Example 1 for several different *modid*'s.

2.14 DISPLAYING THE LIST OF RECORDS IN A SUBTREE

You can use the routine JSLIPR to print a formatted preorder list of records in a subtree in a given output file. Similarly, JSLIBU will return the list in a character buffer. Both these routines use JSLIBG and JSLINX to traverse the tree as described above, and do the output formatting for you. The display will include the physical address in the data file as well as the name of each record.

Usage and argument description:

```

CALL JSLIBU (name_string,mxnbuf,buf,nbuf,a_flag,*label)
CALL JSLIPR (name_string,lun,a_flag,*label)

```

name_string	C*(*)	i	Record or subtree name, in string form
mxnbuf	I	i	Number of "records" in buf
buf	C*(*)	o	Array of mxnbuf character strings
nbuf	I	o	Number of display "records" placed in buf
lun	I	i	Unit for output
a_flag	I	i	For error alternate return 1, otherwise 0

The character buffer, buf, can be thought of a file of mxnbuf fixed length records, each record LEN(buf(1)) long. We require that LEN(buf(1)) be at least 32.

Example 1:

```

      CALL JSLIBU ('ASSEMBLY',500,BUF,N,1,*10)
      NPAGES = 0
      L = 60
      DO 10 I = 1,N
        IF (L .EQ. 60) THEN
          NPAGES = NPAGES + 1
          L = 1
          WRITE (6,1) NPAGES
1         FORMAT ('0PAGE',I6,/)
          ENDIF
          L = L + 1
          WRITE (6,*) BUF(I)
10      CONTINUE

```

Example 2:

```

      CALL JSLIPR ('ASSEMBLY',6,1,*99)

```

2.15 LISTING THE CHILDREN OF A SUBTREE

A list children operation is initiated by calling JS�CBG. Subsequent calls to JS�CNX will return the 8 character names of children nodes, concatenated in a single string. The length of the string, which determines the maximum number of children that can be returned by one call, is your choice; the only restriction is that the length must be a multiple of 8. Since the number of children in a subtree is not known in advance, multiple calls to JS�CNX may be necessary to obtain the names of all the children. When JS�CNX returns a number of children less than the maximum that can fit in the return string, it has found the last child and terminated the operation. Any subsequent call to JS�CNX (before another call to JS�CBG) is an error.

Usage and argument description:

```

      CALL JS�CBG (name_string,a_flag,*label)
      CALL JS�CNX (child_string,nchild,a_flag,*label)

name_string  C*(*) i  Record or subtree name, in string form
child_string C*(*) o  List of children returned
nchild      I      o  Number of children returned in child_string
a_flag      I      i  For error alternate return 1, otherwise 0

```

Example 1:

```

      CALL JSERH ('STD      ')
      CALL JS�CBG ('MULTIGRP',1,*99)
      MX = LEN(CHILD) / 8
10     CONTINUE
      CALL JS�CNX (CHILD,N,1,*99)
      IF (N .GT. 0) THEN
        WRITE (6,'(1X,A8)') (CHILD(I:I+7),I=1,N*8,8)
        IF (N. EQ. MX) GOTO 10
      ENDIF

```

This example will print a list of the top level children of the standard data set MULTIGRP.

Example 2:

```

CALL JSSEWR (JSIQWR())
CALL JSLCBG ('',1,*99)
CALL JSLCNX (CHILD,N,1,*99)
IF (N .EQ. 0) THEN
  WRITE (6,*) 'NO CHILDREN'
ELSE
10  CONTINUE
    WRITE (6,'(1X,A)') (CHILD(I:I+7),I=1,N*8,8)
    IF (N .EQ. MX) THEN
      CALL JSLCNX (CHILD,N,1,*99)
      IF (N .GT. 0) GOTO 10)
    ENDIF
  ENDIF
ENDIF

```

The top level children in the WRITE data set will be printed by this example.

2.16 DISPLAYING THE LIST OF CHILDREN OF A SUBTREE

You can call either JSLCBU or JSLCPR to have the children of a subtree listed in a formatted display. JSLCBU returns the display in a character buffer. JSLCPR prints the display in the file of your choice.

Usage and argument description:

```

CALL JSLCBU (name_string,mxnbuf,buf,nbuf,a_flag,*label)
CALL JSLCPR (name_string,lun,a_flag,*label)

```

name_string	C*(*)	i	Record or subtree name, in string form
mxnbuf	I	i	Number of "records" in buf
buf	C*(*)	o	Array of mxnbuf character strings
nbuf	I	o	Number of display "records" placed in buf
lun	I	i	Unit for output
a_flag	I	i	For error alternate return 1, otherwise 0

The character buffer, buf, can be thought of a file of mxnbuf fixed length records, each record LEN(buf(1)) long. We require that LEN(buf(1)) be at least 9.

Examples:

```

CALL JSLCBU ('ASSEMBLY',100,BUF,N,1,*10)

CALL JSLCPR ('ASSEMBLY',6,1,*99)

```

2.17 SETTING THE WRITE DATA SET

The JSSEWR routine can be used to designate a different, possibly new, nonstandard data set to be the WRITE data set, or to eliminate a WRITE data set designation. Changes to the WRITE data set designation will affect only the current module. The original boot record created by JSMON will be used to set the initial WRITE data set in all other modules.

Usage and argument description:

```
CALL JSSEWR (dsname,stat,tree_space,data_space,loc,a_flag,*label)

dsname      C*(*) i  Name of the data set to be designated WRITE
stat        C*(*) i  Status of the data set, 'NEW', 'OLD', or
                    'REUSE'
tree_space  I      i  Minimum number of physical tree records
data_space  I      i  Minimum number of physical data records
loc         C*(*) i  Name of an IBM system data set, or blank
a_flag      I      i  For error alternate return 1, otherwise 0.
```

If the dsname argument is blank, then the current WRITE data set designation will be eliminated and no other designation will be made.

If the dsname argument is not blank and if the status of the data set is 'NEW', a new data set will be created at the first place on the nonstandard path and set to be the WRITE data set. The tree_space, data_space, and loc arguments are not used under VAX/VMS. They are only used under IBM/MVS if the status of the data set to be designated write is 'NEW'. The loc argument specifies where the files for the data set should be created. If loc is blank, the files will be placed on SYSDA. Otherwise, they will be created as if VOL=REF=loc had been coded in a JCL DD statement [6]

Example 1:

```
CALL JSSEWR (' ', 'OLD', 0, 0, ' ', 1, *10)
```

This example eliminates a WRITE data set. A valid status argument must be specified, even though it will be ignored.

Example 2:

```
CALL JSSEWR ('JUNK', 'NEW', 50, 500, ' ', 0, *10)
```

Assume that this call will be made under IBM/MVS and that T7980 is the first path specifier in the nonstandard path. The data set JUNK will be created and designated to be the WRITE data set. The tree file, T7980.JUNK.J80TREE, will have at least 50 physical records, the data file, T7980.JUNK.J80DATA, will have at least 500 records, and both files will be placed on SYSDA.

Example 3:

```
CALL JSSEWR ('TESTDATA', 'OLD', 0, 0, ' ', 1, *10)
```

2.18 SETTING THE NONSTANDARD PATH

Use the JSSEPA routine to set a different search path for nonstandard data sets. Under VAX/VMS the elements in a path are directory specifications. Under IBM/MVS the elements are top level system data set qualifiers, usually a user id's.

Usage and argument description:

```
CALL JSSEPA (2,path,npath,a_flag,*label)

path        C*(*) i  An array of path elements
```

npath	I	i	Number of elements in path
a_flag	I	i	For error alternate return 1, otherwise 0.

Example 1:

```
CALL JSSEPA (2, '[T7980.JOSHUA.DATA]', 1, 1, *99)
```

This is a simple VAX/VMS example.

Example 2:

```
PATH(1) = '[T7980.JOSHUA.DATA]'
PATH(2) = '[T7980.JOSHUA.TEMP]'
CALL JSSEPA (2, PATH, 2, 1, *99)
```

This is a simple VAX/VMS example.

Example 3:

```
PATH(1) = 'T7980'
PATH(2) = 'T8079'
PATH(3) = 'T3895'
CALL JSSEPA (2, PATH, 3, 1, *99)
```

This is a simple IBM/MVS example.

2.19 SETTING THE LOCATION OF A NONSTANDARD DATA SET

The JSSELD routine can be used to set the location of the files used for a given nonstandard data set. If the specified data set is not already known to the job, it will be added to the Data Set Table with the files specified. If the data set is already known to the job, JSSELD will change the file association for the data set.

Usage and argument description:

```
CALL JSSELD (dsname, loc, *label)
```

dsname	C*(*)	i	Name of the data set, <= C*8
loc	C*(*)	i	Location of files to be associated

Under VAX/VMS the loc argument should be a directory specification. Under IBM/MVS it should be a first level system data set qualifier, usually a user id. An alternate return to the specified label will always be made if an error occurs.

Example 1:

```
CALL JSSELD ('JUNK', '[T7980.JOSHUA.TEST]', *10)
```

This is a VAX/VMS example.

Example 2:

```
CALL JSSELD ('JUNK', 'T7980', *10)
```


This is an IBM/MVS example.

2.20 PLACING A COMMENT IN A DATA SET

Use the JSPCDS routine to place a comment in the tree header record of a READ/WRITE JOSHUA data set. Only the WRITE and user and system SCRATCH data sets can be open with READ/WRITE privileges.

Usage and argument description:

```
CALL JSPCDS (dsname,comment,a_flag,*label)
```

dsname	C*(*)	i	Name of the data set
comment	C*(*)	i	Comment, <= C*80
a_flag	I	i	For error alternate return 1, otherwise 0.

Examples:

```
CALL JSPCDS ('REACDATA','HEAVY WATER VERSION',1,*99)
```

```
CALL JSPCDS ('SCRATCH','TEMPORARY FOR TEST JOB XYZ',0,*99)
```

2.21 GETTING DATA SET INFORMATION

The JSIQDS routine will return information about a given data set.

Usage and argument description:

```
CALL JSIQDS (dsname,exst,open,typ,pro,cdat,ctim,rdat,rtim,lcom,com)
```

dsname	C*(*)	i	Name of the data set
exst	L	o	T if dsname is known to the job, else F
open	L	o	T if dsname set is open, else F
typ	I	o	Type, 0=Nonstandard, 1=Standard
pro	I	o	Protection, 1=READ ONLY, 2=READ/WRITE
cdat	C*(*)	o	Creation date, mm-dd-yy, >= C*8
ctim	C*(*)	o	Creation time, hh:mm:ss, >= C*8
rdat	C*(*)	o	Revision date, mm-dd-yy, >= C*8
rtim	C*(*)	o	Revision time, hh:mm:ss, >= C*8
lcom	I	o	Length of com
com	C*(*)	o	Comment in the header record, >= C*80

2.22 DISPLAYING DATA SET INFORMATION

You can use either JSDSBU or JSDSPR to obtain a formatted display of information about JOSHUA data sets known to the current job. JSDSBU places the display in a character buffer. JSDSPR prints the display in a specified file.

Usage and argument description:

```
CALL JSDSBU (dsname,typ,mxnbuf,buf,nbuf,*label)
```

```
CALL JSDSPR (lun,recl,dsname,typ,*label)
```

dsname	C*(*)	i	Name of a data set, or blank for all of typ
--------	-------	---	---

typ	I	i	Type of data set, 0=Nonstandard, 1=Standard, 2=Any type
mxnbuf	I	i	Number of "records" in buf
buf	C*(*)	o	Array of mxnbuf character strings
nbuf	I	o	Number of display "records" placed in buf
lun	I	i	Unit for output
recl	I	i	Maximum record length to be written in lun
a_flag	I	i	For error alternate return 1, otherwise 0

The character buffer, buf, can be thought of a file of mxnbuf fixed length records, each record LEN(buf(1)) long. We require that LEN(buf(1)) be at least 68.

Example 1:

```
CALL JSDSBU (' ',0,100,BUF,N,*10)
```

This call will return formatted information about all the nonstandard data sets known to the job.

Example 2:

```
CALL JSDSPR (6,131,' ',2,*99)
```

This example will print information about all the standard data sets in the JSOUT file.

2.23 CREATING ERROR, WARNING, AND MESSAGE DISPLAYS

The JS routines can be used to create formatted displays with incorporated data for ERROR, WARNING, or MESSAGE purposes. After an ERROR display is made for error number n, the child module is terminated and control is returned to the parent. If n is nonzero, the following error code and message will be posted by the parent:

1228 EXECUTION OF MODULE *child_module* ENDED BY NONZERO ERROR STATEMENT *n*

After a WARNING or MESSAGE display is completed, the child module will continue executing. A count of WARNING and MESSAGE displays can be obtained from the JSIQWC and JSIQMC inquiry functions. The JSIQUE function will return the number associated with the most recent ERROR, WARNING, or MESSAGE display.

You begin a display by calling JSERBG. You can pass text for the display to JSERBG, but text can also be stored in the JOSHUA data base. JSERBG first looks there for a record named:

ERRMSG.module_name.number

If no such record exists, it uses any text passed as an argument.

You can include format fields, described in *The JOSHUA Users' Manual* [2], in the text passed as an argument or read from a record. To do so, set the list_flag argument in the call to JSERBG and call JSERxx routines to insert values in the fields, based on the field types and values to be displayed, where xx is:

IN for INTEGER
 RL for REAL
 DP for DOUBLE PRECISION
 LG for LOGICAL
 CM for COMPLEX
 CH for CHARACTER

If the list_flag was set for JSERBG, call the routine JSEREN to signal that the display is complete.

Usage and argument description:

```
CALL JSERBG (number,text,type,list_flag)
CALL JSERxx (xxval,nxx)
CALL JSEREN
```

number	I	i	ERROR, WARNING, or MESSAGE number
text	C*(*)	i	Text to be included in display, <= C*876
type	I	i	Type of display, 0=MESSAGE, 1=WARNING, 2=ERROR
list_flag	I	i	To include values in display 1, otherwise 0
xxval	xx	i	Values of type xx to be placed in display
nxx	I	i	Number of xxvals

Example 1:

```
CALL JSERBG (0,'ENTERING PHASE SPACE',0,0)
```

The list_flag argument is 0, which indicates that no values will be included in the display. Thus, no calls to JSERxx routines or JSEREN are made. JSERBG will call JSEREN for you. If this call is placed from module XYZ, the display produced will be:

```
**** JOSHUA MODULE XYZ ***** MESSAGE NUMBER 0 *****
*
*   ENTERING PHASE SPACE
*
*****
```

Example 2:

```
CALL JSERBG (1,':* ISOTOPES IN RECORD -\\'//
*           ':AAAAAAAAA.:AAAAAAAAA.MATERIAL.ISOTOPES\\'//
*           'MAXIMUM ALLOWED IS :*',
*           2,1)
CALL JSERIN (NIS,1)
CALL JSERCH (NAMES,2)
CALL JSERIN (MAXNIS,1)
CALL JSEREN
```

The display produced by this sequence of calls from the module SMASH, depending on the values of NIS, NAMES, and MAXNIS, might look something like:

```

**** JOSHUA MODULE SMASH **** MESSAGE NUMBER 1 ****
*
* :113 ISOTOPES IN RECORD -
*
* :ASSEM .:TEST .MATERIAL.ISOTOPES
*
* MAXIMUM ALLOWED IS :100
*
*****

```

2.24 CONVERTING A LIST OF NAMES

The JSCVSS routine can be used to convert a list of names in which the names are delimited by a given character to a list of names of uniform length with no delimiting characters. For example, JSCVSS can be used to convert a JOSHUA namestring from dotted form to string form (see Section 3.1).

Usage and argument description:

```
CALL JSCVSS (str1,delimiter,length,str2,length2,*label)
```

str1	C*(*) i	The list of names separated by delimiter
delimiter	C*(1) i	The delimiter
length	I i	The uniform length of names to be returned in str2
str2	C*(*) o	The list of names of uniform length and no delimiter
length2	I o	The number of characters returned in str2

If a given name in the str1 is longer than the specified uniform length, it will be truncated when it is returned in str2.

Example 1:

```
CALL JSCVSS ('A.B2.3456.UP-DOWN','.',8,STR,L,*99)
```

This call, which converts a name from dotted form to string form, will return:

```
STR='A      B2      3459  UP-DOWN '
L=32
```

Example 2:

```
CALL JSCVSS ('APPLE,PEAR,POMEGRANATE',' ',10,STR,L,*99)
```

This call will convert a list of fruits separated by commas to a list of names each of length 10. Note that the eleven character name of the last fruit in the list will be truncated in the return string:

```
STR='APPLE  PEAR  POMEGRANAT'
L=24
```

3 DESCRIPTION OF THE JOSHUA DATA BASE

An overview of the JOSHUA data base is given in *The JOSHUA Users' Manual* [2]. In this chapter we will provide detail about the storage format for the data base and information about how JOSHUA data sets are handled internally by the JS routines.

