Laser-Scan Ltd.

LSLLIB

Technical Reference Manual

Issue 1.13 - 29-September-1993

The Facility Number Manager is: Paul Hardy
Consult him before allocating a new facility number for message definition.

Science Park, Milton Road, Cambridge, England CB4 4FY tel: (0223) 420414

Document "LSLLIB REF", Category "Technical"
Document Issue 1.13 J Barber, TJ Ibbs, CC Brunt, RW Russell (modified 29-Sep-1993)

CONTENTS

1 *LSLLIB documentation change record*

Issue 1.0 Tony J Ibbs, 27 February 1986
Tim Hartnall
Jamie Hulme

First issue of LSLLIB documentation

Issue 1.1 R W Russell, 12 May 1986

Chapter 10 - STRINGS
New version of RDYES that returns a condition code

Issue 1.2 Tony J Ibbs, 10 June 1986

Chapter 1 - INTRO
Remove discussion of meanings of severities (now in chapter 3). Alter pagination slightly.
Chapter 2 - LSLINIT
Describe check performed by EXPAND that LSL_INIT has been called.
Chapter 3 - ERRORS
Add explanation of how LSL programs should use message severities. Correct description of message definition format ("<" and ">" now used as delimiters). Correct extensions of files produced to match new practice. Add LSL_ADDMSG, and improve layout of routine descriptions. Add an example for MARK_POSN. Correct various typing errors.
Chapter 4 - EXPC
Correct typing errors.
Chapter 5 - EXPAND
Mention EXPAND's check that LSL_INIT has been called. Correct typing errors.
Chapter 7 - EXCEPTION
Correct description of LSL__BADINEQ and LSL__AMBINEQ to reflect new common /INEQUAL/.
Chapter 9 - NUMBERS
Correct informal descriptions of real numbers (form "a.b" was omitted).
Chapter 11 - FILENAME
Correct typing errors. EXPFLN now also returns LSL__NOFIELD, LSL__FILNOLEN.
Chapter 13 - FILE
Correct typing errors. FLRSTR has new argument **nchs**.
Chapter 14 - DCL
Correct typing errors - real**8** replaced by real*8 where necessary. Examples now call LSL_INIT, not TMRINI.
Chapter 15 - COMMANDS
Add new routine LSL_CMDERR, new common /INEQUAL/.
Chapter 18 - SYSTEM
TMRINI now called LSL_TMRINI. New common /EXIT_HANDLER/ used by LSL_TMRINI.

Issue 1.3 R W Russell, 7 July 1986

Chapter 10 - STRINGS
Correct silly message - used to say "Please answer with Y for Yes, N or <return> for YES"

Issue 1.4 Tony J Ibbs, Tim Hartnall, 6 October 1986

Chapter 2 - LSLINIT
change the name of the chapter. Add the LSL_EXIT routine.
Chapter 3 - ERRORS
reformat some of the examples, add example of command file use of NEWMESSAGE. Document the use of LSL$DEBUG_TRACE to convert LSL_PUTMSG to LSL_SIGNAL.
Chapter 9 - NUMBERS
add a description of LSL_RDREAL_WHOLE and LSL_RDDBLE_WHOLE. These were intended as internal routines, but are useful in some programs - for instance IPATCH requires them.
Chapter 10 - STRINGS
READSTR - add documentation for **term_cond**, and clarify what a call of RDCH would return after READSTR has finished. Undo bolding after the RDYES example.
Chapter 14 - DCL
document DCL_PARSE, and remove the "see below" from error return descriptions. Correctly type /CLD/CARRAY as 'char', and note the length of strings as 128 characters.
MAX_ARR parameter removed from CLD.CMN and instead MAX_LONG and MAX_REAL introduced. MAX_REAL is set to be 128 and is used to dimension RARRAY and DBLRAY arrays. MAX_LONG is set to be 1024 and is used to dimension IARRAY. DCL integer range decoding is now less likely run out of storage space for the expanded arguments.
A third Fortran example (EXAMPLE_3) introduced to show use of qualifier keyword arguments.
Fortran examples now show use of LSL_EXIT.
Chapter 17 - MAGTAPE
add new **type** argument to MTINIT and MTONIT
Chapter 18 - SYSTEM
add the routines TEST_FOREIGN and TEST_TERM

Issue 1.5 Tony J Ibbs, 5 November 1986

Chapter 2 - LSLINIT
LSL_EXIT - informations now converted to success
Chapter 3 - ERRORS
split into two chapters - ERROR_DEFN and ERROR_RTNS. Subsequence chapters are therefore renumbered
Chapter 17 - IFF
new IFF opening routines
Appendix B - COMPARISON
new appendix - comparison of VIOLIB/CMDLIB to LSLLIB
Most chapters
correct typos

Issue 1.6 Bill James, Tony J Ibbs, Tim Hartnall, 29 April 1987

Chapter 1 - INTRO
"Differences from VIOLIB/CMDLIB" is actually an appendix of this document, not a separate document. Document that return codes from functions may be (and often are) ignored in VAX Fortran.
Chapter 2 - LSLINIT
LSL_INIT has new argument **tracing**, and now calls LSL_DEBUG_TRACE.
Chapter 3 - ERROR_DEFN
The .GENMSG* files are no longer generated by NEWMESSAGE, and thus the format of message definition lines is simplified.
Chapter 4 - ERROR_RTNS
LSL_PUTMSG no longer checks LSL$DEBUG_TRACE itself. Document LSL_DEBUG_TRACE. Add another example of the use of LSL_PUTMSG and LSL_ADDMSG. Use **ok** as the name of the return variable for all functions.
Chapter 6 - EXPC
DEF_EXPMAX is now 255. New routine SAVE_EXPMAX.
Chapter 7 - EXPAND
%S and %A now default to 255 characters.
Chapter 8 - TXTC
DEF_TXTLIM is now 255. New routine SAVE_TXTLIM.
Chapter 12 - FILENAME
Use **ok** as the name of the return variable for all functions.
Chapter 13 - TERMINAL
Use **ok** as the name of the return variable for all functions.
Chapter 14 - FILE
New routines FLRSVL and FLWSVL save the current file selection.
Improve the documentation of the file open and file select routines.
Use **ok** as the name of the return variable for all functions.
New routine FLWUSH flushes internal buffers to disc.
Chapter 15 - DCL
Use **ok** as the name of the return variable for all functions.
New routines START_LOG and STARTLCM added for creation of Laser-Scan utility /OUTPUT listing file and /LITES2 command file.
Chapter 16 - COMMANDS
Use **ok** as the name of the return variable for LSL_CMDERR.
Chapter 17 - IFF
Improve the introduction to this chapter - mention the fact that the LSL__IFFxxx error messages often require arguments.
Chapter 18 - MAPPED
These routines are now all fully implemented. Document new arguments to VIO$OPEN_SEC, and improve its description. Document VIO$UPDATE_SEC.
Chapter 19 - MAGTAPE
Use **ok** as the name of the return variable for LSL_CMDERR.
Chapter 20 - SYSTEM
New routine TRNALL added - does logical name translation.
New routine SET_OUTBAND_AST added.
Appendix B - COMPARISON
Update the comparison of current LSLLIB with VIOLIB/CMDLIB.

Issue 1.7 Tony J Ibbs, 3 August 1987

Chapter 3 - ERROR_DEFN
Changed totally to reflect the new NEWMSG program, and the new message definition file layout.

Issue 1.8 Tony J Ibbs, Clarke Brunt, R W Russell, Jamie Hulme
11 September 1987 to 3 March 1988

Chapter 1 - INTRODUCTION
Correct the location of th library and its sources.
Chapter 2 - LSLINIT
Add documentation of setting default terminal i/o routines.
Chapter 3 - ERROR_DEFN
Add new REPLACE command and description.
Chapter 4 - ERROR_RTNS
Typographical error corrected.
Chapter 7 - EXPAND
Add new escape sequences %DD and %DT
Chapter 9 - CHARACTER
In documentation of SETWIN, advise use of BSLN first.
Chapter 11 - STRINGS
Correct the documentation of READSTR to reflect the corrected routine.
Add new routine SIGCHS.
Chapter 13 - TERMINAL
New routines LSL_SET_INPUT and LSL_SET_OUTPUT.
VIO$GET_INPUT and VIO$PUT_OUTPUT should no longer be replaced.
Chapter 14 - FILE
Correct FLWOVM to FLWOVR in index.
Chapter 15 - DCL
Correct documentation of behaviour of DCL_PARSE.
Chapter 16 - COMMANDS
Add documentation of REMCMD.
Add documentation of LSL_SORTAB, and LSL_INEQUAL_CMD_TABLE.
Add documentation of FIND_CMDNAME.
Chapter 19 - MAGTAPE
Correct reference to MTICNT (should be bytcnt) in MTIRDB.
Chapter 20 - SYSTEM
Add LSL_NUM_CHAND to index.
Add recommendation of G for VIO$1ST and VIO$GNA.
Add new routines CVT_DATE, CVT_DAY_STR and CVT_TIME_STR.
Add new routines CVT_DMY_DAY and CVT_DAY_DMY.
Chapter 21 - SORT
Correct various typographical errors.

Issue 1.9 Tony J Ibbs, Clarke Brunt, Ron Russell
15 March 1988 to 8 March 1989

All chapters
Amend to use the new standard documentation titling convention.
Chapter 3 - ERROR_DEFN (NEWMSG)
NEWMSG now has the new qualifier /HL=integer, and replaces underlines in message texts output to the RUNOFF file by double underlines.
Chapter 5 - EXPAND, Chapter 9 - NUMBERS
Document new routines READANG and DISPANG

Chapter 14 - FILEIO
New routines FLRFNB, FLWFNB and FLWRDL
Chapter 16 - COMMANDS
Document the new **D** control for command tables - allows commands to contain non-alphabetic, non-underline characters. Also, an extra optional argument to INITAB to allow the same thing.
Document new routine RDINEQ, and remove LSL_INEQUAL_CMD_TABLE which is no longer to be considered public.
Change description of INITAB, bytarr and bytsiz now unused.

Issue 1.10 Clarke Brunt 6 January 1990

Chapter 17 - MAGTAPE
Document new use of <pe> argument to MTONIT to set density 6250.

Issue 1.11 Clarke Brunt 8 August 1990

Chapter 20 - SYSTEM
Change name of routine WAIT to LSL_WAIT.

Issue 1.12 Clarke Brunt 25 March 1992

Chapter 21 - SORT
Change argument type of COUNT argument to the 3 sort routines from word to long. Also the index arguments for the supplied routines CF, COPY, and SWAP. The arguments passed to these were always actually long, but were documented as word.

Issue 1.13 Jon Barber 29 September 1993

Chapter 3 - NEWMSG
New qualifier /DTILIB for NEWMSG to output references to the Matrix DTILIB library messages, rather than to the default IFFLIB messages.

CHAPTER 1

INTRODUCTION

LSLLIB is Laser-Scan's VAX native mode standard subroutine library.

It provides a set of I/O routines, generally used in Laser-Scan in preference to Fortran I/O. These include terminal, file and magnetic tape read/write utilities, flexible encode/decode facilities, and command reading facilities.

LSLLIB also includes various other routines which are of general use to Laser-Scan programmers.

LSLLIB is descended from two previous VAX libraries, VIOLIB and CMDLIB. These in turn are descended from LIOLIB, which was the RSX Laser-Scan I/O library. However LSLLIB has only limited compatibility with any of the previous libraries. The appendix "Differences from VIOLIB/CMDLIB" discusses this in more detail.

The library may be found in LSL$LIBRARY:LSLLIB.OLB, and its common and parameter files are in LSL$CMNLSL:

Note that in general LSLLIB should be the last library in any link instructions as some of the basic routines such as WRITEF are referenced from other libraries. LSLLIB should not be included with VIOLIB and CMDLIB, as many routines have the same name, but different arguments.

The library sources are in LSL750::LSL$SOURCE_ROOT:[LSLMAINT.LSLLIB...], and the documentation sources are in LSL$DOC_ROOT:[LSLMAINT.LSLLIB].

1.1 *Documentation Notation*

The following conventions are followed:

* all arguments are fully declared for each routine.

* the following input/output declarations are made:

  out - this variable will be written to by the routine.

  in - this variable is read by the routine - it is not written to.

  i/o - this variable may be both read by the routine, and written to.

* the following argument types are used:

  word - this is a Fortran INTEGER*2, a 16 bit variable.

  long - this is a Fortran INTEGER*4, a 32 bit variable. Note that this is the default integer size on the VAX, and is assumed unless there are good reasons for using a word.

  logical - this is a Fortran LOGICAL variable

  byte - this is a Fortran BYTE or LOGICAL*1 variable

  real - this is a Fortran REAL variable

  dreal - this is a Fortran REAL*8, or DOUBLE PRECISION variable

  char - this is a Fortran CHARACTER variable, but see the section on 'Fake strings' in the System chapter

  external - this is a routine declared as EXTERNAL in the calling routine

* arguments are compulsory, unless enclosed in brackets (the characters '[' and ']'). Note that the specification of optional arguments is not always formally correct - for instance, if this documentation describes

  call FRED(arg1, [arg2], [arg3], [arg4])

  then it *really* means

  call FRED(arg1 [, [arg2] [, [arg3] [, [arg4]]]])

* VAX FORTRAN allows functions to be CALLed - that is, the return code from a function can be discarded by using CALL <function>. It is thus not uncommon to find programs which ignore return codes documented in this manual, assuming that the relevant function has succeeded - common examples include EXPAND, LSL_PUTMSG, FLWSTR/FLWLIN.

* Occasionally, variable values are represented in angle brackets - for instance in the declaration of EXPAND, the description of the arguments talks about **arg<n>** where **<n>** represents an appropriate integer.

* some common blocks are essentially used as extra I/O to routines (for instance /CMDCOM/ for RDCOMM), and the variables in these are declared as for routine arguments (see above). However, others (such as /TXTC/) are used in a less predictable fashion. In these cases, the following common block declarations are made:

  public - the variable being described is one which the user is free to refer to. In some cases, the user may wish (or need) to set this variable.

  private - it is not expected that the user will wish to refer to this variable. However, there is nothing to stop the user doing so, providing they are sure of the consequences!

* parameter files are described using the same data-types as used for variables (long, word, real, etc)

* common blocks are normally referred to by enclosing a name in slashes - this name is that of the file which contains the common block under discussion. For example, /EXPC/ means the INCLUDEd file LSL$CMNLSL:EXPC.CMN which actually contains the common block COMMON /LSL_EXPC/

1.2 *Naming Conventions*

* all common block names are prefixed by 'LSL_', e.g. COMMON /LSL_EXPC/. The filenames are not prefixed and will normally be called after the standard (i.e. non-character) common block in the file, e.g. LSL$CMNLSL:EXPC.CMN

* all undocumented routine names are prefixed by 'LSL_'. Note that this prefix is also used for some of the documented routine names, as there is no specific naming convention for them.

* the source code for every Fortran routine is contained in its own separate file, filenames being the same as the routine names.

1.3 *Testing the severity of errors*

The error codes returned by LSLLIB routines are either LSL__ error codes, or system error codes. In either case, the bottom three bits of an error code indicate its severity.

The value of the bottom three bits will be one of

```
STS$K_SUCCESS - success
STS$K_INFO    - information
STS$K_WARNING - warning
STS$K_ERROR   - error
STS$K_SEVERE  - fatal error
```

The bottom three bits may be extracted using the mask STS$M_SEVERITY, and the bottom single bit may be extracted using the mask STS$M_SUCCESS.

The STS$ values are declared for Fortran in the FORSYSDEF module ($STSDEF), which can be included by the statement

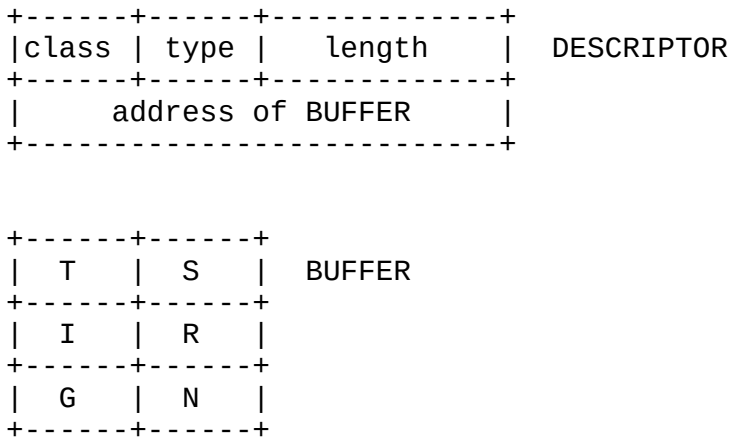
```
INCLUDE    '($STSDEF)'
```

Alternatively, note that errors all test as **false**, and successes as **true** (ie they are respectively even and odd).

Further information can be found in the "Guide to Programming on VAX/VMS (FORTRAN Edition)", Chapter 10. For an explanation of when different severities are conventionally used in LaserScan programs, see the "Error messages" chapter in this document.

1.4 *Creating and using 'Fake strings'*

On the VAX, strings are implemented as follows:



DESCRIPTOR is the character descriptor for the string - length is the size of the BUFFER, in bytes, and class and type are zero for this form of string. It is the character descriptor that is referred to when talking about a string.

In Macro, this could be represented as:

```
DESCRIPTOR:
LENGTH:         .WORD   0               ; no of chars in BUFFER
TYPE:           .BYTE   0               ; type  information
CLASS:          .BYTE   0               ; class information
POINTER:        .LONG   BUFFER          ; address of the buffer

BUFFER:         .BLKB   80              ; the buffer itself
```

It is possible to create a character descriptor in FORTRAN - this is done by mimicking the above structure. For instance:

```
        INTEGER         MAX_BUFLEN
        PARAMETER      (MAX_BUFLEN=255) ! maximum buffer size
        INTEGER*2       STRLEN          ! length field
        INTEGER*4       DESCR(2)        ! descriptor
        EQUIVALENCE    (DESCR(1),STRLEN)
        BYTE            BUFFER(MAX_BUFLEN)

         ...

        DESCR(1) = 0                    ! zero the type/class fields
        DESCR(2) = %LOC(BUFFER)         ! set up the address field
        STRLEN   = MAX_BUFLEN           ! and set the length correctly
```

Note that although this character descriptor may be passed to a subroutine, and used as a character string within that subroutine, the level at which it is declared 'knows' that it is only an integer. Thus it is not possible to do

things such as LEN(DESCR), or REAL_STRING = DESCR in the declaring routine.

Also, care must be taken when passing it to routines that will write to the 'string' - for instance, if routine COPY_STRING is defined as follows:

```
        SUBROUTINE COPY__STRING( FROM, TO )
C
        IMPLICIT NONE
C
        CHARACTER*(*)   FROM    ! source string
        CHARACTER*(*)   TO      ! destination
C
        TO = FROM
C
        RETURN
        END
```

and we are to copy into a 'fake string', then the length of the fake string must be carefully manipulated - for instance

```
C
C set the 'fake string' length - preferably to the length of the
C string we are copying, but not to more than the buffer length!
C
        STRLEN = MIN( LEN(TEXT), MAX_BUFLEN )
C
C and copy the TEXT string into our 'fake' string
C
        CALL COPY__STRING( TEXT, DESCR )
```

This method of declaring 'fake strings' is used with /TXTC/ and /EXPC/, as described above, to produce TXTDSC and EXPDSC. It is also used in /CMDCOM/ (the RDCOMM common block file).

1.5 *Linking with LSLLIB*

There are two versions of LSLLIB available - the normal library and the shared version. Programs may be linked with either. The advantages of using the shared library are that the image will be smaller, that image startup should be quicker, and that LSLLIB bug fixes will take effect without having to relink the program. Much of Laser-Scan's standard software is now standardly linked with the shared version of LSLLIB - in particular, IMP and LITES2.

In order to link with the library *unshared*, a normal link statement of the form:

```
$  link/map  program,sub1,sub2,etc, -
             lsl$library:lsllib/lib
```

In order to link with the shared library, a command of the form

```
$  link/map  program,sub1,sub2,etc, -
             lsl$library:lslshr/opt
```

must be used - this uses a link options file to tell the linker what to do. If sharable IFFLIB is also being used, then we have to use:

```
$  link/map  program,sub1,sub2,etc, -
             lsl$library:iffshr/opt, -
             lsl$library:lslshr/opt
```

Note that the linker requires that options files are specified last - that is, any other libraries, etc, must be specified before the options files.

CHAPTER 2

THE START AND END OF PROGRAMS

2.1 *Initialising the library*

Before using any of the routines in LSLLIB, the library must be initialised by a call of LSL_INIT. Failure to do this will result in obscure and fairly unpredictable errors when trying to use other routines, or a specific complaint when EXPAND is used.

call LSL_INIT([timer], [tracing])

in - logical	**timer**	false if not to call LSL_TMRINI
out - logical	**tracing**	true if logical name LSL$DEBUG_TRACE exists

LSL_INIT performs the following actions:

* references the LSLLIB messages, so that they will be available for the rest of the program
* sets the default input and output routines to be LIB$GET_INPUT and LIB$PUT_OUTPUT respectively.
* sets all the address values in the LSLLIB common blocks - this includes creating the 'fake descriptors' in various blocks.
* sets the default lengths for the 'descriptors'
* sets the /STATUS/LSL_STATUS value to be LSL__NORMAL
* if **timer** is true or absent, calls LSL_TMRINI to set up an exit handler which will use WRITEF to report on the times used by the program when it exits
* calls LSL_DEBUG_TRACE. If the logical name LSL$DEBUG_TRACE is defined, then all calls to LSL_PUTMSG will be converted to calls of LSL_SIGNAL. If **tracing** is present, it will be set true if LSL$DEBUG_TRACE exists, and false otherwise.

The following specific problems result from not calling LSL_INIT:

* any call of EXPAND will result in the message:

EXPMAX is zero - LSL_INIT has not been called

which is output using LIB$PUT_OUTPUT, and LSL_STATUS will be set to SS$_ABORT.

Note that WRITEF and FLWRTF also call EXPAND.

* any attempt to read input or write output using the LSLLIB routines will result in an access violation error.

* calls of LSL_PUTMSG, LSL_ADDMSG and LSL_SIGNAL will fail to recognise the LSLLIB error numbers.

* any attempt to use TXTDSC, EXPDSC or any other supplied 'descriptor' will fail - firstly because the address part is not set up, and secondly because the length part is not set

* the value in /STATUS/LSL_STATUS will probably be zero, rather than LSL__NORMAL. Note that if EXPAND has been called, then it will be set to SS$_ABORT

2.2 *Exiting from the program*

call LSL_EXIT([status])

in - long **status** the final success/error code

LSL_EXIT is used instead of the normal Fortran EXIT routine. It works as follows:

```
if status is supplied
then
   internal_status := status
else
   internal_status := /STATUS/LSL_STATUS
endif

if internal_status is a customer code
then
   remember its severity
   if internal_status is a success or informational message
   then
      internal_status := SS$_NORMAL
   else
      internal_status := SS$_ABORT
      set the severity of internal_status to what we remembered
   endif
endif

call EXIT( internal_status )
```

The effect is thus always to exit with a DEC defined success/error code. If the program would have done that anyway, then LSL_EXIT is equivalent to a direct call of EXIT. Otherwise, it converts the exit status to the simplest form of DEC exit status.

CHAPTER 3

ERROR MESSAGE DEFINITION

3.1 *Introduction*

This chapter describes how to define error message files.

Error messages are defined in a text file in a standard format, and processed by a supplied program to produce the various necessary files.

There are two 'groupings' of error message -

1. Those provided by a library (for instance LSLLIB). These are always available when the library is linked with a program.

2. Those provided for an individual program. Each program will have its own message definition file, which is kept in its source directory.

3.2 *The message definition file*

The error message definition file describes the messages available for a particular library, package or individual program.

3.2.1 *General layout rules*

Blank lines are ignored (except within explanatory texts). Comments start with an exclamation mark ("!") and may be freely used throughout the file. Note that exclamation marks in message and explanatory texts do not start comments.

Spaces and tabs may be used as delimiters between different data fields, and are otherwise ignored (except within message and explanatory texts). Also, the case of the data fields is irrelevant (again, except within message and explanatory texts).

3.2.2 *Message definition*

Each message is defined by the sequence (#1):

```
<severity>    <spc>  <ident>  <gap>  \<message text>\
REPLACE       <spc>  %<expand sequence> = \<replacement text>\
EXPLANATION   <gap>  \<explanatory text>\
USERACTION    <gap>  \<explanatory text>\
```

where:

<severity>	is the severity of this message - it may be any one of SUCCESS, INFORMATIONAL, WARNING, ERROR, SEVERE or FATAL (the last two are equivalent).
<ident>	is the identifying mnemonic for this error message - this should be an abbreviation of the message text. Abbreviations should be made in a consistent manner - for instance, TOOMNYPTS for "Too many points", UNEXPEOF for "Unexpected end of file".
<message text>	is the format statement used when outputting a message with LSL_PUTMSG or LSL_ADDMSG - it is processed by EXPAND before being output.
<expand sequence>	is any sequence of charaters that might follow a percent ("%") symbol in an EXPAND format.
<replacement text>	is the text with which to replace the first occurrence of that expand sequence in <message text> when outputting it to the RUNOFF file.
<explanatory text>	is the EXPLANATION or USERACTION text, which is output to the RUNOFF file to explain the message.
<spc>	is a sequence of spaces and/or tabs.
<gap>	is a (possibly empty) sequence of <spc> and new lines.

The following general notes apply to the message and explanatory texts :-

- Message, replacement and explanatory texts must be enclosed by backward slashes - to include a backward slash in the text itself, type it twice (ie "\\").

- The triangular bracket characters ("<" and ">") are used to delimit the message texts internally, and thus may not be used within a message text. They may, however, be used within the replacement or explanatory

---

#1: Early message definition files may be found to have two text fields defined for some messages. In such cases, the first text is plain text, with no EXPAND formats, and the second text is the one that should be passed to LSL_PUTMSG, etc. This form of message definition file is now obsolete. The message generation utility NEWMSG will output an explanatory error message if it encounters such a message definition, and the file should be corrected.

texts.

- No line in the message file may contain more than 132 characters.
- The "=" sign in a REPLACE clause may have spaces or tabs on either side.
- Explanatory texts are copied directly to the runoff file. They may thus be split over several lines, and may include runoff commands.
- Where a message quotes something the user typed, this should conventionally be enclosed in double quotes.

The REPLACE command is only necessary if there are EXPAND escape sequences in the message text. Several REPLACE clauses may be specified after the command, separated by <spc>. When the message text is being output to the RUNOFF file, each REPLACE clause will be read, and the expand sequence will be found and replaced by the corresponding replacement text.

Technically, the EXPLANATION and USERACTION fields are also optional, in that the NEWMSG program will still process the file succesfully if they are absent. This allows older message files without runoff generation text to be processed. However, all new files should include these commands.

3.2.3 *Example*

The following is a very simple message definition file.

```
! define messages for the IMPOSSIBLE program

INFORM   MISSION   \Your mission, should you choose to accept it, is\
EXPLANATION        \This message is output before the MISSTEXT message.\
USERACTION         \As for the MISSTEXT message.\

INFORM   MISSTEXT  \%S\
REPLACE  %S = \'mission text'\
EXPLANATION
\This message is a short text describing the mission that is being
offered. There is no compulsion to accept it, although we know you will.\
USERACTION
\Reply immediately, with acceptance or rejection of the mission.\

WARNING  DESTRUCT  \This message will self-destruct in %N second%m\
REPLACE  %N = \'integer'\ %m = \s\
EXPLANATION        \This message is self-explanatory.\
USERACTION         \None.\

FATAL    BANG         \*** %S ***\
REPLACE  %S = \'loud noise'\
EXPLANATION
\This message is used to indicate that a loud noise, 'bang', has
been made.\
USERACTION         \Duck.\
```

Processing this file with NEWMSG will define five messages, which might be referred to as IMP__MISSION, IMP__MISSTEXT, IMP__DESTRUCT and IMP__BANG from within a program. Thus

```
CALL LSL_PUTMSG( IMP__MISSION )
CALL LSL_ADDMSG( IMP__MISSTEXT, 'to use this library' )
```

will output the following messages:

```
%IMPOSSIBLE-I-MISSION, Your mission, should you choose to accept it, is
-IMPOSSIBLE-I-MISSTEXT, to use this library
```

NEWMSG can also be used to generate a runoff file which describes the messages, and can be .REQUIREd in the documentation for the utility or library using these messages. See the description of the NEWMSG program below for more details.

3.2.4 *Message Severities*

The message severities used in Laser-Scan programs should normally follow the following conventions:

Success - the message is reporting that the program has performed normally and succeeded.

Informational - the message is for information only, and does not affect the program's successful completion. Informational messages are used to report on the current state of the utility, or to provide further information after an error message.

Warning - the message is to warn the user of something strange happening. The program will attempt to continue, but the results should be checked.

Error - the message is to report a definite error in processing. The program will give up processing the input data, and the output data produced so far is thus suspect - the program may delete it as it exits.

Fatal - the message is to report a very severe error - this normally reflects program failure, or possibly some sort of system failure. This sort of error may need to be reported to Laser-Scan.

3.3 *Building a program with user defined messages*

The following example assumes that the user is creating a new program IMPOSSIBLE. All the steps required to prepare a message definition source file and to generate the required object modules and parameter files (using LSL$EXE:NEWMSG.EXE) are shown. For full details of the use of the NEWMSG utility, see the NEWMSG documentation later in this chapter.

1. The FORTRAN source file for the program should include the message parameter file IMPOSSIBLE.PAR ie:

```
        INCLUDE 'LSL$CMNIMPOSSIBLE:IMPOSSIBLEMSG.PAR'
```

   or if an ADC include statement is preferred:

```
***     PARAMETER/LSL$CMNIMPOSSIBLE:IMPOSSIBLEMSG/
```

2. A Laser-Scan format message definition file should be prepared using the layout defined in "Message definition" above. For our example, this is called IMPOSSIBLE.MES

3. Consult the Facility Number Manager before specifying the facility number. The name of the current FCM is printed on the title page of this document. It is important that numbers are allocated uniquely, because if several programs or libraries had the same facility number, then the wrong error messages could be generated.

4. Run the NEWMSG program. This may be done using the following commands:

```
   $ NEWMSG IMPOSSIBLE
/NUMBER=1234/PREFIX=IMP____/PARAM=(FORTRAN,C)/RUNOFF
   $ @IMPOSSIBLE.TEMP__COM
```

   The facility number assigned to this program is 1234, we are using a facility prefix of IMP__ (since IMPOSSIBLE__ would be rather cumbersome), and we are generating parameter files for FORTRAN and C programs, as well as producing a runoff file, IMPOSSIBLEMSG.RNO

   After the command file has been run, we will have parameter files IMPOSSIBLEMSG.PAR and IMPOSSIBLEMSG.H, and we will also have a message object file IMPOSSIBLE.PROMSG_OBJ. This should then be linked with the rest of the program:

```
   $   link'debug' impossible,subroutines,-
                   impossible.promsg_obj,-
                   lsl$library:lsllib/lib
```

PROGRAM NEWMSG

FUNCTION

NEWMSG reads a user supplied message definition file, and can be used to produce:

- a runoff file describing the messages for inclusion in program documentation
- an object module which is linked with a program to make the messages available
- parameter files defining parameters (constants) for each message's identifying number, for inclusion in program sources

---

FORMAT

$ NEWMSG input-message-file

Command qualifiers	**Defaults**
/DESTINATION=directory	/DESTINATION=SYS$DISK:[]
/DTILIB	No DTILIB references produced
/HL=integer	/HL=1
/KEEP	Do not keep temporary files
/[NO]LOG	Minimal logging
/NAME=string	See below
/NOOBJECT	Generate message object file
/NUMBER=integer	Required
/[NO]PARAMETERS=(type,...)	/PARAMETERS=FORTRAN
/PREFIX=string	See below
/RUNOFF	No RUNOFF file produced

PROMPTS

Input message file: input-message-file

PARAMETERS

input-message-file

- specifies the message definition file that is to be read. The contents of this file is described below. Any parts of the filespec that are omitted will be filled out from the default of SYS$DISK:[].MES

COMMAND QUALIFIERS

/DESTINATION=directory

- specifies the directory in which the parameter and runoff files should be created. All temporary files (including the .TEMP_COM command file) will also be created in this directory. If /DESTINATION is not specified, then the current directory will be used. Any parts of a full directory specification that are omitted will be taken from the default of SYS$DISK:[]

  Note that only a device and directory may be specified. Node names are not supported, and file names, extensions and version numbers are obviously not allowed.

/DTILIB

- specifies that a runoff file is to be produced with references to the Matrix DTILIB library messages, instead of to the default IFFLIB messages. This is useful for the automatic message generation for those programs that use mainly DTI files, such as those in the Matrix or TVES packages.

/HL=integer

- specifies that the header level (.HL) commands output to the /RUNOFF file should use the level number specified. This qualifier is ignored if /RUNOFF is not present. The header level defaults to 1.

