DATAPOINT

DASL™

User's Guide

50807

NOVEMBER 14, 1984

Document No. 50807-01. 11/84
Copyright © 1984 by DATAPOINT Corporation. All rights reserved The "D" logo, DATAPOINT, DATABUS, DATAFORM, DATAPOLL, DATASHARE, Lightlink, Integrated Electronic Office, DATACCOUNTANT, ARC, Attached Resource Computer and ARCNET are trademarks of DATAPOINT Corporation registered in the U.S. Patent and Trademark Office. AIM, Associative Index Method, ARCGATE, ARCLINK, DASP, RMS, Resource Management System, EMS, DASL, RASL, NOSL, EASL and DATASORT are trademarks of DATAPOINT Corporation.
System features and technical details are subject to change without notice.

Preface

The DASL USER's GUIDE, Vol. III, is another reference in the DASL library. Vol. III describes the NOSL I/O package.

This work is a DATEC production. NOSL I/O was written on a DATAPOINT processor using an experimental writing tool. Then, the masters for printing were produced on a DATAPOINT 9660 Laser Printer.

Please forward your comments on this document to:

DATEC Publications
DATAPOINT Corporation
9725 Datapoint Dr. MS T-72
San Antonio, Texas 78284

TABLE OF CONTENTS

Chapter 1.
INTRODUCTION
OVERVIEW

About this manual

This manual describes the NOSL I/O package. It contains general information about the NOSL I/O package for programmers who are unfamiliar with NOSL, and contains instructions for including NOSL, calling sequences for the NOSL functions, and descriptions of the internal memory manager and error handler.

Intended audience

This manual is intended for programmers writing software in DASL on the RMS operating system.

It contains code segments of DASL programs that reference the RMS operating system.

How this manual is organized

This manual is divided into seven chapters.

Chapter	*Content*
1	An introduction to the NOSL I/O package.
2	The procedures for including the NOSL I/O package into your DASL program.
3	A description of the NOSL functions.
4	A description of the different types of nosls and instructions for opening a nosl.
5	A description of the numeric scanners that are part of the NOSL I/O package.
6	A description of the NOSL internal memory manager and its relationship to virtual files and overlays.
7	A description of the NOSL error handler and instructions for customizing error handling.

INTRODUCTION TO NOSL

Introduction

As a DASL programmer, you are probably aware that DASL has no internal I/O facility. In order to generate I/O, you must use one of the RMS I/O packages, or create your own I/O package.

NOSL has been written as an easy to use, but powerful I/O package for the DASL programming language.

NOSL sets up tables and buffers

Performing I/O under RMS often requires you to provide memory for tables and buffers. The fields in these tables must in turn be initialized.

NOSL allocates memory for these tables and buffers, and initializes the tables with common default values, while giving you the flexibility to override the default values.

NOSL has common interfaces called nosls

NOSL provides a common interface to access a variety of devices.

Individual modules, called nosls, interface to the different devices. For every device supported by the NOSL I/O package, there is a corresponding nosl.

NOSLS AND STREAMS

Introduction

The term "device" is often used to refer to printers, keyboards, and screens. In the previous section we learned that there is a corresponding nosl for every device supported by the NOSL I/O package.

In this section, we will find out that a nosl can also interface to another nosl. Because "device" does not accurately describe this interface, the term "stream" will be used to describe all NOSL interfaces instead.

Definition of a stream

A stream is anything whose contents can be treated as a sequence of bytes. Some examples of streams are

- a text file,
- a printer,
- a keyboard, and
- a nosl.

n$Stream type

NOSL provides a type, n$Stream, to describe a stream. The NOSL open function returns a variable of this type, while the other NOSL I/O functions require a variable of this type as an input parameter.

Example:

```
in n$Stream;
```

Streams are sequences of bytes

A stream is treated as a sequence of bytes.

The length of a stream is always measured as the number of bytes from the beginning to the ending of a stream.

Stream pointers

NOSL maintains a pointer to the current byte position of each stream. This stream pointer advances whenever reading or writing is performed on the stream, and can be examined and changed on certain streams.

Note: The first byte in a stream is at position zero. The stream pointer is generally set to zero when a stream is opened.

Nosls translate data to and from streams

Nosls are used to translate data to and from a stream. Since data is formatted differently within each stream, the nosls will

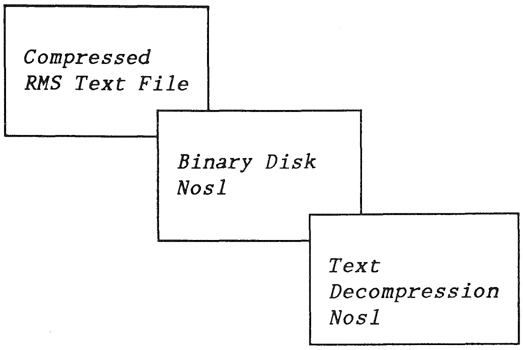
- remove device dependencies from input streams, and
- add device dependencies into output streams.

Example: The Decomp nosl translates compressed text from an input stream into uncompressed text. The CmpOut nosl translates uncompressed text into compressed text.