3.1 JOSHUA RECORD AND SUBTREE NAMES

A JOSHUA *namestring* is an ordered list of qualifiers. A *qualifier* is a string of from 1 to 8 characters. The characters that may be used are:

0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ,~!@#\$%^&*()_{};"|\/?><'

A user generally uses the *dotted form* of a JOSHUA name, in which the qualifiers are separated by dots. For example, a dotted record name such as A.B.C might be used in a JOSHUA statement in an application module, while using the terminal system, or in writing documentation for an application. In speaking about the record we typically say "A dot B dot C".

The JS routines expect to receive a JOSHUA record or subtree name in *string form*, that is as a single string of concatenated qualifiers, each qualifier 8 characters long, left justified and padded on the right with blanks if necessary. For example, using "Δ" to represent a blank:

DOTTED FORM

RECORD.ONE.22

STRING FORM

"RECORDΔΔONEΔΔΔΔΔ22ΔΔΔΔΔΔ"

The JSCVSS routine can be used to convert a JOSHUA name from dotted form to string form.

A qualifier that begins with a "?", the *wildcard* character, is treated differently during catalog searches when a template record is being sought to *match* a record. Template matching is described in *The JOSHUA Users' Manual* [2].

The JS routines use a few special names internally that include characters not usually valid so that they cannot be confused with user specified names. These are:

.SYSCRAT	Name of the system SCRATCH data set
.DELETED	Name of a deleted qualifier
..leaf..	Name of a leaf node

3.2 JOSHUA DATA SET FILES

A *JOSHUA data set* is a collection of JOSHUA records. Physically, a JOSHUA data set consists of two files, a tree catalog file which catalogs the record names and a data file which contains the data associated with the record names. Both are FORTRAN unformatted direct access files.

3.2.1 TREE FILES

A tree file consists of a header record followed by node records, which catalog the record names and point to the data file. The record length for tree files can be set for a given installation of JOSHUA using the symbolic constants TRRECL and RECLLN in the JSPARM include. At Savannah River under VAX/VMS we use a record length equal to one block, that is 512 bytes, which can store 32 nodes. For consistency, we use the same record length in our IBM/MVS version.

3.2.1.1 TREE HEADER RECORDS

A tree header record contains the following information, in the following order:

INTEGER	Number of last created tree node
INTEGER	Number of physical record at end of data file
C*8	Date of creation, in form mm-dd-yy
C*8	Time of creation, in form hh:mm:ss
C*8	Date of last revision, in form mm-dd-yy
C*8	Time of last revision, in form hh:mm:ss
INTEGER	Number of characters, n, in comment (next) field
C*n	Comment of 1 to 80 characters

3.2.1.2 TREE NODE RECORDS

JOSHUA records are cataloged using a tree structure. Each qualifier in a record name corresponds to an *internal node* in the catalog. There is an additional *leaf node* linked by the JS routines to the end of each record name.

Each node is assigned a number, 1, 2, ... N, which is used as its address in the catalog. We can find any given node in the tree catalog file by using the node number and number of nodes in a single tree record to compute the physical tree record number and offset of the node in that record. The routine JSGENA computes a tree file address from a node address.

Each node consists of an 8 character name field and two integer fields. A leaf node has the special name "..leaf..", which is not a valid JOSHUA qualifier name. The leaf integer fields contain, respectively, the physical record number where the data for the record begins in the data file and the number of bytes in the entire JOSHUA record. Both integer fields are stored as the negation of the actual number.

Figure 3.1 The structure of a leaf node.

The name field of an internal node stores the corresponding qualifier name, and the integer fields are used to link the nodes together. Each internal node has a one or more *children*,

the nodes immediately below, and zero or more *siblings*, the nodes at the same level in the same branch of the tree. Each set of siblings is maintained in a linked list in which the first sibling is considered to be the oldest and the last the youngest. For internal nodes, the first integer field points to the oldest child. Note that a child may be a leaf node. The second integer field points to the next younger sibling. The youngest sibling points back to the oldest, a negative pointer flagging the back pointer. A wildcard node, when present, is always maintained as the youngest sibling at its level.

Figure 3.2 The structure of an internal node.

A tree node record that contains n nodes is actually written in a tree file with the io-list:

$(\text{node_name}(i), i=1, n), (\text{first_integer_field}(i), i=1, n), (\text{second_integer_field}(i), i=1, n)$

To get a better idea of the tree catalog structure, suppose there is a JOSHUA data set with the records listed below. One possible tree configuration for the data set is illustrated in Figure 3.3.

```

A.B1.C
A.B2.C.D
A.B2.C.E
A.B3
TEMPLATE.A.B2.C.?
TEMPLATE.A.?
  
```

3.2.2 DATA FILES

A good way to describe the data in a JOSHUA record is using the storage sequence concept used in ANSI FORTRAN 77 [1]:

A storage sequence is a sequence of storage units. A storage unit is either a numeric storage unit or a character storage unit.

An integer, real, or logical datum has one numeric storage unit in a storage sequence. A double precision or complex datum has two numeric storage units in a storage sequence. A character datum has one character storage unit in a storage sequence for each character in the datum.

A JOSHUA record is simply a storage sequence of FORTRAN data.

The FORTRAN standard specifically avoids defining a relationship between a numeric storage unit and a character storage unit, or a relationship between the storage sequence concept and the physical properties or implementation of storage.

Figure 3.3 The tree structure of a sample JOSHUA data set.

However, these relationships are defined for each implementation of FORTRAN, and, in practice, the relationships are very similar in all commonly used computer systems. The smallest unit of storage is a *bit*. Eight bits form a *byte*. A group of *n* bits is a *word*. The number of bits in a word is on most computers evenly divisible by 8, so a word is alternately considered to have *n* bits or *n/8* bytes. Conveniently, under VAX/VMS and IBM/MVS one character storage unit is one byte, and one numeric storage unit is one word.

The data file for a JOSHUA data set is a FORTRAN unformatted direct access file, which by definition has fixed length records. The symbolic constant DARECL, defined in the JSPARM include, sets this record length. At Savannah River we use 512 byte records for both VAX/VMS and IBM/MVS.

JOSHUA records are allowed to be of variable length, so a given JOSHUA record can occupy one or more physical records. The data for a given JOSHUA record is stored in binary form, packed in contiguous bytes. A JOSHUA record is stored in one or more adjacent physical records, spanning record boundaries as necessary. As explained above, the leaf nodes in the tree catalog contain for each JOSHUA record the physical record number where the data starts and the amount of data stored.

Space in the data file can be "lost" when a record or subtree is deleted or when a record shrinks in size. Since the data for a given record is stored contiguously, space can also be lost when a record expands. If a record that needs more space is not at the end of the file it will be moved there if access to additional physical records is required, and the space

formerly occupied will be lost. There is no automatic mechanism for recovering lost space in the data files. To recover the space, use a JOSHUA COPY to make a copy of the data set which will have the records packed; or use the JSJTCP Compress data set routine (MA terminal system function).

3.2.3 NAMING AND STORING DATA SET FILES

There are two types of JOSHUA data sets: nonstandard and standard. The nonstandard data sets include all user owned and managed data sets. The standard data sets include all the public data managed for users at a given installation. Each JOSHUA installation may have its own versions of standard data sets depending on its JOSHUA applications. However, in most cases there will be a special standard data set named JS. The JS standard data set contains a standard table record, which provides a look up table for the other standard data sets. It also will usually contain a module table record that lists the local production versions of publicly used and maintained modules. The default names of the files for the JS data set itself are hard coded in JSMON. The name of the JS data set can be specified by using the JS_DATASET key word in the JSIN file.

On all systems, for all types of data sets, the names for the tree and data files end with, system dependent suffixes. Under VAX/VMS we use the file types ".TREE" and ".DATA". Similarly, under IBM/MVS we end the file names with ".J80TREE" and ".J80DATA". The rest of a file name depends on the data set and its type. The suffixes can be set for a given installation in the JSPARM include file.

3.2.3.1 NONSTANDARD DATA SET FILES

The nonstandard data set file names consist of a path specifier, the data set name, and the tree or data suffix. Under VAX/VMS a path specifier is a directory name, optionally prefixed with node and/or device specifiers. The file names for the nonstandard data set MYDATA might be something like:

```
SRLUSER1:[T7980.JOSHUA]MYDATA.TREE
SRLUSER1:[T7980.JOSHUA]MYDATA.DATA
```

The IBM/MVS path specifier is an upper level name, in most cases at Savannah River a user id. Thus, a pair of data set files might be:

```
T7980.MYDATA.J80TREE
T7980.MYDATA.J80DATA
```

A nonstandard path is used by the JS routines to find the files to be associated with a given nonstandard data set. The implementation of paths in the JS routines is straightforward. The path specifiers are maintained in an array. The file names are constructed using the concatenation:

```
path_specifier // data_set_name // tree_suffix
path_specifier // data_set_name // data_suffix
```

Additionally, under IBM/MVS we concatenate a "." between the path specifier and data set name.

3.2.3.2 STANDARD DATA SET FILES

The JS routines use a *standard table*, which is a JOSHUA record, to find the files for standard data sets except, as noted above, JS. A standard record contains for each standard data set a complete file name excluding the suffixes. Since the JS routines construct complete file names for standard data sets by adding only the tree and data suffixes, the JOSHUA data set name does not have to be a part of the file name. This allows different versions of the standard data sets to be easily maintained in files with different names. A standard table may be supplied by the system or users may supply their own.

3.3 CREATION OF JOSHUA DATA SETS

The creation of a JOSHUA data set requires the creation of two FORTRAN direct access files and the initialization of the tree header record. The creation of files is handled differently on different systems.

3.3.1 VAX/VMS

Under VAX/VMS a FORTRAN OPEN with STATUS of NEW is all that is required to create a file. Since the VAX/VMS file naming structure includes version numbers, the new file will be given the next highest version number.

For a VAX/VMS file used in FORTRAN, we consider the *file name* to be the name used to open the file and the *allocation name* to be the full file name used by the system. We obtain the allocation name by using a FORTRAN INQUIRE on NAME after an OPEN.

VAX FORTRAN direct access files grow as needed, so no preallocation of space is necessary.

3.3.2 IBM/MVS

Under IBM/MVS, JOSHUA data sets are created in two different ways depending on whether the data set is temporary (exists only for the duration of the job) or permanent. Temporary data sets are placed on VIO (virtual IO) and are created when they are allocated. Permanent data sets are created as VSAM (Virtual Storage Access Method) clusters using the IBM utility IDCAMS which is called by the JOSHUA routine JSMKVS.

There are two ways that files can be allocated under MVS: preallocation, such as with DD statements in JCL or with ALLOCATE statements in TSO, and dynamic allocation. In JOSHUA, we use dynamic allocation for the files associated with JOSHUA data sets so that new JOSHUA data sets can be created while a job is in progress, old data sets can be easily accessed, and users do not have to be concerned with the details of data set organization, record length, block size, etc. IBM/MVS dynamic allocation is done in the JS routines using SVC99 [7].

Normally, the amount of space allocated to a permanent JOSHUA data set need not concern the user as VSAM files will extend as needed. However temporary data sets (using VIO) do not extend. The amount of space allocated to the tree and data files can be controlled by use of the WRITE_SPACE and SCRATCH_SPACE keywords in the JSIN file and by the parameters passed to the JSSEWR routine. Defaults for WRITE_SPACE and SCRATCH_SPACE are defined in the JSJBD block data before installation.

At Savannah River, JOSHUA data sets are placed by default on SYSDA, a generic name for direct access devices locally defined for short term storage. The location of JOSHUA data sets can be controlled by use of the JSIN keyword `WRITE_VOLREF` and by the parameters passed to JSSEWR. The defaults for VOLREF can be set for your installation by modifying the initialization of the JSIN Table in the JSJIBD block data.

For an IBM/MVS file used in FORTRAN, we consider the *file name* to be the fully qualified data set name for the file and the *allocation name* to be the data definition name (ddname) used to allocate the file.

3.3.3 SIZE LIMITATIONS

If N is the largest FORTRAN integer on your system, you can easily see that a tree catalog can, at least in theory, have as many as N nodes, and each JOSHUA record can be as long as N bytes. Similarly, a data file can have as many as $(N-1)+N/RECL$ physical records, where RECL is the data file record length in bytes, if the N th JOSHUA record is N bytes long. Since N is usually quite large, in practice the sizes of JOSHUA data sets are limited only by the direct access storage resources of your system.

3.4 THE JOSHUA DATA SET TABLE

The *JOSHUA Data Set Table* is used in the JS routines to store and manage the information about the JOSHUA data sets known to a job.

3.4.1 CONTENTS OF THE DATA SET TABLE

The JOSHUA Data Set Table is stored in the common blocks `/JSCJDS/` and `/JSNJDS/`, which are defined in the JSJDST include file. It consists of the following fields:

- Data set name
- Creation date and time
- Revision data and time
- Data set comment
- Tree file name
- Data file name
- Tree allocation name
- Data allocation name
- Unit to which tree file is connected for I/O, or 0 if none
- Unit to which data file is connected for I/O, or 0 if none
- Type of data set (0 = Nonstandard, 1 = Standard)
- Protection assigned to data set (1 = READ ONLY, 2 = READ/WRITE)
- Number of last used tree node
- Number of last used physical record in the data file

In the Data Set Table, a tree or data *file name* is a name that can be used to reference the file at the command level on a given computer system. A tree or data *allocation name* is an unambiguous name that can be used to reference the file in a FORTRAN OPEN statement. Under VAX/VMS an allocation name is a complete file specification including device and version specifiers, as returned by a FORTRAN INQUIRE; a file name may be the same as an allocation name but typically might not include version, directory, or device information. Under IBM/MVS an allocation name is a data definition name (ddname); a file name is a complete system level name.

3.4.2 STORAGE AND USE OF DATA SET TABLE

The entries in the table are stored in a binary search tree, keyed on the data set name. Each entry carries with it two extra integer fields that index the left child and right child respectively of the entry. New entries are made to the Data Set Table in such a way that the left child of any given entry is lower alphabetically than the parent and the right child is higher. The binary search tree for a sample Data Set Table is illustrated in Figure 3.4.

Figure 3.4 Binary search tree for a sample JOSHUA Data Set Table.

This storage method allows data sets to be easily added to the table as they are encountered, and at the same time provides a quick look up method.

3.5 SPECIAL DATA SETS

During a JOSHUA job, certain data sets may be given special designations. The possible designations are WRITE, USER, user SCRATCH, and system SCRATCH. Like all other data sets, these data sets are stored in the Data Set Table. For easy access, their indices are maintained in pointers, respectively WRITPT, USERPT, SCRTPT, and SYSCPT, stored in the common block /JSNJDS/.

3.6 THE READ HIERARCHY

The *read hierarchy* defines a sequence of JOSHUA data sets to be searched for a record or subtree in an output operation. The read hierarchy consists of a list of as many as 8 elements. An element maybe a data set name or the special literal "STD", used to designate the standard data sets.

Each job has a *default read hierarchy*, initially specified in the JSIN file with the READ_HIERARCHY keyword. If no READ_HIERARCHY keyword is used, the sequence used is:

the WRITE data set, if any
the USER data set, if any
"STD"

During execution, the *current read hierarchy* can be set to some sequence other than the default and then later set back to the default.

The routine JSSEDR sets the default read hierarchy, and JSSEHR changes the current read hierarchy. These routines may be passed either a list of elements or a blank. If JSSEDR receives a blank, the default read hierarchy will be set to the list given just above. If JSSEHR receives a blank, the current read hierarchy will be set to the default read hierarchy.

The default read hierarchy for a job is set by calling JSSEDR from JSMON based on the JSIN file, and written in the boot record for use by each module (see Chapter 5). The JOSHUA statement DEFINE DATASET, which can be used in an application module to change the current read hierarchy, is translated by the precompiler to a call to JSSEHR with the specified list of elements. The JOSHUA statement DEFAULT DATASET, which restores the current read hierarchy to the default, also translates to a call to JSSEHR, but with a blank list.

If JSSEDR is called from a module the default read hierarchy is changed for that module, but only for that module. Every other module will still get the initial default read hierarchy recorded in the boot record by JSMON.