/KEEP

- specifies that all temporary files should be kept, rather than being deleted. This qualifier is useful when trying to diagnose a problem with the program. The effect of /KEEP is as follows

  - both the sequential and indexed /RUNOFF workfiles (.TEMP_SEQ and .TEMP_IDX) will be kept
  - the temporary command file (.TEMP_COM) will not delete
    - itself
    - the output message definition file (.PROMSG and .PROMSG_TEMP) and its ADC.IDE
    - the GENPAR program files, GENPAR.FOR, .OBJ and .EXE

/LOG
/NOLOG

- if the /LOG qualifier is specified, then a message will be output to the terminal whenever any file is opened, closed or otherwise manipulated. Also, each message written to the output message definition file (.PROMSG file) will be reflected. If the /NOLOG qualifier is specified, then no information will be output (apart from the timing information presented when the program terminates). If neither qualifier is specified, minimal information about what the program is doing will be output as it starts.

/NAME=string
- specifies the facility name to be used for the messages being defined. This defaults to the first 9 characters of the message definition file name, and should normally be the same as the program or utility name.

/NOOBJECT
- specifies that the message object file is not to be generated. If both /NOOBJECT and /NOPARAMETERS are specified, then the temporary command file will not be generated. The default is to produce the message object file.

/NUMBER=integer
- specifies the facility number to be used for the messages being defined. This is a required qualifier - the facility number must always be specified.

  The facility number must be unique for each message definition file. Consult the Facility Number Manager (see the title page of the LSLLIB manual) for an appropriate number.

/PARAMETERS=(choice,...)
/NOPARAMETERS
- specifies which parameter files are to be generated. The possible choices are:

  - o FORTRAN - generate a .PAR file for inclusion in FORTRAN programs. This is the default.
  - o MACRO32 - generate a .MAR file for assembling with MACRO programs.
  - o C - generate a .H file for inclusion in C programs

  The parameter files will be created in the destination directory, and will have a file name obtained by appending "MSG" to the message definition file's name.

  By default, the program will produce a .PAR file for inclusion in FORTRAN programs. If /NOPARAMETERS is specified, then the program will not generate any parameter files. If both /NOPARAMETERS and /NOOBJECT are specified, then the temporary command file will not be generated.

/PREFIX=string

- specifies the facility prefix. This is the prefix used for all message names. For non-DEC messages it is conventionally the program name followed by two underlines, and thus the default prefix is the message definition file name (truncated to 14 characters if necessary) followed by two underlines. This qualifier is normally used for programs like IDIFFERENCE, where the program name is rather long, and a prefix such as IDIFF__ is preferred.

NOTE

Currently, a bug in the system MESSAGE utility restricts the prefix string to 9 characters. This problem may be fixed in later versions of VMS (eg, version 5).

/RUNOFF

- specifies that a runoff file is to be produced, documenting the messages defined in the message file. The file produced will be placed in the destination directory. Its file name will be produced by appending "MSG" to the message definition file's name, and it will have extension .RNO

  In creating the runoff file, NEWMSG uses a temporary workfile, which is converted from sequential to indexed sequential form. In order to perform this conversion, the file LSL$LOOKUP:MESSAGE_INDEX.FDL is required. This file is supplied with the program.

DESCRIPTION

Files produced

NEWMSG reads a message definition file and produces various other files as output. One of these files is a temporary command file, which can then be obeyed to generate the actual parameter files.

The following files are generated in order to create the message object file and the parameter files:

- **intermediate message file** - extension .PROMSG
  This is the input for the MESSAGE utility, and is thus not created if /NOOBJECT is specified.

- **message generation program** - called GENPAR.FOR (temporary)
  This is the source file for a program to create the parameter files requested, and is thus not generated if /NOPARAMETERS is specified.

- **command file** - extension .TEMP_COM (temporary)
  This command file is created to run the MESSAGE utility on the .PROMSG file, creating a .PROMSG_OBJ file, and to compile, link and run the GENPAR.FOR file, creating the required parameter files. Its file name is the same as that of the input message file. It is not created if both /NOOBJECT and /NOPARAMETERS are specified.

  After it has created the .PROMSG_OBJ file, it will purge it, and after it has run the GENPAR program, it will purge the parameter files that it has created. It will also, unless /KEEP is specified, delete the .PROMSG file, GENPAR.* and itself.

The following files are created in order to create the .RNO output file (and are thus not created if /RUNOFF is not specified):

- **sequential runoff workfile** - extension .TEMP_SEQ (temporary)
  Contains messages and their descriptions

- **indexed runoff workfile** - extension .TEMP_IDX (temporary)
  This is an indexed sequential version of the sequential runoff workfile, and is used for looking up the text associated with each message. The FDL file LSL$LOOKUP:MESSAGE_INDEX.FDL is used to specify the form of this file.

- **runoff file** - extension .RNO
  This is the file documenting the messages in the input file, and can be included in the documentation for the utility to which the messages belong.

How the program works

The sequence of operations performed by the program is:

- Open the message definition file

- For each message in the file:
  - Output the message severity, ident and text to the .PROMSG file
  - Output code for the message to the GENPAR.FOR file
  - Output the message severity, ident, text and descriptions to the sequential runoff workfile
  - Place the message ident and severity into an internal array

- Create the temporary command file:
  - Write the DCL to produce the .PROMSG_OBJ file from the .PROMSG file
  - Write the DCL to compile, link and run GENPAR.FOR

- Produce the runoff file:
  - Convert the sequential runoff workfile into indexed form
  - Sort the internal array of message severities and idents
  - For each message, in order of increasing severity, and then alphabetical ident order:
    - Read the message text from the indexed workfile
    - Convert any underlines in the message text to double underlines
    - Perform any %string replacements indicated for this message text
    - Write the amended message text to the .RNO file
    - Read each line of explanatory text from the indexed workfile, and write it to the .RNO file

---

EXAMPLES - THE MESSAGE DEFINITION FILE

The following is an extract from the message definition file used to produce the messages for NEWMSG. The result of running NEWMSG/RUNOFF on the full file (followed by suitable editing of the actual message texts) can be seen in the MESSAGE documentation for NEWMSG, below.

```
!
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
! E R R O R  messages
!
ERROR   BADFACNUM       \facility number %N is not in the range %N to %N\
REPLACE %N = \'integer'\   %N = \'integer'\   %N = \'integer'\
EXPLANATION
\Only a certain range of facility numbers is supported. Any numbers below
1025 or above 2047 will not be allowed.\
USERACTION
\All facility numbers should have been approved or assigned by the facility
numbers manager. Do not try to invent your own number - this could cause all
sorts of problems.\

ERROR   CONVCONV        \error in CONV$CONVERT - converting runoff workfile\
EXPLANATION
\An error occurred when trying to convert the sequential runoff workfile into
an indexed file, so that the .RNO file can be created. This message should
be preceded by a message from the CONVERT utility itself, explaining what
went wrong.\
USERACTION      \Dependant upon the associated CONVERT error message.\

ERROR   CONVERROR       \%I4 message records rejected\
REPLACE %I4 = \'integer'\
EXPLANATION
\This message indicates that some of the records in the sequential runoff
workfile could not be converted to the indexed form. A message from the
CONVERT utility may precede this message, giving more details. The offending
records will have been written to a file for inspection - the EXCEPTION
message will follow, indicating the name of that file.\
USERACTION
\Inspect the original message file, and the exception file. The records in
the exception file ^*should\\* be of the form
.blank
        _<severity_> _<type_> _<count_> _<ident_> _<text_>
.blank
where these fields are (respectively) one of S,I,W,E or F, one of M,E or U,
a two digit number, a fifteen character message name and a line of text.
If the record in the exception file is not of this form, then report the
problem, with as many details as possible.\
```

EXAMPLES - USING THE PROGRAM

```
$ NEWMSG NEWMSG /NUMBER=1026/RUNOFF<CR>
Message file:     NEWMSG.MES
Facility number:  1026
Facility name:    NEWMSG
Facility prefix:  NEWMSG__
Generating output for FORTRAN RUNOFF
Generating output for message object module
$ SET VERIFY<CR>
$ @NEWMSG.TEMP_COM<CR>
$!
$! Temporary command file generated by NEWMSG
$!
$       on error then continue
$!
$       adc  SYS$DISK:[]NEWMSG.PROMSG_TEMP/co=SYS$DISK:[]NEWMSG.PROMSG
$       message/nolist/obj=SYS$DISK:[]NEWMSG.PROMSG_OBJ  SYS$DISK:[]NEWMSG.PROM
SG_TEMP
$       if .not.$status then goto had_an_error
$       pvv  SYS$DISK:[]NEWMSG.PROMSG_OBJ
$       purge/nolog  SYS$DISK:[]NEWMSG.PROMSG_OBJ
$       delete/noconfirm  SYS$DISK:[]NEWMSG.PROMSG;
$       delete/noconfirm  SYS$DISK:[]NEWMSG.PROMSG_TEMP;
$       delete/noconfirm  ADC.IDE;
$!
$       fortran/nolist  GENPAR
$       if .not.$status then goto had_an_error
$       link/nomap      GENPAR,SYS$DISK:[]NEWMSG.PROMSG_OBJ
$       if .not.$status then goto had_an_error
$       run  GENPAR
$!
$       purge/nolog  SYS$DISK:[]NEWMSGMSG.PAR
$!
$       delete/noconfirm  GENPAR.FOR;
$       delete/noconfirm  GENPAR.OBJ;
$       delete/noconfirm  GENPAR.EXE;
$!
$! ************************************************
$!
$       delete/noconfirm  LSL$SOURCE_ROOT:[LSLMAINT.LSLLIB.MESSAGE]NEWMSG.TEMP_C
OM;1
$       exit 1
$ DIR<CR>

Directory LSL$SOURCE_ROOT:[LSLMAINT.LSLLIB.MESSAGE]

NEWMSG.PROMSG_OBJ;2
                        12/12      28-JUL-1987 15:17  [LSL,TONY]
NEWMSGMSG.PAR;1         15/16      28-JUL-1987 15:18  [LSL,TONY]
NEWMSGMSG.RNO;1         46/48      28-JUL-1987 15:17  [LSL,TONY]

Total of 6 files, 76/116 blocks.
$
```

This example shows the use of NEWMSG to generate its own message definitions. It has not specified which parameter files are required, and has thus obtained .PAR files for inclusion in FORTRAN sources. The command file NEWMSG.TEMP_COM has then been obeyed, and DIRECTORY used to show what files have been created. Using NEWMSG and then obeying the command file produced in this manner is the normal way that the message utility would be used in command files.

```
$ NEWMSG SMALL /NUMBER=1234/PREFIX=SML__/PARAM=(FORT,C)/LOG<CR>
Message file:     SMALL.MES
Facility number:  1234
Facility name:    SMALL
Facility prefix:  SML__
Generating output for FORTRAN C
Generating output for message object module
%NEWMSG-I-MESOPN, opened  input message file "SMALL.MES"
%NEWMSG-I-PROOPN, opened output message file "SYS$DISK:[]SMALL.PROMSG"
.SEVERITY INFORMATIONAL
 SMALL                  <This is SMALL - type HELP for help>
.SEVERITY WARNING
 PARDON                 <Please type that again, correctly>
 BADVALUE               <The value %N is not allowed>
.SEVERITY ERROR
 NOFILE                 <You have not specified a file to SMALLify>
.SEVERITY FATAL
 BUG                    <Internal error - please report error %S to Laser-Scan>
%NEWMSG-I-MESCLO, closed  input message file
%NEWMSG-I-PROCLO, closed output message file
%NEWMSG-I-FOROPN, opened output FORTRAN file "SYS$DISK:[]GENPAR.FOR"
-NEWMSG-I-CREFOR, writing FORTRAN program to generate parameter files
%NEWMSG-I-FORCLO, closed output FORTRAN file
%NEWMSG-I-COMOPN, opened output command file "SYS$DISK:[]SMALL.TEMP_COM"
-NEWMSG-I-CRECOM, writing DCL command file to use MESSAGE and GENPAR
%NEWMSG-I-COMCLO, closed output command file
$ @SMALL.TEMP_COM<CR>
$
```

In this example, a message file called SMALL.MES has been read. The facility prefix has been changed to "SML__", and parameter files for FORTRAN and C are requested. The /LOG qualifier has been used to produce extra output describing the progress of the program. Since SET VERIFY has not been used, no messages are produced by executing the .TEMP_COM command file.

```
$ NEWMSG<CR>
Input message file: SMALL<CR>
%NEWMSG-E-NOFACNUM, facility number must be specified - use /NUMBER
$
```

In this example, the program has prompted the user for a message definition file. The user has not specified the facility number for the messages, and thus the program has aborted.

```
$ NEWMSG SMALL /NUMBER=1234/RUNOFF/NOPARAM/NOOBJ/LOG<CR>
Message file:     SMALL.MES
Facility number:  1234
```

```
Facility name:    SMALL
Facility prefix:  SMALL__
Generating output for RUNOFF
%NEWMSG-I-MESOPN, opened  input message file "SMALL.MES"
%NEWMSG-I-SEQOPN, opened sequential runoff workfile "SYS$DISK:[]SMALL.TEMP_SEQ"
%NEWMSG-I-MESCLO, closed  input message file
%NEWMSG-I-SEQCLO, closed sequential runoff workfile
%NEWMSG-I-SORTING, sorting messages
%NEWMSG-I-CONVERT, converting runoff workfile to indexed form
%NEWMSG-I-CONVERTED,   15 message records converted
%NEWMSG-I-SEQDEL, deleted sequential runoff workfile
%NEWMSG-I-IDXOPN, opened indexed runoff workfile "SYS$DISK:[]SMALL.TEMP_IDX"
%NEWMSG-I-RNOCRE, opened output runoff file "SYS$DISK:[]SMALLMSG.RNO"
%NEWMSG-I-RNOCLO, closed output runoff file
%NEWMSG-I-IDXDEL, deleted indexed runoff workfile
$
```

In this example, only the runoff file is required. Note the use of both /NOPARAMETERS and /NOOBJECT to suppress all other files. Also note that, because /NOOBJECT has been specified, the /LOG does not reflect the message texts from the file.

```
$ NEWMSG WRONG /NUMBER=1111/NOLOG<CR>
%NEWMSG-W-RUBBISH, found an unexpected '\' on line 5 - may be an old form file
-NEWMSG-I-OLDFORM, the old NEWMSG program used two texts for each message
-NEWMSG-I-NOTEXT2, this one doesn't - it only uses the second one
%NEWMSG-W-MISSEXPL,   6 messages had no EXPLANATION text
%NEWMSG-W-MISSUSER,   6 messages had no USERACTION text
$
```

The message file used in this example appears to have included a message description in the old form. This seems plausible, since six messages are reported not to have EXPLANATIONs, and six not to have USERACTIONs (in fact, the same six). Note the use of /NOLOG to suppress the normal informational output.

MESSAGES (INFORMATIONAL)

These messages give information only, and require no immediate action by the user. They are used to provide information on the current state of the program, or to supply explanatory information in support of a warning or error message.

COMCLO, closed output command file

Explanation: This message is output if /LOG has been specified.

User action: None.

COMOPN, opened output command file "'file-spec'"

Explanation: This message is output if /LOG has been specified.

User action: None.

CONVERT, converting runoff workfile to indexed form

Explanation: When creating data for a .RNO file, NEWMSG assembles the message texts in a sequential file which is then converted to an indexed file, so that the messages may be accessed in alphabetical order. If /LOG is specified, then this message is output before performing the transformation.

User action: None.

CONVERTED, 'integer' message records converted

Explanation: This message follows the CONVERT message, if /LOG has been specified

User action: None.

CRECOM, writing DCL command file to use MESSAGE and GENPAR

Explanation: If /LOG is specified, then this message is output before NEWMSG creates the DCL command file that can be used to run MESSAGE to generate the .PROMSG_OBJ message object file, and to compile, link and run GENPAR.FOR

User action: None.

CREFOR, writing FORTRAN program to generate parameter files

Explanation: If /LOG is specified, then this message is output before NEWMSG creates the FORTRAN program source GENPAR.FOR

User action: None.

EXCEPTION, rejected records are listed in "'file-spec'"

Explanation: This message is output after the CONVERROR message, to indicate the name of the file which contains a list of the runoff workfile records that could not be converted to indexed form.

User action: As for the CONVERROR message.

FORCLO, closed output FORTRAN file

Explanation: This message is output if /LOG has been specified.

User action: None.

FOROPN, opened output FORTRAN file "'file-spec'"

Explanation: This message is output if /LOG has been specified.

User action: None.

IDXCLO, closed indexed runoff workfile

Explanation: This message is output if /LOG has been specified.

User action: None.

IDXDEL, deleted indexed runoff workfile

Explanation: This message is output if /LOG has been specified.

User action: None.

IDXOPN, opened indexed runoff workfile "'file-spec'"

Explanation: This message is output if /LOG has been specified, when the indexed runoff workfile is being opened for read.

User action: None.

LINENO, in line 'integer' of the input message file

Explanation: This message is output after a warning or error message to indicate on which line of the input message file an error occurred.

User action: Depends upon the preceding message.

MESCLO, closed input message file

Explanation: This message is output if /LOG has been specified.

User action: None.

MESOPN, opened input message file "'file-spec'"

Explanation: This message is output if /LOG has been specified.

User action: None.

MESSAG, for the message with ident "'ident'"

Explanation: This message is output after a warning or error message to indicate which message an error occurred with.

User action: Depends upon the preceding message.

NOTEXT2, this one doesn't - it only uses the second one

Explanation: This message is output after the OLDFORM message.

User action: See the OLDFORM message.

OLDFORM, the old NEWMSG program used two texts for each message

Explanation: If an old form NEWMESSAGE file is processed, it may contain two text fields for each message. The new .MES file syntax is not compatible with this form, and will only read the first message text. Unfortunately, in the old form files, it is the second text that is the one needed. To allow the file to be processed with NEWMSG, remove the first message text of each pair.

User action: Change the message definition file - in each case where message descriptions have two message texts, retain the second one (which should be the one with EXPAND sequences in it).

PROCLO, closed output message file

Explanation: This message is output if /LOG has been specified.

User action: None.

PROOPN, opened output message file "'file-spec'"

Explanation: This message is output if /LOG has been specified.

User action: None.

READING, reading REPLACE clause 'integer'

Explanation: This message may follow the BADREPL error message. It indicates that the error was in the n'th REPLACE clause in the REPLACE command for that message.

User action: Check that the escape clause has correct syntax. It should be of the form

%<thing> = <text>

rNOCLO, closed output runoff file

Explanation: This message is output if /LOG has been specified.

User action: None.

RNOCRE, opened output runoff file "'file-spec'"

Explanation: This message is output if /LOG has been specified.

User action: None.

SEQCLO, closed sequential runoff workfile

Explanation: This message is output if /LOG has been specified.

User action: None.

SEQDEL, deleted sequential runoff workfile

Explanation: This message is output if /LOG has been specified.

User action: None.

SEQOPN, opened sequential runoff workfile "'file-spec'"

Explanation: This message is output if /LOG has been specified, when the sequential runoff workfile is being created.

User action: None.

SORTING, sorting messages

Explanation: Before creating a .RNO file, NEWMSG sorts the messages into order, by severity and then by alphabetical order of their idents. If /LOG has been specified, then this message is output before sorting.

User action: None.

TRUNCAT, when used as an output name, it will be truncated to "'string'"

Explanation: This message is output after the NAMTOOLNG message.

User action: See the NAMTOOLNG message.

UNEXPCH, unexpected character "'char'" instead of "'char'"

Explanation: This message may follow the BADREPL error message, to explain it.

User action: Correct the error as reported.

MESSAGES (WARNING)

These messages are output when an error has occurred that can be corrected immediately by the user or that the program will attempt to overcome.

ERRDEL, error deleting file "'file-spec'"

Explanation: This message is output if an error occurs in deleting a file. The program will continue.

User action: Since the program has not deleted the file, the user should remember to do so. Further action may be suggested by the system message following this one, describing why the file could not be deleted.

MISSEXPL, 'integer' messages had no EXPLANATION text

Explanation: This message is output at the end of the program run, and is self-explanatory.

User action: Insert the required texts in the message file.

MISSUSER, 'integer' messages had no USERACTION text

Explanation: This message is output at the end of the program run, and is self-explanatory.

User action: Insert the required texts in the message file.

NOEXPL, no explanatory text following EXPLANATION

Explanation: The text field after the EXPLANATION command was empty.

User action: Insert the relevant text into the field.

NOUSER, no user action text following USERACTION

Explanation: The text field after the USERACTION command was empty.

User action: Insert the relevant text into the field.

READREP, error reading the REPLACE command on line 'integer'

Explanation: The program could not read that line correctly. An associated LSLLIB message should explain the problem.

User action: Depends upon the associated message.

RUBBISH, found an unexpected ''char'' on line 'integer' - may be an old form file

Explanation: The specified character was found after the terminating backslash of a message text.

User action: Check why the character was there. If it is indeed an old form file, convert it to the current specification.

MESSAGES (ERROR)

These messages indicate an error in processing which will cause the program to terminate. The most likely causes are a corrupt or otherwise invalid input file, or an error related to command line processing and file manipulation.

BADFACNUM, facility number 'integer' is not in the range 'integer' to 'integer'

Explanation: Only a certain range of facility numbers is supported. Any numbers below 1025 or above 2047 will not be allowed.

User action: All facility numbers should have been approved or assigned by the facility numbers manager. Do not try to invent your own number - this could cause all sorts of problems.

BADHL, header level number 'integer' is not 1 or greater

Explanation: Runoff header level numbers must be one or greater.

User action: Run the program again with a sensible number. Note that 1 is the default, and the most common alternative is 2.

BADREPL, error in REPLACE clauses for message "'ident'"

Explanation: One of the REPLACE clauses for this message is not of the form

```
%<thing> = <text>
```

a following message should explain the problem.

User action: Depends upon the following message.

CONVCONV, error in CONV$CONVERT - converting runoff workfile

Explanation: An error occurred when trying to convert the sequential runoff workfile into an indexed file, so that the .RNO file can be created. This message should be preceded by a message from the CONVERT utility itself, explaining what went wrong.

User action: Dependant upon the associated CONVERT error message.

CONVERROR, 'integer' message records rejected

Explanation: This message indicates that some of the records in the sequential runoff workfile could not be converted to the indexed form. A message from the CONVERT utility may precede this message, giving more details. The offending records will have been written to a file for inspection - the EXCEPTION message will follow, indicating the name of that file.

User action: Inspect the original message file, and the exception file. The records in the exception file **should** be of the form

```
<severity> <type> <count> <ident> <text>
```

where these fields are (respectively) one of S,I,W,E or F, one of M,E or U, a two digit number, a fifteen character message name and a line of text. If the record in the exception file is not of this form, then report the problem, with as many details as possible.

CONVOPTS, error in CONV$OPTS - passing options to CONVERT

Explanation: An error occurred when preparing to convert the sequential runoff workfile into an indexed file, so that the .RNO file can be created. This message should be preceded by a message from the CONVERT utility itself, explaining what went wrong.

User action: Dependant upon the associated CONVERT error message.

CONVPASS, error in CONV$PASS - passing filespecs to CONVERT

Explanation: An error occurred when preparing to convert the sequential runoff workfile into an indexed file, so that the .RNO file can be created. This message should be preceded by a message from the CONVERT utility itself, explaining what went wrong.

User action: Dependant upon the associated CONVERT error message.

DELETE, error deleting runoff workfile

Explanation: After each runoff workfile has been finished with, the program will delete it. This message is given if that deletion did not work, and should be followed by a system message from the DELETE utility, explaining what went wrong. The program will continue.

User action: Dependant upon the associated DELETE error message.

DESTN, the destination directory may not include a file name, extension or version number

Explanation: The /DESTINATION qualifier was used, but the destination specified was not just a device and directory specification.

User action: Specify the destination correctly.

DESTNOD, the destination directory may not include a node name

Explanation: An attempt was made to use the /DESTINATION qualifier to specify a destination on another node (i.e. another computer). This is not supported.

User action: Create the workfiles on the current node, or move to the remote node and use the program there.

ERROPN, error opening file "'file-spec'"

Explanation: An error occurred when trying to open the specified file. This message should be followed by an LSLLIB or FORTRAN error message indicating what the problem was.

User action: Dependant upon the associated error message.

FACNAMLEN, facility name may not be longer than 'integer' characters

Explanation: The facility name specified with the /NAME qualifier was too long.

User action: Specify a facility name which is not too long.

FACPRELEN, facility prefix may not be longer than 'integer' characters

Explanation: The facility prefix specified with the /PREFIX qualifier was too long.

User action: Specify a facility prefix which is not too long.

FINDEXP, error finding explanatory text line with key "'string'"

Explanation: When creating the .RNO file, the lines of explanatory text for each message are looked up in the indexed runoff workfile using appropriate keys. This message indicates that something has gone wrong with this process. It should be followed by an LSLLIB message explaining what the problem is.

User action: Dependant upon the associated error message.

FINDKEY, error finding message text for message "'ident'"

Explanation: When creating the .RNO file, the message texts are looked up in the indexed runoff workfile. This message indicates that something has gone wrong with this process. It should be followed by an LSLLIB message explaining what the problem is.

User action: Dependant upon the associated error message.

IDENTLEN, ident "'string'" truncated to %N characters

Explanation: The identification field for a message was too long, and has been truncated.

User action: Edit the original message file, and use a shorter ident for that message.

MAXMESS, cannot store more than 'integer' messages

Explanation: The program uses an internal buffer to store message idents, and thus has a limit on how many messages it can store. This message indicates that the message file being processed contained too many messages.

User action: Either use fewer messages, or submit a modification request for the limit on the number of messages to be increased.

MESSLEN, message text is longer than 'integer' characters

Explanation: The text field for a message was too long, and has been truncated.

User action: Edit the original messages file, and use a shorter text for the message.

NAMTOOLNG, filename is longer than 'integer' characters

Explanation: The filename specified for the message file is used to work out the output name for the parameter and .RNO files. This is normally obtained by adding "MSG" to the end of the specified name. However, if the filename is so long that adding "MSG" would generate an illegal name, it will be truncated sufficiently to allow the appendage to be added. This message is thus followed by the TRUNCAT message.

User action: None.

NOFACNUM, facility number must be specified - use /NUMBER

Explanation: The facility number identifies the utility producing the messages, and must always be specified.

User action: Specify the facility number using the /NUMBER qualifier. Facility numbers are assigned by the faciltiy number manager, who should be consulted before you create the messages for a new utility.

NOIDENT, no ident found for message

Explanation: As it says, the identification field was missing for a message in the input message definition file.

User action: Edit the message file to add the correct ident.

NOREPL, unable to replace string "'text'" in message "'ident'"

Explanation: The program could not find the specified EXPAND sequence in the message text, and was thus not able to replace it, or an error occurred in the replacement. In the latter case, a system error message will be output to explain the problem. The program will abandon any further replacements for this message text.

User action: Check the REPLACE clause for this message.

NOTEXT, no message text found

Explanation: As it says, the message text was missing for a message in the input message definition file.

User action: Edit the message file to add the correct text.

NULLMESS, message text has zero length

Explanation: The backslash ("") delimitors for the message text were found in the message file, but there was nothing between them. Blank messages are not allowed.

User action: If the message text is truly missing, edit the message file to insert the correct text. If you actually want a blank message text, then insert a space character between the backslash delimitors - this is the best approximation available.

PARSE, error in internal parsing of the destination

Explanation: An error has occurred in trying to parse the destination for work files as by /DESTINATION. This will be followed by an LSLLIB message giving more information on the problem.

User action: Dependant upon the associated error message.

READERR, error reading line 'integer' from input message file

Explanation: This message will be followed by an LSLLIB message giving more information on the problem.

User action: Dependant upon the associated error message.

READEXP, error reading message text line with key "'string'"

Explanation: This message indicates an error in reading a mesage text from the indexed runoff workfile .It will be followed by an LSLLIB message giving more information on the problem.

User action: Dependant upon the associated error message.

READIDX, error reading message text for message "'ident'"

Explanation: This message will be followed by an LSLLIB message giving more information on the problem.

User action: Dependant upon the associated error message.

READREPX, error reading message REPLACE line with key "'string'"

Explanation: This message indicates an error in reading a mesage REPLACE clause from the indexed runoff workfile .It will be followed by an LSLLIB message giving more information on the problem.

User action: Dependant upon the associated error message.

TOOLONG, line 'integer' of message file is longer than 'integer' characters

Explanation: No line of the input message file may be longer than 132 characters. This is to avoid problems with FORTRAN output when outputting RUNOFF files.

User action: Split the offending line into two shorter lines.

TRIBRAC, triangular brackets are not allowed in message texts

Explanation: The characters "<" and ">" are used to delimit message texts internally within the .PROMSG file generated by the program. They are thus not allowed within the actual message text itself.

User action: Replace the offending characters by a description of them.

VERSION, version numbers are not allowed

Explanation: When specifying the input message file, a version number may not be specified.

User action: Do not specify a version number.

MESSAGES (FATAL)

These messages indicate a severe error in processing, or some form of system failure, which has caused the program to terminate.

BADCMD, unrecognised command number 'integer' (command "'string'")

Explanation: The message severities, and the EXPLANATION and USERACTION keywords, are read as LSLLIB commands. This message indicates that the part of the program which interprets these commands is not working correctly.

User action: Report this problem as a bug.

BUG, please report the following message - it is a bug

Explanation: As it says.

User action: As it says.

STATE, unknown internal state in READ__MESSAGES

Explanation: The routines that read the message file have become confused as to what it is that they are currently reading.

User action: Report this problem as a bug.

UNKSEVLET, unrecognised severity mnemonic "'string'"

Explanation: When retrieving a message from the indexed runoff workfile, the program has not recognised the severity code for the message.

User action: Report this problem as a bug, including the runoff workfiles.

MESSAGES (OTHER)

In addition to the above messages which are generated by the program itself, other messages may be produced by the command line interpreter (CLI) and by Laser-Scan libraries. In particular, messages may be generated by the IFF library and by the Laser-Scan I/O library, LSLLIB. IFF library messages are introduced by '%IFF' and are documented in the IFF library users' guide. In most cases IFF errors will be due to a corrupt input file, and this should be the first area of investigation. If the cause of the error cannot be traced by the user, and Laser-Scan are consulted, then the output file should be preserved to facilitate diagnosis. LSLLIB messages are introduced by '%LSLLIB' and are generally self-explanatory. They are used to explain the details of program generated errors.

CHAPTER 4

ERROR MESSAGE ROUTINES

4.1 *Introduction*

This chapter describes how to output the error messages defined using the LSLLIB message definition procedure (see the previous chapter), and how to perform other forms of diagnostic message outputting.

4.2 *General error message routines*

LSL_PUTMSG and LSL_ADDMSG are the recommended routines to use when outputting messages from a program.

ok = LSL_PUTMSG(errnum, [arg1], [arg2], ...)

out	- long	**ok**	return code from getting the message string
in	- long	**errnum**	LSL utility error number
in	- variable	**arg<n>**	arguments as for EXPAND

If the logical name LSL$DEBUG_TRACE was defined when LSL_INIT was called, or when LSL_DEBUG_TRACE was last called, then all calls of LSL_PUTMSG are converted into calls of LSL_SIGNAL. This is intended to help in debugging programs.

Otherwise, LSL_PUTMSG looks up the error number **errnum** in the list of error message formats, and calls EXPAND on the result, using the given arguments. The resulting string is then assembled into a system style message (according to the severity of **errnum**), and output by VIO$PUT_OUTPUT.

If **errnum** is not found in the supplied list, then it is looked up as a system error number, and the corresponding text is output instead. If it is still not found, then the error LSL__NOMESSAGE is output.

In addition to outputting the appropriate message, LSL_PUTMSG sets LSL_STATUS in /STATUS/ to (**errnum**.OR.LSL_QUIET).

The following values of **ok** may be returned:

LSL__NORMAL	- the message text was successfully found, EXPANDed and output
LSL__BUFFEROVF	- the message text was too long to fit into the internal buffers used by the routine. A warning message (SS$_BUFFEROVF) is output, and then the message is truncated and the truncated version is output

LSL__MSGNOTFND - the message text cannot be found. The function does not output anything.

ok = LSL_ADDMSG(errnum, [arg1], [arg2], ...)

out - long **ok** return code from getting the message string
in - long **errnum** LSL utility error number
in - variable **arg<n>** arguments as for EXPAND

LSL_ADDMSG is identical to LSL_PUTMSG, except that

1. it does not set LSL_STATUS

2. it prefixes its message with "-" (hyphen), rather than "%"

3. it is never converted into a call of LSL_SIGNAL

An example of use of these routines might be:

```
OK = LSL_PUTMSG( EXAMPLE__UNEXPEOL )
IF ( .NOT.OK ) CALL MSGERR( OK )
OK = LSL_ADDMSG( EXAMPLE__READCNT, LINENO )
IF ( .NOT.OK ) CALL MSGERR( OK )
CALL LSL__EXIT
```

which would output the messages:

```
%EXAMPLE-E-UNEXPEOL, unexpected end of line
-EXAMPLE-I-READCNT, read 2500 lines of text
```

and, since EXAMPLE_UNEXPEOL is of severity E (Error), would set $STATUS to SS$_ABORT with a severity of Error.