Nosls may be layered

Nosls may be layered on top of one another if more than one translation of a stream is needed.

Example: In order to read a standard RMS text file, a text decompression nosl must be layered on top of a binary disk nosl which inputs the compressed RMS text file.



Chapter 2.
INCLUDING NOSL

OVERVIEW

Introduction

This chapter contains the procedures that are needed to include the NOSL I/O package into your DASL program.

Overview of the procedure

The procedures for including the NOSL I/O package are outlined in the following table. A description of each procedure in this table is included in this chapter.

Task	*Action*
1	Include the NOSL definition file into your DASL program.
2	Include the NOSL relocatable library in your LINK directives.

INCLUDE NOSL DEFINITION FILE

Introduction

The NOSL definition file must be included in the INCLUDE portion of your DASL program.

Task	*Action*
1	Include the NOSL definition file into your DASL program.

What is the NOSL definition file

The NOSL definition file is a text file that contains definitions and external references that are needed to reference the NOSL I/O package.

Its name is "N$/DEFS".

Example

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(N$/DEFS)
:
: (rest of the program)
```

INCLUDE NOSL RELOCATABLE LIBRARY

Introduction

The NOSL relocatable library must be included by the LIBRARY directive in LINK.

Task	*Action*
2	Include the NOSL relocatable library in your LINK directives.

Selecting the correct NOSL library

There are a couple of copies of the NOSL relocatable library that support two of the DATAPOINT instruction sets. Use this table to select the proper NOSL library for your instruction set.

If you are writing for the...	*then use...*
5500 instruction set	N$/REL5
6600 instruction set	N$/REL

Also include the CUFs library

NOSL uses several of the common user functions, CUFs. The appropriate CUFs relocatable library must be included after the NOSL relocatable library in your LINK directives.

Example

If you had selected N$/REL, your LINK directives might look like this:

```
SEGMENT MYPROG
INCLUDE DASLASM
INCLUDE D$LIB.D$START
LIBRARY N$
LIBRARY $CUFS
LIBRARY D$LIB
LIBRARY RMSUFRS
*
```

Chapter 3.
THE NOSL FUNCTIONS
OVERVIEW

Introduction

This chapter contains descriptions of the eleven NOSL functions that may be called by your DASL program to access the nosls.

Description

The NOSL functions are a set of functions that give you the ability to open, close, and manipulate a stream.

Note: A description of the individual nosls are found in the next chapter.

Parameterization of the NOSL functions

Parameters in some of the NOSL functions are optional.

Optional parameters are enclosed in square brackets for identification in the function diagrams. They may be ignored by omitting them completely, or by replacing them with a comma if a later parameter is to be specified.

NUMBER type

NOSL defines a type, NUMBER, which represents the addressable memory area of a machine. This type is used in the NOSL read and write functions.

On machines with 16 bit pointers, NUMBER is defined as follows:

Type Definition	*Description*
`TYPDEF NUMBER UNSIGNED;`	Addressable memory area of a machine.

Note: On machines with 32 bit pointers, NUMBER is defined as LONG.

Selecting a NOSL function

Use the following table to determine which NOSL function to use.

If you want to...	*then use...*	*page*
open a stream	n$open.	3-4
read from a stream	n$read.	3-5
write to a stream	n$write.	3-6
determine position of a stream	n$tell.	3-7
position to a location in a stream	n$seek	3-8
determine the size of the stream	n$size	3-9
truncate a stream	n$trunc	3-10
determine the status of a stream	n$status	3-11
empty internal buffers of a stream	n$flush	3-13
close a stream	n$close	3-14
output formatted data to a stream	n$format	3-15

n$open function

n$open opens a stream.

Note: The n$open function is the only function that is parameterized differently for different nosls. See the next chapter for the different parameterizations.

All stream types support n$open.

Function Syntax	*Input*	*Output*
n$open (st StreamType,	Type of stream to open.	
[eF ∧ n$ErrorFunc],	Address of a function to call if an error occurs. Note: See Error Handling.	
[eD ∧ n$ErrorData],	Address of data to pass into the error function. Note: See Error Handling.	
[other parameters]	Other parameters depending on the type of stream opened.	
) n$Stream;		Opened stream. NIL if unable to open the stream.

See the next chapter for examples.

n$read function

n$read reads bytes from an input stream.

Some nosls do not support n$read.

Function Syntax	*Input*	*Output*
n$read (s n$Stream,	Stream to read from.	
p ∧ BYTE,	Address of the first byte to store the data read from the stream.	
len NUMBER	Number of bytes to read.	
) NUMBER;		Number of bytes actually read. Note: 0 if end-of-file.

Example:

```
in n$Stream;

function () :=
VAR   string [81] CHAR;
      n BYTE;
{
   n := n$read(in, &string[0], SIZEOF string);
```

n$write function

n$write writes bytes to an output stream.

Note: If there is already data where the stream is positioned, it will be overwritten.

Some nosls do not support n$write.

Function Syntax	*Input*	*Output*
n$write (s n$Stream,	Stream to write to.	
p ^ BYTE,	Address of the first byte to write to the stream.	
len NUMBER);	Number of bytes to write.	

Example:

```
out n$Stream;

function () :=
VAR   STATIC s [] CHAR := 'Hello';
{
   n$write(out, &s[0], SIZEOF s);
```

n$tell function

n$tell returns the current position of the stream in bytes.

Some nosls do not support n$tell.

Function Syntax	*Input*	*Output*
n$tell (s n$Stream	Stream to determine the current byte position.	
) LONG;		Current byte position of the stream.

Example:

```
file n$Stream;

function () :=
VAR   pos LONG;
{
   pos := n$tell(file);
```

n$seek function

n$seek positions the stream to a certain byte position.

Some nosls do not support n$seek.

Function Syntax	*Input*	*Output*
n$seek (s n$Stream,	Stream to change the byte position.	
pos LONG	Byte location to position to.	
) LONG;		New byte position.

Example:

```
file n$Stream;

function () :=
VAR   pos LONG;
{
   pos := n$seek(file, 0);
```

n$size function

n$size returns the size of a stream in bytes.

Some nosls do not support n$size.

Function Syntax	*Input*	*Output*
n$size (s n$Stream	Stream to determine the size.	
) LONG;		Size of the stream in bytes.

Example:

```
file n$Stream;

function () :=
VAR   size LONG;
{
   size := n$size(file);
```

n$trunc function

n$trunc truncates the stream to the current position.

Some nosls do not support n$trunc.

Function Syntax	*Input*	*Output*
`n$trunc ( s n$Stream );`	Stream to truncate.	

Example:

```
file n$Stream;

function () :=
{
   n$trunc(file);
```

n$status

Description

n$status includes a type, n$Status, that defines the different stream statuses, and a function, n$status, that returns the current status of a stream.

n$Status type

n$Status is defined as follows:

Type Definition	*Description*
TYPDEF n$Status SETW (	
n$busy,	Stream is busy with another I/O.
n$EOR,	Stream is positioned between logical records.
n$EOF	Stream is positioned at the end of the stream. Note: Positioned just past the last character of the stream.
);	

n$status function

n$status returns the current status of the stream.

All nosls support n$status.

Function Syntax	*Input*	*Output*
n$status (s n$Stream	Stream to check the status.	
) n$Status;		Status of the stream.

Example:

```
file n$Stream;

function () :=
{
   LOOP {
    WHILE ~ ( n$status(file) && n$EOF );
```

n$flush function

n$flush flushes any internal buffers used by the stream.

- Read buffers are simply marked empty, and
- Write buffers are dumped to the underlying stream or device.

All nosls support n$flush.

Function Syntax	*Input*	*Output*
`n$flush ( s n$Stream );`	Stream to flush.	

Example:

```
out n$Stream;

function () :=
{
   n$flush(out);
```

n$close function

n$close completes all pending operations and closes the stream. Any storage that was allocated by the nosl is released.

All nosls support n$close.

Function Syntax	*Input*	*Output*
n$close (s n$Stream,	Stream to close.	
mode BYTE);	RMS close mode unless otherwise stated by the particular nosl.	

Example:

```
out n$Stream;

function () :=
{
   n$close(out, $CMCHOP);
```

n$format

Description

The n$format function outputs formatted data to an output stream.

n$format is actually a macro that calls other NOSL functions to perform various operations.

n$format function

n$format outputs formatted data to the output stream.

Since n$format is a macro that calls n$write and n$flush, any nosl that does not support n$write and n$flush will not support n$format.

Function Syntax	*Input*	*Output*

`n$format (`
 `s n$stream,`
	Stream to write formatted data to.	

`[ other parameters ]`
 `);`
	Up to eight n$format output specifications.	

n$format output specifications

There are several n$format output specifications. Use the following table to select the type of formatted output needed for your data.

If you want to...	*see page...*
output a single character	3-17
output a string	3-18
output a quoted string	3-19
output a decimal number	3-20
output an octal number	3-21
output a hexadecimal number	3-22
output a real number	3-23
output an end of line character	3-27
n$flush the output stream	3-27

Single character output

The "C" output specification outputs up to seven single characters.

Note: If more than one character is specified, the characters are stored in a temporary array and written with a single n$write.

n$format Syntax	*Description*
C (character CHAR,	A single character to output to the output stream.
[characters]	Up to six more characters to output to the output stream separated by commas.
)	

Example:

```
function (v, h BYTE) :=
{
   n$format(out, C( $CP, v+$WSTL, h+$WSLC, '*' ) );
```

String output

The "S" output specification outputs an "object in memory" (typically an array of characters but can be an INT, STRUCT, etc).

n$format Syntax	*Description*
S (object in memory,	The string or object to output. Note: n$format expects the actual object and not the address of the object.
[length UNSIGNED])	Number of characters to output. Note: If this parameter is not specified, SIZEOF the object will be used as the length.

Example:

```
function () :=
VAR   STATIC array [] CHAR := 'Hello';
{
   n$format(out, S( array ) );
```

Quoted string output

The quoted string output specification outputs the characters within single quotation marks.

n$format Syntax	*Description*
'string'	Quoted string to output.

Example:

```
function () :=
{
   n$format(out, 'Hello' );
```

Decimal number output

The "D" output specification outputs the decimal value of a number in ASCII format.

n$format Syntax	*Description*
D (	
number LONG,	Number to output in decimal. <u>Note</u>: Negative numbers are output with a preceding minus sign.
[length UNSIGNED]	Minimum length of the field where the number is output. <u>Note</u>: The number is right justified and padded with blanks on the left if it is smaller than the minimum length.
)	

<u>Example</u>:

```
function () :=
{
  n$format(out, D( 4000, 10 ) );
```

<u>Result</u>:

```
      4000
```

Octal number output

The "O" output specification outputs the octal value of a number in ASCII format.

n$format Syntax	*Description*
O (number LONG,	Number to output in octal. Note: The number is treated as an unsigned LONG. That means a -1 will be output as a 037777777777. The number may be cast to a BYTE or UNSIGNED to suppress this.
[length UNSIGNED])	Minimum length of the field where the number is output. Note: The number is right justified and padded with zeros on the left if it is smaller than the minimum length.

Example:

```
function () :=
{
   n$format(out, O( 4000, 10 ) );
```

Result:

```
0000007640
```

Hexadecimal number output

The "H" output specification outputs the hexadecimal value of a number in ASCII format.

n$format Syntax	*Description*
H (number LONG,	Number to output in hexadecimal. Note: The number is treated as an unsigned LONG. That means a -1 will be output as a FFFFFFFF. The number may be cast to a BYTE or or UNSIGNED to suppress this.
[length UNSIGNED])	Minimum length of the field where the number is output. Note: The number is right justified and padded with zeros on the left if it is smaller than the minimum length.

Example:

```
function () :=
{
   n$format(out, H( 4000, 10 ) );
```

Result:

```
0000000FA0
```

Real number output

There are three real number output directives. Use the following table to determine which real output directive to use.

If you want to...	*see page...*
output a real number using scientific notation	3-24
output a real number using floating point notation	3-25
output a real number using the shorter of the two notations, scientific and floating point	3-26

Real number output in "e" format

The "Re" output specification outputs a real number in ASCII scientific notation.

n$format Syntax	*Description*
Re (	
number DOUBLE,	Real number to output in the form [-]m.nnnnnnE[-]xxx
places UNSIGNED,	Number of digits to output following the decimal point.
[length UNSIGNED]	Minimum length of the field where the number is output. Note: The number is right justified and padded with blanks on the left if it is smaller than the minimum length.
)	

Example:

```
function () :=
{
   n$format(out, Re( 1234.56789, 2, 20 ) );
```

Result:

```
              1.23E3
```

Real number output in "f" format

The "Rf" output specification outputs a real number in ASCII floating point notation.

n$format Syntax	*Description*
Rf (number DOUBLE,	Real number to output in the form [-]mmm.nnnnnn
places UNSIGNED,	Number of digits to output following the decimal point.
[length UNSIGNED])	Minimum length of the field where the number is output. Note: The number is right justified and padded with blanks on the left if it is smaller than the minimum length.

Example:

```
function () :=
{
   n$format(out, Rf( 1234.56789, 2, 20 ) );
```

Result:

```
            1234.57
```

Real number output in "g" format

The "Rg" output specification outputs a real number in the shorter of the two ASCII notations, scientific and floating point.

n$format Syntax	*Description*
Rg (	
number DOUBLE,	Real number to output in the "e" or "f" format, whichever is shorter. Note: Does not output insignificant zeros.
places UNSIGNED,	Number of digits to output following the decimal point.
[length UNSIGNED]	Minimum length of the field where the number is output. Note: The number is right justified and padded with blanks on the left if it is smaller than the minimum length.
)	

Example:

```
function () :=
{
   n$format(out, Rg( 1234.56789, 2, 20 ) );
```

Result:

```
                1.23E3
```

New line output

The "LN" output specification outputs a single $LEOR.

n$format Syntax	*Description*
LN	

Example:

```
function () :=
{
   n$format(out, LN );
```

Flush n$format internal buffers

The "F" output specification causes an n$flush to be performed on the output stream.

n$format Syntax	*Description*
F	

Example:

```
function () :=
{
   n$format(out, F );
```

Chapter 4. THE NOSLS

OVERVIEW

Introduction

This chapter contains a description of the individual nosls and a guide for selecting a nosl.

Each nosl is a tool

The philosophy of NOSL is that of a toolbox. Each nosl described in this chapter is a small, easy to understand tool, designed to handle a particular job.

Parameterization of the NOSL open call

Each of the nosls in this chapter will include a description of their nosl open calls. Several parameters in each nosl open call are optional.

Optional parameters are enclosed in square brackets for identification in the function diagrams. They may be ignored by omitting them completely, or by replacing them with a comma if a later parameter is to be specified.

NUMBER type

As mentioned in the previous chapter, NOSL defines a type, NUMBER, which represents the addressable memory area of a machine. In this chapter, this type is used to define the buffer size in the NOSL open function.

On machines with 16 bit pointers, NUMBER is defined as follows:

Type Definition	*Description*
`TYPDEF NUMBER UNSIGNED;`	Addressable memory area of a machine.

Note: On machines with 32 bit pointers, NUMBER is defined as LONG.

NOSL open error parameters

The NOSL open function contains two optional parameters (eF and eD) that may be used to customize error handling for each nosl. See the Error Handling chapter for further details.

SELECTING A NOSL

Overview

This section contains the description of a function that determines the device kind of an RMS filename, and a table of the different nosls available.

n$fKind function

n$fKind returns the device kind of an RMS filename.

This function is useful for programs that do not know what kind of input or output device to read or write until run time.

Function Syntax	*Input*	*Output*
n$fKind (sfentP ^ $SFENT,	Address of the file entry table of the file to determine the device kind.	
[pSubKind ^ BYTE],	Address of a byte.	Initialized with subkind code.
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData]	Address of error data.	
) BYTE;		RMS device kind. Example: $DKDISK

n$fKind function (continued)

Example: The following program segment scans an output device from the command line and calls an appropriate function based upon the device kind returned.