3.6.1 MANAGEMENT OF READ HIERARCHY INFORMATION

Much like record names, the list of read hierarchy elements is passed to the JS routines as a single string of concatenated elements, each a data set name or the literal "STD", if necessary padded on the right with blanks to obtain 8 characters. This string is split into 8 character pieces, and each piece is converted to upper case and checked to see if it a legal name. For each element that is not "STD", the Data Set Table is searched. For calls to JSSEDR, if the data set is not in the Data Set Table or if it is not nonstandard, an attempt will be made to add it to the table or find a nonstandard data set by that name. JSSEHR requires that the data set already be in the table. If it is not, a fatal error occurs.

The read hierarchy is stored internally as an integer array, each element being the index of an entry in the Data Set Table or 0 for "STD". Separate arrays are maintained for the default read hierarchy and the current read hierarchy. Both arrays are defined in the JSJDST include and maintained in the /JSNJDS/ common block.

4 JOSHUA DATA BASE OPERATIONS

The Input/Output (I/O) operations that can be performed on the JOSHUA data base are:

INPUT

- Write a record
- Rename a record or subtree
- Delete a record or subtree
- Delete the user SCRATCH data set

OUTPUT

- Read a record
- Print a record using the format specified by a template
- List the records in a subtree
- List the children of a subtree

INPUT AND OUTPUT

- Copy a record or subtree

All of these operations initially involve searching for a JOSHUA record or subtree in the tree catalogs. Then, depending on the operation, further use of the tree files or data files or both is made. Writing might involve adding nodes to the catalog and/or updating the integer fields in a leaf node. Renaming and deleting records or subtrees involves changing the tree catalog, while listing operations involve only reading the catalog file. Reading and printing require only output from data files. A copy is simply a combination of read and write operations.

The specific JS routines that are called to perform I/O operations and their calling sequences are given in Chapter 2. In the remainder of this chapter, we will provide detail about how these operations are handled by the JS routines.

4.1 SEARCHING THE TREE CATALOGS

To find a name in the tree catalogs, first the namestring is checked and prepared for the look up. Then, one or more catalogs, depending on the type of operation to be performed, is searched for a matching name. The steps involved in these tasks are detailed in this section.

4.1.1 SETTING UP A NAME STRING

As explained in Chapter 3, a JOSHUA record or subtree name is passed to the JS routines in string form, that is, as a single string of concatenated qualifiers, each 8 characters long. The routine JSSENM sets up the name for use by the JS routines. JSSENM calls JSCVSA to break the name string into qualifiers and place them in an array. JSCKNM converts any lower case letters to upper case and checks the overall validity of each qualifier. If the name is for a record or subtree in the system SCRATCH data set, the .SYSCRAT qualifier is added to the beginning of the name. Also, if the name is supposed to be a complete record name, the "..leaf.." qualifier is added at the end. At the completion of JSSENM, the JOSHUA name string has been installed in the JOSHUA Name Table.

4.1.1.1 JOSHUA NAME TABLE

The *JOSHUA Name Table* is used by the JS routines to store information about JOSHUA namestrings currently involved in I/O operations. There are three entries in the JOSHUA Name Table, one each for reading data records, reading template records, and writing records. Three entries are used since some I/O operations involve more than one namestring. For example, to print a record we need to read a data record and at the same time read formatting information from a template record. Copying one record or subtree to another requires reading from one record and writing to another.

Each entry in the JOSHUA Name Table consists of a name array of the JOSHUA qualifiers of the record or subtree name for which I/O is being performed and the following fields:

- Index in the Data Set Table of the associated data set
- Count of the qualifiers in the name array
- Index of the qualifier in the name array that begins the record or subtree name
- Index of the qualifier in the name array that ends the record or subtree name

4.1.2 CHOOSING DATA SETS TO BE SEARCHED

The routine JSLKUP decides which tree catalog will be searched next for a name path that matches the JOSHUA name most recently entered in the JOSHUA Name Table. JSMTCH is called by JSLKUP to actually perform the next search.

If "SCRATCH" is the first qualifier in the search name, then only the user SCRATCH data set is searched. Similarly, if the first qualifier is ".SYSCRAT", the internal JS name for the system SCRATCH data set, then only that data set will be searched.

If neither scratch data set is explicitly referenced by the first qualifier, then the logic for searching catalogs depends on whether the operation to be performed is input or output. For input operations, which include those that change the tree file or the data file or both, that is, write, copy, rename, and delete, the WRITE data set is the only one considered. In all other cases the current read hierarchy determines the search order.

To set up for the next call to JSMTCH, JSLKUP uses the logic stated above to make modifications to the JOSHUA Name Table to indicate which data set should be searched. The indices for the user and system SCRATCH and WRITE data sets are set using the pointers to the data set table described in Chapter 3, respectively SCRTPT, SYSCPT, and WRITPT. When the read hierarchy is in use, the indices of nonstandard data sets are available directly from the current read hierarchy list. In the case when STD is next in the read hierarchy, the first qualifier in the JOSHUA name is used as the data set name and the rest of the qualifiers are used as the record name. The routine JSFIDS is called to find the index of the standard data set in the Data Set Table.

The exact list of qualifiers to be matched is specified by the indices in the Name Table of the beginning and ending qualifiers of the record or subtree name in the table name array. JSLKUP sometimes adjusts the index to the first name. For the scratch data sets, the index of the first qualifier to be matched in the catalog is set so that the "SCRATCH" or ".SYSCRAT" qualifier is skipped, since neither is part of the record name in the catalog file. The second qualifier also begins the record or subtree name to be matched when the first qualifier is used as the name of a standard data set.

4.1.3 FINDING A MATCH FOR A NAMESTRING

Finding a given record or subtree name in a tree catalog can be thought of as finding a *path* through the tree in which each qualifier matches some tree node on the path. A qualifier *matches* a node if the names are identical or, when wildcards are being used, if the node name begins with the "?" wildcard character.

Of course, there is a specific, logical method used to traverse the tree to find a matching path for a sequence of qualifiers, $qual_1, qual_2, \dots, qual_n$. We begin at node₁. If $qual_1$ matches node₁, we try to match $qual_2$ with one of the children of node₁, looking in order oldest to youngest. If $qual_1$ does not match node₁, we examine the siblings of node₁, oldest to youngest. In general, each time a match is successfully made between $qual_i$ and node_j, we next compare $qual_{i+1}$ with the children of node_j. If at any point no match can be found for a given qualifier among the appropriate siblings, the entire search fails. An important feature of this traversal method is that since a wildcard node is always maintained as the youngest sibling, any exact match is discovered before a wildcard match. Figure 4.1 shows a tree with a matching path marked.

Figure 4.1 The tree illustrated in Figure 3.3 with the path through the tree for the record name A.B2.C.E.leaf_node marked and the matching nodes shaded.

It is possible when wildcard matching is in effect that a matching record might exist but not be found. If we are searching for the record to match the name TEMPLATE.A.C and the tree is as illustrated in Figure 4.2, successful matches will be made at levels 1 and 2. At level 3, however, the only child of A does not match, but we do not back up to level 2, so the matching record TEMPLATE?.C will not be found.

Figure 4.2 The tree structure of a sample data set with levels labeled. In this tree, a template to match the record A.C will not be found.

4.1.3.1 TREE AND PATH TABLES

Under the assumption that JOSHUA record and subtree names will often be referenced approximately in order of creation, we keep the most recently used catalog information in memory to improve the speed of searches. Copies of the most recently used tree node records are stored in the *Tree Table* to decrease the number of disk accesses to tree files. The most recently followed paths through the trees are stored in the *Path Table* to further expedite searching. When looking for a given record, first, the longest matching path in the Path Table is found, then any additional comparisons to nodes are made using the Tree Table. If the record with a needed node is not in the Tree Table, then it is added to the table, replacing the least recently used record.

The Tree and Path Tables are defined in the JSTRPA include file. The number of tree records in the Tree Table is MXTRCT, defined in the JSPARM include file. Similarly, MXPACT defines the number of paths in the Path Table, and PANDCT defines the maximum number of nodes in each path.

Each entry in the Tree Table corresponds to a single tree node record. It stores the name and index fields of the nodes in the record as well as the following information:

- Index to the Data Set Table for the corresponding data set
- Physical record number in the tree file
- Modification flag
- Time when last used

The modification flag is turned on when a node in the entry is modified. When the entry is removed from the table, if the record has been modified it is written to the data set and physical record specified in the table.

Each entry in the Path Table consists of a sequence of nodes. To follow a path for matching we need to know whether each node is a sibling or a child of the previous node in

the path. Thus, in addition to the name and index fields of each node, there is an integer tag field to mark the node type. The tags are 1 for child and 2 for sibling, which correspond directly to the indices of the first and second integer fields in internal nodes. Each path table entry also has the following fields:

- Index to the Data Set Table for the corresponding data set
- Number of nodes in the path
- Time when last used

To keep track of the time when an entry was last used, a "clock" is "turned on" at the beginning of the module. The clock is simply an integer counter initialized to the smallest integer on the system. Each time a tree or path entry is accessed the clock is "ticked" by calling JSTICK to increment the clock by one. The new clock reading is used to update the time last used field for that entry. When a new entry is needed in the Tree or Path Table and there are no free places, the least recently used entry is the one with the lowest time field and will be the one deleted.

4.2 CONTINUING TO TRAVERSE THE CATALOG

In most cases, both the list subtree and list children operations require further use of the catalog after the initial searching steps.

4.2.1 LISTING THE RECORDS IN A SUBTREE

To list the records in a subtree, we make a *preorder traversal* to find each leaf and thus each record name. This means that, starting from the last node matched in the subtree name, we follow one tree branch all the way to a leaf before investigating a side branch. We do stop to record the name of each node as it is visited before (*pre*) continuing. At a leaf, the recorded node names plus the subtree name comprise the corresponding record name. When we reach a leaf, we back up to the closest side branch and repeat, continuing the process until we are back where we started. For JOSHUA trees, to follow a branch to the leaf means following successive child pointers. Backing up to the closest side branch means backing up to the closest parent, grandparent, etc. that has a younger sibling. Backing up is easy if we also save the node indices when we record the node name. The common blocks /JSCLIS/ and /JSNLIS/, defined in the JSLIST include, are used to record the node information during a list subtree operation, so the record name most recently found is always available, and so that backing up is easy.

For the tree in Figure 4.1, the records listed in preorder are:

- A.B1.C
- A.B2.C.D
- A.B2.C.E
- A.B3
- TEMPLATE.A.B2.C.?
- TEMPLATE.A.?

4.2.2 LISTING THE CHILDREN

To complete a list children operation after the subtree name has been found in a tree catalog, we proceed to the oldest child of the last matched node, then follow the sibling list of that child.

The subtree name passed to the list children operation can be blank to request the first level children in a tree catalog. The oldest child of the "last matched node" in that case is simply the first node in the first data set in the read hierarchy. When STD is the first element in the read hierarchy slightly different logic is used. No use of a catalog is necessary. We simply list all the standard data sets from the Data Set Table.

The children of the subtree TEMPLATE.A in the tree illustrated in Figure 4.1 are B2 and ?.

4.3 REVISING THE CATALOG

Several operations require revising the tree catalog.

4.3.1 ADDING NAMES

When a write operation, by itself or as a part of a copy operation, adds a record to the JOSHUA data base, the routine JSADND is called by the Write Begin routine, JSWRBG or JSWSBG, to add the nodes needed to catalog the new name. The JSMTCH routine tells us where in the catalog to begin adding the nodes and whether the first node to be added should be a child or a sibling. After the first new node, additional nodes are added as successive generations of children. The last node added is a leaf node.

Wildcard nodes, those named with the first character a "?", are always placed at the end of the sibling list at their level. Nonwildcard nodes are also added at the end if no wildcard sibling exists, otherwise they are added as the next to youngest so that the wildcard node is always maintained in the youngest position. This is so that when the catalog is being searched, exact matches will be made before wildcard matches.

If you examine the internal node index fields in the tree in Figure 4.1, you can see that the records were added in the following order:

```
A.B1.C
TEMPLATE.A.?
TEMPLATE.A.B2.C.?
A.B2.C.D
A.B2.C.E
A.B3
```

Note that this list differs from the preorder list of the tree, given in Section 4.2.1.

4.3.2 DELETING

In most cases, deleting a subtree involves simply renaming its last qualifier to the special qualifier ".DELETED". Internally, in fact, the delete routine, JSDLTE, calls the rename routine, JSRNME.

There is one special case in the delete operation, that of deleting the user SCRATCH data set. The files for the data set are not actually deleted, they are reused. To delete user SCRATCH, we make sure it is open, then reset its entry in the Data Set Table to indicate zero nodes and zero physical records. We also reset its creation and revision dates and times.

4.3.3 RENAMING

To rename a node, we make sure that the record that contains it is in memory as one of the Tree Table entries, change the name, mark the Tree Table entry modified, and update the Path Table.

4.4 READING FROM AND WRITING TO DATA FILES

Each entry in the JOSHUA Name Table has an internal *I/O buffer*, a character string in which one character is assumed to occupy one byte. Each buffer consists of a main section, long enough to hold one physical record from a data file, and a pad section. The pad section is for "wrap around" when a single data element spans physical records. It is long enough to hold the longest numeric data element on a given system, under VAX/VMS and IBM/MVS a double precision or complex datum, 2 numeric storage units (words) long. We do not need to provide pad space for character data since we can "wrap around" character data in the character buffers using standard FORTRAN substring expressions.

Each I/O buffer carries with it the following fields:

- Index in the data set table of the associated data set
- Physical record number where the record begins in the data file
- Number of bytes in the JOSHUA record
- Number of bytes read or written so far
- Number of physical records in the JOSHUA record
- Number of physical records read or written so far
- Index of the next unread or unwritten byte in the buffer
- Index of the last data byte in the buffer during a read
- In use flag

As described in Chapter 2, the Read and Write Begin routines prepare a record for I/O. The Read and Write Value routines then read or write a given number of a given type of data elements. The values are consecutively fetched from or stored to the appropriate I/O buffer until the buffer is "exhausted". At that point in a read operation the next physical record, if any, from the JOSHUA record is read into the buffer by JSRDBF. In a write operation the buffer is output to the data file by JSWRBF, and the buffer is available for more input.

Figure 4.3 A read I/O buffer.

Figure 4.4 A write I/O buffer.

The JSRDBF and JSWRBF routines manage the pad sections of the I/O buffers, see Figures 4.3 and 4.4, since a data element can span physical records. When an I/O buffer is exhausted during a read operation, an incomplete element may remain in the buffer, the rest of the element being in the next physical record. JSRDBF moves any partial element to the end of the pad section, which precedes the main section of the buffer, fills the main section with the next physical record, and adjusts the pointers to the buffer. During a write operation, the element that "exhausted" the buffer may have overflowed from the main section into the pad section. After the main section is written to the data file, any overflow

in the pad section is moved to the beginning of the buffer and the pointers are adjusted so that it will be included at the beginning of the next physical record.

Storing and fetching character values in the I/O buffers is straightforward for the Read and Write Value routines. However, to handle numeric data in these character buffers, we use nonstandard FORTRAN, since as noted in Chapter 3, ANSI FORTRAN 77 [1] does not define a relationship between character and numeric storage. In particular, we use our knowledge of the relationship between numeric and character storage for VAX/VMS and IBM/MVS systems. Fortunately, both VAX FORTRAN [8] and IBM VS FORTRAN [13] allow, as an extension to standard FORTRAN, character and numeric data types to be equivalenced.

To "read" a numeric value of a type that is *n* bytes, we copy the appropriate *n* bytes (characters) from the I/O buffer to a local character variable that is *n* bytes (characters) long. Using nonstandard FORTRAN, that local character variable is equivalenced to a numeric variable of the type being read. Then, a standard numeric assignment statement copies the value from local storage to the caller's return argument.

Similarly, to "write" a numeric value of *n* bytes, we store the numeric value in a local numeric variable that has been equivalenced to a character variable of *n* bytes. A standard character string assignment is then used to copy the value from local storage to the I/O buffer.

4.4.1 COPYING

To copy a subtree of records, JSCOPY calls uses the List Subtree routines with the `list_flag` set to traverse the subtree to be copied. First, JSLIBG is called to initiate the operation. Then, each record in the subtree is found and prepared for reading by successive calls to JSLINX. As each record is found by JSLINX, JSCOPY calls JSWRBG to initiate a write operation for the copy, then reads from the source record and writes to the new copy, one physical record at a time. When all the physical records have been copied, JSWREN is called to complete the write operation for the copy of the record, and JSLINX is again called to get the next record to be copied.