A more usual piece of code is:

```
OK = TTRSTR( ,, '> ', IERR )
IF ( .NOT.OK ) THEN
   CALL LSL_PUTMSG( EXAMPLE__READLINE )
   CALL LSL_ADDMSG( OK )
   IF ( OK.EQ.LSL__SYSERR ) CALL LSL_ADDMSG( IERR )
   GOTO 9999
ENDIF
```

which produces the normal message sequence for most Laser-Scan utiltiies.

call LSL_SETMSG(facility-name, severity, message-name)

in - logical **facility-name**
in - logical **severity**
in - logical **message-name**

LSL_SETMSG is used to set the format of the message string which is output by LSL_PUTMSG or LSL_ADDMSG. Each of the arguments should be TRUE if that field of the message should be present, and FALSE if it should be absent. The default state is as if LSL_SETMSG had been called with all arguments TRUE.

Using the same definition of message EXAMPLE_UNEXPEOL as above, the code

```
call LSL_SETMSG( .FALSE., .TRUE., .TRUE. )
call LSL_PUTMSG( EXAMPLE__UNEXPEOL )
call LSL_SETMSG( .FALSE., .FALSE., .TRUE. )
call LSL_PUTMSG( EXAMPLE__UNEXPEOL )
call LSL_SETMSG( .FALSE., .FALSE., .FALSE. )
call LSL_PUTMSG( EXAMPLE__UNEXPEOL )
```

will output the message sequence

```
%E-UNEXPEOL, unexpected end of line
%UNEXPEOL, unexpected end of line
Unexpected end of line
```

4.3 *Outputting an erroneous text string*

LSL_ADDSTR and LSL_ADDBUF are used to output a text string or buffer between backslash (\) characters.

ok = LSL_ADDSTR([string], [start], [end], [ierr])

ok = LSL_ADDBUF([buffer], [buflen], [start], [end], [ierr])

out - long	**ok**	return code
in - char	**string**	the string to output
in - byte	**buffer**	the buffer to output
in - word	**buflen**	the length of **buffer**
in - word	**start**	index of the first character
in - word	**end**	index of the last character
out - long	**ierr**	system error code

In both routines, the actual text to be output defaults to that in /TXTC/ - that is, **string** defaults to TXTDSC, **buffer** defaults to TXTBUF and **buflen** defaults to TXTPTR.

start and **end** are used to specifiy that a substring should be output. **start** defaults to 1 (start with the first character) and **end** defaults to either LEN(**string**) or to **buflen** (ie end with the last character).

Thus a typical example of use would be:

```
ok = LSL_PUTMSG( EXAMPLE__UNEXPEOL )
.......
```

```
ok = LSL_ADDSTR()
```

producing the following output:

```
%EXAMPLE-E-UNEXPEOL, unexpected end of line
\This line ends unexpect\
```

ok returns LSL_NORMAL if the output succeeded, and otherwise LSL_SYSERR if the output failed. In the latter case, **ierr** contains the relevant system error code.

4.4 *Producing a traceback*

LSL_SIGNAL is used to produce a traceback when some very severe error occurs, and LSL_DEBUG_TRACE is called by LSL_INIT to establish whether calls of LSL_PUTMSG should be converted to calls of LSL_SIGNAL.

ok = LSL_SIGNAL(errnum, [arg1], [arg2], ...)

out - long	**ok**	return code
in - long	**errnum**	LSL utility error number
in - variable	**arg<n>**	arguments as for EXPAND

LSL_SIGNAL calls LSL_PUTMSG with the supplied arguments. It then provokes a traceback by calling LIB$SIGNAL with the message LSL__SIGNAL, with its severity code set to be the same as the severity of the original message.

Note that this routine is not generally recommended for use in message production, as tracebacks can be very confusing to ordinary users.

ok returns LSL__MSGNOTFND if the message text cannot be found (in which case no message is output), and otherwise the return code from the call of LIB$SIGNAL.

ok = LSL_DEBUG_TRACE([tracing])

out - long	**ok**	return code
out - logical	**tracing**	true if LSL$DEBUG_TRACE is defined

This routine uses the system function SYS$TRNLNM to determine whether the logical name LSL$DEBUG_TRACE is defined. If it is, then future calls of LSL_PUTMSG will be converted to calls of LSL_SIGNAL. If it is not defined, then calls will not be converted.

If **tracing** is present, then it returns true if the logical name was found, and false if it was not. **ok** returns the return code from the call of SYS$TRNLNM - see the DEC system services documentation for a list of the possible system return codes.

Note that LSL_INIT calls LSL_DEBUG_TRACE when the library is initialised.

4.5 *Extracting the message text*

LSL_GETMSG and LSL_GETFORMAT are used to get the message string, or message format string, so that the user program can process them further.

ok = LSL_GETMSG(errnum, string [,strlen])

out - long	**ok**	return code from getting message string
in - long	**errnum**	error number
out - char	**string**	basic text string corresponding to the error
out - char	**strlen**	length of text string

LSL_GETMSG looks up the error number **errnum** in the error message symbols, and returns the appropriate message text in **string**.

Note that this string may include EXPAND escapes (ie %<character>), but that the initial '%' character will NOT be doubled. If you intend to use this string as a format string to EXPAND, then you must prefix it with an additional '%'.

The length of the text string is optionally returned in **strlen**.

For instance,

```
ok = LSL_GETMSG( LSL__NORMAL, string, length )
```

would return the string '%LSLLIB-S-NORMAL, normal, successful completion'.

If this function is used on a system error number, the appropriate system message is returned in **string**.

The following values of **ok** may be returned:

LSL__NORMAL	- the message text was successfully found, EXPANDed and output
LSL__BUFFEROVF	- the message text was too long to fit into **string**. A warning message (SS$_BUFFEROVF) is output, and then the message is truncated and the truncated version is output
LSL__MSGNOTFND	- the message text cannot be found. The function does not output anything.

ok = LSL_GETFORMAT(errnum, string [,string_length])

out - long	**ok**	return code from getting message string
in - long	**errnum**	LSL utility error number
out - char	**string**	format string string corresponding to the error
out - char	**string_length**	length of text string

LSL_GETFORMAT looks up the error number **errnum** in the list of error message formats, and returns the EXPAND format string for the message text in **string**. The length of the text string is optionally returned in **string_length**.

For instance,

```
ok = LSL_GETFORMAT( LSL__NORMAL, string, length )
```

would return the string 'normal, successful completion'.

If this function is used on a system error number, the appropriate system message text is returned in **string**.

The following values of **ok** may be returned:

LSL__NORMAL	- the message text was successfully found, EXPANDed and output
LSL__BUFFEROVF	- the message text was too long to fit into **string**. A warning message (SS$_BUFFEROVF) is output, and then the message is truncated and the truncated version is output
LSL__MSGNOTFND	- the message text cannot be found. The function does not output anything.

4.6 *Marking current position in TXTBUF*

call MARK_POSN([prompt], [offset], pointer)

in	- char	**prompt**	the prompt used when requesting the string at fault - only required to ascertain its length
in	- long	**offset**	the offset to use from the current position - an offset of zero will point directly to the last character read
in	- char	**pointer**	the character(s) to use as a pointer, typically ^

MARK_POSN can be used to point to an error in a string typed by a user - to highlight where in the input an error occurred.

The default action is to point at the last character read (using the value of DCPTR to establish where this is). If **prompt** is specified, then it is assumed that the **pointer** must be moved LEN(**prompt**) characters to the right. If **offset** is specified, then the pointer will be output at the default position plus **offset**.

If either of these results in the pointer being before the beginning or after the end of the line, then no line is output and a warning message (either LSL__NEGPOSNMK or LSL__POSNMKOVF) is issued.

For example:

```
1000    OK = TTRSTR( ,, 'Next line> ', IERR )
        ......
        IF ( READ_ERROR ) THEN
           OK = MARK_POSN( 'Next line> ',, '^error' )
           CALL EXPAND('Expression ends before end of line')
           CALL WRITAP(' - too many brackets?')
           GOTO 1000
        ENDIF
        ......
```

which might result in the following conversation:

```
        Next line> (ADD (MULT 2 3)) (DIV 5 2))
```

```
                              ^error
Expression ends before end of line - too many brackets?
Next line> (ADD (MULT 2 3) (DIV 5 2))
Result is 8
Next line>
```

CHAPTER 5

/EXCEPTION/ - ERRORS AND EXCEPTIONS

5.1 *Introduction*

The /EXCEPTION/ common block is used to report errors that occur whilst reading things from the current input buffer - via the number and command reading routines, which do not explicitly return an error themselves.

The /STATUS/ common block is used to provide a default exit status - see below.

5.2 *The exception common block*

The exception handling common block is in LSL$CMNLSL:EXCEPTION.CMN, and contains:

public - long **ERRNUM**
The LSL error code for the latest error is placed here. If no error has occurred, then it is set to LSL__NORMAL. Note that all operations which might produce an error will unset ERRNUM before doing anything which might cause an error

public - long **LSL_EXCP**
If a numeric exception occurs, then the error code LSL_HADEXCP is placed into ERRNUM, and a code specifying the error is placed here.

5.3 *Numeric errors*

5.3.1 *The condition handler*

Numeric exceptions are detected by the condition handler LSL_NUM_CHAND. This is declared as an exception handler (using LIB$ESTABLISH) at the start of each number reading routine. If a numeric exception occurs, then it sets ERRNUM to LSL_HADEXCP, and LSL_EXCP to an appropriate error.

Note that the exception handler is only declared within the number reading routines, so will not detect numeric exceptions in the calling program. However, if the user wishes, they may establish LSL_NUM_CHAND themselves.

The following conditions are handled by LSL_NUM_CHAND (and reduced in severity to informational, so that the program continues without complaint):

SS$__FLTDIV, SS$__FLTDIV__F, SS$__FLTOVF, SS$__FLTOVF__F,

SS$__FLTUND, SS$__FLTUND__F, SS$__INTDIV, SS$__INTOVF

For a list of the error codes returned in LSL_EXCP, see the list of errors whilst reading numbers, below.

For more details on reading numbers, see the relevant chapter.

5.3.2 *Errors whilst reading numbers*

The variable ERRNUM in /EXCEPTION/ is set by the number reading routines. The following values may be found:

LSL__NORMAL - success- number read successfully
LSL__BASECH - failure - unrecognised base character (integers only), the "^" signifying change of base for an integer was found, but the character after it was not one of the allowed ones. The unrecognised character is unread, and will be read by the next RDCH
LSL__UNEXPEOL - failure - the end of the line (or input buffer) was found before starting to read the numeric part of a number
LSL__NONUM - failure - there was no number to read - a non-numeric character was found when a number was expected. The unexpected character is unread
LSL__HADEXCP - failure - a numeric exception occurred whilst reading the number - a more precise definition of the error is in LSL_EXCP, which may have one of the values:

LSL__NORMAL - no error in reading the last number
LSL__FLTDIV - floating divide by zero, reading a number of the form a/b
LSL__FLTOVF - floating overflow
LSL__FLTUND - floating underflow
LSL__INTDIV - integer divide by zero
LSL__INTOVF - integer overflow

5.4 *Errors whilst reading commands*

The command reading routine RDCOMM also uses ERRNUM to return errors. Note that, since it can read numeric arguments, all of the numeric error codes may be found, for the same reasons.

By default, RDCOMM will produce its own error messages for each of the errors using LSL_PUTMSG of the appropriate error code - this may be disabled by setting /CMDCOM/NOMESS to be true. There is also a routine LSL_CMDERR which will output an appropriate error message for the following errors.

For more details on command reading and command error reporting, see Chapter 15.

The following values may be placed in ERRNUM by RDCOMM :

LSL__NORMAL - success - command read successfully

LSL__UNEXPEOL - failure - the end of the line (or input buffer) was found before starting to read the command, or whilst looking for compulsory arguments

LSL__UNEXPCH - failure - an unexpected character was found whilst attempting to read a command name. The character is placed in /CMDCOM/UNXCHR, and the character *after* it will be read by the next call of RDCH

LSL__UNEXPCMD - failure - an unexpected (ie unrecognised) primary command name was found - one that is not in the primary command table. The string read is held in the descriptor /CMDCOM/CMDNST

LSL__UNEXPCMD2 - failure - an unexpected (ie unrecognised) secondary command name was found - one that is not in the secondary command table. The string read is held in the descriptor /CMDCOM/SECNST

LSL__AMBIG - failure - the primary command name given is ambiguous. A descriptor for the command name read is in /CMDCOM/CMDNST, and descriptors for (a sample of) the names in the table evincing the ambiguity are in CMDFST and CMDAST (not necessarily in alphabetical order)

LSL__AMBIG2 - failure - the secondary command name given is ambiguous. A descriptor for the command name read is in /CMDCOM/SECNST, and descriptors for (a sample of) the names in the table evincing the ambiguity are in CMSFST and CMDAST (not necessarily in alphabetical order)

LSL__BADINEQ - failure - a bad Fortran style inequality name was found. A descriptor for the inequality name read is in /INEQUAL/INEQNAME.

LSL__AMBINEQ - failure - ambiguous Fortran style inequality name was found. A descriptor for the inequality name read is in /INEQUAL/INEQNAME, and descriptors for (a sample of) the names in the table evincing the ambiguity are in CMDFST and CMDAST (not necessarily in alphabetical order)

5.5 *The STATUS common block*

This is held in LSL$CMNLSL:STATUS.CMN, and contains the following:

public - long parameter **LSL_QUIET**
This is set to the value '10000000'X, and is the value that must be OR'ed with a $STATUS value to stop the system printing the appropriate message when $STATUS is set.

public - long **LSL_STATUS**
This is initially set to LSL__NORMAL.OR.LSL_QUIET by LSL_INIT, and is then altered by LSL_PUTMSG or LSL_SIGNAL to the value that they have been passed (again, OR'ed with LSL_QUIET). This value is then used by LSL_EXIT in determining how to set the return code ($STATUS) for the program.

For instance, if a program calls

```
CALL LSL_PUTMSG( LSL__EOF )
```

to output the "End of file" message, then LSL_STATUS will be set to

```
LSL__EOF.OR.LSL_QUIET
```

The program could then exit with the call

```
CALL LSL_EXIT
```

which would set $STATUS to SS$ABORT with the same severity as LSL__EOF (and with the QUIET bit set, so that the system does not give a message as the program exits). A command file running the program could then check for this status value.

Although it is not standard LSL practice, it might also be required to output the LSL message itself to $STATUS, and in that case the program would exit with

```
CALL EXIT( LSL_STATUS )
```

CHAPTER 6

/EXPC/ - THE OUTPUT COMMON BLOCK

6.1 *Introduction*

This chapter describes the text output common block

6.2 */EXPC/ - the output common block*

Those LSLLIB routines that perform output, or encode data, use the output common block by default. This common block is defined in Fortran in

LSL$CMNLSL:EXPC.CMN

and in Macro in

LSL$CMNLSL:EXPC.MAR

The common block contains the following:

public - long parameter **MAX_EXPMAX**
the maximum length of EXPBUF (ignoring the extra byte), thus the maximum value to which EXPMAX may be set. This is currently 1024, which is the maximum buffer size that the DCL routines will write.

public - long parameter **DEF_EXPMAX**
the default length of EXPBUF (ignoring the extra byte), thus the default value to which EXPMAX is set. This is currently 255.

public - word **EXPLEN**
the number of characters placed in **EXPBUF** so far

public - word **EXPMAX**
the maximum number of characters which may be inserted into **EXPBUF**. This is set to DEF_EXPMAX by LSL_INIT, and may be changed using the routine SET_EXPMAX.

private - word **EXPCTF**
holds control flags for EXPAND and APPEND.

public - byte **EXPBUF(MAX_EXPMAX+1)**
the buffer in which the expanded characters are stored. Note that it is always one byte longer than expected from the value in **EXPMAX**. This is to allow a null (zero) byte to be added to the end of any

string in EXPBUF. Byte strings produced by EXPAND/APPEND are normally terminated by a zero byte.

public - alias **EXPDSC**
equivalenced onto the common block to produce a fake string. It is made up of **EXPLEN**,**EXPTYP**,**EXPCLA** and **EXPPTR**. For further explanation, see the section on "Fake strings" in Chapter 1.

private - long **EXPPTR**
the address of **EXPBUF**. This is set by LSL_INIT

private - byte **EXPTYP**, **EXPCLA**
the type and class fields of **EXPDSC**. These are initially zero, and should be left so.

6.2.1 *Manipulating the size of EXPBUF*

call SET_EXPMAX([length])

in - word **length** the new length for EXPBUF

SET_EXPMAX will change the value of EXPMAX, and hence the size of EXPBUF, to the given **length**. Note that EXPAND and APPEND will write a null byte after the last character output to EXPBUF, and may thus write to the **length**+1th byte.

If **length** is omitted, then EXPMAX is set back to the default value of DEF_EXPMAX. If **length** is less than 1, or greater than MAX_EXPMAX, then the call is ignored.

call SAVE_EXPMAX(length)

out - word **length** the current length of EXPBUF

SAVE_EXPMAX returns the current value of EXPMAX, and hence the current size of EXPBUF, in **length**.

6.2.2 *Saving and restoring EXPBUF and EXPLEN*

call SAVE_EXPC(buffer, bufptr, bufmax)

out - byte **buffer** the buffer to save EXPBUF in
out - word **bufptr** how many characters were saved
in - word **bufmax** maximum number of characters to save

SAVE_EXPC saves the contents of EXPBUF in **buffer**, and EXPLEN in **bufptr**. The number of characters actually saved will be the minimum of EXPLEN and **bufmax**.

It is assumed that **buffer** is at least **bufmax** bytes long.

call RESTORE_EXPC(buffer, bufptr, bufmax)

in - byte **buffer** the buffer to copy into EXPBUF
in - word **bufptr** how many characters to copy in
in - word **bufmax** maximum number of characters to copy

RESTORE_EXPC is used to restore the contents of EXPLEN and EXPBUF. The number of characters copied from **buffer** into EXPBUF is the minimum of **bufptr** and EXPMAX, and EXPLEN is set to **bufptr** regardless.

It is assumed that **buffer** is at least **bufmax** bytes long.

CHAPTER 7

USING /EXPC/ - TEXT EXPANSION

7.1 *Introduction*

These routines will carry out formatted expansion of binary data into character form. This is usually in order to produce an output line, and is often done by calling a routine (such as WRITEF or WRITAP) which uses them.

7.2 *Encoding into the output buffer*

len = EXPAND (format, arg1, arg2, arg3, arg4, ...)

out	- word	**len**	the length of the expanded string
in	- char	**format**	the format describing the required output
in	- variable	**arg<n>**	the arguments to fill out the format

The string format (usually a literal string) is copied into /EXPC/ character by character. If the escape character % is found, then the action determined by the sequence **%<escape sequence>** is obeyed - this may use an argument from the argument list following **format**.

Calling EXPAND with no arguments sets up a null line in /EXPC/ - a line of zero length. Calls to EXPAND reset the default state of the **%^** mode switches.

NOTE that EXPAND will check whether LSL_INIT has been called - see the description in Chapter 2 for details.

call APPEND(format, arg1, arg2, ...)

APPEND is the same as EXPAND, but appends the text to the end of the string already in /EXPC/.

Calling APPEND with no arguments produces undefined results. Calls to APPEND continue to use the values of the **%^** mode switches set up in earlier calls of EXPAND/APPEND.

7.3 *How the string in /EXPC/ is ended*

/EXPC/ is fully described in Chapter 6. As mentioned there, the actual text buffer, EXPBUF, is declared as one byte longer than would be expected from the maximum length of line. This allows EXPAND/APPEND to output a final null byte at the end of any string.

Thus the string placed into /EXPC/ will always terminate in a zero byte, which will not be included in the count of characters held in EXPLEN.

Note that this zero terminating byte is not output when output is redirected to another destination (with %W or %WS - see below).

7.4 *EXPAND escape sequences*

The escape sequences allowed in the format string to EXPAND/APPEND are described below. Note that (unless otherwise stated) alphabetic escape sequences must be upper-case.

7.4.1 *Expanding a text string*

The following expand a byte buffer or string argument in place of the escape sequence.

%A<n> Synonym of %AZ

%AC<n> Includes the given ASCII byte array, assuming that the 1st byte is a count of the number of characters. Maximum length expanded is <n>, default 255

%AZ<n> Includes the given ASCII byte array, assuming that it is terminated by a zero byte. Maximum length expanded is <n>, default 255

%AD<n> Synonym of %S

%S<n> Includes the given ASCII string, assuming it is a character string. Maximum length expanded is <n>, default 255

%C Includes 1-4 ASCII characters (depending on current mode) from the argument - the characters are stored as bytes. Terminated by zero byte

%R Includes three radix-50 characters from the word argument

%5 Synonym of %R

%RZ,%5Z Effect as for %R or %5, but trailing spaces are suppressed

7.4.2 *Expanding integers*

The following expand an integer argument in place of the escape sequence.

If the output is signed (the default) then negative numbers are preceded by a minus sign. If the output is unsigned (set by the %U flag) then the number is output as an unsigned number - ie it is always positive. Positive numbers are never preceded by a plus sign.

A minus sign is included in the count of characters for the field width, and is always output adjacent to and immediately preceding the first digit output.

%I<n> Includes a decimal integer, field width <n> (default 6). If the integer won't fit in the field width, then it is output in the minimum field width that it will fit in.

%N<n> As %I, except that the default field width is 0

%O<n> Includes an octal number (default field 0). This is always assumed unsigned.

%X<n> Includes a hexadecimal number (default field 0). This is always assumed unsigned.

7.4.3 *Expanding real numbers*

The following expand a real (floating point) number in place of the escape sequence.

%F<f>.<d> Includes a floating decimal, where <f> is the total field width to use (including the decimal point), and <d> is the number of digits to be output after the decimal point. The default field values are 9.3

Note that there will always be at least one digit output before the decimal point, although there need not be any digits output after the decimal point. The sequence %F0.<n> can be useful for outputting in the minimum field width - compare with %I0 for integers.

Provided that the field (f) is large enough to contain the number, the number will be right justified in the field, and trailing zeroes will not be truncated. This means that the decimal points of numbers will always appear in the same character position.

If the field is insufficient to contain the number, then the number will be left justified, and will take up as many characters as required. Trailing zeroes may be truncated or not, depending on the use of %^T or %^P (default).

Thus the number 1234.5678 can be output as follows

```
with   %F         1234.568
with   %F0.3     1234.568
with   %F7.0       1235.
with   %F0.0     1235.
```

%E<d> Includes a real number with decimal exponent, where <d> is the number of significant digits to output - that is the number of digits between the decimal point and the 'E' delimiting the exponent. The default field value is 4.

Thus the number 1234.5678 can be output as follows

```
with   %E      .1235E 004
with   %E0     .E 004
with   %E1     .1E 004
with   %E2     .12E 004
with   %E8     .12345678E 004
```

and for the number -1234.5678
with %E -.1235E 004

%G<f>.<d> Includes a general floating decimal. <d> significant figures are always output. Zero is output using %F<f>.<d>, numbers with absolute value less than 0.1 or greater than 10**<d> are output using %E<d>, while numbers within this range are output using %F<f>.<x>, where <x> (between 0 and <d>) is chosen to give <d> significant figures. The default field values are 9.3

%G is useful for outputting numbers where significance must not be lost, but greater readability than that provided by %E is desired. The sequence %G0.<n> can be useful for outputting in the minimum field width.

Thus the number 1234.5678 can be output as follows

```
with   %G        .123E 005
with   %G0.4     1235.
with   %G7.5      1234.6
```

7.4.4 *Expanding dates and times*

The following expand an (integer) argument representing a date or time in place of the escape sequence. There are routines to convert from standard VMS date/time strings to these date and time representations (see chapter 20).

A date is either the number of days since 17-NOV-1858, or if it is negative then it is a "delta date" i.e. the number of days from today.

A time is the number of 10 millisecond units since midnight.

%DD Translates the argument to a date, and includes it in standard VMS date format, e.g. either "22-OCT-1987" (for standard dates) or "54" (for delta dates)

%DT Translates the argument to a time since midnight, and includes it in standard VMS time format, e.g. "14:04:23.56"

7.4.5 *Expanding into a different destination*

The following redirect text expansion into a different buffer or string, instead of the default /EXPC/

%W Further output will be expanded into a user-specified buffer. The argument is the address of the required byte array.

Note that no overflow checking is performed - thus EXPAND/APPEND will quite happily attempt to write off the end of the buffer. This is accomplished by assuming that the destination buffer is of the same maximum length as EXPBUF - ie 1024 characters.

%WS As %W but the argument is the address of a string descriptor. Overflow checking is enabled - the expansion will stop when the string is full, and the string is padded with spaces (if necessary)

7.4.6 *Repetition*

The following cause a piece of text to be repeated in the expansion.

The enclosing repetition brackets must be matched within any call of EXPAND or APPEND (that is, %(with %) and %[with %]). There may be up to 64 repetition sequences (!) in any one call of EXPAND/APPEND, and these may be nested. If the number of repetition sequences is exceeded, then the string %<*> will be output instead of the offending repetition sequence.

%[...%] The enclosed section ("...") will be repeated. The argument is a word specifying how many times the section should occur in the expanded result. If the argument is less than 1, then 1 is assumed.

%(<n>...%) The enclosed section ("...") is expanded <n> times (default number of times is 1)

7.4.7 *Formatting*

The following influence how further escape sequence arguments will be output or interpreted

%P<c> Sets the padding char to <c> (default space). This character will be used in padding fixed field integers from now on. Note that it does not affect the placement of the '-' sign for negative numbers - these are still output directly preceding the integer.

%U Set unsigned mode for next integer - the next integer (only) output with %I or %N is output as an unsigned number.

%^B Set Byte mode - integer arguments are assumed to be bytes (BYTE or LOGICAL*1) from now on

%^W Set Word mode - integer arguments are assumed to be words (INTEGER*2) from now on

%^L Set Long mode (default) - integer arguments are assumed to be longwords (INTEGER or INTEGER*4) from now on

%^F Set single precision floating point mode (default) - real arguments are assumed to be single precision reals (REAL or REAL*4) from now on

%^D Set double precision floating point mode - real arguments are assumed to be double precision reals (REAL*8) from now on

%^P Set padding of numbers output using %F - numbers will always have the requested number of decimal places, including trailing zeroes.

%^T Set truncation of numbers output using %F (default) - numbers for which the specified field is insufficient will have any trailing zeroes after the decimal point truncated. The first digit after the point will never be truncated.

7.4.8 *Miscellaneous*

The following escape sequences do not take an argument

%M Multiplicity - if the last integer output with %I, %N, %O, or %X was not 1, then an 'S' is expanded, otherwise this sequence is ignored. The multiplicity is singular immediately after EXPAND or APPEND has been called - that is the default situation is as if 1 had been expanded.

%m As %M, but produces 's' (lower case letter)

%T Expands a tab character

%% Expands a single '%' character

% %<space> is ignored and may be used as a terminator

7.4.9 *Unrecognised escape sequences*

The following are used if an escape sequence is not recognised

%<unexpected char> is expanded as ?<unexpected char>

%^<unexpected char> is expanded as %^<unexpected char>

7.5 *Output routines using EXPAND/APPEND*

call WRITEF(<args as for EXPAND>)

This routine calls EXPAND on its arguments, and then calls TTWSTR. This latter call writes the contents of /EXPC/ to SYS$OUTPUT using VIO$PUT_OUTPUT.

call WRITAP(<args as for APPEND>)

This routine calls APPEND on its arguments, and then calls TTWSTR. This latter call writes the contents of /EXPC/ to SYS$OUTPUT, using VIO$PUT_OUTPUT.

The alias **WRTAPP** is provided for WRITAP.

7.6 *Routine to output angles*

char = DISPANG(secs, flg)

out -	character*14	**char**	the angle in DD MM SS.SSS format. The format depends on the value of **flg**
in -	real*8	**secs**	the angle to output in second

in - integer **secs** output format to use; see below

This routine is used to output an angle in the format that the routine READANG can read. See the chapter on reading numbers for details of these formats.

The value of **flg** should be one of the following values, as defined in LSL$CMNLSL:READANG.PAR

READANG_ANGLE	for angle with no hemisphere
READANG_LONGITUDE	for angle represents a longitude
READANG_LATITUDE	for angle represents a latitude

CHAPTER 8

/TXTC/ - THE INPUT COMMON BLOCK

8.1 *Introduction*

This chapter describes the text input common block.

8.2 */TXTC/ - the input common block*

Those LSLLIB routines that perform input, or process input, from within a program, use the text input common block by default. This common block is defined in both Fortran and Macro. The Fortran definition is kept as

LSL$CMNLSL:TXTC.CMN

and the macro definition as

LSL$CMNLSL:TXTC.MAR

The common block contains the following:

public - long parameter **MAX_TXTLIM**
the maximum length of TXTBUF, thus the maximum value to which TXTLIM may be set. This is currently 1024, which is the maximum buffer size that the DCL routines will read.

public - long parameter **DEF_TXTLIM**
the default length of TXTBUF, thus the default value to which TXTLIM is set. This is currently 255.

private - word **DCPTR**
the decoding pointer into **TXTBUF** - it is used by RDCH (and hence all other /TXTC/ reading routines) to determine which character is to be read next. If you want to manipulate the decoding position within **TXTBUF**, see the section "Manipulating the decoding pointer" in Chapter 9

public - word **TXTPTR**
holds the number of characters currently in **TXTBUF**, and can thus also be regarded as an "end of line" indicator

public - word **TXTLIM**
the maximum number of characters which may be held in **TXTBUF**. This is set by LSL_INIT to be DEF_TXTLIM, but the user may reset it to any

value up to MAX_TXTLIM, using SET_TXTLIM.

public - alias **TXTDSC**
equivalenced onto the common block to produce a fake string. It is made up of **TXTPTR**,**TXTTYP** and **TXTADD**. See the section on fake strings in Chapter 1 for an explanation of this.

private - byte **TXTTYP**
the type of the /TXTC/ fake string

private - byte **TXTCLA**
the class of the /TXTC/ fake string

private - long **TXTADD**
the address of **TXTBUF**. **TXTADD** is set by the initial call to LSL_INIT.

public - byte **TXTBUF(MAX_TXTLIM)**
the buffer to hold the actual characters in /TXTC/. Note that only TXTLIM of it will be used at any one time.

8.2.1 *Manipulating the size of TXTBUF*

call SET_TXTLIM([length])

in - word **length** the new length for TXTBUF

SET_TXTLIM will change the value of TXTLIM, and hence the size of TXTBUF, to the given **length**. If **length** is omitted, then TXTLIM is set back to the default value of DEF_TXTLIM. If **length** is less than 1, or greater than MAX_TXTLIM, then the call is ignored.

call SAVE_TXTLIM(length)

in - word **length** the current length of TXTBUF

SAVE_TXTLIM returns the current value of TXTLIM, and hence the current size of TXTBUF, in **length**.

8.2.2 *Saving and restoring TXTBUF and TXTPTR*

call SAVE_TXTC(buffer, bufptr, bufmax)

out - byte **buffer** the buffer to save TXTBUF in
out - word **bufptr** how many characters were saved
in - word **bufmax** maximum number of characters to save

SAVE_TXTC saves the contents of TXTBUF in **buffer**, and TXTPTR in **bufptr**. However, if TXTPTR is greater than **bufmax**, then only **bufmax** characters are stored, but **bufptr** is still set to TXTPTR.

It is assumed that **buffer** is at least **bufmax** bytes long.

call RESTORE_TXTC(buffer, bufptr, bufmax)

in	- byte	**buffer**	the buffer to copy into TXTBUF
in	- word	**bufptr**	how many characters to copy in
in	- word	**bufmax**	maximum number of characters to copy

RESTORE_TXTC is used to restore the contents of TXTPTR and TXTBUF. If **bufptr** is greater than **bufmax**, then only **bufmax** characters are restored, but TXTPTR is set to **bufptr**

It is assumed that **buffer** is at least **bufmax** bytes long.

CHAPTER 9

BASIC ROUTINES FOR READING FROM /TXTC/

9.1 *Initialising the input buffer*

9.1.1 *Initialising /TXTC/ for read*

By default, the input buffer is that in /TXTC/. Before it is used, the decoding pointer must be reset:

call BSLN

BSLN resets the decode pointer to the start of TXTBUF. It should thus always be called before starting to decode a line held in /TXTC/

Note that since BSLN resets the pointer to the beginning of TXTBUF, it unsets the effect of any previous call of SETAUX

9.1.2 *Choosing to read from another buffer*

call SETAUX(buffer, length)

in	- byte	**buffer**	new text input buffer
in	- word	**length**	the length of buffer required

SETAUX defines a new text input buffer. All calls of RDCH and the other input buffer reading routines will now refer to this new buffer. The end-of-line is defined by **length** - there are **length** characters available to be read in **buffer**.

Note that SETAUX has the effect of BSLN on the new buffer; BSLN cancels the effect of SETAUX. BSCH and RDCH, and hence all the other decoding routines, are unchanged in their application to the auxiliary buffer.