```
fileSpk $FILESPK := {
   { 'IN      ', '               ', '     ', '           ' },
     $FILNAMR,     $NOADR,        &'TEXT',&'          '
   };

function () :=
VAR   dK BYTE;
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   dK := n$fKind(&fileSpk.$FSOSFT);
   CASE dK {
      $DKPIPE  : openPipe();
      $DKPRINT : openPrinter();
      DEFAULT  : error();
      };
```

Which nosl?

Use the following table to determine which nosl to use.

If you want to...	*then use...*	*page*
input/output to a binary disk file	Disk, or DiskId nosl.	4-6 4-9
input from a binary disk file	DiskR nosl.	4-12
input/output to a virtual file	VirFil nosl.	4-15
provide a buffer for another nosl	BufPag nosl.	4-17
decompress an input text nosl	Decomp nosl.	4-19
compress to an output text nosl	CmpOut nosl.	4-22
input from the workstation	Workstation input nosls.	4-24
output to the workstation	Workstation output nosls.	4-34
output to a printer	Print nosl.	4-43
output to a pipe	Pipe nosl.	4-46
duplicate output to two nosls	Split nosl.	4-48
input/output to memory	Mem nosl.	4-50
ignore nosl operations	Null nosl.	4-52

Disk NOSL

Introduction

NOSL provides the Disk nosl for input and output of binary data to a disk file.

End-of-file pointer

The Disk nosl uses the RMS end-of-file pointer to define the end-of-file rather than the first occurrence of a $LEOF character.

NOSL function support

The Disk nosl supports all of the NOSL functions.

Disk nosl n$open

This function diagram describes the parameterization of the n$open for the Disk nosl.

Function Syntax	*Input*	*Output*
n$open (Disk,	Binary disk I/O stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
sfentP ∧ $SFENT,	Address of the file entry table of the file to open or create.	
[formatP ∧ BYTE],	Address of a byte. Specifies file format if creating a file. <u>Note</u>: Defaults to $FFMTTXT.	File format of file opened.
[mode BYTE],	RMS open mode. <u>Note</u>: Defaults to $OMREAD.	
[bufSize NUMBER],	Size of buffer to use in bytes. <u>Note</u>: Defaults to 2048.	
[openPTP ∧ $OPENPTP]	Address of an open parameter table. <u>Note</u>: Sector length and increment default to 200. formatP takes precedence over openPTP format.	
) n$Stream;		Opened stream.

Disk nosl n$open (continued)

Example:

```
out n$Stream;

fileSpk $FILESPK := {
   { 'OUT     ', '                ', '    ', '          ' },
     $FILNAMR,    $NOADR,        &'TEXT',&'          '
   };

function () :=
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   out := n$open(Disk,,, &fileSpk.$FSOSFT,, $OMPREP);
```

DiskId NOSL

Introduction

In addition to the Disk nosl, NOSL provides the DiskId nosl for input and output of binary data to a disk file that is already open.

The file is opened by FAVID

The DiskId nosl opens the disk file by its file access variable (FAVID).

Uses for the DiskId nosl

The DiskId nosl is useful for

- opening several nosls on the same file, or
- keeping several files open while using a single set of buffers.

NOSL function support

The DiskId nosl supports all of the NOSL functions.

NOSL function results

The following NOSL functions have unique results when used with the DiskId nosl.

Function	*Results*
n$close	closes the nosl, but not the file.

DiskId nosl n$open

This function diagram describes the parameterization of the n$open for the DiskId nosl.

Function Syntax	*Input*	*Output*
n$open (DiskId,	Binary disk FAVID I/O stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData],	Address of error data.	
id UNSIGNED,	File access variable ID of an open file.	
[bufSize NUMBER]	Size of buffer to use in bytes. Note: Defaults to 2048.	
) n$Stream;		Opened stream.

DiskId nosl n$open (continued)

Example: This program segment opens two nosls on one disk file.