4.4.2 PRINTING

The JSRNT routine displays a JOSHUA record or subtree using template records for formatting information. Like JSCOPY, JSRNT uses the List Subtree routines with the `list_flag` set to find the records to be printed. As each record is found, a matching template record is found and read using the JSRTBG, JSRTxx, JSR2xx, and JSRTEN routines. The routines used to interpret a template record and create a display are from the Display Data routines from the JOSHUA Terminal System.

5 THE MONITOR PROGRAM, JSMON

The monitor program, JSMON, establishes a JOSHUA execution environment, and uses JSEXEC to execute the first module in the job. That module in turn may execute other modules. At the completion of module execution, control is returned to JSMON for job termination.

The parameters for a given job are specified by placing a sequence of keywords and associated values in a JSIN file. Under VAX/VMS these keywords and values can optionally be specified as JSMON command parameters. The keywords and values are described in *The JOSHUA Users' Manual* [2].

In this chapter we will describe how the JSIN file is processed by JSMON and how JSMON supervises a JOSHUA job.

5.1 INVOKING JSMON

Under VAX/VMS, JSMON is invoked using the JSMON command. The command is created using the VMS Command Definition Utility [9] by invoking the DCL command SET COMMAND for the JSMON command definition file, JSMONversion.CLD.

Under IBM/MVS, JSMON is simply a FORTRAN program that is invoked in the standard manner. Examples of JCL to execute JSMON are in *The JOSHUA Users' Manual* [2].

5.2 SPECIFYING JOB PARAMETERS

The syntax for a JSIN file follows a subset of the Little Input Language (LIL). A set of JS routines, named JSLILx and documented in Appendix A, are used to parse a sequence of LIL statements. Information obtained from the JSLILx routines is used to build a JSIN Table which is in turn used by JSMON to set up the JOSHUA execution environment.

5.2.1 THE JSIN FILE

The JSIN file is used by JSMON as a sequential input file. Under VAX/VMS, JSMON uses the CLI\$PRESENT and CLI\$GET_VALUE routines [9] to get the name of the JSIN file, if any, from the JSMON command line. Under IBM/MVS, JSMON uses the ddname "JSIN" to OPEN the JSIN file.

5.2.2 THE JSIN TABLE

The *JSIN Table* consists of a value string and a keyword table. The values associated with JSIN keywords are packed in the *value string*. The keywords and additional information are stored in the *keyword table*. The table has one entry for each keyword and is ordered alphabetically by keyword. The fields in the table are:

- Keyword
- ON/OFF flag
- Minimum number of values that may be associated with keyword
- Maximum number of values that may be associated with keyword
- Length of a single value
- Actual number of values specified
- Index in value string of the beginning of first value

5.2.2.1 INITIALIZING THE JSIN TABLE

The block data JSJIBD is used to initialize the JSIN Table and provide default values for the keywords by initializing the ON/OFF flags and the value string.

At run time the JSLILx routines are called from JSJIRJ to read the JSIN file and modify the initial values to meet the user's specifications.

Under VAX/VMS after the JSIN file, if any, is read and processed, information about the command line qualifiers and values is obtained by calling the CLI\$PRESENT and CLI\$GET_VALUE command language routines [9] and added to the JSIN Table in the JSVXJI routine. Since the command line is processed after the JSIN file, the command line parameters take precedence over those specified in the JSIN file.

Under IBM/MVS after the JSIN file, if any, is read and processed, information from the PARM field of the EXEC statement is obtained by calling JSMVGP and added to the contents of the JSIN file.

5.2.2.2 PRINTING THE JSIN TABLE

The routine JSJIPR can be called to print the JSIN parameters in effect for a given job. The print format used creates a display that resembles a JSIN file. The characters "***JS**" are printed at the beginning of each line to flag that the output is from the JS routines. If MESSAGES is in effect, JSJIPR is called by JSMON to print the parameters in the message file.

5.2.2.3 USING THE JSIN TABLE

After the JSIN table has been fully initialized, it is used by JSMON to complete the definition of the JOSHUA job environment. The routine JSJIGV will return the values, if any, associated with a given keyword.

5.3 CREATING SCRATCH DATA SETS

Creation of JOSHUA data sets is described in Chapter 3. A system SCRATCH data set is always created by JSMON. User SCRATCH and WRITE data sets are created by JSMON depending on the JSIN job parameters.

5.3.1 THE SYSTEM SCRATCH DATA SET

The system SCRATCH data set should be used exclusively by the JS routines. It is named ".SYSCRAT", an invalid name according to the rules in *The JOSHUA Users' Manual* [2], so that it can be included in the Data Set Table like all other data sets, but cannot inadvertently be referenced by a user. The names of all other data sets are checked for conformance to the naming rules before they are added to the Data Set Table. The files associated with the system SCRATCH data set are given a unique base name which is generated by JSTAMP based on time and date. The method used by JSTAMP varies from system to system.

5.3.2 THE USER SCRATCH DATA SET

The user SCRATCH data set is always named "SCRATCH". The files associated with it by default will be given unique names generated by the JS routines, as for system

SCRATCH. In contrast to the system SCRATCH data set, however, the user SCRATCH can be named by the user by using `SCRATCH=name` in the JSIN file.

5.4 INITIALIZING THE DATA SET TABLE

The JOSHUA Data Set Table, described in Chapter 3, is initialized by JSMON. The routine JSADDS adds an entry to the Data Set Table and calls the routine JSALDS, which obtains allocation names for the data set files.

JSMON first call JSADDS to enter the system SCRATCH data set in the data set. Then, entries are made for the user SCRATCH, WRITE, and USER data sets as specified in the JSIN Table. Any additional nonstandard data sets listed with the `READ_HIERARCHY` and `NONSTD_DATASETS` JSIN keywords are then added to the table.

If a standard table record was specified for the job, it is read and processed in the routine JSADST. For each data set in the table record, the tree and data file names are constructed by concatenating the tree and data suffixes to the file specification in the record, and JSADDS is called to add the data set to the Data Set Table.

5.5 INITIALIZING SEQUENTIAL FILES

Several different types of formatted output may be produced by a JOSHUA job, depending on the JSIN job parameters:

- Terminal/console/log information (JSTERMO)
- Program output (JSOUT)
- I/O trace
- Messages
- Error trace

Files for this output are initialized by JSMON in the routine JSINSQ. The output unit to be used by each module for a given file is assigned in JSMON. Each job has unit 51 connected for terminal output. Also, each job has the file specified with the JSOUT keyword connected to unit 6 for program output. Each of the other types of output can be directed to a separate file or to the JSOUT file. The variables IOTLUN, MESLUN, and ERTLUN, defined in the /JSNDAT/ common block in the JSDATA include, store the unit numbers, or zero for no output, to be used for I/O trace, messages, and error trace output respectively.

The Sequential File Table, which contains the following fields, is used to store information about the output files:

- Name of file
- Unit associated for output

5.6 WRITING THE BOOT RECORD

A *boot record* contains the job parameters and file names to be used by each module executed in a given job. The monitor program, JSMON, creates this record, named `BOOT.INFO`, in the system SCRATCH data set at the beginning of each JOSHUA job. Each at the beginning of execution of each module in a job the JS routine JSMDDBG reads the boot record, and uses the data in it to initialize the JOSHUA environment.

The boot record contains the following information:

- Sequential File Table
- IOTLUN, MESLUN, and ERTLUN (see previous section)
- JOSHUA Data Set Table
- SCRTPPT, WRITPT, and USERPT (see Chapter 3)
- Default read hierarchy
- Nonstandard path
- Module path, if any
- Name of the module table record, if any
- Debug flag
- User message, if any

5.7 TERMINATING THE JOB

When control returns to JSMON at the end of a JOSHUA job, JSMON closes and deallocates all the files used by the job and prints a job statistics table, if requested. The system SCRATCH data set is always deleted by JSMON at the end of a job. The user SCRATCH data set is deleted or kept, based on the value specified with the SCRATCH_CLOSE_STATUS JSIN keyword.

The job is terminated by executing a FORTRAN STOP. Under IBM/MVS if the final completion code of the job is nonzero, a STOP 1 is issued.

5.7.1 PRINTING THE STATISTICS TABLE

At the termination of each module, a statistics record is written in the system SCRATCH data set (see Section 6.3.2). Depending on the JSIN job parameters, JSMON will call the routine JSPRST to use these records to print a job statistics table.

All the statistics records are in the same subtree in the system SCRATCH data set, so JSPRST uses the List Subtree routines with the list_flag argument set to read each record in the subtree. Since the records are added to the system SCRATCH data set in order of completion of execution, the statistics table will be written in this order.

6 MODULE EXECUTION

Under the JOSHUA system, one module, the *parent*, can execute another module, the *child*, and optionally pass the child a list of parameters. The parent is suspended when the child begins execution, and remains suspended until the child completes. Thus, at any given time during a JOSHUA job one module is in active execution and all other modules are in a suspended state.

The Execute Begin routine, JSEXBG, is used by a parent to prepare to execute a child module. The Execute routine, JSEEXEC, actually invokes the child module. In a child, the Module Begin routine, JSMDDBG, initializes the JOSHUA environment for that module. The Return routine, JSRETN, can be called to terminate execution and return a completion code and message to the parent. At the completion of an ERROR display, control is also returned to the parent.

Chapter 2 describes how to use the JS routines for module execution. In this chapter we describe how the routines work.

6.1 PREPARING FOR EXECUTION

The steps that a parent must take in preparation for executing a child are described in the following sections.

6.1.1 PASSING ARGUMENTS

JOSHUA supports one way parameter passing from a parent to a child module. The parameters are written in the record ARGUMNT.PASSED in the system SCRATCH data set by the parent module, and read from that record by the child module. The JOSHUA language makes parameter passing easy by providing an optional io-list on the EXECUTE and MODULE statements, see *The JOSHUA User's Manual* [2].

If parameters are to be passed from a parent to child, the `list_flag` is set in the call to JSEXBG. JSEXBG will initiate writing to the parameter passing record by calling JSWSBG. After JSEXBG has completed but before JSEEXEC is invoked, a sequence of calls is made to the JSWRxx and/or JSWlxx routines to actually write the data, and JSWREN is called to signal that writing is complete.

In the child module, if arguments are expected, the `list_flag` parameter for JSMDDBG is set. JSMDDBG will call JSRSBG to set up for reading. After return from JSMDDBG, calls may be made to the JSRDxx and/or JSRlxx routines to read the argument record. Reading must be terminated by a call to JSRDEN.

It is the users responsibility to make sure that the parent and child "argument lists" match in number and type.

6.1.2 SENDING THE BOOT MESSAGE

The *boot message* is a character string created by the parent module in JSEEXEC and "sent" to the child module. In the child module, the message is "received" in JSMDDBG, where it is split into pieces and used to help initialize the child for JOSHUA execution. In particular the message provides the following information:

Name of the system SCRATCH data set
Names of the tree and data files for the system SCRATCH data set
Name of parent module
Name of child module
Child module's internal id
Existence of an io-list record of arguments created by parent

Using this information, JSMDBG can find the job's system SCRATCH data set, add it to the Data Set Table, read the boot record, and complete initialization of the child module for execution.

Since module execution itself is system dependent, the method used to send and receive the boot message is different in each system. The following sections describe different message passing methods.

6.1.2.1 VAX/VMS

Under VAX/VMS a logical name string is used for communication of the boot message. The VAX/VMS system service routine SY\$CRELNM [11] is called by the parent module to create the logical name, JS_JOSHUA_LNM1, in the job logical name table. The child module calls SY\$TRNLNM [11] to translate the logical name.

6.1.2.2 IBM/MVS

Under IBM/MVS, as sequential data set on VIO is used for communication of the boot message. JSMON creates this data set and allocates it with the known ddname BOOTMSG. The parent module writes the boot message to file BOOTMSG in JSPUMS. The child module reads the boot message from file BOOTMSG in JSGEMS.

6.1.3 FINDING THE EXECUTABLE FILE

As explained in *The JOSHUA Users' Manual* [2], there are two methods that can be used to locate the executable files for JOSHUA modules: module paths and module tables.

6.1.3.1 MODULE PATH

A module path is an ordered sequence of places, path specifiers, where executable files might be found. For a module located on a module path, the name of the executable file consists of a path specifier and the name of the module itself. The exact method of combining the module name and a path specifier to form the complete name of an executable file varies from system to system. To search a module path, the routine JSFIFP uses path specifiers, in order, to construct executable file names and check their existence until a file is found

6.1.3.1.1 VAX/VMS

Under VAX/VMS a path specifier is a subdirectory name with optional device and node specifiers. All executable files are assumed to have a file type of .EXE., so the complete file name for an executable file is constructed as:

path_specifier // module_name // .EXE

Searching for the file simply involves doing a FORTRAN INQUIRE on the file name.

6.1.3.1.2 IBM/MVS

Under IBM/MVS an executable file is a member of a partitioned data set (PDS). For a module on a module path, a path specifier is either a partitioned data set name or a ddname allocated to a partitioned data set, and the member name is the module name. Unfortunately, we cannot use a VS FORTRAN INQUIRE to determine if a given member exists in a PDS which is allocated by ddname. Since we want to find modules at run time, we use SVC99 [7] to dynamically allocated the partitioned data set and additional system dependent routines to open and search the partitioned data set directory for the given member.

6.1.3.2 MODULE TABLE

A module table is simply a JOSHUA record that provides a look up table for module names and executable files. Under VAX/VMS the executable file name in the table record should not include a file type since the JS routines will add the .EXE suffix. A complete name, including partitioned data set and member names is, however, expected under IBM/MVS.

6.1.4 MAINTAINING FILE INTEGRITY

In order to maintain integrity of the JOSHUA data base, all READ/WRITE data sets are closed by a parent before a child module is invoked. In a child module, JOSHUA data sets are opened as needed. Similarly, when a parent resumes execution, data sets are reopened as needed.

All the sequential output files in the Sequential File Table are closed by a parent before a child module is invoked. The files are all reopened in the child module and later, by the parent, when the parent resumes.

When a sequential output file is reopened, we want to append additional output to the end of the file rather than write again each time from the beginning. In VAX FORTRAN [8] the nonstandard APPEND specifier for the OPEN statement does exactly what we want. Under IBM/MVS we require that sequential output files must be preallocated with a status of MOD, which takes care of appending additional output. So that the job's output starts at the beginning of the file, we REWIND a sequential output file when it is opened for the first time in JSMON.

6.1.5 CREATING A MODULE ID

In JSEXEC, the parent module creates a unique internal id for the instance of the child module about to be executed. The id is passed to the child in the boot message. The child uses the id to label its return record to the parent and to label its statistics record.

6.1.6 SUSPENDING A PARENT AND ACTIVATING A CHILD

The suspension of a parent module and activation of a child module cannot be done using ANSI FORTRAN 77 [1], thus each system on which JOSHUA is installed uses a different system dependent method.

6.1.6.1 VAX/VMS

Under VAX/VMS JSMON is run as a process or subprocess, depending on how it is invoked, and each applications module is run as a subprocess of its parent. While a child module is running, its parent hibernates; control is returned to the parent when the child

completes. The VAX/VMS run time library routine LIB\$SPAWN [12], called from FORTRAN, is used to spawn the subprocess in which the child module is invoked by a RUN command. The debug string flag is concatenated to the RUN command. Thus, if DEBUG was specified as a JSIN parameter, RUN will be in effect, and each module will be invoked under the VAX/VMS symbolic debugger if and only if it was linked with the /DEBUG qualifier. If NODEBUG was specified, each module is simply invoked using RUN/NODEBUG

6.1.6.2 IBM/MVS

Under IBM/MVS a child module is run as a subtask of the parent module. The ATTACH macro is used in a short assembler routine, JSMVAT, to create the subtask for the child module. In the same routine, the WAIT macro is used to suspend execution of the parent module until the subtask is complete. After completion, the subtask is removed from the system using the DETACH macro [4].

The PARAM parameter of the ATTACH macro is used to specify FORTRAN execution-time options for the child module. The execution time options, separated by commas, are placed in the parameter. The execution time options are used by FORTRAN, see Section 6.1.2.2.

Two execution time options, OCSTATUS/NOOCSTATUS and DEBUG/NODEBUG, are explicitly specified; defaults are accepted for the others [14]. The OCSTATUS/NOOCSTATUS option control whether the status specifiers on OPEN and CLOSE statements will be verified. We use NOOCSTATUS, because the JS routines perform their own tests for file existence before opening files. The DEBUG option, in effect if DEBUG is specified in the JSIN file, will cause the VS FORTRAN Interactive Debug to be invoked for each module [15].