9.1.3 *Choosing part of TXTBUF to read*

call SETWIN([ptr], [length])

in	- word	**ptr**	the position to start reading at
in	- word	**length**	number of characters that may be read

SETWIN offers facilities for use when decoding fixed or partially fixed format records. It defines a sub-window of the current buffer (either TXTBUF or one defined by SETAUX). The end-of-line is effectively at the position defined by

length. In the current implementation, however, BSCH can go back beyond the start of the window.

If **ptr** is not given, then it defaults to the current position in the buffer. If **length** is not given, then it defaults to the rest of the current line.

Note that BSLN or SETAUX must still be used to initialise reading from a new buffer before using SETWIN.

9.2 *Manipulating the decode pointer*

The following routines are supplied to change the position from which the next character (in the current input buffer) will be read.

9.2.1 *Backspacing by one character*

call BSCH

BSCH backspaces the buffer pointer one space. If the pointer is already at the beginning of the line, then it does nothing.

Thus the next call of RDCH will return the same character as the last one.

9.2.2 *Saving and restoring the decode pointer*

DCPSET and DCPSAV are provided to allow users of the command routines to remember where decoding had reached, attempt to read more of the buffer, and then return to the original position.

Thus the actual explicit value of DCPTR (the buffer decoding pointer) should never normally be needed.

call DCPSAV(ptr)

out - word **ptr** to save the pointer in

DCPSAV saves the current value of the buffer decoding pointer. The value may then be reset at a later stage using DCPSET

call DCPSET(ptr)

in - word **ptr** pointer saved by DCPSAV

DCPSET restores the buffer decoding pointer to the position current when DCPSAV was used to save **ptr**.

9.3 *Reading a single character*

end = RDCH(ich)

```
     out - logical end        true if no more characters to be read
     out - byte    ich        the ASCII value of the character read
```

RDCH puts the next character from the buffer into **ich**. If there is no next character, then it returns true, and sets **ich** to zero. Otherwise, it returns false.

```
     end = RDCHS( ich )

     out - logical end        true if no more characters to be read
     out - byte    ich        the ASCII value of the character read
```

RDCHS behaves as RDCH, except that it will ignore any spaces or tabs.

CHAPTER 10

READING NUMBERS

10.1 *Introduction*

The routines described in this chapter are used to read numbers from the current input buffer. This latter defaults to /TXTC/

10.1.1 *Errors whilst reading numbers*

Each number reading routine checks for various forms of error, including numeric overflow and underflow (as appropriate). The errors occurring are reported in ERRNUM and LSL_EXCP, both in /EXCEPTION/. See the chapter on errors and exceptions for more details.

If an error occurs whilst reading a number, such that no sensible result is available, then the routine will return zero (0).

10.2 *Reading an integer*

The integer reading routines read an integer from the current buffer, using RDCH. Note that leading spaces and tabs are ignored.

10.2.1 *Reading to different bases*

Although each routine has a defined base in which it reads (usually decimal, octal or hexadecimal), it is possible to accept numbers in other bases. This is done by prefixing the number with a radix escape sequence. These are

```
^B  the number is in binary
^O  the number is in octal
^D  the number is in decimal
^X  the number is in hexadecimal
```

Upper and lower-case are both acceptable (in the escape sequences, and also in hexadecimal numbers). The sign of a number (if any) should precede the radix escape sequence (if any).

10.2.2 *Reading word length integers*

nodig = RDINT(num)

out - logical **nodig** true if doesn't read a legal number
out - word **num** the integer that is found

Reads a signed decimal integer, returning the value in **num**.

The result of the function is true if no number was read, or if overflow was encountered while reading the number; in either case, **num** is set to zero, and /EXCEPTION/ERRNUM and LSL_EXCP are set appropriately. If reading the number succeeds, the function result is false.

nodig = RDOCT(num)

out - logical **nodig** true if doesn't read a legal number
out - word **num** the integer that is found

Octal version of RDINT - reads an octal number

nodig = RDHEX(num)

out - logical **nodig** true if doesn't read a legal number
out - word **num** the integer that is found

Hexadecimal version of RDINT - reads a hexadecimal number.

10.2.3 *Reading longword length integers*

nodig = RDLONG(lnum)

out - logical **nodig** true if doesn't read a legal number
out - long **lnum** the integer that is found

This routine behaves as RDINT, except that it reads a long integer.

nodig = RDLOCT(lnum)

out - logical **nodig** true if doesn't read a legal number
out - long **lnum** the integer that is found

Octal version of RDLONG - reads a long octal number

nodig = RDLHEX(lnum)

out - logical **nodig** true if doesn't read a legal number
out - long **lnum** the integer that is found

Hexadecimal version of RDLONG - reads a long hexadecimal number.

nodig = RDNUM(lnum, base)

out - logical **nodig** true if doesn't read a legal number
out - long **lnum** the integer that is found
in - long **base** the default base in which to read

This routine behaves as RDLONG, except that any default base can be specified. Thus a call of

```
NOTHING = RDNUM( NUMBER, 10)
```

is equivalent to

```
NOTHING = RDLONG( NUMBER )
```

10.3 *Reading a real number*

nodig = RDREAL(rnum)

out - logical **nodig** true if doesn't read a legal number
out - real **rnum** the real number that is found

Reads a real number into **rnum**. The real number may be in any of the following forms (described more formally further on)

```
a             standard integer - coerced to a real
a.b           standard real format
a.bEc         standard exponential format
a.b/c.d       rational format
```

The function result is true if no number was found, or if floating overflow or underflow, or integer overflow was encountered while the number was being read; in any of these cases, the value **rnum** is set to 0.0, and /EXCEPTION/ERRNUM and LSL_EXCP are set appropriately. The function result is otherwise false.

nodig = RDDBLE(dnum)
nodig = RREAL8(dnum)

out - logical **nodig** true if doesn't read a legal number
out - dreal **dnum** the real number that is found

Reads a double precision real number (a Fortran REAL*8) into **dnum**. The real number may be in any of the following forms (described more formally further on)

```
a             standard integer - coerced to a real
a.b           standard real format
a.bEc         standard exponential format
a.bDc         standard double precision exponential format
a.b/c.d       rational format
```

The function result is true if no number was found, or if floating overflow or underflow, or integer overflow was encountered while the number was being read; in any of these cases, the value **dnum** is set to 0.0, and /EXCEPTION/ERRNUM and LSL_EXCP are set appropriately. The function result is otherwise false.

nodig = LSL_RDREAL_WHOLE(rnum,is_real)
nodig = LSL_RDDBLE_WHOLE(dnum,is_real)

out - logical	**nodig**	true if doesn't read a legal number
out - real	**rnum**	the real number that is found
out - dreal	**dnum**	the (double) real number that is found
i/o - logical	**is_real**	set true if number was definitely real

Both functions read a real number of the appropriate size into **rnum** or **dnum**.

These functions are identical to RDREAL and RDDBLE, with the addition of the **is_real** argument. This is set to TRUE if the number read was definitely a real - that is, if it contained any of the symbols '.' (decimal point), 'E', 'e', 'D' or 'd' (exponentiation), or '/' (division). The argument **is_real** will not be altered in any other way, so a calling routine should take care to initialise it to FALSE before using LSL_RDREAL_WHOLE or LSL_RDDBLE_WHOLE.

10.3.1 *Syntax of a real number*

The syntax of a real number may be defined as:

```
          <real number> ::= <unsigned real number> |
                            <sign><unsigned real number>
 <unsigned real number> ::= <basic real number> |
                            <basic real number><exponent> |
                            <basic real number>'/'<basic real number>
                 <sign> ::= '+' | '-'
    <basic real number> ::= <whole part> |
                            <fraction part> |
                            <whole part><fraction part>
           <exponent>  ::= 'E'<signed decimal number>
          <whole part> ::= <decimal number>
        <fraction part> ::= '.'<decimal number>
<signed decimal number> ::= <decimal number> |
                            <sign><decimal number>
       <decimal number> ::= <digit> |
                            <decimal number><digit>
              <digit> ::= '0' | '1' | '2' | '3' | '4' |
                            '5' | '6' | '7' | '8' | '9'
```

The double length real numbers read by RDDBLE/RREAL8 require the modification:

```
           <exponent>  ::= 'E'<signed decimal number> |
                           'D'<signed decimal number>
```

10.4 *Reading angles*

nodig = READANG(secs,flg)

out - logical	**nodig**	true if doesn't read a legal angle; see below for formats of angles
out - real*8	**secs**	the number of seconds that are read from the angle
out - integer	**flg**	one of the parameters defined in

LSL$CMNLSL:READANG.PAR; see below

This routine is designed to read an angle, or a geographical latitude or longitude, from the current position in the input buffer. The angle is expressed in degrees, minutes and seconds, followed by an optional "N", "S","E" or "W" to represent the hemisphere. Angles with an absolute value greater than 360 degrees are invalid, as are latitudes greater than 90 degrees and longitudes greater than 180 degrees.

It returns the angle, as a REAL*8, as seconds of arc.

The value of **flg** is one of the following values, as defined in LSL$CMNLSL:READANG.PAR

Values of **flg** when READANG is .TRUE. are:-

READANG_EOL	end of line
READANG_ILLEGDECPT	error in format of angle (only last element is allowed a ".")
READANG_SGNHANDEMI	sign and hemisphere present
READANG_ILLEGANGVAL	error in value of an element (eg degrees > 360, min > 60 etc)
READANG_ILLEGCHAR	unexpected character at start of number

Values of **flg** when READANG is .FALSE. are:-

READANG_ANGLE	for angle with no hemisphere
READANG_LONGITUDE	for angle represents a longitude
READANG_LATITUDE	for angle represents a latitude

10.4.1 *Syntax of an angle*

Angles are specified in the format sDDD MM SS.SSS or DDD MM SS.SSSh where:

s	is + , - or <blank>
h	is N, n, S, s, E, e, W or w; representing a hemisphere
DDD	is an integer value for the degrees, or a real number, if there are no minutes or seconds.
MM	is an integer value for the minutes, or a standard real number if there are no seconds
SS.SSS	is a standard real value for the seconds.

NOTES

i) In angles the exponential and rational real number formats are not permitted.

ii) Elements can be excluded from the left of an angle, but not from the right. That is the seconds (and minutes) need not be present, but the degrees must always be.

iii) The value in the signed format must be less than 360 degrees. The maximum value of an angle in the E/W hemisphere is 180 degrees and in the N/S hemisphere is 90 degrees.

iv) The value of the minutes and seconds must be less than 60.0.

v) Sign and hemisphere are mutually exclusive. When the hemisphere form is used, south and west are taken as negative.

vi) Angles are terminated by:

1. the occurrence of three elements of an angle
2. the occurrence of a hemisphere sign
3. the end of line

EXAMPLE

The following all represent the same angular value:

```
-00 00 30.00
 00 00 30.00 S
 00 00 30 s
 00 00.5  W
 0.00833333 w
```

CHAPTER 11

READING STRINGS

11.1 *Introduction*

The routine described in this chapter is used to read strings from the current input buffer. This latter defaults to /TXTC/

11.2 *Reading strings*

READSTR is the general string reading function, and reads from the current position in the input buffer (thus normally TXTBUF, unless it has been reassigned).

len = READSTR(string, [term_char], [term_cond], [skip], [retval])

out	long	**len**	the number of characters read into **string**, or zero if none were read
out	char	**string**	the destination character string
in	byte	**term_char**	character to end reading the string on
in	long	**term_cond**	condition to end reading the string on
in	logical	**skip**	true if to skip leading spaces/tabs
out	long	**retval**	error code

The following notes apply:

* if **term_char** is present, then it may be regarded as a "closing quote" character, depending upon the value of **term_cond**. Note that zero and negative values will be accepted as legitimate "characters".

* if **term_cond** is present, then it must be one of the following values, as defined in LSL$CMNLSL:READSTR.PAR

  ON_CMD - terminate as for RDCOMM commands - ie on any character that is not alphabetic or "_". **term_char** will be ignored.
  ON_CHAR - if **term_char** is present, then terminate on that character, otherwise treat as ON_EOL
  ON_CHAR2 - as for ON_CHAR, but the sequence **term_char term_char** inserts **term_char** into **string**
  ON_SPACE - terminate on space or tab (as well as on **term_char**, if it is present)
  ON_EOL - terminate at end of line only

String reading will always terminate at end of line. If **term_cond** is absent, then the default termination condition is ON_CHAR if **term_char** is present, and ON_EOL if it is not.

* **skip** is true if to skip leading spaces and tabs before the string, and false if to read them into the string. If this argument is omitted, then it is assumed true.

* **retval** is set to one of the following values:

LSL__BADTCOND - failure - an invalid value was given for **term_cond**. The routine returns at once, and does not read any characters.
LSL__STRTOOLONG - failure - the string to be read was longer than **string**, and reading was terminated when **string** was filled, rather than when the termination condition was satisfied. A call of RDCH will read the character that would not fit into **string**.
LSL__STREOL - success - the string was terminated by end of line. A call of RDCH will return TRUE (ie end of line).
LSL__STRSPACE - success - the string was terminated by a space. A call of RDCH will read the space.
LSL__STRCHAR - success - the string was terminated by reading **term_char**. A call of RDCH will read the character after **term_char**.
LSL__STRCMD - success - the string was terminated under the ON_CMD condition. A call of RDCH will read the character that stopped string reading.

11.3 *Reading Yes or No*

ret = RDYES(prompt, yesno, [default], [ierr])

out	long	**ret**	returns LSL__NORMAL if the read succeeds, otherwise see below
in	char	**prompt**	the question to ask
out	logical	**yesno**	true if the answer is yes, false if the answer is no
in	logical	**default**	what <return> means
out	long	**ierr**	the system error code - not used if the routine succeeds

* if **default** is present, then it is one of the longword parameters defined in

LSL$CMNLSL:RDYES.PAR

with meaning as follows:

ASSUME_NONE - no default is accepted - a reply of <return> will cause the routine to reprompt with **prompt**

ASSUME_YES - a reply of <return> means yes

ASSUME_NO - a reply of <return> means no

* if **default** is absent, it defaults to ASSUME_NONE

* The following values are returned in **ret**

LSL__NORMAL	- success - the line was read successfully
LSL__EOF	- warning - end of file was read (ie the user typed control-Z)
LSL__SYSERR	- error - an error occurred within the routine. If given, **ierr** will hold an appropriate system error code

The routine works as follows:

1. if **default** is absent or not a value defined in LSL$CMNLSL:RDYES.PAR, then it is regarded as being ASSUME_NONE

2. it uses EXPAND to append '? ' to **prompt**

3. it uses TTRSTR to ask the user for a reply, using EXPDSC as the prompt

4. it uses RDCHS to read the first character of the reply, and interprets it as follows:

Y or y	the answer is Yes - it returns true in **yesno**
N or n	the answer is No - it returns false in **yesno**
<return>	if **default** is ASSUME_NO, then the answer is No - it returns false in **yesno**
<return>	if **default** is ASSUME_YES, then the answer is Yes - it returns true in **yesno**
otherwise	it asks the user to reply with one of the valid answers, prompting with the appropriate one of:

```
Please answer with Y for Yes, N for No
Please answer with Y for Yes, N or <return> for No
Please answer with N for No,  Y or <return> for Yes
```

and asking the question again

Note that each time the question is asked, EXPAND is used to append '? ' to **prompt**. Thus, if **prompt** is EXPDSC, confusing results may ensue.

Any errors in TTRSTR will result in the errors being returned in **ret** and perhaps **ierr**.

11.4 *The number of characters in a string*

len = SIGCHS(string)

out - long	**len**	returns the number of significant characters in a character string
in - char	**string**	the string to test

This function finds the last character in **string** that is not a space or tab, and returns its position in **length**. It thus works out the number of 'significant' characters in **string**.

CHAPTER 12

FILENAME PARSING ROUTINES

12.1 *Introduction*

The routines described in this chapter are used to recognise and parse filenames.

12.2 *The filename common block*

This is kept in the file LSL$CMNLSL:FILENAME.CMN

All variables and parameters are public, and the user is free to alter values within the common block. It is defined as follows:

parameters - file name components sizes
The following integer (longword) parameters define the maximum lengths of the various components of a filename

C_NOD_SIZ = 6 - the node name

C_DEV_SIZ = 20 - the device specification

C_DIR_SIZ = 100 - the directory specification

C_NAM_SIZ = 39 - the actual file name

C_EXT_SIZ = 39 - the file extension or file type

C_VER_SIZ = 6 - the version number

C_MAX_SIZ - the total file name - the sum of the above values, plus seven, to allow for "::", ":", "[]", "." and ";"

character variables - the actual file name components
Each is of the appropriate size defined above

STR_NOD - the node name - does not include username or password

STR_DEV - the device name

STR_DIR - the directory specification - may include embedded periods

STR_NAM - the file name

STR_EXT - the file extension or type

STR_VER - the version number, as a string - may be negative

logical variables - was each part present

These are set TRUE if the relevant entry was present, or is required (see routine documentation for their use). There is one for each file name component, thus **HAD_NOD, HAD_DEV, HAD_DIR, HAD_NAM, HAD_EXT** and **HAD_VER**

longword variables - the lengths

For each component, the actual length of the component is recorded (which may be less than the permitted length). There is a longword for each component, thus **LEN_NOD, LEN_DEV, LEN_DIR, LEN_NAM, LEN_EXT** and **LEN_VER**

12.3 *Reading filenames from the current buffer*

GETFILNAM is the general filename reading function, and reads from the current position in the input buffer (thus normally TXTBUF, unless it has been reassigned). It uses the READSTR function (qv) to read the string in, and then the PARFILN (see below) function to parse the filename and check that it is valid.

Note that this ensures that /FILENAME/ is set up for the final filename.

ok = GETFILNAM(name, namlen, default, allow_ver,
[term_char], [devour])

out	- long	**ok**	LSL__NORMAL if the filename is read successfully, otherwise see below
out	- char	**name**	character string to receive the resulting filename
out	- word	**namlen**	the number of characters placed into **name**
in	- char	**default**	a template to build the filename against
in	- logical	**allow_ver**	true if to allow version numbers on the filename
in	- byte	**term_char**	termination character for the filename
in	- logical	**devour**	true if to 'eat up' the termination character

The function reads the filename, ignoring leading spaces or tabs. It will end the filename on a space, tab, end-of-line, or **term_char** if it is given.

If **devour** is true, or absent, then the next call of RDCH will return the character after the terminating character, but if **devour** is false, then the next RDCH will return the terminating character.

Having obtained the string, the function uses PARFILN to parse it against the default, and then returns with the final filename in **name**.

The following values of **ok** may be returned

LSL__NORMAL - filename successfully read into **name**
LSL__BADTCOND - failure - an invalid value was given for the termination condition to READSTR, called internally. This should not happen.
LSL__STRTOOLONG - failure - the string to be read was longer than **name**, and reading was terminated when **name** was filled, rather than when the termination condition was satisfied.
LSL__DEFTOOBIG - warning - one of the components of **default** was too long - the component will have been truncated
LSL__BADPARSE - failure - too few arguments were supplied to FILE_PARSE - this reflects a bug in the routine - please report it!
LSL__DEFFILNAM - failure - an error occurred in parsing **default**
LSL__DEFVERNUM - failure - a version number was supplied in **default**, although **allow_ver** is false
LSL__SRCTOOBIG - warning - one of the components of the filename read was too long - the component will have been truncated
LSL__SRCFILNAM - failure - an error occurred in parsing the filename read
LSL__SRCVERNUM - failure - a version number was supplied in the filename read, although **allow_ver** is false
LSL__FILTOOLONG - warning - the resulting filename was too long to fit into **name**

12.3.1 *Examples of reading a filename*

These examples demonstrate the use of **devour**. If we have a filename presented as

```
"FILENAME.SRC"
```

then we would detect the first quote with RDCH(ich), and our call might then be

```
ok = GETFILNAM( name, namlen, default, .TRUE.,
                ich, .TRUE.  )
```

which reads the filename and the final quote.

If our filename was presented as

```
/FILE=FILENAME.SRC/FRED=BILL
```

then we would have used RDCOMM to read the FILE= part of the command, and might then call

```
ok = GETFILNAM( name, namlen, default, .TRUE.,
                 '/', .FALSE.  )
```

This would leave us ready to detect the next slash with RDCH

12.4 *PARFILN - parse a file name*

PARFILN is used to parse a filename, filling it in from a default string if required.

ok = PARFILN(result, reslen, source, default, allow_ver)

out - long	**ok**	LSL__NORMAL if the filename is parsed successfully, otherwise see below
out - char	**result**	the final result of the parse is placed here
out - word	**reslen**	the total length of the filename in result (NB if the filename is too long to fit into **result**, the length of **result** is returned)
in - char	**source**	this is the filename as supplied by the user
in - char	**default**	this is the default or template filename
in - logical	**allow_ver**	true if to allow version numbers, false if not to. NB - if this is false, they are not allowed in **default** either. If true then a version number must be supplied in **default**. If the default file is to be that with the latest version number, version ';0' should be specified eg 'LSL$IF:IFF.IFF;0'.

Thus a typical call might be:

```
OK = PARFILN( RESULT, RESLEN,
              TXTDSC, 'LSL$IF:IFF.IFF;0', .TRUE.  )
```

The function works by

1. Unsetting all elements in /FILENAME/ - that is all lengths are set to zero, and all logicals are set false
2. Calling FILE_PARSE on **default** - this parses the default string into the common block
3. Checking **allow_ver** - if HAD_VER is true, and **allow_ver** is false, then exit with LSL__DEFVERNUM
4. Calling FILE_PARSE on **source** - this parses the source string into the common block as well
5. Checking **allow_ver** - if HAD_VER is true, and **allow_ver** is false, then exit with LSL__DEFVERNUM
6. Calling EXPFLN to expand the combined filename into **result** and **reslen**
7. If **reslen** greater than len(**result**)
   then **reslen** := len(**result**), and return with LSL__FILTOOLONG
8. success => return LSL__NORMAL

Possible values returned in **ok** are:

LSL__NORMAL - success - filename successfully parsed
LSL__DEFTOOBIG - warning - one of the components of **default** was too long - the component will have been truncated
LSL__BADPARSE - failure - too few arguments were supplied to FILE_PARSE - this reflects a bug in the routine - please report it!
LSL__DEFFILNAM - failure - an error occurred in parsing **default**
LSL__DEFVERNUM - failure - a version number was supplied in **default**, although **allow_ver** is false
LSL__SRCTOOBIG - warning - one of the components of **source** was too long - the component will have been truncated
LSL__SRCFILNAM - failure - an error occurred in parsing **source**
LSL__SRCVERNUM - failure - a version number was supplied in **source**, although **allow_ver** is false
LSL__FILTOOLONG - warning - the resulting filename was too long to fit into **result**

12.5 *EXPFLN - expand a filename from its parts*

EXPFLN is used to expand a full filename from the parts in /FILENAME/. It does not write anything to the common block.

ok = EXPFLN(string, strlen)

out - long	**ok**	LSL__NORMAL if the filename is expanded successfully, otherwise see below
out - char	**string**	the string to put the filename into
out - word	**strlen**	the length of the filename

Note that if the resultant filename is too long to fit into **string**, then **strlen** will hold the full length, but **string** will hold only a truncated form of the name.

ok may return either of the values

LSL__NORMAL	- success - filename expanded successfully
LSL__FILTOOLONG	- warning - the filename was too long, and has been truncated
LSL__NOFIELD	- warning - either the device or node field was flagged as present, but had zero length. Note that the other fields MAY be present and null (eg as in [].;)
LSL__FILNOLEN	- error - the assembled filename has zero length. This presumably reflects a mismatch between the HAD_xxx flags and the LEN_xxx sizes.

12.6 *PUTFLN - parse a filename into the common block*

PUTFLN places a filename into /FILENAME/, using FILE_PARSE

ok = PUTFLN(name)

where we have:

out - long	**ok**	LSL__NORMAL if the filename is parsed correctly, otherwise see below
in - char	**name**	the filename to parse

Where **ok** may have one of the values

LSL__NORMAL	- success - **name** parsed successfully, and placed into the common block
LSL__SRCTOOBIG	- warning - some component of the filename was too long, and has been truncated
LSL__BADPARSE	- failure - too few arguments to FILE_PARSE - this reflects a bug in the routine, and should be reported!
LSL__SRCFILNAM	- failure - some error occurred parsing **name**

12.7 *FILE_PARSE - the filename parsing routine*

Although not really intended to be called from outside PARFILN, this routine is documented for completeness. FILE_PARSE uses LIB$TPARSE to parse the filename handed to it (see next section for the filename definition it uses).

```
ierr = FILE_PARSE(
                FILE_NAME,
                HAD_NOD, STR_NOD, LEN_NOD,
                HAD_DEV, STR_DEV, LEN_DEV,
                HAD_DIR, STR_DIR, LEN_DIR,
                HAD_NAM, STR_NAM, LEN_NAM,
                HAD_EXT, STR_EXT, LEN_EXT,
                HAD_VER, STR_VER, LEN_VER
                )
```

out - long	**ierr**	system error return - LSL__NORMAL for success
in - char	**FILE_NAME**	the file name to be parsed
out - logical	**HAD_xxx**	true if field **xxx** is present
out - char	**STR_xxx**	field **xxx** is returned here
out - long	**LEN_xxx**	length of field in **STR_xxx**

Note that all arguments must be present. The **HAD_xxx**, **STR_xxx** and **LEN_xxx** arguments would normally be the appropriate variables from /FILENAME/.

FILE_PARSE parses the file name given, and sets the appropriate logicals, strings and lengths. If a filename field is not present, the arguments relating to it are not touched, thus allowing the usage of FILE_PARSE in PARFILN, where the **default** filename is read into the common block, and the **source** filename is then superimposed on it.

Note that all defining punctuation is omitted - eg the brackets around a directory name (although not the dots inside it), the colon after a device name, the semicolon before a version number. Also, blank fields are allowed, so that a filename such as

```
FRED.IFF;
```

will still cause HAD_VER to be set (since the ';' was present) although LEN_VER will be zero, and STR_VER will be empty (full of spaces).

The values normally returned in **ierr** are:

LSL__NORMAL	- success - filename parsed successfully
LSL__SYNTAXERR	- failure - there was a syntax error in the filename being parsed - parsing was abandoned. As much of the filename as was parsed correctly will have been placed into the arguments.
LSL__RESULTOVF	- failure - resultant string overflow. One of the fields of the filename was too long - parsing was abandoned, after placing a truncated version of the field into the appropriate string argument. The length argument for that field will be set to the full (untruncated) length of the field.
LSL__BADPARSE	- failure - too few arguments were supplied to FILE_PARSE - this reflects a bug in the routine and should be reported.
LSL__INTPARSERR	- failure - internal parsing error - should never occur, and should be reported if it does.

12.8 *Definition of a filename*

FILE_PARSE parses the source string into its constituent elements, using the following (rather informal) definition of a file.

A filename is a <FILE> where:

```
<file>           ::=  <node> <device> <directory> <name>
                      <extension><version>

<node>           ::=  null | <alphan> ':' ':'

<device>         ::=  null | <symbol> ':'

<directory>      ::=  null | '[' <dir_text> ']' |
                              '<' <dir_text> '>'

<dir_text>       ::=  <ndot> <dir_start> <dir_rest>

<ndot>           ::=  null | '.'

<dir_start>      ::=  null | '-' | <name>

<dir_rest>       ::=  null | '.' <name> <dir_rest>

<name>           ::=  null | <string>

<extension>      ::=  null | '.' <ext_text>

<ext_text>       ::=  null | <string>

<version>        ::=  null | ';' <ver_text> | '.' <ver_text>

<ver_text>       ::=  null | <signed_number>

<signed_number>  ::=  <sign>  <number>

<sign>           ::=  null | '+' | '-'
```

and, informally, we have

```
<alphan>         ::=  sequence of <letter>s and digits

<symbol>         ::=  sequence of <letter>s, digits, '_' and '$'

<number>         ::=  sequence of digits

<letter>         ::=  upper or lower case alphabetic
```

CHAPTER 13

TERMINAL INPUT/OUTPUT

13.1 *Introduction*

The terminal I/O routines provide the ability to read with prompt, and perform simple output to the terminal. Their main advantages are

* They use the common blocks defined in /TXTC/ and /EXPC/ by default
* They provide a constant interface in a simple manner
* Their i/o can be redirected by redefining the low level routines that they call

13.2 *Reading from the terminal*

ok = TTRSTR([string], [nchs], [prompt], [ierr])

out - long	**ok**	returns LSL__NORMAL if the read succeeds, otherwise see below
out - char	**string**	where to place the line read. Defaults to TXTDSC
out - long	**nchs**	the number of characters read
in - char	**prompt**	the string to prompt the user with. If this is not given, then no prompt string is used
out - long	**ierr**	the system error code - not used if the routine succeeds

TTRSTR reads a line from the terminal, into either **string** or /TXTC/ (as TXTDSC). If **string** is given explicitly, then the line length to be read is limited by the string length. If TXTDSC is defaulted, then the line length to be read is TXTLIM characters. Trailing spaces are ignored in the length returned in **nchs**.

Note that the string read is padded with spaces.

ok = TTRLIN([buffer], [nchs], [lim], [prompt], [ierr])

out - long	**ok**	returns LSL__NORMAL if the read succeeds, otherwise see below
out - byte	**buffer**	where to place the line read. Defaults to TXTBUF
out - word	**nchs**	the number of characters read into **buffer**. Defaults to TXTPTR
in - word	**lim**	the length of **buffer**. Defaults to TXTLIM

in - char	**prompt**	the string to prompt the user with. If this is not given, then no prompt string is used.
out - long	**ierr**	the system error code - not used if the routine succeeds

TTRLIN reads a line from the terminal, into either **buffer** or /TXTC/ (as TXTBUF and TXTPTR). Trailing spaces are ignored in the length returned in **nchs**. The line read is NOT padded with spaces - the characters after the **nchs**th byte are undefined.

13.2.1 *Error returns*

The following values are returned in **ok** for TTRSTR and TTRLIN:

LSL__NORMAL	- success - the line was read successfully
LSL__EOF	- warning - end of file was read (ie the user typed control-Z)
LSL__SYSERR	- error - an error occurred within the routine. If given, **ierr** will hold an appropriate system error code

13.3 *Writing to the terminal*

ok = TTWSTR([string], [ierr])

out - long	**ok**	returns LSL__NORMAL if the write succeeds, otherwise see below
in - char	**string**	the string to be output. Defaults to EXPDSC
out - long	**ierr**	the system error code - not used if the routine succeeds

TTWSTR writes a character string to the terminal. If no string is given, then the contents of /EXPC/ is written out.

ok = TTWLIN([buffer], [nchs], [ierr])

out - long	**ok**	returns LSL__NORMAL if the write succeeds, otherwise see below
in - byte	**buffer**	the buffer to be output. Defaults to EXPBUF
in - word	**nchs**	the number of byte to output. Defaults to EXPLEN
out - long	**ierr**	the system error code - not used if the routine succeeds

TTWLIN writes a byte array to the terminal. If no arguments are given, then it writes out the contents of /EXPC/.

13.3.1 *Error returns*

The following values are returned in **ok** for TTWSTR and TTWLIN:

LSL__NORMAL	- success - the line was written successfully
LSL__SYSERR	- error - an error occurred within the routine. If given, **ierr** will hold an appropriate system error code

13.4 *Changing the terminal I/O routines*

The user may supply routines to be used by LSLLIB for all its terminal input and output, in place of its defaults. This may be required in order to perform input/output to alternate devices, or to perform special carriage control facilities.

13.4.1 *Changing the terminal input routine*

call LSL_SET_INPUT(routine)

in - external **routine** the routine to be used for terminal input, declared EXTERNAL in the calling program

By default, LSLLIB terminal input uses the routine LIB$GET_INPUT. The user may substitute their own routine by calling LSL_SET_INPUT. The supplied routine must accept the same arguments as LIB$GET_INPUT, though at present it is only ever called with one or two arguments. Refer to the documentation for the LIB$ routines.

13.4.2 *Changing the terminal output routine*

call LSL_SET_OUTPUT(routine)

in - external **routine** the routine to be used for terminal output, declared EXTERNAL in the calling program

By default, LSLLIB terminal output uses the routine LIB$PUT_OUTPUT. The user may substitute their own routine by calling LSL_SET_OUTPUT. The supplied routine must accept the same arguments as LIB$PUT_OUTPUT. Refer to the documentation for the LIB$ routines.

13.5 *Low level routines*

The TTxLIN and TTxSTR routines call VIO$GET_INPUT and VIO$PUT_OUTPUT to read or write a line.

LSLLIB itself supplies these routines, to read from SYS$INPUT and write to SYS$OUTPUT using LIB$GET_INPUT and LIB$PUT_OUTPUT. Programs used to redirect terminal input/output by substituting their own routines for VIO$GET_INPUT and/or VIO$PUT_OUTPUT. This will not work if the shareable image form of LSLLIB is used, so LSL_SET_INPUT, and LSL_SET_OUTPUT should be used in preference.

The user may call VIO$GET_INPUT and VIO$PUT_OUTPUT directly, with the same argument lists as for LIB$GET_INPUT and LIB$PUT_OUTPUT. The result will be either a call to the LIB$ routine, or to the user's own routine if LSL_SET_INPUT or LSL_SET_OUTPUT have been called. It is not normal practice to call VIO$GET_INPUT and VIO$PUT_OUTPUT directly - TTRSTR, TTWSTR etc. should normally be used.