One final thing needs to be mentioned concerning the interaction of the VS FORTRAN environment and JOSHUA. According to IBM documentation for VS FORTRAN Version 2.4 [14], you may have only one VS FORTRAN run time environment in your address space at a time. However, the independence of each JOSHUA module requires that each module have its own FORTRAN environment. Fortunately, it appears that this is only a hard and fast rule when running in *load mode*, that is when service routines are loaded during run time. In *link mode*, when all libraries needed by a code are included in the executable program, we are able to successfully create multiple FORTRAN execution time environments. We would prefer to use load mode because the amount of time necessary to create an executable program is reduced, library routines can be changed without requiring relinking, and an executable program will be smaller, and thus require less disk space for storage. Perhaps with later versions of VS FORTRAN we will be able to lift the link mode only restriction.

6.2 TIMING EXECUTION

The JS routine JSMCPU returns the amount of cpu time used by the current module, and JSJCPU returns the amount used by the entire job. Cpu timing is certainly not standard, but we originally assumed that a system dependent routine, JSXCPU, could be written to return the current reading of the cpu clock for the current module. That is, we assumed that a separate cpu clock runs for each module. If we take readings at two different times then we can easily compute how much has been used in the interval. We use a *time record*, named TIME.RECORD, in the system SCRATCH data set to communicate the cumulative job cpu time between modules.

6.2.1 VAX/VMS

On the VAX the run time library routine LIB\$GETJPI [12] can be used to obtain a cpu clock reading for a process or subprocess. At any given time, we consider the cpu clock for a module to be either on or off. When JSMON or a module begins, we turn the clock on, when it ends we turn it off. We also turn it off right before executing a child module, then turn it back on when we return. When the clock is turned on, its value is saved so that next time we look, we can compute the cpu time used over the interval since last looking.

Each module maintains a *module timer* and a *job timer*. These timers are updated intermittently to reflect the cpu time used respectively by the current module and entire job. Updates occur when the cpu clock is turned off. Updates are also made when either JSMCPU or JSJCPU is called if the cpu clock is running for the module.

The time record contains a single value, the job cpu time. It is always updated in JSEXEC just before a child module is executed. It is read in JSMDDBG in the child module, and used to initialize the job timer for that module. The time record is also updated in JSRETN at termination of a child module, then read again in the parent and used to update the parent's job timer.

6.2.2 IBM/MVS

The assumption mentioned above turns out to be true for VAX/VMS, which is not surprising since it was the target system at the time and the system about which we knew the most. Unfortunately, it is not clear that it is true for IBM/MVS. The problem with IBM/MVS is that the cpu timer runs for the whole job, not for each task in the job.

6.3 TERMINATING EXECUTION

There are several conditions under which execution of a child module can terminate:

- Normal, that is no error
- User detected error
- JS routine detected error
- FORTTRAN or operating system detected error

In the last situation, we lose control and return is made to the parent in a system dependent manner. For all the other conditions, return to the parent can and should be made through a JS routine to maintain integrity of the JOSHUA data base and sequential output files. If we return through a JS routine, we can also send completion information to the parent.

For normal completion, the routine JSRETN should be called with a zero completion code. The programmer of a module can use either of two methods of terminating if an error is detected. One method is to call JSRETN with a nonzero code. The other is to call JSERBG, and optionally JSERxx and JSEREN, to create an ERROR display. When the display is complete, JSJRET, an entry point in JSRETN, is called to terminate the child module. JSJRET is also called if the JS routines detect an error.

The same termination activities are performed by JSRETN and JSJRET. The entry point JSJRET was created simply so that we can determine the origination of the termination request. JSJRET is passed a flag value of 3 if it is called due to a JS routine detected error; it is passed a flag value of 2 if it is called at the end of an ERROR display.

The execution of a child module is actually ended by executing a FORTRAN STOP statement. The activities performed in preparation for termination are described in the following sections.

6.3.1 MAINTAINING FILE INTEGRITY

All modified tree node records in the Tree Table are written to the tree files, tree header records are updated, and all the data sets in the Data Set Table are closed. All the files in the Sequential File Table are closed.

6.3.2 UPDATING THE TIME RECORD

The module and job timers are updated one last time and the time record, named TIME.RECORD, is updated.

6.3.3 WRITING THE STATISTICS RECORD

At the termination of the module, a statistics record, named STATS.RECORD.*module_id*, where *module_id* is the internal module id for the child created by the parent, is written in the system SCRATCH data set. It includes the following information:

CHARACTER*8	Module name
DOUBLE PRECISION	CPU time used by module
INTEGER	Number of JOSHUA records read
INTEGER	Number of JOSHUA records written
INTEGER	Number of accesses to tree catalog files
INTEGER	Number of accesses to data files

The statistics records are read in JSMON at the completion of the job if the JSIN parameters specify that a statistics table should be printed.

6.3.4 WRITING THE MODULE COMPLETION RECORD

A completion record, named MSG&CODE.RETURNED.*module_id*, where *module_id* is the module id for the child created by the parent, is written to the system SCRATCH data set. The *module_id* is included as a qualifier so that each instance of each module executed has its own completion record, and so that the parent can find the correct completion record. This record contains the following information:

INTEGER	Completion code
INTEGER	Origination of termination request:
	1 by call to JSRETN
	2 by call JSJRET after an ERROR display
	3 by call to JSJRET after a JS detected error
INTEGER	Number of characters, n, in the message
C*n	Termination message

6.4 RESUMING THE PARENT

When the parent module resumes execution, it first restarts the module timer, reopens all the JOSHUA sequential output files, reads the time record, and updates the job cpu time. It then examines the system completion code from the execution of the child module, before

considering the JOSHUA completion code. The method of obtaining and interpreting the system completion code of course varies from system to system.

Under VAX/VMS a completion status for the spawn itself is available. A spawn might fail, for example, because the subprocess quota has been exceeded. There is also a separate run status available that gives the completion status of the subprocess. If the run status is not normal, then the module did not terminate through a JS routine and the JOSHUA error set by JSEXEC is:

6005, UNTRAPPED FATAL ERROR WHILE EXECUTING MODULE *module*

The run status will not be normal, for example, if a divide by zero or access violation caused termination of the child module. If both the spawn status and run status are normal then the JOSHUA completion record is examined.

Under IBM/MVS the return code from JSMVAT can indicate either that an error occurred creating the subtask for the child module or that the child module abended rather than terminating through a JS routine. If the return code from JSMVAT is 0, then the completion record is processed.

If the system completion codes are all normal, the completion record is read. If the JOSHUA completion code is zero, execution continues. If it not zero, a return is made to the alternate return label if the *a_flag* argument was set on the call to JSEXEC, otherwise, execution of the module is terminated, by a call to JSJRET.

7 THE TERMINAL SYSTEM

One of the original design goals of the J80 system was portability. This proved to be the most difficult with the J80 Terminal System (JSTERM). A practical Terminal System requires full screen editing capability and at the time J80 was written there was no one technology that provided full screen editing across multiple platforms. Hopefully new technologies such as X Windows and Motif will provide the standards needed to make a portable Terminal System, but such a system will have to wait until all the users have the ability to access it.

The approach we have taken in the current system is to write all the logic that is unrelated to full screen I/O in FORTRAN 77 and make it fully portable. This source is then combined with system dependent source which does the full screen I/O to make a fully functional Terminal System. On the VAX/VMS system this system dependent part is a VAX Text Processing Utility (TPU) program and on the IBM/MVS system it is an ISPF dialog written in FORTRAN which calls the ISPF library and uses ISPF panels and messages defined for the Terminal System.

7.1 JSTERM ON VAX/VMS

On the VAX/VMS system, the Joshua Terminal System consists of two separate programs, a FORTRAN program and a full screen text editor. The program is written in FORTRAN 77 and with minor modifications should run on any machine. The name of the FORTRAN program is JSTERM.EXE and consists of the main program JSTERM.FOR and its associated subprograms. JSTERM.EXE may be run by itself and it is useful to do so for the purposes of testing and debugging. The only input to JSTERM.EXE is lines of text read from standard input and the only output is lines of text written to standard output. The second part of JSTERM is the full screen text editor. The editor is written in the VAX Text Processing Utility (TPU) language (See the VAXTPU Reference for a complete description of TPU). The TPU procedures that are unique to JSTERM are in a file called JSTERM.TPU\$SECTION. This source is combined with the source for the JED editor which itself is combined with the standard VAX editor EVE to create the complete TPU program for JSTERM. When JSTERM is invoked the editor is executed and it creates a sub-process in which JSTERM.EXE is run. When a terminal system command is entered, the editor parses the command and sends a message to JSTERM.EXE requesting the appropriate function. The output from JSTERM.EXE is automatically placed in the main buffer of the editor. The process tree looks like this:

7.1.1 THE TPU PROGRAM

In EVE a command is defined by writing a TPU procedure with a name beginning with "EVE_", for example the procedure name for the CD command is EVE_CD. There are some procedures in JSTERM.TPU that do not begin with "EVE_", these are utility procedures which are called by the main procedures.

When a Terminal System command is entered the command is parsed by EVE and control is passed to the appropriate TPU procedure. After whatever preprocessing needed is done, the TPU procedure DO_TERM_COMMAND is called to pass control to the FORTRAN routines. The arguments for DO_TERM_COMMAND are a command verb, a character string containing any operands and a flag indicating whether the contents of the TPU buffer should be sent following the command.

DO_TERM_COMMAND sends a one line message to the FORTRAN program that indicates what function is to be performed. This single input line contains the function code and the operands in the following general format:

```
CMD  [ARG1  [ARG2]]
```

CMD is the command, such as "DD", which may be followed by one or two text arguments separated by spaces.

Some functions such as the save command "WD" require that the contents of the main buffer be sent also. After the FORTRAN program receives such a command it continues reading from the input unit, placing each subsequent line into its internal array until it detects an end of file condition. The TPU program sends the contents of its main buffer one line at a time via the TPU SEND command and signals that it is done via the TPU SEND_EOF command. After the FORTRAN program processes the command any output information is written to the output unit and automatically is placed at the end of the main TPU buffer.

A list of all the command functions appears in Section 7.5.

After control is passed back from the FORTRAN program, the TPU main buffer will have appended to it the output from the FORTRAN program. At the end of any output the FORTRAN program adds two or more lines. The first line is the full J80 record name for whatever operation was done and the second line is a completion message of the form:

```
LLLLCCctext_string
```

Text_string is a short text message stating whether or not the command completed successfully. LLLL is a 4 digit line number and CCC is a 3 digit column number within the buffer that was sent from TPU. If no error occurred they will be zero. In the event of an error they will indicate where the error was detected so that TPU can place the cursor at the location of the offending text.

After this one line status message an additional multi-line error message from the JS system routines may be written. If the last line is of the form "%%%nnn", where n is a numeric digit, then the nnn lines prior to the last line form a message from the Joshua System routines and are moved from the main buffer to the error buffer.

At this point the record name is moved to the record name line, the status message is displayed in the message window and the any remaining lines constitute the main output of the function and are left in the main buffer for display.

7.2 JSTERM ON IBM/MVS

The IBM/MVS version of JSTERM differs from the VAX/VMS version in that it is a single program. The full screen editing functionality is provided by the ISPF facility on the IBM. The FORTRAN 77 standard program is linked with several system specific routines along with the ISPF library to form an ISPF program. This program along with a few other ISPF components forms an ISPF dialog. The other components consist of panels, messages and clists. Panels are screen definitions with input and output fields defined. Messages are lists of text strings with identifiers which are used to display error and status information. Clists are command procedures that consist of TSO commands. The panels are stored in 'JSS.SPF.ISRPLIB', the messages are stored in 'JSS.SPF.ISRMLIB' and the clists are stored in 'JSS.SPF.ISRCLIB'.

The JSTERM dialog is invoked by the NJSTERM clist from a site specific menu. NJSTERM may take 5 parameters passed from the menu. These parameters are 5 lines of text which are written to a temporary file and used as the JSIN file. NJSTERM also defines the ISPF panel and message libraries, allocates the input and output files and invokes JSMON.

JSMON is invoked with a parm list that contains a list of JSIN keywords. The keywords are separated by commas with one exception. A semi-colon may be used in place of one of the commas or at the beginning or ending of the list. This semi-colon denotes where in the JSIN list the information from the JSIN file should be inserted. Think of the JSIN files as containing user specified JSIN information and the parm list as containing system specified JSIN information. Information in the parm list before the semi-colon may be overridden by the user, information appearing after the semi-colon may not. Information that you don't want the user to change includes the module to be run, in this case JSTERM. Also NOUSER is specified after the semi-colon due to the fact that the IBM/MVS version of JSTERM makes a temporary copy of the user data set and thus any changes made to it would be lost.

The purpose of the keywords in the parm list appearing before the semi-colon is to change some of the normal default values. The PDS that contains JSTERM is specified by the MODULE_PATH keyword, the version of the STD database to be used is specified by the JS_DATASET keyword. IMMEDIATE_UPDATE is specified so that the user's data sets are not corrupted if JSTERM terminates abnormally. Also WRITE_STATUS = OLD is specified so that if the user specifies a WRITE data set, an existing one will be used.

The main program JSTERM passes control to the subroutines JSJTGC which contains the main control loop. Before entering the control loop JSJTGC defines a 'link' between several of the variables in the FORTRAN program and the corresponding ISPF variables. For example, the command line field on the main panel is defined to be the same as the FORTRAN variable COMLIN. Other information mapped includes such things as the message field and the cursor position. The main display area is defined as an ISPF dynamic area and mapped to the FORTRAN variable DAREA.

The main control loop in JSJTGC handles parsing of each command line from the user as well as scrolling operations.

7.3 STORAGE FORMAT OF TEMPLATE RECORDS

When a template is first created the list is parsed and the structure of the list stored in two integer tables. These two tables, and the text of the template including the I/O list and type declarations, all represented in character format form the template record. The syntax of the template I/O list, type declarations and display image are described in The JOSHUA (J80) User's Manual Section 4.5.3.

The first table is the Element Table and it is a 4 x N array that contains one entry for each variable in the list and two entries for each loop (one for the beginning, and one for the ending of the loop). The first column in the table identifies the type of entry (integer, real, index...). The second column is one of four things, for characters it is the length, for integers which are used as loop limits it is a -1, for loop beginnings and endings it is a pointer into the Loop Table, for all other cases it is a 0. The third and fourth columns are the beginning and ending positions, respectively, of the name of the element in the template list. The general form looks like this:

ELEMENT TABLE

ELEMENT_ID	PTR,LEN,-1 OR 0	NAME BEGIN POSITION	NAME END POSITION
------------	-----------------	---------------------	-------------------

The possible entries for ELEMENT_ID are:

- 1Loop Begin
- 0Loop End
- 1Integer
- 2Real
- 3Double Precision
- 4Character

The second table is the Loop Table and it is a 9 x M array and contains an entry for each of the M loops in the list. The first column is a pointer to the entry in the Element Table for the beginning of the loop, the second column is a pointer to the entry for the index of the loop. The next 6 columns consist of two columns each for the lower limit, the upper limit and the increment of the loop. The last column is 1 if the index for the loop was explicitly given and zero otherwise.

LOOP TABLE

POINTER TO LOOP BEGIN ENTRY	POINTER TO INDEX OF LOOP	LOWER LIMIT INFO		UPPER LIMIT INFO		INCREMENT INFO		KEY / ENUMERATION
		L1	L2	L1	L2	L1	L2	

For each of the loop limits there are two possibilities.

- 1) The loop limit is a constant in which case L1 is -1 and L2 is the value of the constant
- 2) The loop limit is a variable in which case L1 is 0 and L2 is a pointer to the entry in the element table for that variable.

Example: Below is an example of a template and the tables that would be generated for it.