CHAPTER 14

FILE READ AND WRITE ROUTINES

14.1 *Introduction*

These routines are provided for file oriented I/O. That is, they enable the user to read and write any file structured device (note that this includes terminals).

For examples of their use, see the conversion utility I2MOSS.

14.2 *Unit numbers*

These routines do not use Fortran I/O, but interface directly with RMS (Record Management System, see the appropriate DIGITAL manuals). Thus the 'unit number' in use here is not related to the LUN (Logical Unit Number) that would be used for reading or writing a file in Fortran. A file opened with the FILEIO routines cannot be accessed by Fortran using that number, and vice-versa.

Each file is accessed via its own FAB and RAB, and these are associated with a particular unit number. A FAB is a File Access Block, which describes the particular file being dealt with, and a RAB is a Record Access Block, which is associated with a particular FAB and specifies how each record in the file will be dealt with.

Within the library, a table is built up, associating a unit number with each pair of FAB and RAB, and noting whether the unit is being used for input or output:

row	unit	FAB	RAB	i/o	file
0	2	1	1	i	FRED.DAT
1	4	2	2	o	MICK.LEG
2	-	-	-	-	no file, unit unassigned
3	-	-	-	-	" " " "

This example shows a table holding up to four units open at any one time (LSLLIB FILEIO will currently deal with up to fifteen, although this number may be increased by a simple amendment to the sources).

Unit numbers may be any values in the range 1 to 32767 - the actual values chosen are not significant. FILEIO will not allow an active unit number (one with an open file on it) to be reassigned - the file must be closed first.

14.3 *The routines*

14.3.1 *Opening files*

In the current version of LSLLIB FILEIO, no more than fifteen files may be open for FILEIO read/write at any one time.

The file open routines are:

ok = FLROPN(unit, filename, [ierr], [alq])

ok = FLROPB(unit, filename, [ierr], [alq])

ok = FLWOPN(unit, filename, [ierr], [alq])

ok = FLWOPB(unit, filename, [ierr], [alq])

ok = FLWEXT(unit, filename, [ierr], [alq])

ok = FLWOVW(unit, filename, [ierr], [alq])

ok = FLWUPD(unit, filename, [ierr], [alq])

out	- long	**ok**	LSL__NORMAL if the file is successfully opened, otherwise a relevant error code - see below.
in	- long	**unit**	is the unit number, a value between 1 and 32767.
in	- char	**filename**	a string specifying the name of the file to be opened.
out	- long	**ierr**	for some values of **ok**, this returns a supplementary system error code.
out	- long	**alq**	allocation file size - the allocated file size in blocks.

FLROPN The file is opened for reading. It may not be written to. It may be sequential or indexed sequential (both are sensible).

FLROPB The file is opened for block reading. It may not be written to. It may only be read block by block by routine FLRBLK.

FLWOPN A new file of the given name is created and opened for writing. This should only be used for sequential files; the file it creates is a sequential file.

FLWOPB A new file of the given name is created and opened for writing in block mode. It may be written block by block by routine FLWBLK, and the blocks may then be read using FLRBLK.

FLWEXT The file is opened for extending. If the given file does not exist, an error is given and the file open is abandoned. This should be used to open files for extending - the file may be sequential or indexed sequential, but for sequential files data may only be added to the end of the file, and for indexed sequential files only records with an unduplicated key may be inserted.

FLWOVW The file is opened for overwriting. If the file does not exist, an error is given and the file open is abandoned.

For a sequential file, all data in the file is lost, and the effect is of starting a new file with the same version number as the old file.

For an indexed sequential file, this open allows the overwriting of records - if the key of the record to be inserted is unmatched, then the record is just inserted, but if the key is duplicated, then the record is overwritten and the old value lost.

FLWUPD The file is opened for updating. If the given file does not exist, an error is given and the file open is abandoned.

If a file is opened for read then it is automatically selected as the current file for read, and similarly a file selected for write is automatically selected as the current file for write. For a discussion of how selection of files is dependent upon how the file was opened, see the section on "Selecting files" below.

The following values of **ok** may be returned:

LSL__NORMAL - success - file opened successfully
LSL__MISSARGS - failure - one or more required arguments are missing - the routine is abandoned
LSL__LUNINUSE - failure - a file has already been opened with this unit number - the file is not opened
LSL__NOSUCHFILE - failure - the file could not be found - the open has been abandoned. This will never be returned by FLWOPN
LSL__NOLUNS - failure - there is no room for another record in the unit table, so the file is not opened
LSL__FILINUSE - failure - the file has already been opened (probably by someone else), and is locked - the open is abandoned
LSL__SYSOPEN - failure - an error occurred during the open process - a system error which should give more details will be found in **ierr** (for instance, if the $OPEN or $CREATE failed)

14.3.2 *Selecting files*

These routines allow the selection of a particular file for reading or writing. They do not check that the file selected was actually opened in a relevant manner.

The file select routines are:

ok = FLRSEL(unit)

ok = FLWSEL(unit)

out - long **ok** LSL__NORMAL if the file on the required unit is selected successfully, otherwise a relevant error code - see below
in - long **unit** is the unit number, a value between 1 and 32767.

FLRSEL The file associated with the given unit is selected for reading - the FAB/RAB combination associated with this unit will be used for future calls of the read routines. This is sensible if the file was opened for read with FLROPN or FLROPB, if it is an indexed sequential file opened for write with FLWEXT or FLWOVW, or if it is a block oriented file created with FLWOPB.

FLWSEL The file associated with the given unit is selected for writing - the FAB/RAB combination associated with this unit will be used for future calls of the write routines. This is sensible if the file was created with FLWOPN or FLWOPB, or if it was opened with FLWEXT, FLWOVW or FLWUPD.

The following values of **ok** may be returned:

LSL__NORMAL - success - file selected successfully
LSL__NOSUCHLUN - failure - the unit number requested is not present in the unit/FAB/RAB table.
LSL__MISSARGS - failure - one or more required arguments are missing - the routine is abandoned

14.3.3 *Saving current file selection*

These routines allow the user to save the unit number of the currently selected file, so that it may be reselected at a later time.

The save unit routines are:

ok = FLRSVL(savunit)

ok = FLWSVL(savunit)

out - long **ok** LSL__NORMAL if there was a currently selected file, otherwise a relevant error code - see below.
out - long **savunit** is the saved unit number.

FLRSVL The unit number of the file currently selected for reading is returned.

FLWSVL The unit number of the file currently selected for writing is returned.

The following values of **ok** may be returned:

LSL__NORMAL - success - the currently selected file unit number was returned in **savunit**.
LSL__NOSUCHLUN - failure - there was no currently selected file. **savunit** will be zero.
LSL__MISSARGS - failure - one or more required arguments are missing - the routine is abandoned

14.3.4 *Finding records in indexed sequential files*

Four routines are provided for finding records by key in indexed sequential files. They are:

ok = FLRFND(string, [key_of_ref], [greater], [ierr])

ok = FLWFND(string, [key_of_ref], [greater], [ierr])

ok = FLRFNB(buffer, buflen, [key_of_ref], [greater], [ierr])

ok = FLWFNB(buffer, buflen, [key_of_ref], [greater], [ierr])

out	long	**ok**	LSL__NORMAL if the search is successful, otherwise a relevant error code - see below
in	char	**string**	the string to search for.
in	byte	**buffer**	the buffer containing the string to search for.
in	long	**buflen**	the number of characters in **buffer**
in	long	**key_of_ref**	which key to search on. 0 (zero) is the default, and means the primary key. 1 would mean the first alternate key, and so on.
in	long	**greater**	if this argument is zero (the default) then a match will only occur if the string being searched for matches a key exactly. If it is 1 then the search will be satisfied if the key is greater than the string, and if it is 2 then it will be satisfied if the key is greater than or equal to the string.
out	long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLRFND finds a particular record in the file selected for reading
FLRFNB finds a particular record in the file selected for reading

FLWFND finds a particular record in the file selected for writing.
FLWFNB finds a particular record in the file selected for writing.

The following values of **ok** may be returned:

LSL__NORMAL - success - record found successfully
LSL__MISSARGS - failure - one or more required arguments are missing - the routine is abandoned
LSL__ILLEGLUN - failure - the unit number requested is not allowed, i.e. it is 0 or less.
LSL__SYSFIND - failure - an error occurred during the find. **ierr** should contain a system error code, which may give more details.

These routines will only work for indexed sequential files. They assume that any key being searched is a string key, but allow any level of key to be searched. Success is returned on finding a record with a key which matches the equal/greater condition passed in **greater**, and the current record in the file becomes the found record. The key supplied must be shorter than or of equal length to the key being searched. If it is shorter, then the comparison is made on only that number of characters in each key.

If the file is selected for read (whether it was opened for read or for write) then FLRLIN or FLRSTR will return the current (ie found) record after an FLxFND

If the file is selected for write, then FLULIN or FLUSTR may be used to update the current record after an FLxFND.

For a file opened with FLWEXT, the key specified in a record handed to FLWLIN or FLWSTR must be greater than the key of the current record, and unduplicated, for insertion to occur. For a file opened with FLWOVW, the situation is similar, except that a duplicated key will cause the record concerned to be overwritten (as if the relevant FLUxxx routine had been used).

14.3.5 *Rewinding files*

ok = FLRREW([unit], [ierr])

out - long	**ok**	LSL__NORMAL if the file is successfully rewound, otherwise a relevant error code - see below
in - long	**unit**	is the unit number, a value between 1 and 32767. If specified, the given unit is selected for read, otherwise the current unit is used.
out - long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLRREW rewinds the file currently selected for read. The next read will find the first record in the file.

This routine may also be used for files opened for write which have been selected for read. Note that the latter is only really useful for an indexed sequential file (when movement about the file is natural) or for a sequential file opened with FLWOVW (which will allow records to be overwritten again). If a sequential file is rewound and overwritten, everything after the current record (ie the one which has been most recently been written) will be lost on closing the file.

The following values of **ok** may be returned:

LSL__NORMAL	- success - file rewound successfully
LSL__NOSUCHLUN	- failure - the unit number requested is not present in the unit/FAB/RAB table.
LSL__ILLEGLUN	- failure - the unit number requested is not allowed, i.e. it is 0 or less.
LSL__SYSREW	- failure - an error occurred during the rewind. **ierr** should contain a system error code, which may give more details

14.3.6 *Reading a record*

These routines operate on the file currently selected for read. Defaults are taken from the common block /TXTC/.

ok = FLRLIN([buffer], [nchs], [buflen], [ierr])

ok = FLRSTR([string], [nchs], [ierr])

out - long	**ok**	LSL__NORMAL is the line is read successfully, otherwise a relevant error code - see below
out - byte	**buffer**	a byte array to receive the record which is being read. **buffer** defaults to TXTBUF.
out - word	**nchs**	the number of characters read, but see below. **nchs** defaults to TXTPTR.
in - word	**buflen**	the maximum length of **buffer**. **buflen** defaults to TXTLIM.
out - char	**string**	the string into which to read the record. **string** defaults to TXTDSC.
out - long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLRLIN This reads a record into **buffer**, of maximum length **buflen**. If **buffer** is TXTBUF, then BSLN should be called before using LSLLIB routines to read from the buffer. If **buffer** is not filled, then it is padded with spaces. Note that this does not affect the value of **nchs**.

FLRSTR This reads a record into **string**, padding after with spaces up to the string length. Note that if **string** is TXTDSC and **nchs** is TXTPTR (the defaults) then the string returned in TXTDSC is not padded - the string has effectively shrunk in length, instead.

If **string** is TXTDSC, then BSLN should be called before using LSLLIB routines to read from the buffer.

The following values of **ok** may be returned:

LSL__NORMAL	- success - record read successfully
LSL__EOF	- warning - end of file was read
LSL__RECTOOBIG	- warning - the record being read was too long to fit into the buffer supplied (whether a string or line read). The length read is returned correctly (as supplied by the system routine) in **nchs**
LSL__SYSREAD	- failure - an error occurred during the read. **ierr** should contain a system error code, which may give more details

14.3.7 *Reading a block*

This routine reads a block from a file opened with FLROPB or FLWOPB.

ok = FLRBLK(vbn, buffer, [read], [ierr])

out - long	**ok**	returns LSL__NORMAL if the block is read successfully, otherwise a relevant error code - see below
in - long	**vbn**	the virtual block number of the block to be read. If this is 0 then the next block will be read.
out - byte	**buffer**	the buffer to receive the record which is being read. This must be 512 bytes long.
out - long	**read**	the virtual block number of the block that was read
out - long	**ierr**	for some values of **ok**, this returns a

supplementary system error code

FLRBLK This routine reads a requested block into a user-defined buffer. Note that /TXTC/ is NOT involved.

The following values of **ok** may be returned:

LSL__NORMAL - success - block read successfully
LSL__EOF - warning - end of file was read
LSL__MISSARGS - failure - one or more required arguments are missing - the routine is abandoned
LSL__SYSREAD - failure - an error occurred during the read. **ierr** should contain a system error code, which may give more details

14.3.8 *Writing a record*

These routines operate on the file currently selected for write. Defaults are taken from the common block /EXPC/.

ok = FLWLIN([buffer], [nchs], [ierr])

ok = FLWSTR([string], [ierr])

out - long	**ok**	LSL__NORMAL if the line is written successfully, otherwise a relevant error code - see below
out - byte	**buffer**	the buffer from which to output characters. **buffer** defaults to EXPBUF.
out - word	**nchs**	the number of characters to output from **buffer**. **nchs** defaults to EXPLEN.
out - char	**string**	the string to be output. **string** defaults to EXPDSC.
out - long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLWLIN this routine outputs **nchs** characters from **buffer**.

FLWSTR this routine outputs **string**.

The following values of **ok** may be returned:

LSL__NORMAL - success - record written successfully
LSL__SYSWRITE - failure - an error occurred during the write. **ierr** should contain a system error code, which may give more details

14.3.9 *Writing a block*

This routine writes a block to a file opened with FLWOPB.

ok = FLWBLK(vbn, buffer, [read], [ierr])

out - long	**ok**	returns LSL__NORMAL if the block is written successfully, otherwise a relevant error code - see below
in - long	**vbn**	the virtual block number of the block to be

written. If this is 0 then the next block will be written.

in	- byte	**buffer**	the buffer containing the block which is to be written. This must be 512 bytes long.
out	- long	**read**	the virtual block number of the block that was written
out	- long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLWBLK This routine writes a specified block from a user-defined buffer. Note that /EXPC/ is NOT involved.

The following values of **ok** may be returned:

LSL__NORMAL - success - block written successfully
LSL__MISSARGS - failure - one or more required arguments are missing - the routine is abandoned
LSL__SYSWRITE - failure - an error occurred during the write. **ierr** should contain a system error code, which may give more details

14.3.10 *Updating a record*

These routines operate on the file currently selected for write, which is assumed to be an indexed sequential file. Defaults are taken from the common block /EXPC/.

ok = FLULIN([buffer], [nchs], [ierr])

ok = FLUSTR([string], [ierr])

out	- long	**ok**	LSL__NORMAL if the record is updated successfully, otherwise a relevant error code - see below
out	- byte	**buffer**	the buffer from which to output characters. **buffer** defaults to EXPBUF.
out	- word	**nchs**	the number of characters to output from **buffer**. **nchs** defaults to EXPLEN.
out	- char	**string**	the string to be output. **string** defaults to EXPDSC.
out	- long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLULIN this routine outputs **nchs** characters from **buffer** into the current record of an indexed sequential file

FLUSTR this routine outputs **string** into the current record of an indexed sequential file

The following values of **ok** may be returned:

LSL__NORMAL - success - record updated successfully
LSL__SYSUPD - failure - an error occurred during the update. **ierr** should contain a system error code, which may give more details

14.3.11 *Deleting a record*

This routine operates on the file currently selected for write, which is assumed to be an indexed sequential file.

ok = FLWRDL([ierr])

out	- long	**ok**	LSL__NORMAL if the record is deleted successfully, otherwise a relevant error code - see below
out	- long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLWRDL this routine deletes the currently selected record of an indexed sequential file.

The following values of **ok** may be returned:

LSL__NORMAL - success - record updated successfully
LSL__SYSUPD - failure - an error occurred during the deletion. **ierr** should contain a system error code, which may give more details

14.3.12 *Flushing buffers*

This routine allows the internal buffers for a file to be flushed out to disc, ensuring that the disc file is up to date.

The routine is:

ok = FLWUSH([unit], [ierr])

out	- long	**ok**	LSL__NORMAL if the buffers are flushed successfully, otherwise a relevant error code - see below
in	- long	**unit**	is the unit number, a value between 1 and 32767. If specified, the given unit is flushed, otherwise the current unit is used.
out	- long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLWUSH The file associated with the given unit is flushed, ensuring that all data written so far is actually on the disc. This can be useful when it is important that data is protected against system failure.

The following values of **ok** may be returned:

LSL__NORMAL - success - file flushed successfully
LSL__NOSUCHLUN - failure - the unit number requested is not present in the unit/FAB/RAB table.
LSL__ILLEGLUN - failure - the unit number requested is not allowed, i.e. it is 0 or less.
LSL__SYSFLUSH - failure - an error occurred during the flush. **ierr** should contain a system error code, which may give more details

14.3.13 *Closing files*

These routines close the specified file, which defaults to that currently selected, possibly performing some action on the file first. They then remove the file's entry in the unit/FAB/RAB table.

Note that closing a sequential file opened for write will truncate it - that is, the last record in the file will be that which was written last before the close. This is relevant if a file is opened with FLWOVW, rewound and then more data inserted - the original data is lost.

ok = FLRCLO([unit], [ierr])

ok = FLWCLO([unit], [ierr])

ok = FLWPRT([unit], [ierr])

ok = FLWDEL([unit], [ierr])

ok = FLWSPL([unit], [ierr])

ok = FLWSUB([unit], [ierr])

out - long	**ok**	LSL__NORMAL if the file is closed successfully, otherwise a relevant error code - see below
in - long	**unit**	is the unit number, a value between 1 and 32767. If specified, the given unit is selected before attempting the close, otherwise the file on the current unit is closed.
out - long	**ierr**	for some values of **ok**, this returns a supplementary system error code

FLRCLO closes the file currently selected for read. The file must have been opened for read.

FLWCLO closes the file currently selected for write. The file must have been opened for write.

FLWPRT closes the file currently selected for write, and prints it on the printer (SYS$SYSPRINT). The file must have been opened for write.

FLWDEL closes the file currently selected for write, and deletes it. The file must have been opened for write.

FLWSPL closes the file currently selected for write, and 'spools' it - the file is printed (as for FLWPRT) and then deleted. The file must have been open for write.

FLWSUB closes the file currently selected for write, and submits it to the batch queue. The file must have been opened for write. Don't forget that in order to charge the batch job to an account, the file must start with a CHARGE command.

The following values of **ok** may be returned:

LSL__NORMAL - success - file closed successfully
LSL__NOSUCHLUN - failure - the unit number requested is not present in the unit/FAB/RAB table.
LSL__ILLEGLUN - failure - the unit number requested is not allowed, i.e. it is 0 or less.
LSL__FAC - failure - file access conflict - an attempt has been made to close a file open for read with a write routine, or vice versa
LSL__SYSCLOSE - failure - an error occurred during the close. **ierr** should contain a system error code, which may give more details

14.3.14 *WRITEF routines*

The following routines are also supplied:

call FLWRTF(format, arg1, arg2, ...)

call FLWAPP(format, arg1, arg2, ...)

FLWRTF has the same effect as WRITEF, except that output is to the file currently selected for writing. The arguments are passed to EXPAND, and then FLWLIN is called to output them.

FLWAPP has the same effect as WRITAP (aka WRTAPP), except that output is to the file currently selected for writing. The arguments are passed to APPEND, and then FLWLIN is called to output them.

CHAPTER 15

DCL COMMAND LINE INTERPRETATION

15.1 *Introduction*

These routines enable the user to obtain a foreign command line using the VMS Command Line Interpreter (CLI) and to parse that command line against the definition contained in a related Command Language Definition (CLD) module. This means that Laser-Scan utilities can appear to the user with full VMS command syntax.

Routines are provided to

get and parse the command line (DCL_STARTUP)

just parse a command line (DCL_PARSE)

detect the presence of qualifiers (DCL_QUAL)

get arguments (DCL_DBL, DCL_INT, DCL_REAL, DCL_FILE, and DCL_STR)

DCL_REAL provides an extension to normal VMS command decoding as it collects real numbers from a command line. Similarly DCL_INT automatically expands integer ranges specified with the syntax **n:m**, where **n** is the start number and **m** the stop number (inclusive). These routines (collectively referred to here as the DCL_ routines) set up a common block to contain the results of command decoding.

All the routines are longword functions which return with SS$_NORMAL if they succeed and a system error code if they fail. Local error reporting within the routines (using LIB$SIGNAL) may be invoked if desired, or, the status of the function may be tested on return and errors reported using the users own output routines.

One of the initial routines DCL_STARTUP or DCL_PARSE must be invoked BEFORE any of the other DCL_ routines can be used.

NOTE - for conducting command dialogues within a program (as for LITES2 or ISELAC) use the terminal I/O and command reading routines documented in other chapters.

For details of how to create a CLD module, see the "Command Definition Utility" chapter in the VMS Utilities Reference Manual.

For examples of programs using CLD modules and the DCL routines, see the sources for the IMP utilties ISTART and IFILTER.

15.1.1 *Brief description of a command line*

This section establishes the terminology used to describe command lines in the rest of this chapter.

An example command line might have the form:

```
$  ICLIP   /OUTPUT=FRED.IFF   BILL.IFF
```

In this example,

ICLIP is the **verb** - it specifies the action to be performed. For LSL programs, this will normally be a foreign command (ie ICLIP is defined as the symbol "$ LSL$EXE:ICLIP")

/OUTPUT is a **qualifier** - it modifies the action of the program invoked by the **verb**. It has the filename "FRED.IFF" as its argument.

BILL.IFF is a **parameter** - it specifies what the program invoked by the **verb** should act upon.

A more complex example is:

```
$  IMERGE   /LOG   /OUTPUT=OUTFILE.IFF   INFILE1.IFF/LAYER=(1,2) -
$_  INFILE2.IFF/LAYER=3
```

In this case, both INFILE1.IFF and INFILE2.IFF are parameters. However, each parameter has qualifiers associated with itself alone - the /LAYER qualifiers are **positional qualifiers**, in that their meaning depends upon where they are found. Also note that the first /LAYER qualifier has a list of arguments, enclosed in parentheses and separated by commas.

15.2 */CLD/ - the common block*

The DCL_ common block is kept in the file

LSL$CMNLSL:CLD.CMN

and is defined as follows:

public - long parameter **MAX_FIL**
the maximum number of filenames, either from command parameters or from the arguments for qualifiers - thus the length of the FILARY and FIL_LEN arrays

public - long **NUMFIL**
the number of filename arguments or parameters found

public - char **FILARY(MAX_FIL)** holds the **NUMFIL** filenames

public - word **FIL_LEN(MAX_FIL)**
holds the length of each filename in FILARY

public - long parameter **MAX_LONG**
the maximum number of arguments that may be present for integer arguments - thus the length of the IARRAY array

public - long parameter **MAX_REAL**
the maximum number of arguments that may be present for real or real*8 arguments - thus the length of the RARRAY and DBLRAY arrays

public - long **NUMINT**
the number of integer arguments found

public - long **IARRAY(MAX_LONG)**
holds the **NUMINT** integer arguments

public - long **NUMREA**
the number of real arguments found

public - long **RARRAY(MAX_REAL)**
holds the **NUMREA** real arguments

public - long **NUMDBL**
the number of real*8 arguments found

public - long **DBLRAY(MAX_REAL)**
holds the **NUMDBL** real*8 arguments

public - long parameter **MAX_STR**
the maximum number of arguments that may be present for string arguments - thus the length of the CARRAY array

public - long **NUMSTR**
the number of string or character arguments found

public - char **CARRAY(MAX_STR)**
holds the **NUMSTR** string or character arguments, each containing up to 128 characters

public - word **STR_LEN(MAX_STR)**
holds the length of each string in CARRAY

15.3 *The routines*

15.3.1 *Get and parse the command line*

ok = DCL_STARTUP(verb, burst, cldname, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**verb**	the command verb used to invoke this program
in - logical	**burst**	this is for use with positional qualifiers, and should normally be false. Otherwise, see below.
in - long	**cldname**	this is the 'name' of the CLD module referred to by all the routines described here. It must be declared EXTERNAL in the main program.
in - logical	**report**	true if to allow DCL_STARTUP to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_STARTUP( 'ICLIP', .FALSE., ICLIP_CLD, .TRUE.  )
```

The function works as follows:

1. Calling LIB$GET_FOREIGN to get the command line.

   If no arguments were specified on the command line, then CLI$_PARSE prompts for input, using the prompt specified in the CLD module.

2. If **burst** is true, then DCL_COMMA is called to substitute spaces for any commas separating the elements of the parameter lists (note - DCL_COMMA is an internal routine and is not further documented)

3. **verb** is concatenated to the start of the command line

   The symbol used to invoke the program is removed from the beginning of the line returned by LIB$GET_FOREIGN. However, the CLD requires that the command verb be present at the start of a line to be parsed

4. CLI$DCL_PARSE is called to parse the command line against the contents of the CLD module **cld_name**

5. If any parameters are missing from the command string, then CLI$DCL_PARSE will prompt for them with LIB$GET_INPUT, using the prompt specified for that parameter in the CLD module

A completely parsed command line is now available for decoding. In general, the user has no need to see the actual command line, but if necessary the routine DCL_CML may be used to retrieve it.

15.3.1.1 *Bursting positional parameters -*

If positional parameters are being used, then **burst** can be used to control the decoding environment(s) returned.

If **burst** is true, then the commas separating the components of a parameter list are replaced with spaces. This breaks the list into its component parameters, each of which will then have a unique decoding environment for decoding positionally dependent qualifiers.

If **burst** is left false, then the unburst parameter list presents only one decoding environment, which means that only the first of multiple qualifiers which occur between the parameters will be read and decoded.

Note that although the command line will be typed by the user as a parameter list, the CLD module should define one 'REQUIRED' and then up to 7 optional parameters, none of which should be of VALUE (LIST). Also, if **burst** is true, then no output file parameter may be specified on the command line, unless a fixed number of input files can be used.

15.3.2 *Parse a command line provided by the program*

ok = DCL_PARSE(string, verb, burst, cldname, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**string**	**the command line to be parsed**
in - char	**verb**	the command verb used to invoke this program
in - logical	**burst**	this is for use with positional qualifiers, and should normally be false. Otherwise, see below.
in - long	**cldname**	this is the 'name' of the CLD module referred to by all the routines described here. It must be declared EXTERNAL in the main program.
in - logical	**report**	true if to allow DCL_PARSE to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_PARSE( TXTDSC, 'I2OSTF_2', .FALSE., ICLIP_CLD, .TRUE.  )
```

This routine is identical in its action to DCL_STARTUP, except step 1 of the list of actions is omitted. That is, DCL_PARSE does not call LIB$GET_FOREIGN to obtain a command line (however, note that if any parameters are missing, CLI$DCL_PARSE will still prompt the user for them).

Instead, it uses **string** as the line to be parsed. DCL_PARSE is therefore useful for parsing lines built by the program itself, or perhaps lines read from other sources than SYS$INPUT:

If **string** contains no significant characters, then CLI$_PARSE prompts for input, using the prompt specified in the CLD module. This can be useful in programs which must loop and obtain a new command line, rather than exiting.

15.3.3 *Retrieve a parsed command line*

ok = DCL_CML(cmlstr, cmlen, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
out - char	**cmlstr**	the command line, after parsing. This must be a 1024 character variable
out - long	**cmlen**	the length of the command line returned in **cmlstr**
in - logical	**report**	true if to allow DCL_CML to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_CML( CMLSTR, CMLEN, .TRUE.  )
```

The function works by calling CLI$GET_VALUE, with key '$LINE', to retrieve the command line as typed by the user.

15.3.4 *Check presence of a command qualifier*

ok = DCL_QUAL(qualifier, had_qual, present, negated, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**qualifier**	the name of the qualifier as defined in the CLD module - for example 'FEATURE_CODES' for /FEATURECODES
out - logical	**had_qual**	true if the qualifier us present, or present by default, and false if it is absent or negated (eg /NOLOG)
out - logical	**present**	true if the qualifier was detected within the decoding environment of a parameter.
out - logical	**negated**	true if the qualifier was detected in a negated form within the decoding environment of a parameter.
in - logical	**report**	true if to allow DCL_CML to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_QUAL( 'LOG', HAD_LOG, LOCPRES, LOCNEG, .TRUE.  )
```

The function works by:

1. Calling CLI$PRESENT to determine whether the qualifier (eg /LOG) is present

2. Setting HAD_LOG to true if the qualifier found or defaulted.

Note that **present** and **negated** will only be set if the CLD module definition for the qualifier contains the 'PLACEMENT=LOCAL' or 'PLACEMENT=POSITIONAL' conditionals. If qualifiers are to have local meaning then a call to DCL_FILE should be made before the call to DCL_QUAL to detect the presence and position

of the parameter (it is assumed that this will normally be a filename!) and so define the decoding environment of that parameter.

15.3.5 *Get integer arguments*

ok = DCL_INT(qualifier, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**qualifier**	the name of the qualifier as defined in the CLD module - for example 'FEATURE_CODES' for /FEATURECODES=
in - logical	**report**	true if to allow DCL_INT to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_INT( 'FEATURE_CODES', .TRUE.  )
```

The function works as follows:

1. CLI$GET_VALUE is called to determine whether any arguments are present for this qualifier

2. For each argument, it checks that the argument is a valid integer, or integer value range. An integer value range has the syntax **n:m**, where **n** is the start number and **m** the stop number (inclusive).

3. The argument is read as a character string using an internal read - this is done to facilitate the flexible range decoding mechanism employed in DCL_INT. Note that it must thus be specified with "TYPE=$QUOTED_STRING" in the CLD module.

4. Any value ranges are expanded.

5. The results are placed in /CLD/ IARRAY() and the number of integers read (or expanded from range specifications) is held in /CLD/ NUMINT.

6. DCL_INT continues to read and expand arguments in a qualifier argument list until it is exhausted.

DCL_INT expands ranges by reference to the following rules:

Storage exists for only MAX_LONG integers (currently 1024)

The values defining the range are included in the expansion.

The syntax :n is permissible, DCL_INT will assume that the range starts at zero and will expand upwards to n.

The syntax n: is not permissible, the MAX_LONG parameter is currently set to 1024, not infinity!

If negative values are specified then the most negative (!) must be specified first.

As an example our "typical call" above used on:

```
/FEATURE_CODES=(17,:3,11:13,91)
```

would yield:

```
NUMINT=9

IARRAY(1)=17  IARRAY(2)=0   IARRAY(3)=1
IARRAY(4)=2   IARRAY(5)=3   IARRAY(6)=11
IARRAY(7)=12  IARRAY(8)=13  IARRAY(9)=91
```

15.3.6 *Get real arguments*

ok = DCL_REAL(qualifier, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**qualifier**	the name of the qualifier as defined in the CLD module - for example 'TOLERANCE' for /TOLERANCE=
in - logical	**report**	true if to allow DCL_REAL to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_REAL( 'TOLERANCE', .TRUE.  )
```

The function works by:

1. Calling CLI$GET_VALUE to determine whether any argument(s) is present on the qualifier.

2. Taking each of the arguments in turn and checking that they contain valid numeric values (D floating, E floating and G floating numbers are NOT supported).

3. Performing an internal read on the arguments which are treated as character strings (and must be specified with "TYPE=$QUOTED_STRING" in the CLD module). This is because DCL does not currently support the real number type.

4. Continues to read and decode arguments in a qualifier argument list until the list is exhausted.

5. Places the results in /CLD/RARRAY() and the number of reals read in /CLD/NUMREA

As an example our "typical call" above used on:

```
/TOLERANCE=17.8
```

would yield:

```
NUMREA=1

RARRAY(1)=17.8
```

Up to MAX_REAL (currently 128) real numbers may be read from a single qualifier and placed in RARRAY()

15.3.7 *Get real*8 arguments*

ok = DCL_DBL(qualifier, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**qualifier**	the name of the qualifier as defined in the CLD module - for example 'OFFSET' for /OFFSET=
in - logical	**report**	true if to allow DCL_DBL to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_DBL( 'OFFSET', .TRUE.  )
```

The function works by:

1. Calling CLI$GET_VALUE to determine whether any argument(s) is present on the qualifier.

2. Taking each of the arguments in turn and checking that they contain valid numeric values (D floating, E floating and G floating numbers are NOT supported).

3. Performing an internal read on the arguments which are treated as character strings (and must be specified with "TYPE=$QUOTED_STRING" in the CLD module). This is because DCL does not currently support the real*8 number type.

4. Continues to read and decode arguments in a qualifier argument list until the list is exhausted.

5. Places the results in /CLD/DBLRAY() and the number of real*8 numbers read in /CLD/NUMDBL

As an example our "typical call" above used on:

```
/OFFSET=95634451217.84337
```

would yield:

```
NUMDBL=1

DBLRAY(1)=95634451217.84337
```

Up to MAX_REAL (currently 128) real*8 numbers may be read from a single qualifier and placed in DBLRAY()

15.3.8 *Get string arguments*

ok = DCL_STR(qualifier, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**qualifier**	the name of the qualifier as defined in the CLD module - for example 'LOOK_FOR' for /LOOK_FOR=
in - logical	**report**	true if to allow DCL_STR to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
RET = DCL_STR( 'LOOK_FOR', .TRUE.  )
```

The function works by:

1. Calling CLI$GET_VALUE to determine whether any argument(s) is present on the qualifier.

2. Performing an internal read on the arguments which are treated as character strings (and must be specified with "TYPE=$QUOTED_STRING" in the CLD module).

3. continuing to read arguments in a qualifier argument list until the list is exhausted.

4. Placing the results in /CLD/CARRAY() and the number of strings read in /CLD/NUMSTR. The length of each string is stored in STR_LEN().

As an example our "typical call" above used on:

```
/LOOK_FOR=(GOOD_PUBS,WINE,WOMEN,SONG)
```

would yield:

```
NUMSTR=4

CARRAY(1)(1:STR_LEN(1)) = "GOOD_PUBS"
CARRAY(2)(1:STR_LEN(2)) = "WINE"
CARRAY(3)(1:STR_LEN(3)) = "WOMEN"
CARRAY(4)(1:STR_LEN(4)) = "SONG"

STR_LEN(1) = 9
STR_LEN(2) = 4
STR_LEN(3) = 5
```

```
STR_LEN(4) = 4
```

Up to MAX_STR (currently 16) strings may be read from a single qualifier and placed in CARRAY()

15.3.9 *Get filename arguments*

ok = DCL_FILE(label, default, absent, list, report)

out - long	**ok**	SS$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails
in - char	**label**	the name of the the name of the qualifier or parameter as defined in the CLD module - for example 'OUTPUT' for /OUTPUT=file-spec, or perhaps 'INPUT_FILE_SPEC' for an input file parameter.
in - char	**default**	the default file specification against which the first file-spec detected by DCL_FILE will be parsed. A typical value for this argument might be 'LSL$IF:IFF.IFF;0'. If the default file is to be that with the latest version number, version ';0' should be specified eg 'LSL$IF:IFF.IFF;0' The default "rolls through" the list of files - that is, the first file is used as the default for the second, etc. This matches the normal VMS approach.
out - logical	**absent**	returned true if only a single file-spec was expected for parameter **label** (or qualifier **label**), but after searching was found to be absent
in - logical	**list**	controls how DCL_FILE parses the filenames - see below
in - logical	**report**	true if to allow DCL_CML to report its own errors via LSL_PUTMSG

Thus a typical call might be:

```
  RET = DCL_FILE ( 'OUTPUT', 'LSL780::LSL$IF:IFF.IFF;0',
&                  ABSENT, .TRUE., .TRUE.  )
```

The function works by:

1. Calling CLI$GET_VALUE to determine whether any argument(s) is present on the qualifier, the parameter is present (it had better be to have got this far!).

2. Performing an internal read on the arguments which are treated as character strings (and must be specified with "TYPE=$FILE" in the CLD module).

3. Parsing the string read against the default string supplied (if any) - default version numbers are allowed and should be supplied. If the default file is to be that with the latest version number, version ';0' should be specified eg 'LSL$IF:IFF.IFF;0'

4. continuing to read file-spec arguments in a qualifier argument list until the list is exhausted.

5. Placing the results in /CLD/FILARY() and the number of file-specs read in /CLD/NUMFIL. The length of each file-spec is stored in FIL_LEN().

As an example our "typical call" above used on:

```
/OUTPUT=DESTITUTION.MER
```

would yield:

```
NUMFIL=1

FILARY(1)(1:FIL_LEN(1)) = "LSL780::LSL$IF:DESTITUTION.MER;0"
FIL_LEN(1)=32
```

Up to MAX_FIL (currently 16) file-specs may be read from a single qualifier or parameter and placed in FILARY()

15.3.9.1 *Controlling parsing with **list*** -

If **list** is true then DCL_FILE will attempt to read and parse all the files in the parameter list or qualifier argument list specified by **label**.

If this action is required, then the CLD module entry for the parameter or qualifier pointed to by **label** should contain the 'VALUE(LIST)' conditional.

If **list** is specified as false then DCL_FILE will only attempt to read and parse one file-spec (or the first file-spec only in a parameter list or list of qualifier arguments).

list should be set .FALSE. if reading a file parameter for which positionally dependent qualifiers may be available.

15.4 *Creating a CLD file object module.*

In order to use the LSL_DCL routines your program must be linked with a CLD object module containing the definitions of the command structure of the program. To create this use the commands:

```
$ ADC <file>.TMP=<file>.CLD
$ SET COMMAND/OBJECT=<object-file>  <file>.TMP
$ PVV <object-file>
$ DELETE ADC.IDE;, <file>.TMP;
```

Where <file>.CLD is the name of the file containing your command language definitions for the program.

15.5 *User interface service routines*

This section contains service routines which are often needed during or immediately after interpretation of the DCL command line. They are designed to ensure that a common style of user interface is maintained between Laser-Scan programs.

Two qualifiers which are standard features of Laser-Scan utility program command lines are /OUTPUT and /LITES2. Both require a file to be opened and an explanatory header to be written to the file.

Two routines are provided to facilitate log and LCM file opening and header construction. They have two purposes:

1. to simplify and standardise the process of opening the files and then gathering and formatting standard header information.

2. to standardise the messages output by programs which are opening log and LCM files.

15.5.1 *Open and then write header into a /LITES2 LCM file*

ok = LCM_OPEN(lun, file-spec, range, log)

out - long	**ok**	LSL$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails. (All error reporting is **always** done by LCM_OPEN itself).
in - word	**lun**	the lun to be assigned to the LCM file when it is opened using FLWOPN
in - char	**file-spec**	the specification of the LCM file to be opened
in - real	**range(4)**	the coordinate range of the data to be written to the LCM file. Used subsequently to define the extent of the sectors produced by the LITES2 LCMORG utility.
in - logical	**log**	if .TRUE. causes LCM_OPEN to report the successful opening of the LCM file.

Thus a typical call might be:

```
RET = LCM_OPEN( 2, 'LSL$LITES2CMD:EXAMPLE.LCM;0', RANGE, LOG )
```

The function works as follows:

1. Calling FLWOPN to create and open the LCM file

2. Calling DCL_CML to get the whole command line string.

3. Calling SYS$GETJPIW to get the process information required for the header.

4. Selecting the newly created LCM file for write using FLWSEL.

5. Writing the values held in RANGE in LCMORG format to the LCM file.

6. Writing the header information to the LCM file using FLWRTF and FLWSTR

7. Chopping the command line string into <80 byte records at appropriate command element positions such as commas, solidi and spaces.

8. Writing the command line records to the LCM file using FLWRTF and FLWSTR.

As an example, our "typical call" above used in conjunction with the command line:

```
$ EXAMPLE JOES_FILE.IFF;9 JOEFILE/LAYER=12/LITES2=LSL$LITES2CMD:EG.LCM
```

will result in the creation of the LITES2 LCM file LSL$LITES2CMD:EG.LCM;0 which will typically contain a header of the form:

```
%POSITION     0.000     0.000
%POSITION 15014.115 15002.576
%ABANDON
%MESSAGE
%MESSAGE                L I T E S 2   C O M M A N D   F I L E
%MESSAGE
%MESSAGE                              created by
%MESSAGE
%MESSAGE ============================ E X A M P L E ============================
%MESSAGE
%MESSAGE EXAMPLE invoked by TIM using terminal TTE4: at 16-APR-1987 09:25:34.2
%MESSAGE
%MESSAGE Command line:
%MESSAGE
%MESSAGE $ EXAMPLE JOES__FILE.IFF;9 JOEFILE/LAYER=12/LITES2=LSL$LITES2CMD:EG.LCM
%MESSAGE
%MESSAGE ========================================================================
%MESSAGE
%ABANDON
%ABANDON
```

This header may subsequently be followed by any valid LITES2 commands. For a description of LITES2 commands available see the LITES2 Reference Manual.

Note that LCM_OPEN will leave FILEIO write selection directed to the LCM file. Use the FILEIO FLWSEL routine to select an alternative file for write.

Be warned that LCM_OPEN will report all errors itself. The user need only test that the return value is LSL_NORMAL.

15.5.2 *Open and then write header into a /OUTPUT log file*

ok = LOG_OPEN(lun, file-spec, log)

out - long	**ok**	LSL$_NORMAL if it succeeds, and an appropriate CLI$_ or LSL__ message if it fails. (All error reporting is **always** done by LOG_OPEN itself).
in - word	**lun**	the lun to be assigned to the LOG file when it is opened using FLWOPN
in - char	**file-spec**	the specification of the log file to be opened
in - logical	**log**	if .TRUE. causes LOG_OPEN to report the successful opening of the log file.

Thus a typical call might be:

```
RET = LOG_OPEN( 2, 'HERE:EXAMPLE.LIS;0', LOG )
```

The function works as follows:

1. Calling FLWOPN to create and open the log file
2. Calling DCL_CML to get the whole command line string.
3. Calling SYS$GETJPIW to get the process information required for the header.
4. Selecting the newly created LCM file for write using FLWSEL.
5. Writing the header information to the log file using FLWRTF and FLWSTR
6. Chopping the command line string into <80 byte records at appropriate command element positions such as commas, solidi and spaces.
7. Writing the command line records to the LOG file using FLWRTF and FLWSTR

As an example our "typical call" above used in conjunction with the command line:

```
$ EXAMPLE JOES_FILE.IFF;9 JOEFILE/LAYER=12/OUTPUT=HERE:EG.LIS
```

will result in the creation of the /OUTPUT LOG file HERE:EG.LIS;0 which will typically contain a header of the form:

```
============================== E X A M P L E ================================

EXAMPLE invoked by TIM using terminal LTA85: at 16-APR-1987 15:47:57.96

Command line:

$ EXAMPLE JOES__FILE.IFF;9 JOEFILE/LAYER=12/OUTPUT=HERE:EG.LIS

=============================================================================
```

Note that LOG_OPEN will leave FILEIO write selection directed to the log file. Use the FILEIO FLWSEL routine to select an alternative file for write.

Be warned that LOG_OPEN will report all errors itself. The user need only test that the return value is LSL_NORMAL.

CHAPTER 16

COMMAND DECODING

16.1 *Introduction*

Some programs expect the user to supply commands specifying the actions they require. This chapter documents how to define tables of commands, and how to read commands using such tables as syntax definitions.

The command reading routines provide the following advantages:

i) flexible argument decoding;

ii) fast table look up (binary chop), with no requirement for special table ordering (o(nlogn) sorting process if out of order);

iii) command tables (theoretically, at least) extensible, and (in practice) compilable at run time;

iv) abbreviated command names accepted; ambiguities diagnosed;

v) secondary commands available as an alternative to arguments of a primary command.

Static command tables are assembled using MACRO-32, using macros defined in the LaserScan standard macro library, LSL$LIBRARY:LSLMACLIB.MLB.

Dynamic command tables are generated by routines described in this chapter.

For an example of using the command decoding routines, see the sources of the IMP utility IPATCH.

16.2 *Defining Static Command Tables.*

The command table itself is introduced by a **$CMTAB** macro, and terminated by a **$CMEND** macro. Each command is defined by a **$CMD** macro. Thus a command table consists of the sequence **$CMTAB|[$CMD*]|$CMEND**. The first and last may not be omitted.

16.2.1 *The $CMTAB macro*

Takes the form:

$CMTAB name, [controls], [psect]

name is the name of the command table; should be (at most 6) alphanumeric characters, starting with an alpha. The name is used in generating the global symbol to be used to label the table - the symbol generated is **name**_CMD_TABLE. This global symbol name is then passed as an argument to RDCOMM when reading commands using the table.

controls defines controls to be applied to the table; a sequence of characters interpreted as:

- **F** permit user flags on commands in this table;
- **D** digits, and other characters, allowed in command names (note that this overrides **N** and **R** if they are also present);
- **N** permit 'numeric' commands in this table;
- **R** read 'numeric' commands, read against this table, as reals;
- **S** permit secondary commands to commands in this table;
- **U** upper case all command name strings before looking them up in this table; note that this is done in local workspace, so that the original string is not corrupted.
- **X** insist that all command table matches are 'exact' - ie no ambiguous matches are permitted with commands in the table. (This facility is used in language preprocessor and suchlike programs.)

psect defines the PSECT (COMMON block) into which the command table will be assembled. If argument **psect** is absent or blank, the command table is assembled into (concatenated) PSECT $$CMTAB.

16.2.2 *The $CMD macro*

Takes the form:

$CMD mnemonic(s), [controls], [user_flags]

mnemonic(s) define(s) the name(s) of the command. By default, each **mnemonic** may consist of alphabetic characters and underlines. If the **D** control was specified in the table header, each **mnemonic** may be any printing characters except tab and space - it is, however, recommended that only alphabetic characters, numbers, underlines and hyphens be used. Also note that angle brackets and semi-colons will not work, since the command tables are being written using MACRO.

If uppercasing was specified (with the **U** control) in the table header, then all mnemonics must be specified with upper case letters - any mnemonic containing lower case will not be recognised. If uppercasing was NOT specified in the table header, then upper and lower case versions of mnemonics are treated as distinct.

One mnemonic may be expressed on its own, as in
```
$CMD  FRED
```
but more than one must be enclosed in angle brackets, as in
```
$CMD  <JIM,HARRY>
```

controls defines controls and arguments for this command; a sequence of characters interpreted as:

- **.** (dot: commas are precluded by MACRO) comma is ignored immediately after any numeric argument
- **+** command has logical argument (+ or - before the command); only one logical argument per command
- **=** expect an '=' or a ':' character between the command name and its argument(s) (though there is no insistence that the character be there)
- **C** character argument - effectively a one-character string, though not enclosed in quotation marks; note that only one string or character argument per command is permitted
- **F** no command argument may be missing, or, if the command has a secondary command table, the secondary command may not be omitted (note that there's no complaint about an absent logical argument)
- **I** one integer argument; one **I** for each such (at most 7)
- **M** arguments are 'mnemonic', ie are to be decoded from a secondary command table named from the first name given for this command in its macro (eg FRED_CMD_TABLE and JIM_CMD_TABLE in the two examples of the $CMD macro given above); the **M** flag may be associated only with the **F** flag, and may only appear in tables for which the **S** flag was given in the $CMTAB macro
- **Q** expect an ine**Q**uality between the command name and the argument to the command
- **R** one real argument; one **R** for each such (at most 7)
- **S** string argument (enclosed in quotation marks); note that only one string or character argument per command is permitted

user_flags define user flags associated with this command.

User flags provide information about what sort of command this is, and suchlike. They are returned to the user program when the command is found by the command routines. Each character of the **user_flags** argument defines a bit of the user flags longword returned; the characters are interpreted as bit numbers in a 32-bit extension of standard hexadecimal format; characters 0-9 mean bits 0-9 (value H'00000001 to H'00000200), and characters A-V mean bits 10-31 (value H'00000400 to H'80000000).

The **$CMD** macro does check that its invocation doesn't specify anything too silly (like invalid control characters, repeats of flag controls such as **+**, or too many **I** or **R** specifications). It doesn't, however, check for 'indirect' sillies like **.** with only one number argument, or **=** or **Q** with no number argument.

16.2.2.1 *Arguments and secondary commands* -

Arguments and secondary commands are decoded, at the same time as the command itself, by routine RDCOMM; the result of this decoding always appears in /CMDCOM/ (qvi).

Only the terminal secondary command of a primary command may have arguments. There is no reason that a secondary command should not specify itself as having a secondary command; however, the decoding routines only record the command number of the terminal secondary command of a chain thus specified.

16.2.2.2 *A common mistake* -

A common mistake is to get the case of the letters in arguments to the **$CMD** macro wrong. Even if the **U** control was set in the **$CMTAB** macro, the system is sensitive (one way or another) to the case of every letter entered. Incorrect case will manifest itself either as unexpected failure to recognise commands, or as diagnostics from the $CMD macro itself when assembling the table.

16.2.3 *The $CMEND macro*

Takes the form:

$CMEND name

name is the same **name** as appeared in the corresponding **$CMTAB** macro.

16.2.3.1 *Another common mistake* -

Another common mistake is to omit, or to mis-spell, the **name** argument to the **$CMEND** macro. The effect of this is obscure names (of the form '$$<proper_name_of_the_table>_CMCT') appearing unset when the file is linked.

16.2.4 *Command Numbers.*

Command numbers are defined by the order of commands in the table. Commands are allocated numbers 1,2,...; each **$CMD** macro increments a command number which is set 0 by the **$CMTAB** macro. Thus the aliases of a single command (which are all specified by the same **$CMD** macro) all get allocated the same command number.

16.3 *RDCOMM - reading commands*

RDCOMM is the routine that extracts a command (the next command) from the current input buffer (normally the /TXTC/ common block), looks it up in a table (the argument to RDCOMM), gets the command's arguments, and produces any diagnostics that are called for (if the command ain't right!).

ret = RDCOMM(table)

out - long	**ret**	the result of reading the command - zero or the command number
in - external	**table**	the command table - see below

If the command table is static - that is, declared in MACRO and compiled - then it is declared by the macros as a global symbol. The Fortran routine calling RDCOMM must then declare it EXTERNAL before referencing it - for instance

```
INTEGER*4  ret
INTEGER*4  RDCOMM
 :::
EXTERNAL   FRED_CMD_TABLE
 :::
ret = RDCOMM( FRED_CMD_TABLE )
```

The values returned in **ret** are:

0	if no command is found, or if the command was bad in some way (in which case a diagnostic message will have been produced, unless NOMESS was set), or
-1	if numeric command has been found; an integer numeric command will have appeared in INTARG(1), a real numeric command in REALAR(1) (both in /CMDCOM/), or
command number	from the table, if the command was successfully read, and all its arguments have been neatly stuffed away in the appropriate slots in /CMDCOM/.

By default, commands are read as strings which are terminated by any non-alphabetic, non-underline character. If the **D** control is specified for the command table, then commands are read as strings which are terminated by a space or tab. This means that a command such as LITES2 becomes legal, but it also means that ZOOM5 must be typed as ZOOM 5.

16.3.1 *RDCOMM - Error Handling*

RDCOMM normally generates error messages for itself, by use of the LSLPUTMSG routine. Sometimes, it is preferable to generate messages in the calling program instead; to enable users to do this, a logical variable NOMESS (in /CMDCOM/) is supplied. If NOMESS is true then RDCOMM suppresses any error messages that it might otherwise be inclined to produce; NOMESS false (the normal state) permits RDCOMM to produce the messages that are so close to its heart.

The variable ERRNUM (in /EXCEPTION/) signals to the calling program the state of errors encountered while processing a command. Values that may be placed in ERRNUM by RDCOMM are fully documented in the chapter on errors and exceptions.

If RDCOMM fails (returning **ret** as 0), the contents of /CMDCOM/ are in general unpredictable (with some specific exceptions - see the description of /EXCEPTION/ in the relevant chapter, and /CMDCOM/ below).

16.3.1.1 *Error reporting - LSL_CMDERR -*

ok = LSL_CMDERR()

out - long **ok** error return

LSL_CMDERR can be used to output a CMDLIB error message when NOMESS is true. It looks up the error in /EXCEPTION/, gathers the appropriate arguments from /CMDCOM/ and /INEQUAL/, and uses LSL_PUTMSG to output the error message. The values of **ok** returned are identical to those from LSL_PUTMSG, that is:

LSL__NORMAL - the message text was successfully found, EXPANDed and output
LSL__BUFFEROVF - the message text was too long to fit into the internal buffers. A warning message (SS$_BUFFEROVF) is output, and then the message is truncated and the truncated version is output
LSL__MSGNOTFND - the message text cannot be found. The function does not output anything.

16.4 *RDINEQ - reading an inequality*

RDINEQ attempts to read an inequality name from the current input buffer (normally the /TXTC/ common block).

failed = RDINEQ(ineq)

out - logical **failed** false if an inequality was read, else true
out - byte **ineq** inequality number (see below)

RDINEQ reads an inequality in the same way that RDCOMM does. If the routine returns true (failure) then the decoding pointer is restored to its position when the routine was called. If (as is often the case) the lack of any inequality is to be treated as '=', then the function return may be ignored, since **ineq** is set to 0 in the case of failure. RDINEQ uses RDCOMM to read the inequality names, and may therefore produce RDCOMM error messages unless NOMESS (in common /CMDCOM/) is set true. The possible values returned in **ineq** (defined as parameters in LSL$CMNLSL:INEQ.PAR) are:

```
INEQ_EQL (0) - '='  or '.EQ[L].' or no inequality at all
INEQ_GTR (1) - '>'  or '.GT[R].'
INEQ_GEQ (2) - '>=' or '.GE[Q].'
INEQ_LSS (3) - '<'  or '.LT.'  or '.LSS.'
INEQ_LEQ (4) - '<=' or '.LE[Q].'
INEQ_NEQ (5) - '<>' or '.NE[Q].'
```

16.5 *Common blocks*

16.5.1 *The CMDCOM common block*

The main RDCOMM common block is defined in the file LSL$CMNLSL:CMDCOM.CMN for FORTRAN programs.

For MACRO programs, a macro $CMDCOM is defined in the file LSL$CMNLSL:CMDCOM.MAR. The common block may be defined by assembling the relevant MACRO sources with this file, and invoking the macro $CMDCOM within the program. The user should refer to the $CMDCOM source code for the amended names defined in the macro.

NOTE

All *logical* variables in /CMDCOM/ are declared as LOGICAL*1 or BYTE - this should never affect normal use of the variables.

The (Fortran version of the) common block contains the following (presented in alphabetical order):

out - logical **ARGMSG**
true if an argument (other than a logical one) was missing from the command (note that this condition is an error if the command was flagged **F**)

out - long **CMDACT**
returns the total number of arguments found
(ie CMDICT+CMDRCT, or CMDICT+CMDRCT+1 if a string argument is found)

out - long **CMDAST(2)**
is a 'descriptor' for the 'other' string if command name ambiguity is found in either a primary or a secondary command table.

out - long **CMDFLG**
returns command flags from the command read; note that, in the case of a primary followed by a secondary command, the flags of the primary command are not retained, since they consist, perforce, of the **M** flag only

out - long **CMDFST(2)**
is a 'descriptor' for the name of the primary command as it was to be found in the command table (not what the user typed)

out - long **CMDICT**
returns number of integer arguments found

out - logical **CMDNEG**
returns true if the command had a logical argument '-' ('no logical argument' and 'logical argument '+'' are not distinguished, and return CMDNEG false)

out - word **CMDNLE**
word length part of CMDNST 'descriptor' - equivalenced onto CMDNST(1)

out - long **CMDNST(2)**
returns 'descriptor' for the command name (as actually typed by the user)

out - long **CMDNUM**
returns a copy of the primary command number - which is what RDCOMM's **ret** is made up of. CMDNUM is correctly set if RDCOMM returns an error as a result of an error in reading a secondary command; if an error occurs during the processing of the primary command, but after its recognition, CMDNUM returns the negative of the primary command number.

out - long **CMDRCT**
returns number of real arguments found

out - long **CMSFST(2)**
as CMDFST, but for secondary command name

in - long **DEFBASE**
set by the caller of RDCOMM; the default base to which integer arguments are to be read. Default=0 => 'unset', ie decimal base that can be overridden by the user of the program specifying ^<base letter>

out - logical **HADDOT**
set to true if there was(were) real argument(s), and at least one had a '.', '/', or 'E' in it, thus distinguishing them from integers

out - byte **INEQUAL**
returns the inequality encountered in reading the command, if any. Possible values (defined as parameters in LSL$CMNLSL:INEQ.PAR) are:

```
INEQ_EQL (0) - '='  or '.EQ[L].' or no inequality at all
INEQ_GTR (1) - '>'  or '.GT[R].'
INEQ_GEQ (2) - '>=' or '.GE[Q].'
INEQ_LSS (3) - '<'  or '.LT.'  or '.LSS.'
INEQ_LEQ (4) - '<=' or '.LE[Q].'
INEQ_NEQ (5) - '<>' or '.NE[Q].'
```

out - long **INTARG(1...CMDICT)**
returns the CMDICT longword (INTEGER*4) arguments, in order

in - logical **NOMESS**
set by caller of RDCOMM; if it is true then RDCOMM does not output its own error messages. The default value of false allows them to be output

out - logical **NOUFLG**
returns true if user flags were found in the table the last primary command was looked up in

out - logical **NSUFLG**
as NOUFLG, but for the secondary table

out - real **REALAR(1...CMDRCT)**
returns the CMDRCT real (REAL*4) arguments, in order

out - long **SECMDN**
as CMDNUM, but for secondary command (note that it's always CMDNUM that gets returned as the 'result' of RDCOMM)

out - word **SECNLE**
word (INTEGER*2) length part of SECNST 'descriptor' - equivalenced on SECNST(1)

out - long **SECNST(2)**
as CMDNST, for what the user typed for the secondary command

out - long **SECTAB**
pointer to the table in which the secondary command was looked up; may be used (for example) to print the command table by a call such as:

```
CALL   CMDPRT ( %VAL(SECTAB) )
```

out - word **STARLE**
word (INTEGER*2) length part of STARST 'descriptor' - equivalenced on STARST(1)

out - long **STARST(2)**
returns 'descriptor' for any string argument

out - long **SUFLAG**
as UFLAG, but for secondary command

out - long **TABFLG**
returns table flags from the table argument to RDCOMM

out - long **UFLAG**
returns user flag bits of primary command looked up

out - byte **UNXCHR**
returns the character that wasn't expected following an LSL__UNEXPCH 'Unexpected character' error

16.5.2 *The INEQUAL common block*

The INEQUAL common block is defined in the file LSL$CMNLSL:INEQUAL.CMN, and is used to store data about the current inequality name. It is not expected that a user will normally need to refer to it.

private - parameter **INEQ_BUF_LEN**
This defines the maximum length which a string being considered as an inequality name may have. It is currently set at 10.

private - long **INEQ_NAME(2)**
This is a 'fake' descriptor for the inequality name. It is initialised by LSL_INIT.

private - byte **INEQ_BUF(INEQ_BUF_LEN)**
This is the buffer part of the 'fake' descriptor.

16.6 *Dynamic Command Table Routines.*

The library provides a mechanism for command tables to be defined dynamically during the execution of an image. Routine INITAB sets up a common block to point to a command table being developed. Routine ADDNAM adds a name to the table. Routine REMCMD removes a command from the table. The names added by ADDNAM are stored in dynamically obtained memory, so two arguments to INITAB which used to supply the space for this are now unused. The maximum length command name which may be added is 80 characters.

Note that there are no routines yet for dynamic definition of secondary command tables.

16.6.1 *Defining the table*

The table is defined by:

call INITAB(table, tabsiz, bytarr, bytsiz,
[argarr, argsiz], [uflarr], [digits])

in - long	**tabsiz**	the size of the **table** array
out - long	**table**	an array **tabsiz** longwords long, to hold the table of names
in - long	**bytsiz**	unused, pass any integer. This argument used to be the size of the **bytarr** array
out - byte	**bytarr**	unused, pass anything. This argument used to be an array **bytsiz** bytes long, to hold the bytes of names
in - long	**argsiz**	the size of the **argarr** and **uflarr** arrays
out - word	**argarr**	an array **argsiz** words long, to hold the table of argument descriptions
out - long	**uflarr**	an array **argsiz** longwords long, to hold the table of command user flags
in - logical	**digits**	allow digits (and other characters) in command mnemonics

INITAB initialises dynamic table generation. It must be called before any call to ADDNAM.

Commands in the table are assumed not to have arguments if arguments **argarr** and **argsiz** are missing. User flags are marked 'suppressed' in the table if argument **uflarr** is missing.

If **digits** is specified as .TRUE. the effect is as if the **D** control were specified for a static command table.

Note that command table definition may not be nested. A call to INITAB starts definition of a new table, and all subsequent entries made by ADDNAM go into the new table. If another table is to be defined, and there is a need later to return to the original table, routines SAVTAB and SELTAB (qvi) should be used.

16.6.2 *Saving and restoring the state of definition of a table*

Once a table's definition has been started, its current state may be preserved by:

call SAVTAB(savbuf)

out - long	**savbuf**	a 12 longword buffer to hold the saved information

When a table's state has been saved by SAVTAB, definition of the table may be resumed by:

call SELTAB(savbuf)

in - long	**savbuf**	must be a buffer loaded by a call to SAVTAB - it is thus 12 longwords long

16.6.3 *Entering names in the table*

Once the table has been defined by a call to INITAB, names may be entered in it by:

ret = ADDNAM(name, [length], [cmdnum], [argspc], [uflag])

out - long	**ret**	the function result - see below
in - char	**name**	the name to be added to the table
in - long	**length**	defines the relevant substring of **name** to be added
in - long	**cmdnum**	defines the command number to be given to **name** in the table
in - word	**argspc**	defines the arguments to **name** (this argument may be generated by the function ARGSPC)
in - long	**uflag**	contains the user flags for **name**

Adds the name **name** to the currently-defined table. The command is given number **cmdnum** (if that argument is present), or the lowest command number not yet allocated in the table (otherwise).

Caveats:

1. the routine only adds the alphabetic and underline ('_') characters of **name** to the table, so it's quite capable of adding a name of zero length!

2. it makes no check on duplication of names within the table.

Function result is:

> 0	'index' in the table where name has been put - this is the value which will be returned by RDCOMM if the name is looked up in the table. Note that this is the same value as argument <cmdnum>, if that argument was present.
= 0	no room in the table
=-1	cannot obtain memory to store command name (used to be too many bytes for residue of space in byte array)

=-2 command number exceeds <argsiz> argument of INITAB (if that was present)
=-3 <argspc> argument was given, but no <argarr> argument was given to INITAB
=-4 <uflag> argument was given, but no <uflarr> argument was given to INITAB

16.6.4 *Removing commands from the table*

Commands may be removed from a dynamic command table by:

CALL REMCMD(cmdnum)

in - long	**cmdnum**	defines the command number to be removed from the table

All commands with command number **cmdnum** are removed from the current dynamic command table. If no commands have the correct number, then nothing is done. The command numbers of the remaining commands in the table are unchanged.

16.6.5 *Evaluating argument specifications*

Argument specification words are tricky objects to generate. In recognition of this fact, a near-impossibly complex routine is provided to generate the beasts for the the user of ADDNAM. This routine is:

argspec = ARGSPC(numint, numrea, logarg, strarg, chrarg,
argful, prmequ, prmieq, prmcom)

out - word	**argspec**	the resultant argument specification
in - long	**numint**	($CMD flag **I**) the number of integer arguments (0 to 7)
in - long	**numrea**	($CMD flag **R**) the number of real arguments (0 to 7)
in - logical	**logarg**	($CMD flag **+**) true if a logical argument is to be recognised
in - logical	**strarg**	($CMD flag **S**) true if a string argument is required
in - logical	**chrarg**	($CMD flag **C**) true if a character argument is required
in - logical	**argful**	($CMD flag **F**) true if all arguments (other than a logical one) must be present
in - logical	**prmequ**	($CMD flag **=**) true if to permit '=' or ':' after a command, before any arguments
in - logical	**prmieq**	($CMD flag **Q**) true if to permit inequalities
in - logical	**prmcom**	($CMD flag **.**) true if to permit a comma between numeric arguments

The function result is typically to be used as the **argspc** argument to ADDNAM (indeed it's hard to imagine what else it could be used for!).

16.7 *Additional Command Table Routines.*

The routines documented here enable the user to perform some additional manipulation of command tables.

16.7.1 *Command Table Print*

A command table, once handed to the command-reading routines, has a tendency to get sorted, juggled, and otherwise 'improved'. CMDPRT provides the 'ordinary user' with a means of listing the command names in a command table. It ensures, as does RDCOMM, that the command table is in alphabetic order before it starts.

CALL CMDPRT(table)

in -	**table**	the command table - which for static command tables will usually be declared external **name**_CMD_TABLE in the calling routine, but for dynamic command tables is the table defined by the subroutine INITAB.

CMDPRT prints out the commands in **table** using TTWSTR.

16.7.2 *Accessing Command Tables by Command Number*

CALL FIND_CMDNAME(table, cmdnum, cmdnam, cmdlen, context)

in -	**table**	the command table - which for static command tables will usually be declared external **name**_CMD_TABLE in the calling routine, but for dynamic command tables is the table defined by the subroutine INITAB.
in - long	**cmdnum**	the number of the command to be found
out - char	**cmdnam**	the command that was found
out - long	**cmdlen**	the number of characters in the command name
i/o - long	**context**	see below

It is sometimes necessary to get the name of a command, from a command table, when the command number is known. This process is complicated by the fact that different commands may have the same command number (when they are synonyms). This subroutine therefore uses the **context** argument to save the current context of the search whenever an occurrence of the command number has been found. The subroutine should first be called with **context** having the value 0. If a command is found, **context** will be returned as a non-zero value. By calling the subroutine repeatedly with this new value of **context** any subsequent occurences of the command number in the table will be found will be found. If no command is found, **context** is returned as 0.