N,M,(NAMES,(VALUES,I=1,4),ITEM,J=1,N),((LIST1,AMOUNT1,X,Y,Z,J=1,M),
(LIST2,AMOUNT2,X,Y,Z,J=1,M),I=1,N)

```

INTEGER      N,M
REAL          AMOUNT1,AMOUNT2
DOUBLE        VALUES
CHARACTER     NAMES*8,ITEM*4,LIST1*10,LIST2*10

```

THIS IS THE FIRST LINE OF THE DISPLAY IMAGE

VALUE OF N IS :II VALUE OF M IS :II

TABLE A

```

.BEGIN
NAME:01 IS :AAAAAAAAA :E.EEEEEEE :E.EEEEEEE :E.EEEEEEE :E.EEEEEEE :AAAA
.END

```

```

.BEGIN

```

SAMPLE NUMBER :003

=====

LIST 1

```

                                NAME                                AMOUNT
.BEGIN
:04      :AAAAAAAAAAAA                                :FFFF.FF
X =:F.FFFF      Y =:F.FFFF      Z =:F.FFFF
.END

```

LIST 2

```

                                NAME                                AMOUNT
.BEGIN
:05      :AAAAAAAAAAAA                                :FFFF.FF
X =:F.FFFF      Y =:F.FFFF      Z =:F.FFFF
.END
.END

```

Element Table

1	-1	1	1
1	-1	3	3
-1	1	5	5
5	8	6	10
-1	2	12	12
3	0	13	18
0	2	20	20
5	4	27	30
0	1	32	32
-1	3	39	39
-1	4	40	40
5	10	41	45
2	0	47	53
2	0	55	55
2	0	57	57
2	0	59	59
0	4	61	61
-1	5	81	81
5	10	82	86
2	0	88	94
2	0	96	96
2	0	98	98
2	0	100	100
0	5	102	102
0	3	109	109

Loop Table

4	9	-1	1	0	1	-1	1	1
6	7	-1	1	-1	4	-1	1	1
12	25	-1	1	0	1	-1	1	1
12	17	-1	1	0	2	-1	1	1
19	24	-1	1	0	2	-1	1	1

The complete template record consists of first the element table with each entry on a CHARACTER*40 line, then the loop table with each entry on a CHARACTER*80 line. The end of each table is indicated by a blank line. Then there are two CHARACTER*8 lines, the first is the line width of the display image, and the second is the length of the display image in lines. Then finally the lines of the display image are stored. Each element table entry is written with a FORTRAN format '(4I5)' and each loop table entry is written with a FORTRAN format '(11I5)'.

7.4 THE FORTRAN PROGRAM

The FORTRAN program consists of the main program JSTERM.FOR, the subroutines JSJTxx.FOR and the include files JT*.INC. Some of the JS*.INC files are also included. The main program accepts a one line command and then calls the appropriate subroutines to perform the indicated operation. The command line has the following format.

Cmd String1 String2

Cmd is the command function and String1 and String2 are character string operands. A list of all the command functions appears in Section 7.5.

7.4.1 DATA STRUCTURES

Some of the functions read and/or write the contents of the screen buffer. This information is stored in one of two character buffers, BUFFER or PREDIS. These buffers are CHARACTER*80 character arrays of fixed length. There is also a third CHARACTER*80 character array, TIMAGE, for storage of the template display image. The buffer PREDIS contains the information that is returned during a display data, create/change data or update operation, the buffer BUFFER is used for everything else. The lengths of these buffers are defined as parameters in the include file JTPARM.INC.

Besides the parameter specifying the length of the buffers mentioned above, JTPARM.INC contains several other parameters that specify the size of arrays pertaining to the parsing of template records. The most important of these arrays are ELETBL and LPTBL. These two arrays contain the Element Table and the Loop Table for a template as described in Section 7.3. ELETBL and LPTBL are read and written as part of each template record.

ELETBL is dimensioned (8,MXSYMN) where MXSYMN is a parameter defining the maximum number of symbols. The first subscript is the column number of the Element Table and the second subscript is the entry number. Columns 1-4 are used to store the 4 element table entries read from the template record and the remaining columns are used to contain information about the values of the variable. Except during creation of a new template the first four columns are constant and only the values of the last four columns are modified by the FORTRAN program.

The general form of ELETBL is:

1	2	3	4	5	6	7	8
ELEMENT ID	PTR, LEN, -1 OR 0	NAME BEGIN POSITION	NAME END POSITION	VALUE	VALUE DEFINED	NEW VALUE	NEW VALUE DEFINED

Columns 5 through 8 are used to hold information about the values of loop indices and variables that are used as loop limits. Column 6 is a flag that indicates whether or not a value has been defined for that element. A value of 0 means the element is undefined and a value of 1 means that the element has the value stored in column 5. The update display function requires that we know both the previous value and the new value of loop limit variables. Columns 7 and 8 contain the new value information for a variable whose value is changed. Columns 7 and 8 are used only by the subroutine JSJTUD (update display).

LPTBL is dimensioned (10,MXLPN) where MXLPN is a parameter defining the maximum number of symbols. The first subscript is the column number of the Loop Table and the second subscript is the entry number. Columns 1-9 are used to store the Loop Table read from the template record. The last column, column 10, is used during traversal of the display image to store the line number within the template image of the begin directive for the current iteration.

The general form of LPTBL is:

1	2	3	4	5	6	7	8	9	10
POINTER TO LOOP BEGIN ENTRY	POINTER TO INDEX OF LOOP	LOWER LIMIT INFO		UPPER LIMIT INFO		INCREMENT INFO		KEY / ENUMERATION	LINE NUMBER OF BEGIN DIRECTIVE

7.5 JSTERM COMMAND FUNCTIONS

This section lists the names of the command functions and the input arguments for each function.

CMD	Operation	Arg1	Arg2
CA	Cancel DD/CD		
CD	Create-ChangeData	Record Name	
CF	Create Freeform	Record Name	
CL	Close Data Set	Data Set Name	
CO	Copy	Subtree Name	Subtree Name
CT	Create Template	Record Name	
DD	Display Data	Subtree Name	
DE	Delete	Subtree Name	
DF	Display Freeform	Subtree Name	
DT	Display Template	Record Name	
EXIT	End Program		
Export	Export Data	Subtree Name	File Name
Import	Import Data	File Name	Data Set Name
LC	List Children	Subtree Name	

LI	List Records	Subtree Name	
MA	Compress Data Set	Data Set Name	
NF1	Next Record/Display Freeform		
NR1	Next Record/Create-Change Data		
NR2	Next Record/Display Data		
RE	Rename	Subtree Name	Qualifier
RM	Display Reactor Map	Record Name RM-Command ³	
RP	Report RM Position	Line-number	Column-number
SC	Save Comment	Data Set Name	Comment-string
SD	Show Dataset	Data Set Name	
SL	Save Location	Data Set Name:Location	
SMP	Display Module Path		
SNP	Display NONSTD Path		
SNW	Create Data Set	Data Set Name	
SR	Set Read Hierarchy	Data Set Name List ¹	
SSP	Display STD Path		
SW	Set Write	Data Set Name	
TR	Test Record	Record Name	
UD	Update Display	(See Note 2)	
UM	Update Reactor Map	(See Note 2)	
WD	Write Data	Record Name ²	
WF	Write Freeform	Record Name ²	
WM120	Write Reactor Map	Record Name ²	
WM180	Write Reactor Map	Record Name ²	
WM60	Write Reactor Map	Record Name ²	
WMP	Write Module Path	(See Note 2)	

WNP Write NONSTD Path (See Note 2)

WSP Write STD Path (See Note 2)

WT Write Template Record Name²

Note 1 The Set Read Hierarchy function takes as input a string which is an exact copy of the line in the read hierarchy buffer. That is, it is a string containing 0-8 data set names left justified in 8 character fields spaced 2 characters apart.

Note 2 This function requires the contents of the main buffer as input.

Note 3 Valid RM commands are 'ADD', 'HIDE', 'SHOW' and 'SYM'. Each requires a second argument and the contents of the main buffer as input. The second argument for the 'ADD', 'HIDE' and 'SHOW' commands is a list of reactor map locations of the form 'Cn' (cluster n), 'Gn' (gang n) and 'Sn' (system n) separated by spaces. The second argument for 'SYM' is 360, 180, 120 or 60.

A THE LITTLE INPUT LANGUAGE, LIL

The Little Input Language (LIL) is a simple freeform input language. This appendix describes the LIL syntax and a group of subroutines that can be used in a FORTRAN program to parse an input File written in LIL.

A.1 THE LIL SYNTAX

Input instructions to a program can be prepared as a File of card images with a maximum of 80 columns per card. When the File is read in by the LIL routines, the cards are (logically) concatenated into a long character string. Blanks not enclosed in apostrophes are ignored.

Logically a LIL File is a hierarchical arrangement of:

Blocks (separated by a ';') which contain a set of
 Statements (separated by a ',') which contain a
 Keyword (followed by an '=') followed by a
 Value (or a set of Values separated by a ',' and enclosed in '()')

Stated another way, the LIL syntax is

```
File    :=    FirstBlock ; SecondBlock ; ... ; LastBlock
Block   :=    FirstStatement , SecondStatement , ... , LastStatement
Statement :=    Keyword
            |    Keyword = Value
            |    Keyword = (FirstValue , SecondValue , ... LastValue )
```

The first form of a Statement has only a Keyword. The second form has a Keyword and a single Value which, if omitted, is a null Value. The third form has a Keyword and a list of Values. An empty list is a single null Value. An omitted Value is a null Value. For example, statement KW=(,,) has the Keyword KW with 3 null Values.

A Keyword is an arbitrary length alphanumeric character string which will be converted to upper case by the LIL routines. A Keyword must not be enclosed in apostrophes. A Value may contain any valid processor characters and may be enclosed in apostrophes. If the Value contains any of the LIL delimiters, ';', '=', '(', ')', then it must be enclosed in apostrophes. As in FORTRAN, an apostrophe within a Value is represented by two apostrophes.

The ordering of Blocks in a File, Statements in a Block, and Values in a Value list is specified by the programmer. The programmer can prepare code that requires a specific ordering and the user must then prepare Files in that order. However, this would defeat the purpose of the Keywords which is to identify the Values regardless of the order in which they are presented. Alternatively, the programmer can prepare code that will accept an arbitrary ordering. The LIL routines provide assistance to the programmer in preparing the code to accept an arbitrary ordering.

The following is an example of a specification File written in LIL. Note that it has been arranged on the card images in a readable manner. However, it could have been typed as a long string with no blanks.

```
Input, File='S8999[DIF.SRC]*.FOR', Opt=n;
Output, File='S8999[DIF.OUT]*.OUT';
```

```
Print, File=PRN;  
Punch
```

There are 4 Blocks specified. The Keywords (Input, File, Opt, Output, Print, and Punch) will all be converted to upper case when read by the LIL routines. The Values in apostrophes will not be changed. Note that the Keyword File is used as a generic Keyword meaning the name of a File. The Keyword is taken in the context of the Block in which it occurs. The first occurrence means Input File, the second occurrence means Output File, etc. The above example could also have been specified as a single Block using expanded Keywords as follows:

```
InputFile='S8999[DIF.SRC]*.FOR',  
InputOpt=n,  
OutputFile='S8999[DIF.OUT]*.OUT',  
PrintFile=PRN,Punch
```

The design of the LIL File is up to the programmer who selects the Blocks, Statements, Keywords, allowed Values, and the order (if any) in which they must occur in the File.

A.2 THE LIL ROUTINES

The programmer preparing the FORTRAN code to read the LIL File may use a set of pre-written routines to simplify the task. The routines are FORTRAN subroutines with the general name JSLILx, where x takes on several values. The routines communicate with each other via two small Common areas named /JSLIL1/ and /JSLIL2/. The routines are described below in the order that they would be used in a program.

The File containing the input specifications written in the LIL syntax must be opened. Assume that it is opened as unit UNIT. The LIL routines are initialized for reading this File with the statement:

```
CALL JSLILI (UNIT,*n)
```

where the *n return is taken when an error occurs. Whenever an error occurs, the error number ERRN and a 28-character error message MSG (both listed in Section A.4) can be obtained with the statement:

```
CALL JSLILE (ERRN,MSG)
```

The error number and message can then be printed. Another way to handle the error is to print on unit PRNU the above error number and message along with the offending line of the File using the statement:

```
CALL JSLILP (PRNU)
```

The error number can be used to determine the severity of the error and the subsequent action. If there is no error, the normal return is taken.

We imagine a pointer which points to characters in the File. Following the call to JSLILI the pointer points to the first character of the File which is the first character of the first Keyword of the first Block. (If the Block is null, then this character is a ';'.) As we process characters in the Block, the pointer moves forward. The current implementation provides a single 80-character card buffer and no provision to reread the buffer or the File.

Therefore, the File is read sequentially and the program must be written to process the Blocks and Statements in the order presented in the File.

We need not read all of the Statements in a Block. Whenever we have finished reading the current Block, we can move to the start of the next Block (skipping over any remaining Statements) using the statement:

CALL JSLILB (RETC,*n)

where RETC is a return code as follows:

RETC=1	means there is another Block and the pointer now points to the start of it.
RETC=2	means there are no more Blocks and all of the File has been processed. If necessary, the File can be rewound and processed again starting with another call to JSLILI.

Assume that the pointer points either to the start of a Keyword (or to a Value from the prior Keyword). There are two approaches. First, simply read the next Keyword KEY (skipping over any remaining Values from a prior Keyword) using the statement:

CALL JSLILK (KEY,RETC,*n)

where RETC is a return code as follows:

RETC=1	means a Keyword was returned in KEY but there is no Value.
RETC=2	means a Keyword was returned in KEY and there are one or more (possibly null) Values which are ready to be read.
RETC=3	means there are no more Keywords in this Block. JSLILB must be called to move to the start of the following Block before it can be read.

The Keyword container KEY has a implicit length (i.e. LEN(KEY)) determined by a declaration statement such as CHARACTER*8 KEY. An error occurs if the Keyword is longer than KEY. Otherwise, the Keyword is converted to upper case, left-adjusted, packed, and blank right-filled in the container KEY. We can next test KEY against the possible Keywords and branch to the proper handler. We can make this a bit easier by using the routine JSLILF. Assume that we prepared a list of the possible Keywords in the array (KEYS (IKEY), IKEY=1,NKEY). Then the statement:

CALL JSLILF (KEYS,NKEY,IKEY,RETC,*n)

reads the next Keyword from the File (skipping over any remaining Values) and compares it with the Keywords in the array KEYS. The return code RETC is:

RETC=1	means the Keyword was found at index IKEY but there is no Value.
RETC=2	means the Keyword was found at index IKEY and there are one or more (possibly null) Values.
RETC=3	means a Keyword was read but it does not match any Keyword in the array KEYS.
RETC=4	means there are no more Keywords in this Block. JSLILB must be called to move to the start of the following Block before it can be read.

If RETC=1 or 2, we can branch on IKEY to the proper handler.

Assume that a Keyword has been read and that we are in the proper handler. If there are Values given for the Keyword (RETC=2 above), then the Values can be read into the array VALS of length MVAL using the upper case specifier:

UPCT=0	means no Value conversion.
UPCT=1	means convert Values to upper case if they are not in apostrophes.
UPCT=2	means convert Values to upper case even if they are in apostrophes.

and the statement:

```
CALL JSLILV (UPCT,VALS,MVAL,NVAL,*n)
```

The programmer must allocate the array with sufficient width (e.g. CHARACTER*64) and length to hold the Values. An error occurs if either the width or length overflow. A null Value is returned as a blank Value.

Frequently a Keyword will have a single Value meaning "Yes" or "No". There are many ways to represent this Value. A "binary" Value of this type can be read with the Statement:

```
CALL JSLILY (RETC,*n)
```

where the return code RETC means:

RETC=0	means the Value was ' ', 'No', 'Off', 'False', 'N', '0', '.False.', '.F.', or any upper or lower case version of these.
RETC=1	means the Value was 'Yes', 'On', 'True', 'T', '1', '.True.', '.T.', or any upper or lower case version of these.
RETC=2	means the Value was none of the above.

In addition to the routines listed above, there are two routines designed for use only by other LIL routines that should never be called by the application programmer. They are JSLILC and JSLILT.

A.3 SUMMARY OF THE JSLILX ROUTINES

The following is a summary of the routines used by the application programmer:

CALL JSLILI (UNIT, *n)	Initialize for reading File
CALL JSLILB (RETC, *n)	Skip to next Block
CALL JSLILK (KEY, RETC, *n)	Read the next Keyword
CALL JSLILF (KEYS, NKEY, IKEY, RETC, *n)	Read and find the next Keyword
CALL JSLILV (UPCT, VALS, MVAL, NVAL, *n)	Read the set of Values
CALL JSLILY (RETC, *n)	Read a binary Value
CALL JSLILE (ERRN, MESG)	Get the error number and message
CALL JSLILP (PRNU)	Print the error and card buffer

A.4 ERROR NUMBERS AND MESSAGES

The following is a list of the error numbers ERRN and messages MESH used in the LIL routines:

ERRN=1, MESH='The input unit is undefined'

The UNIT was less than or equal 0 in the call to JSLILI.

ERRN=2, MESH='The input unit is empty'

In JSLILI, the File on unit UNIT is empty.

ERRN=3, MESH='Improper use of apostrophes'

The apostrophes are unbalanced. There is no recovery since the routines read to the end of the File looking for a matching apostrophe.

ERRN=4, MESH='A key or values is too long'

The field given in the File is longer than the character variable KEY in JSLILK, KEYS in JSLILF, or VALS in JSLILV.

ERRN=5, MESH='Improper delimiter syntax'

An improper delimiter or delimiter ordering has been used.

ERRN=6, MESH='Too many values are given'

The number of Values NVAL exceeds the allocated size MVAL of the Value array VALS in JSLILV.

ERRN=7, MESH='Logic error in JSLIL'

A fatal logic error in one of the JSLIL routines has been detected. Consult a JOSHUA systems programmer.

ERRN=8, MESH='Incorrect call to JSLILV'

JSLILV can be called only when the pointer points to an '=' delimiter as it is when RETC=2 is returned from JSLILK or JSLILF.

ERRN=9, MESH='No place given to put value'

The variable MVAL=0 in a call to JSLILV.

ERRN=10, MESH='Key must not be in quotes'

A Keyword must not be bracketed by apostrophes.

ERRN=11, MESH='Null key is not allowed'

There must be a non-null Keyword ahead of an '=' sign.

A.5 A SAMPLE PROGRAM

The fragment of a subroutine listed below illustrates the use of the JSLILx subroutines to read the following sample File:

```
Input, File='S8999[DIF.SRC]*.FOR', opt=n;  
Output, File='S8999[DIF.OUT]*.OUT';  
Print, File=PRN
```

```
      SUBROUTINE LILINP
C
C.....Dimensions, Data, and Defaults
C
      INTEGER RETC,IKEY,OPT
      CHARACTER*64 INPF,OUTF,PRNF
      CHARACTER*8  BLKS(3),KEYS(2)
C
      DATA BLKS/'INPUT','OUTPUT','PRINT'/
      DATA KEYS/'FILE','OPT'/
      DATA INPF/' ','OUTF/' ','PRNF/' ','OPT/0/
C
C.....Open the LIL Input File
C
      100 OPEN (UNIT=2,FILE=input_file_name)
          CALL JSLILI (2,*999)
C
C.....Get the first Keyword in this Block (Block Type)
C
      110 CALL JSLILF (BLKS,3,IKEY,RETC,*999)
          GO TO (120,901,190,190),RETC
      120 GO TO (130,170,180),IKEY
C
C.....Input Block, find next Keyword
C
      130 CALL JSLILF (KEYS,2,IKEY,RETC,*999)
          GO TO (190,140,190,190),RETC
      140 GO TO (150,160),IKEY
C
C.....Process Input Block File Keyword
C
      150 CALL JSLILV (2,INPF,1,N,*999)
          GO TO 130
C
C.....Process Input Block Opt Keyword
C
      160 IF (RETC.EQ.1) THEN
          OPT=0
        ELSE
          CALL JSLILY (OPT,*999)
          IF (OPT.GE.2) GO TO 903
        ENDIF
          GO TO 130
C
C.....Process Output Block File Keyword
C
      170 CALL JSLILK ('FILE',RETC,*999)
          IF (RETC.NE.2) GO TO 190
          CALL JSLILV (2,OUTF,1,N,*999)
          GO TO 190
C
C.....Process Print Block File Keyword
C
      180 CALL JSLILK ('FILE',RETC,*999)
          IF (RETC.NE.2) GO TO 190
          CALL JSLILV (2,PRNF,1,N,*999)
```



```
C
C.....Move to next Block
C
  190  CALL JSLILB (RETC,*999)
        IF (RETC.EQ.1) GO TO 110
C
C.....Finished reading LIL File
C
        (check results)
        RETURN
C
C.....Errors
C
  901  CALL ERROR ('Block Keyword may not have values',*999)
  902  CALL ERROR ('Illegal value given for Opt',*999)
  999  CALL JSLILP (6)
        STOP
        END
```

There were many shortcuts taken in this example to keep it readable. A "real" subroutine would have much stronger checking. Here we set the default file names to blank in a data statement and then check them later. There is no checking done on extra (undefined) Keywords. However, the fragment does illustrate the technique for reading the Blocks and Keywords in the order given in the File.

B SUMMARY OF JS ROUTINES

Most of the JS routines are written in ANSI FORTRAN 77 [1]. Common blocks and symbolic constants used by several different routines are defined in separate files and included in the routines at compile time using INCLUDE statements. INCLUDE statements are not defined in ANSI FORTRAN 77 [1], but are available in VAX FORTRAN [8] and IBM VS FORTRAN [13-15]. (INCLUDE statements are also available in the JOSHUA language.) For each different computer system, there are a few non-FORTRAN files: assembler routines for both the VAX/VMS and IBM/MVS and a MESSAGE routine [10] for VAX/VMS.

Each JS routine is maintained as a separate unit, that is, as a separate file. Development of the routines was done under VAX/VMS, and the following file types have been used:

FORTTRAN	.FOR
INCLUDE	.INC
VAX assembler	.MAR
VAX message	.MES
IBM assembler	.ASM
IBM assembler macro	.MAC

The JS routines and common blocks have 6 character names, beginning with the letters "JS". The monitor program, JSMON, has a 5 character name, also beginning with the letters "JS". Some groups of routines are named with common letters following the "JS" to indicate additional relatedness:

JSLIL	for the Little Input Language (LIL) routines, see also Appendix A
JSMV	for routines used only under IBM/MVS
JSVX	for routines used only under VAX/VMS

The JS routines are listed and briefly described in the following sections. Entry points are listed under the routine containing them.

Certain other routines used in the IBM/MVS version of JOSHUA which are not exclusive to JOSHUA do not start with JS.

B.1 GENERAL ROUTINES

B.1.1. INCLUDE FILES

JSDATA.INC	General data
JSERSY.INC	System error information
JSERUS.INC	User error information
JSIOBF.INC	I/O buffers
JSJDST.INC	JOSHUA Data Set Table
JSJINT.INC	JSIN Table

JSJNME.INC	JOSHUA Name Table
JSLIST.INC	List and list children information
JSPARM.INC	Parameters (symbolic constants)
JSSQFT.INC	Sequential File Table
JSSTAT.INC	Job Statistics Table
JSTRPA.INC	Catalog Tree and Path Tables

B.1.2. FORTRAN FILES

JSADAT.FOR	To return the current date in character form "MM-DD-YY".
JSADDS.FOR	To add a data set to the Data Set Table.
JSADND.FOR	To add a group of nodes to a JOSHUA tree catalog.
JSADST.FOR	To add the STD data sets listed in the STD Table record to the data set table.
JSALDS.FOR	To allocate the files for a data set.
JSATIM.FOR	To return the current time in character form "HH:MM:SS".
JSBIGL.FOR	Print big block letters.
JSBLDA.FOR	To initialize commons used by all JS routines. Any program/module, including JSMON, that call JS routines should use this block data.
JSCHDS.FOR	To change the protection, type, and/or files used for an entry in the data set table. The data set will always be changed to NONSTD.
JSCKNM.FOR	To check validity of each name in an array of JOSHUA names.
JSCLAL.FOR	To close all open data sets.
JSCLDS.FOR	To close a data set, making sure to maintain data set catalog integrity.
JSCLFI.FOR	To close a file.
JSCMPR.FOR	Compare strings.
JSCNTR.FOR	To print a given string, centered in a line of given length
JSCOPY.FOR	To copy the records, one at a time, from one subtree to another.
JSCPUS.FOR	To copy the USER data set to a "temporary" data set

so the user can continue to write to their USER data set.

JSCVCH.FOR	To convert a character string of arbitrary length to length 8.
JSCVCI.FOR	To convert a character string to an integer.
JSCVDC.FOR	To convert a double precision to a character string.
JSCVIC.FOR	To convert an integer to a packed character string.
JSCVIN.FOR	To convert an integer to a character string of length 8.
JSCVRC.FOR	To convert an real to a character string.
JSCVSA.FOR	To convert a string to an array of C*8 strings.
JSCVSS.FOR	To break a string into the pieces indicated by a given separator character, pad or truncate the pieces to a uniform length.
JSDEBL.FOR	To return a given string deblanked, left justified, and right padded with blanks.
JSDLTE.FOR	To delete a subtree and to handle the special case of deleting the SCRATCH data set.
JSDSB1.FOR	To place in a buffer information about a given data set.
JSERBG.FOR	To begin processing of a user MESSAGE, WARNING, or ERROR statement .
JSEREN.FOR	To end processing of a user ERROR, WARNING, or MESSAGE.
JSERIO.FOR	To create (part of) an error message for system IO errors.
JSERNF.FOR	While processing a user ERROR, WARNING, or MESSAGE statement, find the next format specifier in the text.
JSEROR.FOR JSERCL JSERSE	To clear or set error status.
JSERPL.FOR	While processing a user ERROR, WARNING, or MESSAGE statement, print the next display line.
JSERRF.FOR	While processing a user ERROR, WARNING, or MESSAGE statement, get ready to output the next formatted element from the IO list.
JSERTR.FOR	To print the current error message, if any, in the error trace file.
JSERXX.FOR JSERIN	While processing a user ERROR, WARNING, or MESSAGE statement, to add the next element in the IO list to

JSERRL JSERDP JSERLG JSERCH	the display.
JSEUDS.FOR	Convert input string of extended data set names (may contain '<E>') to a string of C*8 names without the exclusive use(EU) indicator.
JSIQUEU	Inquire if a data set is reserved for exclusive use.
JSEXBG.FOR	To begin the setup necessary for execution of a module.
JSEXEC.FOR	To suspend the current module, execute another module, then resume the current module.
JSFIDF.FOR	To find the names of the tree and data files for a given JOSHUA data set. Or to construct the names of the files if they do not yet exist, but will be created.
JSFIDS.FOR	To find a data set in the Data Set Table. Or if the data set is not in the table, to find the location where it should be added.
JSFIFP.FOR	To find a file by searching along a file path. Or to find where the file should be placed along a path and to construct its name.
JSFIFP.FOR	To find a file by searching along a file path. Or to find where the file should be placed along a path and to construct its name.
JSFIFT.FOR	To find an executable file by using a file lookup table.
JSFILL.FOR	To fill a string with a given number of occurrences of a given character.
JSFORF.FOR	Find and open file.
JSGEBR.FOR	To get the boot record from the System SCRATCH data set.
JSGEMS.FOR	To get a message sent by parent module.
JSGENA.FOR	To get the tree address = (Record#,Offset) of a given node number.
JSGENN.FOR	To get the next node of a given type in a given data set.
JSGEPA.FOR	To get a (catalog) path for a data set; an unused path if one exists, otherwise the 'least recently used'.
JSGETN.FOR	To get a tree node, specifically, a given node number in a given data set.
JSGEUN.FOR	To find and return a valid, available FORTRAN unit number.

JSICHR.FOR	Generalized ICHAR function.
JSINSQ.FOR	To initialize a file for sequential output.
JSIOTR.FOR	To provide, in a sequential file, a trace of all JOSHUA I/O operations.
JSIQCO.FOR	To return the number of records copied in the most recent copy operation.
JSIQDR.FOR	To return the name of the Ith element in the DEFAULT Read Hierarchy. If there is no Ith element, then blanks are returned.
JSIQDS.FOR	To satisfy an inquiry about a data set.
JSIQEC.FOR	To return the current ERROR CODE.
JSIQEM.FOR	To print the current error message on unit 6.
JSIQJN.FOR	To return a JOSHUA name in dotted form.
JSIQLN.FOR	To return the JOSHUA record name, not including data set name, most recently found by preorder traversal.
JSIQMC.FOR	To return the count of JOSHUA MESSAGE statements processed so far during execution of this module.
JSIQMS.FOR	To return the current JOSHUA message.
JSIQNC.FOR	To return the number of qualifiers in the record name most recently looked for in the JOSHUA data base.
JSIQNM.FOR	To return the Ith qualifier in the record name most recently looked for in the JOSHUA data base.
JSIQPA.FOR	To return the ISTEPth step in a file path.
JSIQPL.FOR	To return the number of steps in a file path.
JSIQRH.FOR	To return the name of the Ith element in the current Read Hierarchy. If no Ith element, return blanks.
JSIQSE.FOR	To return the current SYSTEM ERROR number.
JSIQUE.FOR	To return the current number of the most recently executed ERROR, WARNING, or MESSAGE statement.
JSIQUM.FOR	To return the current JOSHUA user message defined with USER_MESSAGE keyword as part of the JSMON command or in a JSIN file.
JSIQUS.FOR	To return the name of current USER data set.

JSIQWC.FOR	To return the count of JOSHUA WARNING statements processed so far during execution of this module.
JSIQWR.FOR	To return the name of the current WRITE data set.
JSJCPU.FOR	To return the CPU time used so far by the current job in units of hundredths of a second.
JSJDAT.FOR	To return the current Julian date in the character form "YY.DDD ".
JSJIBD.FOR	To initial the JSIN Keyword Table and value string.
JSJIGV.FOR	To get the value(s) associated with a given JSIN keyword.
JSJIPR.FOR	While processing JSIN, to print the current JSIN data in the messages file.
JSJIRJ.FOR	While processing JSIN, to read the JSIN file and set values into the JSIN Keyword Table and value string.
JSLCBG.FOR	To signal that a new list children operation is beginning.
JSLCNX.FOR	To next provide a list or partial list of the children in a list children operation. Children named ".DELETED" will not be included in the list.
JSLENG.FOR	Find non-blank length of string.
JSLIBG.FOR JSLSBG	To begin "list" (preorder traversal) of a tree or subtree.
JSLINX.FOR	To find the next JOSHUA name in the preorder traversal of a tree or subtree.
JSLKUP.FOR	To supervise the lookup of a JOSHUA name string in the JOSHUA tree catalog(s).
JSMACH.FOR	To provide identification of machine on which running.
JSMCPU.FOR	To return the CPU time used so far by the current module in units of hundredths of a second.
JSMDBG.FOR	To perform initializations at the beginning of execution of a module.
JSMNIV.FOR	To find the minimum in an integer vector.
JSMON.FOR	To oversee execution of a JOSHUA job.
JSMOVS.FOR	Move string.
JSMTCH.FOR	To match a given JOSHUA name string in a given JOSHUA

tree catalog.

JSNLEN.FOR	To return the non-blank length of a given string.
JSOPDS.FOR	To open a data set. The data set is assumed to be already allocated.
JSOPSY.FOR	To provide identification of operating system and version.
JSOPXX.FOR JSOPDA JSOPSQ	To open a file.
JSPCDS.FOR	To put a comment in a JOSHUA data set catalog header.
JSPMXX.FOR JSPMBG JSPMIN JSPMRL JSPMDP JSPMCH JSPMNL JSPMEN	Supervise printing of a message.
JSPOTH.FOR	To 'Page Out' to disk the current tree header record for a given data set.
JSPOTR.FOR	To 'Page Out' a tree record.
JSPREM.FOR	To print the current error message, level and code in a given file. Message lines will be preceded by "***JS***".
JSPRST.FOR	To get and print job statistics.
JSPUBR.FOR	To 'put' boot record in the System SCRATCH data set.
JSPUMS.FOR	To 'send' a message to a child module.
JSQILM.FOR	To return the index of the end qualifier in the main subtree of a preorder traversal.
JSQILS.FOR	To return the index of the first qualifier in current subtree of a preorder traversal.
JSQINC.FOR	To return the number of qualifiers in the record name most recently looked for in the JOSHUA data base.
JSQINM.FOR	To return the Ith qualifier in the Jth name in the JOSHUA Name Table.
JSQUXX.FOR JSQUFN JSQUFE JSQUUN JSQUUO	To satisfy inquiries about files.