16.7.3 *Command Table Sort*

Before use, or after adding new commands, a command table must be sorted into alphabetical order. This is performed automatically by the LSLLIB routines, but if the user wishes to write their own code to manipulate a command table (look at the LSLLIB source code to determine the command table structure), then

LSL_SORTAB may be used to sort the table.

CALL LSL_SORTAB(table)

in	-	**table**	the command table - which for static command tables will usually be declared external **name**_CMD_TABLE in the calling routine, but for dynamic command tables is the table defined by the subroutine INITAB.

LSL_SORTAB sorts the commands in **table** into alphabetical order. A flag is set in the table to indicate that it has been sorted, so that future calls to LSL_SORTAB just return immediately.

CHAPTER 17

IFF FILE OPENING ROUTINES

17.1 *Introduction*

The following routines are provided as a standard route for opening IFF files. They have two purposes:

1. to simplify the arguments required for opening an IFF file - the IFFLIB routines IFFOPN and IFFOPI are over-complex to use

2. to standardise the messages output by programs which are opening IFF files - all of the IMP utilities now use these routines to open their IFF files

It is assumed that the routines will normally be allowed to report their own errors (with **report** set to true). However, if the calling program intends to use the return code from these routines, care should be taken to supply the arguments required by several of the error codes. The IFF filename is required by the messages LSL_IFFOPEN, LSL_IFFCREATE, LSL_IFFMODIFY and LSL_IFFPARSE. The message LSL_IFFSIZE requires the **size** argument to IFFCREATE as its argument.

17.2 *Opening a file for read*

ok = IFFOPEN(lun, file, [fid], [report], [revision])

out - long	**ok**	returns LSL__NORMAL if the open succeeds, otherwise see below
in - word	**lun**	the IFFLIB unit to open the file on
in - char	**file**	the name of the file to open
in - long	**fid(7)**	the file ID of the file to open
in - logical	**report**	true if to report a successful open
in - long	**revision**	the input revision level for this file

IFFOPEN opens an IFF file for read (only).

If the **fid** argument is absent, then a call to IFFOPN is generated.
file is the name of the file to open, defaulting to "LSL$IF:IFF.IFF". If **file** does not have a version number, then the most recent version of the file is opened. The default applied to the filename is "LSL$IF:IFF.IFF"

If the **fid** argument is present, then a call to IFFOPI is generated.
fid is the file ID obtained by a call of IFFID when the file was previously opened, and **file** should be the full filename as returned by a call of IFFINQ at

the same time.

When the file has been succesfully opened, IFFRLA is called to switch read-look-ahead on. If **report** is present and TRUE, then a message of the form

LSLLIB-I-IFFOPENED, **file** opened for read

is output. If **report** is absent, it defaults to FALSE, and no message is output.

If **revision** is specified, then IFFIRV will be called after the file has been opened, to set the input revision level. If **revision** is not specified, then the input revision level defaults to 0.

The values returned in **ok** are listed below. Note that if an error occurs, the routine will report it itself.

LSL__NORMAL - success - the IFF file was opened succesfully
LSL__IFFERR - error - an IFF error occurred whilst trying to open the file. The function will output the error messages

LSLLIB-E-IFFOPEN, IFF error opening file "**file**"
<appropriate IFFLIB error messages>

LSL__IFFPARSE - error - an error occurred whilst parsing the IFF file, to see if it had a version number. The function will output the error messages

LSLLIB-E-IFFPARSE, error parsing IFF filename "**file**"
<appropriate LSLLIB error message>

17.3 *Creating a new file*

ok = IFFCREATE(lun, file, [history], [size], [report], [revision])

out - long	**ok**	returns LSL__NORMAL if the open succeeds, otherwise see below
in - word	**lun**	the IFFLIB unit to open the file on
in - char	**file**	the name of the file to open
in - char	**history**	the annotation for the history record
in - long	**size**	the initial size of the file, in words
in - logical	**report**	true if to report a successful open
in - long	**revision**	the output revision level for this file

IFFCREATE creates a new IFF file, and leaves it selected for write.

file is the name of the file to open, defaulting to "LSL$IF:IFF.IFF". Note that **file** may not contain a version number (or a ";" at the end).

If **history** is present, then it may be up to 12 characters to be inserted into the descriptive part of the first history record (in the HI entry), when the file is closed by IFFCLO. If **history** is absent, then it defaults to 'Create'. If the file as closed does not contain an HI entry, then this argument has no effect.

If **size** is present, then it is the required initial size of the IFF file in words. If not present, then this defaults to 25,600 (ie 100 blocks of 256 words each). Note that the file is truncated when closed by IFFCLO, anyway.

If **revision** is present, then IFFORV will be called before the IFF file is opened, to set the output revision level. If **revision** is not specified, then the value stored in the logical name LSL$IFF_OUTPUT_REVISION will be used. If the logical name does not exist, then an output revision level of zero is used.

When the file is created, then if **report** is present and TRUE, a message of the form

LSLLIB-I-IFFOPENED, **file** opened for write

is output. If **report** is absent, it defaults to FALSE, and no message is output.

The values returned in **ok** are listed below. Note that if an error occurs, the routine will report it itself.

LSL__NORMAL - success - the IFF file was created succesfully
LSL__IFFERR - error - an IFF error occurred whilst trying to open the file. The function will output the error messages

LSLLIB-E-IFFCREATE, IFF error creating file "**file**"
<appropriate IFFLIB error messages>

LSL__IFFPARSE - error - an error occurred whilst parsing the IFF file, to see if it had a version number. The function will output the error messages

LSLLIB-E-IFFPARSE, error parsing IFF filename "**file**"
<appropriate LSLLIB error message>

LSL__IFFVERNUM - error - **file** contained a version number (or ";" character), and IFFOPN does not allow a version number on the filename it is asked to create. The function will output the error message

LSLLIB-E-IFFVERNUM, version number not allowed when creating new IFF file **file**\

LSL__IFFSIZE - error - the **size** argument was zero or negative. IFFOPN would interpret a value of zero as meaning open for readonly, and a negative argument is not sensible. The function will output the error message

LSLLIB-E-IFFSIZE, IFFCREATE - initial size **size** should be greater than zero

17.4 *Updating a file*

ok = IFFMODIFY(lun, file, [history], [fid], [report])

out - long **ok** returns LSL__NORMAL if the open succeeds, otherwise see below
in - word **lun** the IFFLIB unit to open the file on

```
in  - char    file       the name of the file to open
in  - char    history    the annotation for the history record
 in - long    fid(7)     the file ID of the file to open
 in - logical report     true if to report a successful open
 in - long    revision   the input revision level for this file
```

IFFMODIFY opens an existing IFF file, and leaves it selected for write.

If the **fid** argument is absent, then a call to IFFOPN is generated. In that case, the arguments are interpreted as follows:

file is the name of the file to open, defaulting to "LSL$IF:IFF.IFF". If **file** does not have a version number, then the most recent version of the file is opened.

If **history** is present, then it may be up to 12 characters to be inserted into the descriptive part of the next history record (in the HI entry), when the file is closed by IFFCLO. If **history** is absent, then it defaults to 'Update'. If the file as closed does not contain an HI entry, then this argument has no effect.

If the **fid** argument is present, then a call to IFFOPI is generated. In that case, the arguments are interpreted as follows:

fid is the file ID obtained by a call of IFFID when the file was previously opened, and **file** should be the full filename as returned by a call of IFFINQ at the same time.

If the **history** argument is present, then a new history record will be generated in the HI entry when IFFCLO is called, and **history** will be the annotation placed into that record. If the **history** argument is absent, then the old history record will be updated - the annotation will not be changed.

When the file is opened then if **report** is present and TRUE, a message of the form

LSLLIB-I-IFFOPENED, **file** opened for update

is output. If **report** is absent, it defaults to FALSE, and no message is output.

If **revision** is specified, then IFFIRV will be called after the file has been opened, to set the input revision level. If **revision** is not specified, then the input revision level defaults to 0.

The values returned in **ok** are listed below. Note that if an error occurs, the routine will report it itself.

```
LSL__NORMAL     - success - the IFF file was opened succesfully
LSL__IFFERR     - error - an IFF error occurred whilst trying to open the file.
                  The function will output the error messages
```

LSLLIB-E-IFFMODIFY, IFF error opening file "**file**" for update
<appropriate IFFLIB error messages>

LSL__IFFPARSE - error - an error occurred whilst parsing the IFF file, to see if it had a version number. The function will output the error messages

LSLLIB-E-IFFPARSE, error parsing IFF filename "**file**"
<appropriate LSLLIB error message>

CHAPTER 18

MAPPED SECTION FILES

18.1 *Introduction*

The mapped section routines allow a program to map a file into its virtual memory space. This means that the file can be accessed as if it were an array, rather than by conventional record or block access methods.

Note that the terms "page" and "block" both refer to units of 512 bytes - the first is normally used for virtual memory, and the second for disk based data.

The current routines support up to 9 sections being mapped (not necessarily all from different files). Unit numbers in the range 0 to 8 are used to identify which section is relevant to a routine, and an internal table is used to relate that number to the actual channel used. Negative unit numbers will be treated as 0.

For examples of the use of these routines, see the sources of DTILIB.

18.2 *Opening and mapping a file*

ok = VIO$OPEN_SEC(file, pagcnt, write, create,
array, bytlen, [unit], [cluster], [vbn])

out	long	**ok**	returns SS$_NORMAL if the routine works, otherwise an appropriate system error - see below
in	char	**file**	the name of the file to be mapped
in	long	**pagcnt**	the number of pages we want, ignored if the file already exists and **vbn** is not specified
in	logical	**write**	true if write access is allowed to the file
in	logical	**create**	true if a new file is to be created - note that this implies write access
out	long	**array**	the address in program (P0) space of the "array" that the file has been mapped to
out	long	**bytlen**	the number of bytes that were mapped
in	long	**unit**	the unit number, between 0 and 8 - defaults to 0
in	long	**cluster**	page fault cluster size - number of pages to be brought in on a page fault - defaults to 0
in	long	**vbn**	virtual block number of the first block to be mapped - defaults to 0

VIO$OPEN_SEC opens the given file, and maps it into memory, returning the address of the mapped part of the file in **array**. The user may then access the file by passing its address to a routine - ie as %VAL(**array**)

There are three ways of using VIO$OPEN_SEC

- If **create** is true, then a new file is created with **pagcnt** blocks. The file is created "contiguous best try" - that is, contiguous if possible. The file is then mapped demand-zero - all of its pages are set to zero as they are first read into memory. Note that a file created by VIO$OPEN_SEC will be mapped as writable, regardless of the value of **write**.

- If **create** is false and the **vbn** argument is specified, then the first **pagcnt** blocks starting at block **vbn** will be mapped. This allows a program to map part of a file into memory.

- If **create** is false but the **vbn** argument is not specified, then the whole file will be mapped.

The file is mapped into the first available space in the P0 (program) region, to a temporary, private section, with no name. Thus no other process may access this mapped section.

The routine itself may fail with any of the following values of **ok**:

SS$_INSFARG - if any of the compulsory arguments are missing.

SS$_IVCHAN - if the unit chosen is already in use - another file has been mapped on this unit. A unit is detected as being in use if a previous call of VIO$OPEN_SECTION has set the internal record of the files mapped address range (by a call of $CRMPSC).

SS$_BADPARAM - if **create** is true and the **vbn** argument is present.

A full list of the system values which may be returned in **ok** is not given here, but may be obtained by looking up the following system services:

$FAB_STORE - used to store file characteristics in the file's FAB (file access block)
$CREATE - used to create the file if **create** is true
$OPEN - used to open the file if **create** is false. If the file does not exist, then **ok** will be set to RMS$_FNF (file not found)
$CRMPSC - used to map the file to a global section

Note that if an error occurs in creating/mapping the section (ie in $CRMPSC), then the channel associated with that unit will be deassigned before the routine returns.

18.3 *Extending a mapped file*

ok = VIO$EXTEND_SEC(pages, array, bytcnt, [unit])

out	long	**ok**	returns SS$_NORMAL if the routine works, otherwise an appropriate system error - see below
in	long	**pages**	how many pages we want to add to the file
out	long	**array**	the new address in program (P0) space of the mapped file. Note that the file is remapped, so that the address will change
out	long	**bytlen**	the number of bytes mapped in the (extended) file
in	long	**unit**	the unit number, between 0 and 8 - defaults to 0

VIO$EXTEND_SEC extends an already open mapped section file. Note that:

1. The file is unmapped, extended, and then remapped. Thus the address of the mapped section will change.

2. The requested extension is rounded up to the next multiple of the cluster size (#1), and this number of blocks is added to the current extent. This means that the file can end with up to

   <cluster_size-1> * <number of extends done> blocks

   of 'wasted' space at the end.

Apart from SS$NORMAL, the following values of **ok** can be returned by the routine itself:

SS$_INSFARG - if any of the compulsory arguments are missing

SS$_IVCHAN - if the unit chosen is not in use - no file has been mapped on this unit.

SS$BADPARAM - if the number of pages is invalid - less than 1 block was requested.

Other system values which may be returned in **ok** may be obtained by looking up the following system services:

$DELTVA - used to delete the global section containing the mapped file
$QIOW - used to access and modify the file attributes
$CRMPSC - used to map the extended file to a global section

Note that as in VIO$OPENSEC,if an error occurs in creating/mapping the section (ie in $CRMPSC), then the channel associated with that unit will be deassigned before the routine returns.

#1 - the cluster size for a disk is the number of blocks that will be allocated when any number between 1 and the cluster size is requested - that is it is the 'granularity' of block allocation. Thus, if the cluster size is 3, then files will have allocated block sizes of 3, 6, 9, etc

18.4 *Updating a mapped file*

ok = VIO$UPDATE_SEC([unit])

out	- long	**ok**	returns SS$_NORMAL if the routine works, otherwise an appropriate system error - see below
in	- long	**unit**	the unit number, between 0 and 8 - defaults to 0

VIO$UPDATE_SEC updates the mapped section to disk - it uses SYS$UPDSEC to force a write of the section.

Normally, a section is written back to disk when a page fault requires it, when the section is deleted (so VIO$CLOSE_SEC and VIO$UPDATE_SEC will both update it), or when the process creating the section exits. Sometimes it is necessary to update the section more frequently, for instance to ensure that information is not lost if the system crashes.

If the routine does not work, the only error it sets itself is:

SS$_IVCHAN - if the unit chosen is not in use - no file has been mapped on this unit.

Other system values which may be returned in **ok** may be obtained by looking up the following system service:

$UPDSECW - used to update the section

18.5 *Closing a mapped file*

ok = VIO$CLOSE_SEC([unit])

out	- long	**ok**	returns SS$_NORMAL if the routine works, otherwise an appropriate system error - see below
in	- long	**unit**	the unit number, between 0 and 8 - defaults to 0

VIO$CLOSE_SEC closes the section down by deleting the virtual addresses to which it is mapped, and then deassigning the channel associated with the file. The addresses are then deleted from the internal record associated with that unit number.

If the routine does not work, the only error it sets itself is:

SS$_IVCHAN - if the unit chosen is not in use - no file has been mapped on this unit.

Other system values which may be returned in **ok** may be obtained by looking up the following system services:

$DELTVA - used to delete the global section containing the mapped file
$DASSGN - used to deassign the i/o channel associated with the file

Note that if an error occurs while trying to delete the global section, the i/o channel will not be deassigned.

CHAPTER 19

BASIC MAGNETIC TAPE I/O ROUTINES

19.1 *Introduction*

This chapter documents the general purpose magtape I/O routines.

19.2 *Magtape common blocks*

Two common blocks are provided, one for input and one for output. These are MTIVCM.CMN, and MTOVCM.CMN, both in LSL$CMNLSL:

19.2.1 *Input common block*

The input common block has the following form:

out - long **MTIERR**
If an error occurs, then **MTIERR** contains the system error code. This is sometimes useful as an auxiliary to the value returned by the function being used. If no error occurs, then this value is not touched.

out - long **MTINBL**
records the number of data blocks on the tape, before (ie not including) the current read position.

19.2.2 *Output common block*

The output common block has the following form:

out - long **MTOERR**
If an error occurs, then **MTOERR** contains the system error code. This is sometimes useful as an auxiliary to the value returned by the function being used. If no error occurs, then this value is not touched.

out - long **MTONBL**
records the number of data blocks on the tape, before (ie not including) the current write position.

i/o - long **MTOPTR**
points to the next available slot in the user's buffer. It is cleared by the routines where appropriate, but must otherwise be maintained by

the user.

out - long **MTOLST**
records the length in bytes of the last buffer written to magtape.

19.2.3 *Examples*

For examples of the use of the magtape routines (although output only), see the sources of the conversion utility I2MOSS.

19.3 *Input routines*

19.3.1 *MTINIT - initialise tape*

ok = MTINIT([name], [norew], [type])

out	long	**ok**	LSL__NORMAL if initialisation succeeds, otherwise see below
in	char	**name**	the magtape drive to use
in	logical	**norew**	if true, suppresses tape rewind
in	long	**type**	the type of controller protocol to use

MTINIT initialises the tape drive **name**: (default MTA0) for input, and assigns a channel to it.

If **norew** is false (or absent), then the tape is rewound, positioning it at BOT. Otherwise, the tape is not moved. Regardless, MTINBL will be set to zero.

type is used to indicate what sort of communications protocol is required to communicate with the magnetic tape drive:

* If **type** is zero (or absent) then a standard tape drive is assumed, and the tape is assumed mounted foreign (ie by the DCL command MOUNT/FOREIGN).

* If **type** is one, then a DIL serial interface controller is assumed. The tape drive will appear to the VAX as a normal serial port.

* Further values of **type** are reserved for possible future uses.

The following values of **ok** may be returned:

LSL__NORMAL	- success - tape drive initialised correctly
LSL__DEVALLOC	- failure - the magtape drive is already allocated to another process (MTIERR will contain SS$_DEVALLOC)
LSL__NOSUCHDEV	- failure - there is no device with the name specified by **name** (this error may also be given if **name** defaults, but MTA0 does not exist). Note that this error will also be given if **name** does not make sense as a device name. MTIERR will give a more precise system error, describing the problem.
LSL__SYSERR	- failure - some other error occurred - the system error code is in MTIERR

19.3.2 *MTIRWD - rewind magtape*

ok = MTIRWD()

out	long	**ok**	LSL__NORMAL if the rewind succeeds, otherwise see below

MTIRWD rewinds the tape to BOT. MTINBL is set to zero.

The following values of **ok** may be returned:

LSL__NORMAL - success - tape rewound successfully
LSL__SYSERR - failure - some other error occurred - the system error code is in MTIERR

19.3.3 *MTIRDB - read block*

ok = MTIRDB(buffer, length, bytcnt)

out - long	**ok**	LSL__NORMAL if the read succeeds, otherwise see below
out - byte	**buffer**	a byte buffer to receive data from the tape
in - long	**length**	the maximum number of bytes to read into buffer
out - long	**bytcnt**	the number of bytes actually read into buffer

MTIRDB reads the next block on the tape, block of maximum length **length** into the user specified byte buffer **buffer**. The number of bytes actually read is returned in **bytcnt**.

Note that it is not possible to read a block of **length** less than 14 bytes, due to physical limitations of the tape drive - any such blocks on the tape are ignored as noise.

MTIRDN increments MTINBL, and sets **bytcnt**, whether it succeeds or not.

The following values of **ok** may be returned:

LSL__NORMAL - success - block read successfully
LSL__BUFOVFLW - error - the block on the tape was too large to fit into **buffer**. The first **length** bytes are transferred, and **bytcnt** is set to the actual length of the block on the tape.
LSL__EOF - error - end of file - this is returned if a tapemark is read, or if the EOT marker is found. No data will be transferred to **buffer**, and **bytcnt** will be zero. Note, however, that MTINBL will still be incremented.
LSL__SYSERR - failure - some other error occurred - the system error code is in MTIERR

19.3.4 *MTISPC - space forwards/backwards*

ok = MTISPC(blkno)

out - long	**ok**	LSL__NORMAL if the blocks are skipped successfully, otherwise see below
in - long	**blkno**	number of blocks to space forwards

MTISPC causes the read head to space forwards by **blkno** blocks. If **blkno** is negative, then it spaces backwards. The value of MTINBL is adjusted accordingly, whether the routine succeeds or not.

The following values of **ok** may be returned:

LSL__NORMAL - success - tape moved successfully
LSL__ENDOFTAPE - error - returned if either EOT (moving forwards) or BOT (moving backwards) is found
LSL__ENDOFVOL - error - returned if two consecutive tapemarks are read (moving forwards only). The tape remains positioned between the tapemarks, and MTISPC may not be used to space past the second tapemark.
LSL__SYSERR - failure - some other error occurred - the system error code is in MTIERR

19.3.5 *MTIBCK - backspace one block*

ok = MTIBCK()

out - long **ok** result as for MTISPC

MTIBCK causes the read head to move backwards by one block. It is identical to a call of MTISPC(-1)

19.3.6 *MTIEOV - find end of volume*

ok = MTIEOV()

out - long **ok** LSL__NORMAL if end of volume is reached, otherwise see below

MTIEOV causes the read head to space over blocks until the end of volume is reached. This is considered to be marked by two consecutive tapemarks, and the tape is left positioned between the two tapemarks. Note that another call of MTIEOV will return (successfully) immediately.

MTINBL is incremented for each block read in this process.

The following values of **ok** may be returned:

LSL__NORMAL - success - tape moved successfully
LSL__ENDOFTAPE - error - returned if either EOT (moving forwards) or BOT (moving backwards) is found
LSL__SYSERR - failure - some other error occurred - the system error code is in MTIERR

19.3.7 *MTISNS - sense characteristics*

ok = MTISNS(result)

out - long **ok** LSL__NORMAL if sense succeeds, otherwise see below
out - long **result** current tape characteristics

Senses the tape characteristics and returns them in **result**. See the appropriate chapter of the VMS I/O systems manuals for how to interpret **result**

The following values of **ok** may be returned:

LSL__NORMAL - success - tape characteristics sensed successfully
LSL__SYSERR - failure - some other error occurred - the system error code is in MTIERR

19.4 *Output routines*

19.4.1 *MTONIT - initialise tape*

ok = MTONIT([name], [den], [norew], [type])

out	long	**ok**	LSL__NORMAL if initialisation succeeds, otherwise see below
in	char	**name**	the magtape drive to use
in	integer	**den**	tape density (see below)
in	logical	**norew**	if true, don't rewind the tape
in	long	**type**	the type of controller protocol to use

MTONIT initialises the tape drive **name**: (default MTA0) for input, and assigns a channel to it.

If **norew** is false (or absent), then the tape is rewound, positioning it at BOT. Otherwise, the tape is not moved. Regardless, MTINBL will be set to zero.

If **den** is -1 (or absent), then it sets the tape characteristics to 1600 bpi Phase Encoded. Other possible values are 0 for 800 bpi NRZI, or -2 for 6250 bpi GCR. In previous versions, this argument was just a logical, which will still work (FALSE = 0, TRUE = -1). Only the bottom two bits of the number are significant.

type is used to indicate what sort of communications protocol is required to communicate with the magnetic tape drive:

* If **type** is zero (or absent) then a standard tape drive is assumed, and the tape is assumed mounted foreign (ie by the DCL command MOUNT/FOREIGN).

* If **type** is one, then a DIL serial interface controller is assumed. The tape drive will appear to the VAX as a normal serial port.

* Further values of **type** are reserved for possible future uses.

The following values of **ok** may be returned:

LSL__NORMAL	- success - tape initialised successfully
LSL__DEVALLOC	- failure - the magtape drive is already allocated to another process (MTOERR will contain SS$_DEVALLOC)
LSL__NOSUCHDEV	- failure - there is no device with the name specified by **name** (this error may also be given if **name** defaults, but MTA0 does not exist). Note that this error will also be given if **name** does not make sense as a device name. MTOERR will give a more precise system error, describing the problem.
LSL__SYSERR	- failure - some other error occurred - the system error code is in MTOERR

19.4.2 *MTORWD - rewind magtape*

ok = MTORWD()

out	- long	**ok**	LSL__NORMAL if the rewind succeeds, otherwise see below

MTORWD rewinds the tape to BOT. MTONBL is set to zero.

The following values of **ok** may be returned:

LSL__NORMAL - success - tape rewound successfully
LSL__SYSERR - failure - some other error occurred - the system error code is in MTOERR

19.4.3 *MTOWRB - write block*

ok = MTOWRB(buffer, length)

out	- long	**ok**	LSL__NORMAL if the write succeeds, otherwise see below
in	- byte	**buffer**	buffer of data to write to the tape
in	- long	**length**	number of bytes of data to write

MTOWRB writes a block of length **length** to the magtape from **buffer**. Before output, it preserves **length** in MTOLST, and after output it increments MTONBL, and clears MTOPTR. These latter are regardless of success or failure.

Note that it is not possible to write a block of less than 14 bytes, due to physical limitations of the tape drive - any such blocks on the tape are ignored as noise. An attempt to do such will result in the routine failing with LSL__SYSERR, and a bad parameter value error in MTOERR.

The following values of **ok** may be returned:

LSL__NORMAL - success - block written successfully
LSL__ENDOFTAPE - error - returned if writing the block causes the tape to be positioned past the EOT, or if the tape was already there before the write. Part of the block may have been written.
LSL__SYSERR - failure - some other error occurred - the system error code is in MTOERR

19.4.4 *MTOEOF - write tapemark*

ok = MTOEOF()

out	- long	**ok**	LSL__NORMAL if the write succeeds, otherwise see below

MTOEOF writes a tape mark. It increments MTONBL, whether it succeeds or not.

The following values of **ok** may be returned:

LSL__NORMAL - success - tapemark written successfully
LSL__ENDOFTAPE - error - returned if writing the tapemark causes the tape to be positioned past the EOT, or if the tape was already there before the write.
LSL__SYSERR - failure - some other error occurred - the system error code is in MTOERR

19.4.5 *MTOSPC - Space backwards/forwards*

ok = MTOSPC(blkno)

out - long	**ok**	LSL__NORMAL if the blocks are skipped successfully, otherwise see below
in - long	**blkno**	number of blocks to space forwards

MTOSPC causes the write head to space forwards by **blkno** blocks. If **blkno** is negative, then it spaces backwards. The value of MTONBL is adjusted accordingly, whether the routine succeeds or not.

The following values of **ok** may be returned:

LSL__NORMAL - success - tape moved successfully
LSL__ENDOFTAPE - error - returned if either EOT (moving forwards) or BOT (moving backwards) is found
LSL__ENDOFVOL - error - returned if two consecutive tapemarks are found (moving forwards only). The tape remains positioned between the tapemarks, and MTOSPC may not be used to space past the second tapemark.
LSL__SYSERR - failure - some other error occurred - the system error code is in MTOERR

19.4.6 *MTOBCK - Space backwards one block*

ok = MTOBCK()

out - word	**ok**	as for MTOSPC

MTOBCK causes the write head to move backwards by one block. It is identical to a call of MTOSPC(-1).

19.4.7 *MTOEOV - find end of volume*

NOTE that this routine does not *write* end of volume!

ok = MTOEOV()

out - long	**ok**	LSL__NORMAL if end of volume is reached, otherwise see below

MTOEOV causes the write head to space over blocks until the end of volume is reached.

This is considered to be marked by two consecutive tapemarks, and the tape is left positioned between the two tapemarks. Note that another call of MTOEOV will return (successfully) immediately.

MTONBL is incremented for each block read in this process.

The following values of **ok** may be returned:

LSL__NORMAL - success - end of volume found successfully
LSL__ENDOFTAPE - error - returned if either EOT (moving forwards) or BOT (moving backwards) is found
LSL__SYSERR - failure - some other error occurred - the system error code is in MTOERR

19.4.8 *MTORDB - read block*

ok = MTORDB(buffer, length, [bytcnt])

out - long	**ok**	LSL__NORMAL if the read succeeds, otherwise see below
out - byte	**buffer**	a byte buffer to receive data from the tape
in - long	**length**	the maximum number of bytes to read into buffer
out - long	**bytcnt**	the number of bytes actually read into buffer

MTORDB reads the next block from the tape, of maximum length **length** into the user specified byte buffer **buffer**. The number of bytes actually read is returned in **bytcnt**.

Note that it is not possible to read a block of **length** less than 14 bytes, due to physical limitations of the tape drive - any such blocks on the tape are ignored as noise

MTORDB increments MTONBL, whether it succeeds or not.

The following values of **ok** may be returned:

LSL__NORMAL - success - block read successfully
LSL__BUFOVFLW - error - the block on the tape was too large to fit into **buffer**. The first **length** bytes are transferred, and **bytcnt** is set to the actual length of the block on the tape.
LSL__EOF - error - end of file - this is returned if a tapemark is read, or if the EOT marker is found. No data will be transferred to **buffer**, and **bytcnt** will be zero. Note, however, that MTINBL will still be incremented.
LSL__SYSERR - failure - some other error occurred - the system error code is in MTOERR
LSL__MISSARGS - failure - one of the mandatory arguments is missing

19.4.9 *MTOSNS - sense characteristics*

ok = MTOSNS(result)

out - long	**ok**	LSL__NORMAL if the sense succeeds, otherwise see below
out - long	**result**	current tape characteristics

Senses the tape characteristics and returns them in **result**. See the appropriate chapter of the VMS I/O systems manuals for how to interpret **result**

The following values of **ok** may be returned:

LSL__NORMAL	- success - sense performed successfully
LSL__SYSERR	- failure - some other error occurred - the system error code is in MTOERR

CHAPTER 20

ROUTINES TO EASE USE OF SYSTEM FACILITIES

20.1 *Setting up a control-C AST*

call SET_CTRLC_AST([routine], [input])

in - external	**routine**	the routine to set up. Defaults to the last used AST_routine
in - char	**input**	the name of the input channel to place the QIO on. Defaults to SYS$INPUT, or the last **input**

SET_CTRLC_AST places a control-C AST routine on an input channel. Then, if control-C is received on that channel, the AST routine will be called.

If **routine** is given, then it is used as the AST routine, and must be declared EXTERNAL in the calling Fortran routine. If it is not given, then the value given in the last call of SET_CTRLC_AST is used. If this is the first call, then address zero (an unset AST) is used.

If **input** is given, then that is the name of the input channel to place the QIO request for the AST on. If it is not given, then the value given in the last call of SET_CTRLC_AST is used. If this is the first call, then SYS$INPUT is used. **input** should be omitted when re-enabling an AST on a channel - if it is given, then another channel will be needlessly assigned.

If an error occurs within SET_CTRLC_AST, then it will use LIB$SIGNAL to output the appropriate system error message, and will then return.

The following points should be noted:

1. ASTs will queue up. Thus, if more than one control-C AST is defined, they will all be called (in last in, first out order) when a control-C is received

2. It is not possible to place a control-C AST on an input channel which is not a terminal. Routine TEST_TERM may be used to test whether a channel is a terminal or not.

3. Control-C ASTs are one-shot. Once triggered, the AST must be re-enabled, possibly by calling SET_CTRLC_AST with no arguments. This can conveniently be done in the AST routine itself.

20.2 *Setting up an out-of-band character AST*

call SET_OUTBAND_AST([routine], [mask], [include], [input])

in - external	**routine**	the routine to set up. If omitted, then the AST is cancelled (provided that **input** is omitted also).
in - long	**mask**	a mask containing bits set corresponding to the control characters to trap. For example, to trap Control-C and Control-D, set bits 3 and 4, giving a value of 8+16=24. Defaults to zero, which is only useful if cancelling the AST.
in - logical	**include**	if true then any trapped control character is still included in the input stream from the device. If omitted or false, then the character is not included.
in - char	**input**	the name of the input channel to place the QIO on. Defaults to SYS$INPUT, or the last **input**

SET_OUTBAND_AST places an out-of-band character AST routine on an input channel. If any of the selected control characters are received, then the AST routine will be called. The particular characters are passed to the AST routine as the AST parameter. This parameter value (0-31) may be accessed as %LOC(first argument) within the AST routine.

One reason for using SET_OUTBAND_AST is to prevent control-C from aborting any input/output operations in progress on the channel. This can cause certain graphics devices to hang because partial commands are received. Provided that **include** is false, then the control characters are not seen by any control-C handlers, or system handlers. See the DEC VMS system programming (IO Part 1) manual (terminal drivers) for details.

If **routine** is given, then it is used as the AST routine, and must be declared EXTERNAL in the calling Fortran routine. If it is not given, and **input** is not given either, then any AST is cancelled.

If **input** is given, then that is the name of the input channel to place the QIO request for the AST on. If it is not given, then the value given in the last call of SET_OUTBAND_AST is used. If this is the first call, then SYS$INPUT is used.

If an error occurs within SET_OUTBAND_AST, then it will use LIB$SIGNAL to output the appropriate system error message, and will then return.