JSRDBF.FOR	To read the next record from disk into the given buffer.
JSRDBG.FOR JSRSBG JSRTBG	To signal that a new read operation is beginning.
JSRDEN.FOR JSRTEN	To signal that a read operation has ended.
JSRDXX.FOR JSRDIN JSRTIN JSRDRL JSRTRL JSRDDP JSRTDP JSRDLG JSRTLG JSRDCM JSRTCM JSRDCH JSRTCH	To read values of type xx from a JOHSHA record on disk to locations in core.
JSREPL.FOR	Replace file name.
JSRESA.FOR	Resolve abbreviations for command processing. Used by JSMON for JSIN keywords and by JSTERM on the IBM.
JSRETM.FOR	To prepare to end module (or JSMON) by closing sequential files.
JSRETN.FOR JSJRET	To return from an EXECUTE.
JSRNME.FOR	To rename a qualifier in a data set tree catalog.
JSRTIM.FOR	To return the time elapsed since a (system dependent) fixed base time in units of hundredths of a second.
JSRXXX.FOR JSR1IN JSR2IN JSR1RL JSR2RL JSR1DP JSR2DP JSR1LG JSR2LG JSR1CM JSR2CM JSR1CH JSR2CH	To read values of type xx from a JOHSHA record on disk to locations in core, with error alternate returns.
JSSEDR.FOR	To set the DEFAULT Read Hierarchy for this job. Also

	to set the current Read Hierarchy to this new default.
JSELD.FOR	To (re)set the (physical) location of a NONSTD data set.
JSENM.FOR	To setup a name for lookup.
JSSEPA.FOR	To set a different file search path.
JSSERH.FOR	To set the current Read Hierarchy.
JSSEWR.FOR	To set a different NONSTD data set to be the WRITE data set or to set NO WRITE data set.
JSSRCH.FOR	To search for K in sorted array, A(I), I = 1,N. If found return its index I, otherwise return 0.
JSTADS.FOR	Get time and date stamp.
JSTAMP.FOR	To provide unique "stamp" based on time and date.
JSTICK.FOR	To tick the internal clock and return new internal time.
JSTIOF.FOR	To turn off the module cpu timer if it is not already off. Update the cpu times before doing so.
JSTION.FOR	To turn on the module CPU timer, if it is not already on.
JSTIUD.FOR	To update the cpu timer, JOBCPU, and MODCPU.
JSUCAS.FOR JSUCA2	To change each lower case letter in a string to the corresponding upper case letter. Other characters remain unchanged.
JSWRBF.FOR	To write the WRITE I/O buffer to disk.
JSWRBG.FOR JSWSBG	To signal that a new write operation is beginning.
JSWREN.FOR	To end a write operation.
JSWRXX.FOR JSWRIN JSWRRL JSWRDP JSWRLG JSWRCM JSWRCH	To write values of type xx to a JOHSHA record.
JSWXXX.FOR JSW1IN JSW1RL JSW1DP JSW1LG JSW1CM	To write values of type xx to a JOHSHA record, with error alternate returns.

JSW1CH

JSXCPU.FOR

To return the current CPU clock reading for this job,
process, or whatever in units of hundredths of a second.

B.2 TERMINAL SYSTEM ROUTINES

B.2.1. INCLUDE FILES

JTFLDM.INC

Define the characters used at the beginning and ending of template data
fields.

JTJSJTDS.INC

Commons containing the current read hierarchy and default write data
set.

JTPARM.INC

General size and length parameters.

B.2.1. FORTRAN FILES

JSJTBD.FOR

Block data defining initial screen display. (IBM Only)

JSJTCD.FOR
JSJTC2

Create data function.

JSJTCF.FOR

Create freeform function.

JSJTCL.FOR

Command initialize. (IBM Only)

JSJTCP.FOR

Compress a JOSHUA data set.

JSJTCS.FOR

Prompt user to save buffer if it has been changed. (IBM Only)

JSJTCT.FOR

Create template function.

JSJTD2.FOR

Fill in index fields in a template during a data display operation.

JSJTDB.FOR

Removes blanks from a text string.

JSJTDD.FOR

Display data function.

JSJTDF.FOR

Display freeform function

JSJTDI.FOR

Fill in index fields in a template during a data display operation.

JSJTDO.FOR

Main command handler.

JSJTEM.FOR

Return the error message associated with a given error number.

JSJTEX.FOR

Export records function.

JSJTFD.FOR

Text find function for IBM. (IBM Only)

JSJTFI.FOR

Return the type and length of a display field from a template.

JSJTFM.FOR	Fill in numeric information in an error message.
JSJTFF.FOR	Insert text information into an error message.
JSJTFR.FOR	Return the full record name of the last JOSHUA record used.
JSJTFS.FOR	Return the type and length information for a variable from the I/O list of a template.
JSJTFX.FOR	JOSHUA record name format routine. (IBM Only)
JSJTGC.FOR	Main command parser for the IBM version. (IBM Only)
JSJTGD.FOR	Prompt for and check JOSHUA data set names. (IBM Only)
JSJTGN.FOR	Prompt for and check JOSHUA record names. (IBM Only)
JSJTGW.FOR	Remove and return the first word in a string. (IBM Only)
JSJTIC.FOR	Update data field and loop directive pointers within the current display during a data display operation.
JSJTIH.FOR JSJTHI	Convert numbers between decimal and hex. Used for Z type display fields.
JSJTIM.FOR	Import JOSHUA data records from a text file and write them to a binary data set.
JSJTIO.FOR	Update data field and loop directive pointers within the output buffer during a data display operation.
JSJTND.FOR	Find the next .Begin or .End directive in a template definition.
JSJTNE.FOR	Advance the element pointer within the element table of a template definition.
JSJTNF.FOR	Update data field and loop directive pointers within the template buffer during a data display operation.
JSJTNX.FOR	Return the position of the next delimiter within a text string.
JSJTPB.FOR	Display the contents of the screen buffer.
JSJTCB	Clear the screen buffer (IBM Only)
JSJTIL	Insert a line in the screen buffer. (IBM Only)
JSJTDL	Delete a line from the screen buffer (IBM Only)
JSJTRB.FOR	Return the next line in the buffer. (IBM Only)
JSJTRL.FOR	Returns the text from the command line. (IBM Only)
JSJTRD.FOR	Read the contents of the screen into a buffer. (VAX Only)
JSJTRM.FOR	Reactor Map function.
JSJTRO.FOR	Rotate changed assembly positions for the RM function.

JSJTRT.FOR	Read a template definition into the template buffer.
JSJTSA.FOR	Save record function. (IBM Only)
JSJTSL.FOR	Store an output line of text to a buffer or write it to an output unit.
JSJTTD.FOR	Check to see if display field or a loop directive follows the current display field.
JSJTTR.FOR	Test for the existence of a JOSHUA data record.
JSJTUC.FOR	Convert a text string to upper case.
JSJTUD.FOR JSJTMD	Update the display of the current data record.
JSJTWL.FOR	Write a text string without the trailing spaces.
JSTERM.FOR	Main program for JSTERM.

B.3. VAX/VMS ROUTINES

B.3.1. INCLUDE FILES

JSVXMS.INC VAX, inter-module communication information

B.3.2. FORTRAN FILES

JSDSPR.FOR	To print information from the current Data Set Table.
JSFILE_PARSE.FOR	To parse VAX/VMS file specifications and apply appropriate default fields.
JSLCPR.FOR	To perform a complete LIST CHILDREN operation and print the results in 'pretty' form.
JSLIPR.FOR	To perform a LIST operation (pre-order traversal) and print the results in 'pretty' form.
JSPRBU.FOR	To print the contents of a buffer.
JSVXBD.FOR	VAX/VMS block data.
JSVXEh.FOR	VMS error condition handler.
JSVXEX.FOR	VAX exit handler to delete scratch, user scratch and "temporary" user data sets.
JSVXJI.FOR	VAX command line interface(CLI) routine.
JSVXSM.FOR	To add the next VMS error message line to the VXEMSG buffer.

B.3.3. MISCELLANEOUS FILES

JSVXLN.MAR Logical name definitions, needed by JSGEMS and JSPUMS.

JSVXTB.MSG Cause the system to produce trace back onabend.

B.4. IBM/MVS ROUTINES

B.4.1. FORTRAN FILES

DAF\$AL.FOR To dynamically allocate a data set.

DAF\$IL.FOR Used by the other DAF\$ routines.
DAF\$IX

DAF\$IN.FOR To dynamically inquire on a data set.

DAF\$UN.FOR To dynamically unallocate a data set.

JSDEDS.FOR To delete a data set from the Data Set Table

JSDSBU.FOR To place in a buffer information from the current
Data Set Table.

JSLCBU.FOR To perform a complete LIST CHILDREN operation and
place the results in 'pretty' form in a character buffer.

JSLIBU.FOR To perform a LIST operation (pre-order traversal) and
place the results in 'pretty' form in a character buffer.

JSMKVS.FOR To create a VSAM file.

JSMVAD.FOR Under IBM MVS, allocate a sequential file for FORTRAN
direct access I/O. If DD is blank use a system generated
ddname otherwise use DD.

JSMVAP.FOR Under IBM/MVS, to allocate an existing partitioned data
set with disposition of SHR.

JSMVBG.FOR Under IBM/MVS, to return the user's badge (user ID).

JSMVFM.FOR Under IBM/MVS, to find the first occurrence of a given
member along a path of partitioned data sets (PDSs).

JSMVFN.FOR Under IBM/MVS, to provide the data set name associated
with a given DDNAME.

JSMVIZ.FOR Under IBM/MVS, To convert an integer to a hexadecimal string.

JSMVME.FOR Under IBM/MVS, to determine if a given member exists in

a given partitioned data set (PDS).

JSMVML.FOR Under IBM/MVS, to get a list of members in a given partitioned data set (PDS).

JSMVUN.FOR Under IBM/MVS, to deallocate a data set.

UT\$PRS.FOR Find a portion of one string inside another string.

B.4.2. ASSEMBLER FILES

\$INFO\$.ASM Return the current user ID.

JSMVAT.ASM Use the ATTACH macro to execute a JOSHUA module. Documented in the IBM manual MVX/XA Supervisor and Macro Instructions.

JSMVGP.ASM To get the PARM message "sent" by the operating system

JSMVLO.ASM To use the LOCATE macro to see if a given data set exists. The locate macro is documented in the IBM manual "MVS/XA Catalog Administration Guide".

JSMVOB.ASM To use the OBTAIN macro to get info about a given data set. The obtain macro is documented in the IBM manual "MVS/XA System-Data Administration".

JSMVWT.ASM Wait a specified number of seconds.

JSVSAM.ASM Use the LINK macro to execute the access services utility program IDCAMS to create VSAM datasets (IBM/MVS). IDCAMS is documented in the MVS/XA Integrated Catalog Administration: access method services reference manual. The LINK macro is documented in the IBM manual "MVS/XA SUPERVISOR AND MACRO INSTRUCTIONS".

SVC\$99.ASM Execute an SVC 99 call.

UT\$ADR.ASM Get absolute address of symbol.

UT\$CAP.ASM Branch to an absolute address passing an argument list.

UT\$CPY.ASM Memory to Memory copy.

UT\$EOL.ASM Set end of parameter list.

UT\$EOP.ASM Set end of list bit for a parameter list.

UT\$PAC.ASM Parse an argument list and return the length of the character argument.

UT\$PAL.ASM Parse the argument list of the calling routine.

UT\$PAR.ASM Return the address of the argument list.

UT\$PTR.ASM Return the word at an absolute address.

UT\$TRN.ASM Translate string.

VS\$FRE.ASM Free previously allocated area of virtual storage.

VS\$GET.ASM Allocate virtual storage.

B.5. LITTLE INPUT LANGUAGE (LIL) ROUTINES

JSLILB.FOR LIL, block.

JSLILC.FOR LIL, character.

JSLILE.FOR LIL, error.

JSLILF.FOR LIL, find.

JSLILI.FOR LIL, initialize.

JSLILK.FOR LIL, key.

JSLILT.FOR LIL, token.

JSLILV.FOR LIL, value.

JSLILY.FOR LIL, yes.

C JS INTERNAL ERROR MESSAGES

A JS *internal error* in most cases indicates a programming error in the JS routines. This appendix lists the internal errors and the routines in which they can occur.

9001	JSADDS	Invalid value for TYP argument.
9002	JSADDS	Invalid value for PRO argument.
9003	JSADDS	TFIL and DFIL not both blank or both non-blank.
9004	JSADDS	Data Set Table full. Increase MXDSCT in the JSPARM include.
9005	JSCHDS	Invalid index in Data Set Table.
9006	JSCLDS	Attempt to close improperly open data set. Only one, not both, of tree and data files is open.
9007	JSGENA	Invalid value for NODENO argument.
9008	JSLKUP	Invalid value for LKFLG argument.
9009	JSPOTH	Invalid index to Data Set Table.
9010	JSPOTH	Attempt to revise READONLY data set.
9011	JSPOTR	Attempt to update catalog of READONLY data set.
9013	JSCHDS	Invalid value for PRO argument.
9014	JSMTCH	If JNMEBG > JNMECT, then there are no qualifiers to be matched. Therefore, must have LKFLG=5, to flag an operation on an entire tree.
9015	JSEXEC	Return message from child module too long.
9016	JSADDS	Error 1155 came back but should not even send STD to JSADDS.
9017	JSEXEC	Unable to read time record in reactivation of module.
9019	JSMDBG	Unable to setup to read io-list sent by parent.
9020	JSEXEC	Unable to update time record in preparation to execute module.
9021	JSMDBG	Unable to read time record.
9022	JSMON	Got an error code from JSCHDS but should not have even called it because not have to be changing data sets from JSMON.

9024	JSMON	Unit 6 not assigned to JSOUT, but it should have been initialized to 6 in JSBLDA.
9025	JSJIRJ	Error in JSIN block data, invalid JILVAL value
9027	JSPRNT	Should not get RETCOD > 0 from JSJTPD.
9028	JSPRNT	String passed to JSIQLN not long enough to hold all name.
9030	JSINSQ	Invalid blank file name argument for nonempty Sequential File Table.
9040	JSMON	VAX error signaled by CL\$PRESENT.
9041	JSJIRJ	Error in use of LIL routines.
9042	JSEEXEC	Invalid value for MSGFLG returned from child in completion record.
9044	JSFIFP	Under MVS JSFIFP is being called for a file other than a JOSHUA data set file.
9045	JSFIFT	File spec will be too long.
9047	JSMACH	Under VAX/VMS, LIB\$GETSYI error getting system id.
9048	JSMACH	Under VAX/VMS, LIB\$GETSYI error getting machine type.
9049	JSMON	The data set requested for WRITE is already in the data set table which should not happen.
9050	JSOPSY	Under VAX/VMS, LIB\$GETSYI error getting operating system version.
9060	JSADDS	Attempt to add a standard data set to the Data Set Table with READ/WRITE access.
9061	JSCHDS	Invalid request to JSCHDS. You either have to change the file association or change the protection for an existing data set.
9062	JSMVFN	Under IBM/MVS, data set name too long.
9063	JSJIGV	Request for value for unknown keyword.

D REFERENCES

- [1] *American National Standard Programming Language FORTRAN*, ANSI X3.9-1978, American National Standards Institute, Inc., 1978.
- [2] *The JOSHUA (J80) Users' Manual*, WSRC-IM-92-97, B. W. Westmoreland, Westinghouse Savannah River Company, 1992.
- [3] *Assembler H Version 2 Application Programming: Language Reference*, GC26-4037, IBM Corporation.
- [4] *MVS/Extended Architecture Supervisor Services and Macro Instructions*, GC28-1154-2, IBM Corporation, 1985.
- [5] *MVS/Extended Data Administration: Macro Instruction Reference*, GC26-4014-1, IBM Corporation, 1984.
- [6] *MVS/Extended JCL Reference*, GC28-1352-1, IBM Corporation, 1985.
- [7] *OS/VS2 MVS System Programming Library: Job Management*, GC28-0627, IBM Corporation.
- [8] *Programming in VAX FORTRAN*, AA-D034D-TE, Digital Equipment Corporation, 1984.
- [9] *VAX/VMS Command Definition Utility Reference Manual*, AA-ZA08A-TE, Digital Equipment Corporation, 1984.
- [10] *VAX/VMS Message Utility Reference Manual*, AA-Z422A-TE, Digital Equipment Corporation, 1984.
- [11] *VAX/VMS System Services Reference Manual*, AA-Z501C-TE, Digital Equipment Corporation, 1984.
- [12] *VAX/VMS Run-Time Library Routines Reference Manual*, AA-Z502C-TE, Digital Equipment Corporation, 1984.
- [13] *VS FORTRAN Version 2 Language and Library Reference, Release 2*, SC26-4221-2, IBM Corporation, June 1987.
- [14] *VS FORTRAN Version 2 Programming Guide, Release 2*, SC26-4222-2, IBM Corporation Edition, June 1987.
- [15] *VS FORTRAN Version 2 Interactive Debug Guide and Reference, Release 2*, SC26-4223-0, IBM Corporation, January 1986.

DATE

FILMED

1 / 10 / 94

END