The following points should be noted:

1. Only one AST may be declared on a given channel, but more than one can be enabled by explicitly giving the **input** argument.

2. It is not possible to place an out-of-band character AST on an input channel which is not a terminal. Routine TEST_TERM may be used to test whether a channel is a terminal or not.

20.3 *GETID - get LSL standard date stamp*

call GETID(date)

out - char **date** receives the date stamp - 6 characters long

This routine produces the LSL standard 6 character IDENT for today. This is in the form **ddMMyy**, where

dd is from 01 to 31 (as appropriate for the month!)

MM is one of JA, FE, MR, AP, MY, JN, JL, AU, SE, OC, NO or DE

yy is up to 73 (representing a year between 1974 and 2073)

20.4 *Testing a device's status*

20.4.1 *Is device mounted /FOREIGN?*

ok = TEST_FOREIGN(file, yes, ierr)

out - long	**ok**	LSL__NORMAL if routine succeeds, otherwise see below
in - char	**file**	the name of the file/device to check
out - logical	**yes**	set true if the device is mounted /FOREIGN
out - long	**ierr**	system error code from $GETDVI call

TEST_FOREIGN uses the $GETDVI system service to find out if the device name in **file** is mounted foreign. It looks for the characteristic DVI$_FOR.

$GETDVI will logically translate **file**, and treat the translation as a device name. Note that if there is a colon (":") in the filename, then any characters after the colon are ignored. If **file** starts with an underline ("_") then it will be treated as a physical device name, and will not be logically translated.

If the device is mounted foreign (ie by the MOUNT/FOREIGN command) then **yes** is returned as true, and otherwise it is returned as false.

The following values of **ok** may be returned

LSL__NORMAL	- success - an appropriate value will be returned in **yes** (but see below).
LSL__NOSUCHDEV	- failure - the device does not exist. **yes** is undefined.
LSL__SYSERR	- failure - the actual error returned by $GETDVI is returned in **ierr**. **yes** is undefined.

The result of calling the $GETDVI system service is always returned in **ierr**. Note that if **file** does not contain a device name, then **ierr** will be set to SS$_IVDEVNAM (invalid device name).

The function interprets this condition to mean that it should

* set **yes** to false

* return **ok** as LSL__NORMAL

assuming that a filename was specified, which did not contain a device name and was therefore on the current device (presumably a disk).

20.4.2 *Is device a terminal?*

ok = TEST_TERM(file, yes, ierr)

out - long	**ok**	LSL__NORMAL if routine succeeds, otherwise see above
in - char	**file**	the name of the file/device to check
out - logical	**yes**	set true if the device is a terminal
out - long	**ierr**	system error code from $GETDVI call

TEST_TERM uses the $GETDVI system service to find out if the device name in **file** is a terminal device. It is identical to TEST_FOREIGN, except that it looks for the characteristic DVI$_TRM.

20.5 *Check for presence of an argument*

20.5.1 *Optional arguments in Fortran*

This routine can be used to check for an absent argument, allowing Fortran routines to implement optional arguments.

yes = HAVE_I_AN_ARG(argnum)

out - logical	**yes**	true if argument present, false otherwise
in - long	**argnum**	number of the argument to check

This routine checks the calling routines argument pointer, to see if argument number **argnum** was supplied to the routine. The value of **yes** is set as follows:

false if **argnum** is zero - argument zero is always absent (!)

false if the **argnum**th argument is missing

true if the **argnum**th argument is present

The following problem should be noted:

* If the argument is declared as a character string, then the Fortran optimiser performs actions which assume that the argument is present. If the argument is then not passed, the program will fall over.

It is possible that future versions of Fortran might extend this problem to other data-types. For maximum safety, restrict optional arguments to byte, word or longword quantities (ie BYTE, LOGICAL, INTEGER, REAL), and do not use it for arrays.

20.5.2 *Optional arguments in MACRO-32*

JSB VIO$1ST

This routine initialises the argument handling interface, and must be called before the presence of any arguments is checked. Register R11 is used to hold a count of the number of arguments found so far, and VIO$1ST simply clears that register.

JSB VIO$GNA

This is the routine which is used to get the next argument. Before calling it, VIO$1ST must have been used to initialise argument handling, and R11 must contain the number of arguments so far obtained. The routine first increments R11, and if the current argument is present, the carry bit in the Program Status Word is cleared and the address of the argument is returned in R0. If there are no arguments left, or if the current argument is missing, the carry bit in the Program Status Word is set and R0 is left untouched. It is imperative that the contents of R11 are not disturbed between calls to VIO$GNA.

20.5.2.1 *Example -*

```
        JSB     G^VIO$1ST               ; basic initialisation
        MOVAL   DEFAULT,R0              ; address of default
        JSB     G^VIO$GNA               ; get first argument
        PUSHL   R0                      ; save it
        JSB     G^VIO$GNA               ; and next arg (no default)
        BCS     ERROR                   ; missing arg error
        ...
```

In this example, DEFAULT is the default to use in lieu of an argument, so it is put into R0 after argument handling initialisation. The first argument is requested, and if there isn't one, R0 will still contain the default. R0 can therefore be saved for later use with no further tests necessary. The next argument is then requested, and this time the carry bit is tested to see if it was actually there. If the carry bit is set, indicating that the argument was missing, execution branches to the label ERROR.

Note that general mode addressing (G^) should be used when calling all LSLLIB routines from Macro, and also when referencing LSLLIB common variables.

20.6 *JPINFO - print process information*

call JPINFO

This subroutine will call the SYS$GETJPI system service to get the current user name, process name, terminal name, and image name. It then prints these out using WRITEF.

20.7 *LSL_TMRINI - set up a timer exit handler*

call LSL_TMRINI

This subroutine calls LIB$INIT_TIMER, and then declares an exit handler which will print timer statistics using LIB$SHOW_TIMER when the program exits.

The exit handler will get called on image exit unless the program is killed by CTRL/Y followed by STOP.

The common file LSL$CMNLSL:EXIT_HANDLER.CMN contains the four longword description block DESBLK and the longword EXIT_STATUS. The former is required if the exit handler is to be cancelled, and the latter is where the exit handler will return its exit status.

NOTE that LSL_TMRINI is called by LSL_INIT (unless the **timer** argument is false).

20.8 *TRNALL - recursively translate a logical name*

ret = TRNALL(lognam , translation)

out - long	**ret**	SS$_NORMAL if the logical name translation succeeds, otherwise see below
in - char	**lognam**	the logical name to be translated
out - char	**translation**	contains the result of logical name translation

Translates a logical name recursively until it will translate no further.

The following values of **ret** may be returned:

SS$_NORMAL	- success - the translation of the logical name will be returned in **translation**
SS$_INSFARG	- failure - insufficient arguments supplied to TRNALL
SS$_NOLOGNAM	- failure - the logical name doesn't translate at all
SS$_RESULTOVF	- failure - the result of logical name translation overflowed the supplied character string **translation**

20.9 *VIOCLR - clear or set an array*

ret = VIOCLR(array, size, [fill])

out - long	**ret**	returns either SS$_NORMAL, or SS$_INSFARG if required arguments are missing
i/o - byte	**array**	the array to be initialised
in - long	**size**	the number of bytes to initialise, offset from the beginning of **array**
in - byte	**fill**	the value to fill with

VIOCLR will set an area of memory to all zero (the default if **fill** is not given), or to a byte value. The latter case is only normally useful for **fill** = -1, or an ASCII character.

The only advantage that this routine has over LIB$MOVC5 is that it takes a longword length, rather than a word. It will use MOVC5 sufficient times to fill the space requested.

20.10 *VIOMV3 - move an array*

ret = VIOMV3(source, size, destn)

out - long	**ret**	returns either SS$_NORMAL, or SS$_INSFARG if required arguments are missing
in - byte	**source**	the array to be copied
in - long	**size**	the number of bytes to copy
out - byte	**destn**	the array to copy to

VIOMV3 will copy one array into another.

The only advantage that this routine has over LIB$MOVC3 is that it takes a longword length, rather than a word. It will use MOVC3 sufficient times to copy the number of bytes requested.

20.11 *LSL_WAIT - wait for a time*

call LSL_WAIT(time, units) [routine used to be called WAIT]

in - word	**time**	the length of time to wait for
in - word	**units**	the units of **time** - see below

LSL_WAIT causes the calling process to wait for **time** units, where the units depend upon the value of **units** as follows:

0 ticks, 50 per second
1 milliseconds
2 seconds
3 minutes
4 hours

The routine uses the system $HIBER routine to hibernate for a specified period of time.

The maximum time for a wait is 2**31-1 milliseconds (ie 23 days).

This routine has little advantage over LIB$WAIT, which takes a real time in seconds.

20.12 *WFLOR - wait for event flags*

ret = WFLOR(efn1, [efn2], [efn3], ...)

out - long	**ret**	returns the result of calling $WFLOR to wait - not normally used
in - word	**efn<n>**	an event flag to wait for

WFLOR provides a simplified interface to the system routine $WFLOR. It takes each event flag number specified, and forms the mask required to call $WFLOR. It then waits for one of the event flags to be set (ie it performs a wait for the logical OR of one of the event flags).

WFLOR mimics the RSX-11M library routine WFLOR.

20.13 *LSLLIB's condition handler*

call LIB$ESTABLISH(LSL_NUM_CHAND)

Numeric exceptions are detected by the LSLLIB condition handler LSL_NUM_CHAND. This is declared as an exception handler (using LIB$ESTABLISH) at the start of each number reading routine. If a numeric exception occurs, then it sets ERRNUM to LSL_HADEXCP, and LSL_EXCP to an appropriate error.

Note that the exception handler is only declared within the number reading routines by LSLLIB, so it will not detect numeric exceptions in the calling program. However, it may be established within a user program if required - declare it as EXTERNAL, and use LIB$ESTABLISH.

The following conditions are handled by LSL_NUM_CHAND (and reduced in severity to informational, so that the program continues without complaint):

SS$_FLTDIV, SS$_FLTDIV_F, SS$_FLTOVF, SS$_FLTOVF_F,
SS$_FLTUND, SS$_FLTUND_F, SS$_INTDIV, SS$_INTOVF

For a list of the error codes returned in LSL_EXCP, see the list of errors whilst reading numbers, documented in the chapter on exceptions.

For more details on reading numbers, see the chapter on reading numbers.

20.14 *Date and time conversions*

Dates and times are stored as integers. There are routines to convert from standard VMS date/time strings to these numbers and vice versa. There are also routines to convert from day, month and year to day number and vice versa.

1. Dates are either the number of days since 17-NOV-1858 (up to 31-DEC-9999) stored as a positive integer, or are the number of days from today - the "delta date" - which is stored as a negative integer and is in the range 000 - 9999.

2. Times are the number of 10s of milliseconds since midnight.

20.14.1 *Convert from date/time string*

ret = CVT_DATE(date, day, time)

out - long **ret** SS$_NORMAL if the conversion succeeds, otherwise see below
in - char **date** a standard DEC VMS date/time string (see VMS

documentation for details)
out - long **day** the day number (-ve for a delta date)
out - long **time** the time (in 10s of milliseconds) translation

Converts from a standard DEC VMS date/time string to integers representing date and time.

The following values of **ret** may be returned:

SS$_NORMAL - success - the conversion completed successfully
SS$_IVTIME - failure - The syntax of the specified date/time string is invalid, or the time component is out of range

20.14.2 *Convert from date to string*

ret = CVT_DAY_STR(date, length, date_str)

out - long **ret** SS$_NORMAL if the conversion succeeds, otherwise see below
in - long **date** an integer representing a date
out - long **length** the length of the date_str
out - char **date_str** the ASCII representation of the date translation

Converts from an integer representing a date to a standard DEC VMS ASCII date.

The following values of **ret** may be returned:

SS$_NORMAL - success - the conversion completed successfully
SS$_IVTIME - failure - The specified delta time is equal or greater than 10,000 days
SS$_BUFFEROVF - success - date_str is too short to receive the string. It has been truncated.

20.14.3 *Convert from time to string*

ret = CVT_TIME_STR(time, length, date_str)

out - long **ret** SS$_NORMAL if the conversion succeeds, otherwise see below
in - long **date** an integer representing a time
out - long **length** the length of the time_str
out - char **time_str** the ASCII representation of the time translation

Converts from an integer representing a time to a standard DEC VMS ASCII time.

The following values of **ret** may be returned:

SS$_NORMAL - success - the conversion completed successfully
SS$_BUFFEROVF - success - time_str is too short to receive the string. It has been truncated.

20.14.4 *Convert from day,month,year to date*

ret = CVT_DMY_DAY(date, day, month, year)

out	long	**ret**	SS$_NORMAL if the conversion succeeds, otherwise see below
out	long	**date**	an integer representing a date
in	long	**day**	the day of the month (in the range 1-31)
in	long	**month**	the month number (in the range 1-12)
in	long	**year**	the year (in the range 1858-9999)

Converts from day,month,year to an integer representing the date. If either the day of the month or the month number are out of range, they are set to 1

The following values of **ret** may be returned:

SS$_NORMAL	- success - the conversion completed successfully
SS$_IVTIME	- failure - The year as input is out of range, causing the syntax of the date/time string used internally to be invalid.
SS$_INTOVF	- failure - The input arguments represent a date past the year 8600

20.14.5 *Convert from date to day,month,year*

ret = CVT_DAY_DMY(date, day, month, year)

out	long	**ret**	SS$_NORMAL if the conversion succeeds, otherwise see below
in	long	**date**	an integer representing a date
out	long	**day**	the day of the month (in the range 1-31)
out	long	**month**	the month number (in the range 1-12)
out	long	**year**	the year (in the range 1858-9999)

Converts from an integer representing a date to the equivalent day,month,year

The following values of **ret** may be returned:

SS$_NORMAL	- success - the conversion completed successfully
SS$_IVTIME	- failure - The specified delta time is equal or greater than 10,000 days
SS$_ACCVIO	- failure - The 64-bit time value cannot be read internally, or the buffer used internally for the result cannot be written.

CHAPTER 21

SORT ROUTINES

21.1 *Introduction*

These routines are designed for sorting non-specific forms of data. That is, the user supplies the routines that actually compare and move elements of the data. The sort routine can then just call these as appropriate, without needing to know the actual form of the data.

21.1.1 *Routines supplied by the user*

All three of the sort routines require that the user supply a comparison routine, which is defined as follows:

ret = CF(table, index1, index2)

out -	long	**ret**	the result of the comparison - see below
i/o -		**table**	the array to be sorted. This is passed through from the sort routine, and must be declared as the relevant type
in -	long	**index1**	the number of the first element to be compared
in -	long	**index2**	the number of the second element to be compared

cf then compares the two elements, and returns the following values:

-1 if element **index1** < element **index2**
0 if element **index1** = element **index2**
1 if element **index1** > element **index2**

Quicksort and Shell's sort also require a routine to swap two elements of the input array, **table**. This is defined as:

call SWAP(table, index1, index2)

where the arguments are as for **cf**, but the two elements are swapped rather than compared.

Heapsort requires a routine to copy an element of the array to be sorted, **table**, from somewhere to somewhere else. This is defined as:

call COPY(table1, index1, table2, index2)

in	-	**table1**	the 'array' to take the element from. This is passed through from the sort routine, and must be declared as the relevant type
in	- long	**index1**	the number of the element to be taken from **array1**
out	-	**table2**	the 'array' to move the element to. This is passed through from the sort routine, and must be declared as the relevant type
in	- long	**index2**	the number of the element to be overwritten in **array2**

The routine is used in the following ways (where **store** is described in the definition of HEAP_SORT):

call COPY(table, index1, table, index2)

to move the element in position **index1** in **table** to position **index2** in **table**

call COPY(table, index1, store, 1)

to store the element of **table** at position **index1** in the workspace variable **store**

call COPY(store, 1, table, index2)

to recover an element from **store** and place it in position **index2** of **table**

21.2 *Quick sort*

Quicksort - otherwise known as partition exchange, or Hoare's method

call QUICK_SORT(table, count, cf, swap, stack)

i/o	-	**table**	the array to be sorted
in	- long	**count**	the number of elements in **table**
in	- external	**cf**	longword function to compare two elements of **table**
in	- external	**swap**	routine to swap two elements of **table**
	- long	**stack**	array to use as workspace

The data to be sorted is held in the array **table, and there are count** elements in that array.

The longword function **cf** and the routine **swap** must be provided by the user, and they are described above.

stack is a stack or workspace supplied by the calling routine, and must be of length 2*log2(**count**), in order to prevent overflow.

For an explanation and discussion of Quicksort (CA Hoare's method), see

Knuth - The Art of Computer Programming, vol 3, p114 sqq
Wirth - Algorithms + Data Structures = Programs, p76 sqq

Comments on quicksort:

* the average running time is very good, but the worst case is very bad!
* it needs a stack of size longword(2,log2(**count**))
* equal keys will be swapped

Also note that this version doesn't do a straight insertion sort for (sufficiently) small partitions - instead it just Quicksorts all the way down.

21.3 *Heap sort*

call HEAP_SORT(table, count, cf, copy, store)

i/o	-	**table**	the array to be sorted.
in	- long	**count**	the number of elements in **table**
in	- external	**cf**	longword function to compare two elements of **table**
in	- external	**copy**	routine to copy an element of **table** elsewhere
	-	**store**	this is used to hold a single element of **table**, and should be declared appropriately

The data to be sorted is held in the array **table**, and there are **count** elements in that array.

The longword function **cf** and the routine **copy** must be provided by the user, and are described above.

store is a single element version of **table**, used to temporarily remember one element during the sort.

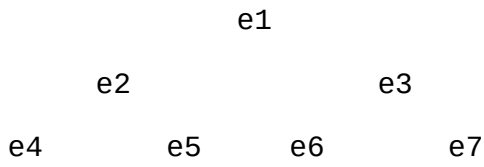
For an explanation and discussion of Heapsort, see

Knuth - The Art of Computer Programming, vol 3, p145 sqq
Wirth - Algorithms + Data Structures = Programs, p73 sqq

Comments on heap sort:

* the average and worst cases take about the same time
* it takes about twice the average time of Quicksort
* equal keys will be swapped
* it needs workspace of one element of whatever is being sorted

This sort works by creating a HEAP - an ordered tree of data, for instance:



such that

e(j/1) >= ej for 1 <= (j/2) < j <= **count**

ie we have e1 >= e2, e1 >= e3, e2 >= e4, etc

This can then be traversed to determine the correct sorting.

Thus this algorithm first rearranges the data into a heap, and then repeatedly removes the top of the heap and transfers it to its proper final position

21.4 *Shell sort*

Shell's sort - otherwise known as a diminishing increment sort

call SHELL_SORT(table, count, cf, swap)

i/o -	**table**	the array to be sorted
in - long	**count**	the number of elements in **table**
in - external	**cf**	longword function to compare two elements of **table**
in - external	**swap**	routine to swap two elements of **table**

For an explanation and discussion of Shell's sort, see

Knuth - The Art of Computer Programming, vol 3, p84 sqq
Wirth - Algorithms + Data Structures = Programs, p68 sqq

Comments on Shell's sort:

* reasonably efficient for a moderately large **count** (say less than or around 1000)
* equal keys will be swapped
* doesn't require any extra elements to use as workspace

APPENDIX A

LSLLIB SUCCESS/ERROR CODES

A.1 *Introduction*

This appendix lists the LSLLIB error codes, which may be returned by LSLLIB routines, or otherwise used from within LSLLIB. The actual message numbers are defined within the parameter files:

LSL$CMNLSL:LSLLIBMSG.PAR	for FORTRAN
LSL$CMNLSL:LSLLIBMSG.MAR	for MACRO/32
LSL$CMNLSL:LSLLIBMSG.H	for C

For each message, the message name and text are listed. The message name is the name of the message parameter, without the LSL__ prefix.

A.2 *Success messages*

CREATED, new file created for writing

NORMAL, normal, succesful completion

SIGSUCC, LSL_SIGNAL called with severity SUCCESS

STRCHAR, string terminated succesfully by given character

STRCMD, string terminated succesfully - as for a command

STREOL, string terminated succesfully by end of line

STRSPACE, string terminated succesfully by space

A.3 *Informational messages*

IFFOPENED, <file> opened for <action>

SIGINFO, LSL_SIGNAL called with severity INFORMATION

A.4 *Error messages*

AMBIG, ambiguous command '<string>', as between <string> and <string>

AMBIG2, ambiguous secondary command '<string>', as between <string> and <string>

AMBINEQ, ambiguous inequality '<string>', as between <string> and <string>

BADEXCEP, bad exception code detected by numeric excep. handler

BADINEQ, invalid inequality '<string>'

BADPARSE, too few arguments given to FILE_PARSE routine

BADTCOND, illegal termination condition to READSTR routine

BASECH, unknown base character whilst reading integer, after ^

BUFFEROVF, message buffer overflow

BUFOVFLW, magnetic tape buffer overflow

COMMA, number missing - comma followed by comma

DEFFILNAM, error parsing default filename

DEFVERNUM, unexpected version number in default file name

DEVALLOC, device is already allocated

ENDOFTAPE, end of tape detected

ENDOFVOL, end of volume detected

FAC, file access conflict

FILINUSE, file has already been opened

FILNOLEN, expanded filename has zero length

FLTDIV, floating division overflow

FLTOVF, floating overflow

FLTUND, floating underflow

HADEXCP, an arithmetic exception occurred in reading the number

IFFCREATE, IFF error creating file "<file>"

IFFMODIFY, IFF error opening file "<file>" for update

IFFOPEN, IFF error opening file "<file>" for read

IFFPARSE, error parsing IFF filename "<file>"

IFFSIZE, IFFCREATE - initial size <number> should be greater than zero

IFFVERNUM, version number not allowed when creating new IFF file

ILLEGLUN, illegal unit number

INTDIV, integer division overflow

INTOVF, integer overflow

INTPARSERR, internal parsing error

INVALSPEC, invalid syntax for value specification

LUNINUSE, a file is already open with the given unit number

MAXPAREX, maximum parameter count exceeded

MISSARGS, argument(s) to routine missing

MSGNOTFND, message not found

NOLUNS, no room in unit table

NONUM, no number to read, where number expected

NOSUCHDEV, no such device

NOSUCHFILE, file cannot be found

NOSUCHLUN, the unit number requested is not in the unit table

RESPARSOVF, result of parse overflowed buffer

SIGERR, LSL_SIGNAL called with severity ERROR

SRCFILNAM, error parsing source filename

SRCVERNUM, unexpected version number in source file name

STRTOOLONG, string is too long - truncated

SYNTAXERR, syntax error

SYSCLOSE, system error while closing file

SYSERR, i/o system error

SYSFIND, system error while finding record in file

SYSOPEN, system error while opening file

SYSREAD, system error while reading from file

SYSREW, system error while rewinding file

SYSUPD, system error while updating record in file

SYSWRITE, system error while writing to file

UNEXPCH, unexpected character '<byte>'

UNEXPCMD, unexpected '<string>' found instead of command

UNEXPCMD2, unexpected '<string>' found instead of secondary command

UNEXPEOF, unexpected end of file

UNEXPEOL, unexpected end of line

A.5 *Warning messages*

DEFTOOBIG, part of the default filename was too long

EOF, end of file

FILTOOLONG, filename is too long - truncated

NEGPOSNMK, position marker under-flowed start of TXTBUF

NOFIELD, filename node or device field is blank

POSNMKOVF, position marker over-flowed end of TXTBUF

RANREV, range specification in wrong order - numbers reversed

RECTOOBIG, record being read from file is too long - truncated

SIGWARN, LSL_SIGNAL called with severity WARNING

SRCTOOBIG, part of the source filename was too long

A.6 *Fatal messages*

SIGSEVER, LSL_SIGNAL called with severity SEVERE

APPENDIX B

DIFFERENCES FROM VIOLIB/CMDLIB

B.1 *Introduction*

LSLLIB is intended to replace both VIOLIB and CMDLIB, and is essentially the result of merging the two libraries and then producing a VAX style library, rather than a rather ad-hoc RSX style library.

This appendix attempts to describe the differences between VIOLIB/CMDLIB and LSLLIB, and is thus intended as a guide in both updating programs to use the newer library, and in deciding what routine to use in the first place.

The document includes:

* a general discussion of major differences between the libraries
* a note on changes from CMDLIB routines, including details for each of the number reading routines
* a routine by routine discussion of VIOLIB, with notes on what to use from LSLLIB instead
* a discussion of the changes to common blocks
* some general notes on other changes

It does not discuss things that are *new* in LSLLIB - see the rest of the LSLLIB reference manual for these!

B.2 *Major differences*

* Before using LSLLIB, a call must be made to the LSL_INIT routine. This sets up various things, including the link to the LSLLIB error messages. Every call of EXPAND makes a simple check that LSL_INIT has been called.

* System style error handling has been introduced, and is intended to be used in all future LSL programs.

* Most routines are now functions returning LSL__ error codes, rather than positive/negative values. Specifically, end-of-file is no longer signalled by returning -10.

* Command line handling is provided to allow VAX style line processing (using system supplied routines) to be performed. The old routines TXTCML and TXTCLI have been withdrawn.

* TXTBUF and EXPBUF are increased to a maximum of 1024 meaningful characters in length (the maximum that DCL is likely to handle), and parameters are set up for the maximum length of these buffers. EXPMAX and TXTLIM now default to 255 characters initially, instead of 80, and new routines are provided to change them - it is not envisaged that users will alter them explicitly (in the same way that they don't normally touch DCPTR explicitly).

* Anything that was included purely for compatibility with LIOLIB or RSX has been removed (note that WFLOR has been retained, as being genuinely useful).

* All common blocks, and all internal routines (those which are globally defined, but are not described in the LSLLIB reference manual) have names that start with the characters "LSL_"

B.3 *CMDLIB - comments*

The changes in placing CMDLIB into LSLLIB are fairly minor.

* The command reading and and manipulating routines are basically the same, except that they now reference different common blocks.

* The routine CMDERR has been replaced by LSL_CMDERR, which takes no arguments.

* The number reading routines (which superceded the previous VIOLIB versions) now have extended functionality in some cases, and they also reference different common blocks. Changes in number syntax are :-

  i) the "&" character is no longer supported as a 'power of 10' notation - this was only retained in VIOLIB for compatibility with PDP-11 libraries.

ii) all of the integer reading routines now accept numbers to different bases, through the use of B, O, D, or X as prefixes - previously RDLONG was the only routine which supported this.

Changes in the routine specifications are :-

i) the real number reading routines do not now allow the second optional "had-dot" argument.

ii) RDLONG does not now allow a default base as an optional second argument.

For changes in common blocks, see below.

See the relevant chapter in the main body of the LSLLIB Reference Manual for full details of number reading routines.

B.4 *VIOLIB - comments by routine*

This section itemises each VIOLIB routine, and how the corresponding LSLLIB routine differs.

APPEND	see EXPAND
BSCH	no change
BSLN	no change
CMLTIT	withdrawn - use the new DCL command line routines to fetch and process the initial (foreign) command line
CMLTWO	see CMLTIT
DCPSAV	no change
DCPSET	no change
DEFGBLSYM	withdrawn - use the appropriate LIB$ routine
DEFLOCSYM	withdrawn - use the appropriate LIB$ routine
EDIV	withdrawn
ERRFLN	withdrawn - use LSL_PUTMSG or LSL_ADDMSG
EXPFLN	routine is the same, but filenames are now V4 syntax, and the common blocks are different
EXPAND	(1) the default integer size is now longword (%^L), rather than word (%^W), as the VAX default integer is longword (2) the default length for string output (%S,%A) is now 255, rather than 72 (3) the 'power of 10' indicator '&' is withdrawn - use 'E' or 'e'

(4) vertical format controls, which are not supported on the VAX, have been withdrawn
(5) the maximum length of EXPBUF has been increased - see EXPC below

FILEIO the routines have been changed to return LSL__ error codes, and FLRSTR has a new argument to allow it to return the length of the string read. Various new routines have been added.

FILE_PARSE this now recognises VMS version 4 filename syntax (including ;-0 !)

FLTERR withdrawn - use LSL_PUTMSG or LSL_ADDMSG

GEN_SYSMSG withdrawn - use LSL_PUTMSG or LSL_ADDMSG

GETCLI withdrawn - use the LSL DCL routines to get and process the command line

HAVE_I_AN_ARG no change

ICL tape routines are withdrawn from LSLLIB - specialist tape routines should eventually be collected in a separate library

JPINFO no change

LTH withdrawn - Laseraid help facilities are not kept in LSLLIB

Magtape routines - the low level magtape routines (MTIV and MTOV) have been enhanced to return LSL__ error codes, and to maintain internal variables in a more sensible manner

RDCH no change

RDCHS no change

RDDBLE see CMDLIB changes

RDFILN this routine is now called PARFILN (parse file name), and returns LSL__ error codes

RDFILT withdrawn - use GETFILNAM

RDHEX see CMDLIB changes

RDINT see CMDLIB changes

RDLHEX see CMDLIB changes

RDLOCT see CMDLIB changes

RDLONG see CMDLIB changes

RDOCT see CMDLIB changes

RDREAL see CMDLIB changes

RDSWCH withdrawn - use the LSL DCL routines to handle command line switches

RDSWVI as for RDSWCH

RDSWVR as for RDSWCH

RREAL8 see CMDLIB changes

SUBQUAD withdrawn

Tek drawing routines (TX...) are not included in LSLLIB

TMRINI now called LSL_TMRINI, and called automatically by LSL_INIT

TTATT withdrawn

TTATC withdrawn

TTDET withdrawn

TTRLIN returns LSL__ error code (thus not number of characters read)

TTRSTR returns LSL__ error codes now (thus not number of characters read); has a new argument to return length of string read

TTWLIN returns LSL_ error codes now

TTWSTR returns LSL_ error codes now

Unsolicited input routines are superseded by various system functions, and are not included in LSLLIB

VAXRSX the following 'RSX mimics' have been withdrawn -
ALTPRI, AZTOAD, CLREF, DSASTR, ENASTR, READEF, SETEF, WAITFR

VIO$1ST no change

VIO$xxxx_SEC the mapped section routines are identical to the VIOLIB versions

VIO$GET_INPUT no change

VIO$GNA no change

VIO$PUT_OUTPUT no change

VIOCLR no change

VIOMV3 no change

VT52 drawing routines (VT...) are not included in LSLLIB

WAIT	no change
WFLOR	no change
WRITAP	no change, except that the only alias supplied is WRTAPP
WRITEF	no change, except that there are no aliases for this routine

B.5 *Common blocks*

The following is a list of VIOLIB/CMDLIB common block filenames, and notes on their corresponding LSLLIB files (which all live in LSL$CMNLSL:)

* EXPC EXPC - but note that the length of EXPBUF is vastly increased, and the default length is not the same. Routines are provided to change and save EXPMAX and to save and restore TXTBUF/TXTPTR.
* TXTC TXTC - but note that the length of TXTBUF is vastly increased, and the default length is not the same. Routines are provided to change and save TXTLIM and to save and restore EXPBUF/EXPLEN.
* CMDCOM CMDCOM - but note that the variables ERRNUM and RDCOMM_EXCP have been replaced by variables ERRNUM and LSL_EXCP in common file EXCEPTION, and the parameters defining error numbers are replaced by LSL__ error codes
* MTIVCM MTIVCM - all variables are now longwords
* MTOVCM MTOVCM - all variables are now longwords
* RDFILN FILENAME - lengths of fields is greater (to fit version 4)

EXPC, TXTC, and CMDCOM all retain their macro definitions, as well, with the same caveats as above.

B.6 *Specific routines*

B.6.1 *Comparison of GETFILNAM with RDFILT*

The main differences are:

- GETFILNAM is able to cope with a full VMS version 4 filename, whereas RDFILT would only cope with version 3 and below (new versions of VIOLIB may improve their file parsing).
- GETFILNAM reads the filename as a string, and then parses it, whereas RDFILT read the filename as it parsed it.

  Thus GETFILNAM parses a filename after reading it from the line, whilst RDFILT parsed the filename as it read it from the line. It is conceivable that this could produce different results from the two routines, but only in pathological cases.

B.6.2 *Using READSTR instead of RDSTR*

It is possible to produce the effects of using the RDSTR function with READSTR. The equivalent calls are as follows:

- length = RDSTR(string,char), or RDSTR(string,char,.TRUE.)

  Terminate a string on a particular character, with double that character inserting it once - use a call of

  ```
  length = READSTR( string, char, ON_CHAR2, .FALSE.  )
  ```

- length = RDSTR(string,char,.FALSE.)

  Terminate a string as soon as a particular character is found - use a call of

  ```
  length = READSTR( string, char, ON_CHAR, .FALSE.  )
  ```

- length = RDSTR(string,0)

  Terminate a string as if it were a CMDLIB command - use a call of

  ```
  length = READSTR( string, 0, ON_CMD, .TRUE.  )
  ```

- length = RDSTR(string,' ')

  Terminate a string at the first space or tab - use a call of

  ```
  length = READSTR( string, 0, ON_SPACE, .TRUE.  )
  ```

- RDSTR(string,-1)

  Terminate a string at the end of the line - use a call of

  ```
  length = READSTR( string, 0, ON_EOL, .TRUE.  )
  ```

INDEX