```
fI $PFDB;

fileSpk $FILESPK := {
   { 'OUT      ', '                ', '     ', '           ' },
     $FILNAMR,     $NOADR,         &'TEXT',&'           '
   };

openPT $OPENPT := {
   &fI,
   &fileSpk.$FSOSFT.$SFTENV[0],
   &<$NAMEEXT>fileSpk.$FSOSFT.$SFTNAM
   };

in1 n$Stream;
in2 n$Stream;

function () :=
VAR    fav UNSIGNED;
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   IF $OPEN($OMREAD, &openPT) && D$CFLAG THEN
      $ERMSG();
   fav := openPT.$OTPFDB^.$PFVID;
   in1 := n$open(DiskId,,, fav, 4096);
   in2 := n$open(DiskId,,, fav, 256);
```

DiskR NOSL

Introduction

NOSL provides the DiskR nosl for reading data from a disk file. It is a read only version of the Disk nosl.

NOSL function support

The DiskR nosl supports all of the NOSL functions except

- n$write,
- n$trunc, and
- n$format.

DiskR nosl n$open

This function diagram describes the parameterization of the n$open for the DiskR nosl.

Function Syntax	*Input*	*Output*
n$Open (DiskR,	Binary disk input stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
sfentP ∧ $SFENT,	Address of the file entry table of the file to open.	
[formatP ∧ BYTE],	Address of a byte.	File format of file opened.
[mode BYTE],	RMS open mode. Note: Defaults to $OMREAD.	
[bufSize NUMBER],	Size of buffer to use in bytes. Note: Defaults to 2048.	
[openPTP ∧ $OPENPTP]	Address of an open parameter table.	
) n$Stream;		Opened stream.

DiskR nosl n$open (continued)

Example:

```
in n$Stream;

fileSpk $FILESPK := {
   { 'IN      ', '            ', '     ', '         ' },
     $FILNAMR,    $NOADR,       &'TEXT',&'         '
   };

function () :=
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   in := n$open(DiskR,,, &fileSpk.$FSOSFT,,,4096);
```

VirFil NOSL

Introduction

NOSL provides the VirFil nosl as a virtual file facility. The VirFil nosl simulates a virtual file for both input and output.

Definition of a virtual file

A virtual file is a file that looks and acts like a real file, except it is maintained in memory. Input and output to a virtual file is therefore much faster than a disk file.

Note: The memory used by a virtual file is released by the virtual file handler when the virtual file is closed. If more than one virtual file is in use, the memory will not be released until all of the virtual files have been closed.

The spill file

If the virtual file runs out of memory, a "spill file" is created on disk to hold the data that did not fit in memory. Multiple virtual files will all use the same spill file if they run out of memory.

Note: NOSL will delete the spill file when all virtual files are closed.

Improving performance

VirFil may realize a performance improvement by opening a BufPag nosl over the VirFil nosl.

NOSL function support

The VirFil nosl supports all of the NOSL functions.

VirFil nosl n$open

This function diagram describes the parameterization of the n$open for the VirFil nosl.

Function Syntax	*Input*	*Output*
n$open (VirFil,	Virtual binary disk I/O stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData]	Address of error data.	
) n$Stream;		Opened stream.

Example:

```
virtual n$Stream;

function () :=
{
   virtual := n$open(VirFil);
```

BufPag NOSL

Introduction

NOSL provides the BufPag nosl for buffering the data input or output from another nosl.

Performance improvement

The BufPag nosl is useful for improving performance by reducing the number of operations of another nosl.

Note: The potential improvement in performance is bought at the price of memory allocated from your address space.

NOSL function support

The BufPag nosl supports all of the NOSL functions except n$trunc.

NOSL function results

The following NOSL functions have unique results when used with the BufPag nosl.

Function	*Results*
n$open	the position of the nosl stream is the same as the position of the underlying stream.
n$close	the nosl is closed, but the underlying stream is not.

BufPag nosl n$open

This function diagram describes the parameterization of the n$open for the BufPag nosl.

Function Syntax	*Input*	*Output*
n$open (BufPag,	Page aligned buffer stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
under n$Stream,	Stream to buffer.	
[bufSize NUMBER]	Size of buffer to use in bytes. <u>Note</u>: Defaults to 256.	
) n$Stream;		Opened stream.

<u>Example</u>:

```
buff n$Stream;

function () :=
VAR   virtual n$Stream;
{
   virtual := n$open(VirFil);
   buff := n$open(BufPag,,, virtual, 512);
```

Decomp NOSL

Introduction

NOSL provides Decomp as a text decompression nosl for decompressing text read in from one of the binary disk nosls.

End-of-file pointer

The Decomp nosl defines the end-of-file as the position specified by the RMS end-of-file pointer or the first occurrence of a $LEOF character, whichever comes first.

NOSL function support

The Decomp nosl supports all of the NOSL functions except

- n$write,
- n$trunc, and
- n$format.

NOSL function results

The following NOSL functions have unique results when used with the Decomp nosl.

Function	*Results*
n$open	the position of the nosl stream is the same as the position of the underlying stream.
n$read	• stops reading if a $LEOR character or the maximum number of characters is read. • sets the n$EOF status but does not return the $LEOF if a $LEOF terminates the read rather than a $LEOR. The n$EOR status is not set in this situation.
n$tell	• only valid when n$EOR status is set. • returns position of $LEOF if current position and $LEOF are separated by $LDELs or $LEOBs.
n$status	returns n$EOF status if current position and $LEOF are separated by $LDELs or $LEOBs.
n$seek n$flush	force n$EOR status.
n$close	the nosl is closed, but the underlying stream is not.

Decomp nosl n$open

This function diagram describes the parameterization of the n$open for the Decomp nosl.

Function Syntax	*Input*	*Output*
n$open (Decomp,	Text decompression stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData],	Address of error data.	
under n$Stream	Text stream to decompress.	
) n$Stream;		Opened stream.

Example:

```
in n$Stream;
decomp n$Stream;

fileSpk $FILESPK := {
   { 'IN       ', '                ', '     ', '          ' },
     $FILNAMR,     $NOADR,        &'TEXT',&'          '
   };

function () :=
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   in := n$open(DiskR,,, &fileSpk.$FSOSFT,,,4096);
   decomp := n$open(Decomp,,, in);
```

CmpOut NOSL

Introduction

NOSL provides CmpOut as a text compression nosl for writing compressed text to one of the binary disk output nosls.

NOSL function support

The CmpOut nosl supports all of the NOSL functions except n$read.

NOSL function results

The following NOSL functions have unique results when used with the CmpOut nosl.

Function	*Results*
n$open	the position of the nosl stream is the same as the position of the underlying stream.
n$tell n$seek n$size n$trunc	forces a n$flush before performing normal operations.
n$close	the nosl is closed, but the underlying stream is not.

CmpOut nosl n$open

This function diagram describes the parameterization of the n$open for the CmpOut nosl.

Function Syntax	*Input*	*Output*
n$open (CmpOut,	Text compression stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData],	Address of error data.	
under n$Stream	Text stream to output compressed text to.	
) n$Stream;		Opened stream.

Example:

```
out n$Stream;
cmp n$Stream;

fileSpk $FILESPK := {
   { 'OUT      ', '                ', '     ', '          ' },
     $FILNAMR,     $NOADR,          &'TEXT',&'          '
   };

function () :=
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   out := n$open(Disk,,, &fileSpk.$FSOSFT,, $OMPREP);
   cmp := n$open(CmpOut,,, out);
```

WORKSTATION INPUT NOSLS

Introduction

NOSL provides several input nosls to input data from the workstation.

The workstation input nosls are WSIn, ChaIn, BWSIn, and BChaIn.

Unbuffered and buffered input

The workstation input nosls can be separated into unbuffered, WSIn and ChaIn, and buffered, BWSIn and BChaIn input nosls. When doing an n$read,

- the unbuffered nosls will read up to 81 characters from the workstation. They will accept no more than the number of characters specified by the n$read.
- the buffered nosls will read up to 81 characters to fill the buffer and then return the number of characters specified by the n$read. If more characters were entered into the buffer than were specified by the n$read, the characters are retained in the buffer and returned on subsequent reads.

Predefined stream constants

Each of the workstation nosls have a corresponding predefined stream constant. These constants may be used as the stream parameter to the NOSL functions.

The only reason to actually "open" the workstation nosls, instead of using these stream constants, is to specify an optional parameter with a value other than their defaults.

NOSL function support

The workstation input nosls support all of the NOSL functions except

- n$write,
- n$tell,
- n$seek,
- n$size,
- n$trunc, and
- n$format.

Which nosl?

Use the following table to determine which nosl to use.

If you want to...	*then use...*	*page*
input data from the keyboard	WSIn.	4-26
input data from either the keyboard or chain file if chaining is active	ChaIn.	4-28
buffer input from the keyboard	BWSIn.	4-30
buffer input from either the keyboard or chain file if chaining is active	BChaIn.	4-32

WSIn nosl

The WSIn nosl inputs from the keyboard.

n$WSIn constant

n$WSIn is a constant that is available for you to use.

Constant Definition	*Description*
n$WSIn n$Stream;	Predefined WSIn input stream.

Example:

```
function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   n := n$read(n$WSIn, &s[0], SIZEOF s);
```

WSIn nosl n$open

This function diagram describes the parameterization of the n$open for the WSIn nosl.

Note: This function will return a stream with the same value as the n$WSIn constant described above.

Function Syntax	*Input*	*Output*
n$open (WSIn,	Workstation input stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData],	Address of error data.	
[noEnter BOOLEAN]	TRUE disables a $LEOR from being written to the workstation every time one is read from the keyboard. Note: Default is FALSE.	
) n$Stream;		Opened stream.

Example:

```
in n$Stream;

function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   in := n$open(WSIn,,, TRUE);
   n := n$read(in, &s[0], SIZEOF s);
```

ChaIn nosl

The ChaIn nosl inputs from either the keyboard or chain file if chaining is active.

n$ChaIn constant

n$ChaIn is a constant that is available for you to use.

Constant Definition	*Description*
n$ChaIn n$Stream;	Predefined ChaIn input stream.

Example:

```
function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   n := n$read(n$ChaIn, &s[0], SIZEOF s);
```

ChaIn nosl n$open

This function diagram describes the parameterization of the n$open for the ChaIn nosl.

Note: This function will return a stream with the same value as the n$ChaIn constant described above.

Function Syntax	*Input*	*Output*
n$open (ChaIn,	Chain input stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
[noEnter BOOLEAN]	TRUE disables a $LEOR from being written to the workstation every time one is read from the keyboard or chain file. Note: Default is FALSE.	
) n$Stream;		Opened stream.

Example:

```
in n$Stream;

function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   in := n$open(ChaIn,,, TRUE);
   n := n$read(in, &s[0], SIZEOF s);
```

BWSIn nosl

The BWSIn nosl buffers input from the keyboard.

n$BWSIn constant

n$BWSIn is a constant that is available for you to use.

Constant Definition	*Description*
n$BWSIn n$Stream;	Predefined BWSIn input stream.

Example:

```
function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   n := n$read(n$BWSIn, &s[0], SIZEOF s);
```

BWSIn nosl n$open

This function diagram describes the parameterization of the n$open for the BWSIn nosl.

Note: This function will return a stream with the same value as the n$BWSIn constant described above.

Function Syntax	*Input*	*Output*
n$open (BWSIn,	Buffered workstation input stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData]	Address of error data.	
) n$Stream;		Opened stream.

Example:

```
in n$Stream;

function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   in := n$open(BWSIn, &errorF);
   n := n$read(in, &s[0], SIZEOF s);
```

BChaIn nosl

The BChaIn nosl buffers input from either the keyboard or chain file if chaining is active.

n$BChaIn constant

n$BChaIn is a constant that is available for you to use.

Constant Definition	*Description*
n$BChaIn n$Stream;	Predefined BChaIn input stream.

Example:

```
function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   n := n$read(n$BChaIn, &s[0], SIZEOF s);
```

BChaIn nosl n$open

This function diagram describes the parameterization of the n$open for the BChaIn nosl.

Note: This function will return a stream with the same value as the n$BChaIn constant described above.

Function Syntax	*Input*	*Output*
n$open (BChaIn,	Buffered chain input stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData]	Address of error data.	
) n$Stream;		Opened stream.

Example:

```
in n$Stream;

function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   in := n$open(BChaIn, &errorF);
   n := n$read(in, &s[0], SIZEOF s);
```

WORKSTATION OUTPUT NOSLS

Introduction

NOSL provides several output nosls to output data to the workstation.

The workstation output nosls are WSOut, Line25, and LogOut.

Common buffer

The workstation output nosls share a common buffer. This buffer is flushed when

- the buffer is full,
- an n$read operation is performed by the n$WSIn or the n$ChaIn nosl,
- an $LEOR character is written, or
- a write is performed on one of the other two output nosls.

Note: The common buffer is initialized with a $NL (newline character) which causes the first output to the screen to be written on its own line.

Writing $LEORs

When a $LEOR is written to one of the workstation output nosls,

- the common buffer is flushed first, and
- a $NL is left as the first character in the common buffer.

Note: This treatment of $LEORs enables you to write to the workstation exactly as if it was a file.

WSIO control code support

The following WSIO control codes are not supported by the workstation output nosls.

- $WSIS,
- $WSISI,
- $WSIN,
- $WSINI,
- $WSIDOCS, and
- $WSCONFD.

Predefined stream constants

Each of the workstation nosls have a corresponding predefined stream constant. These constants may be used as the stream parameter to the NOSL functions.

The only reason to actually "open" the workstation nosls, instead of using these stream constants, is to specify optional parameters with values other than their defaults.

NOSL function support

The workstation output nosls support all of the NOSL functions except

- n$read,
- n$tell,
- n$seek, and
- n$size.

Note: n$trunc is supported but performs no function.

Which nosl?

Use the following table to determine which nosl to use.

If you want to...	*then use...*	*page*
output to the workstation screen	WSOut.	4-37
output to a pseudo 25th line on the workstation screen	Line25.	4-39
output to the workstation screen and log file if logging is active	LogOut.	4-41

WSOut nosl

The WSOut nosl outputs to the workstation screen.

n$WSOut constant

n$WSOut is a constant that is available for you to use.

Constant Definition	*Description*
n$WSOut n$Stream;	Predefined WSOut output stream.

Example:

```
function () :=
{
   n$format(n$WSOut, 'Welcome to NOSL', LN);
```

WSOut nosl n$open

This function diagram describes the parameterization of the n$open for the WSOut nosl.

Note: This function will return a stream with the same value as the n$WSOut constant described above.

Function Syntax	*Input*	*Output*
`n$open (` `WSOut,`	Workstation output stream indicator.	
`[ eF ∧ n$ErrorFunc ],`	Address of error function.	
`[ eD ∧ n$ErrorData ]`	Address of error data.	
`) n$Stream;`		Opened stream.

Example:

```
out n$Stream;

function () :=
{
   out := n$open(WSOut, &errorF);
   n$format(out, 'Welcome to NOSL', LN);
```

Line25 nosl

The Line25 nosl outputs to a pseudo 25th line on the workstation screen for status and operator entertainment messages.

After a message has been displayed by this nosl, the common buffer is primed with a $EEOL (erase to end of line) which will cause the message to disappear when something is written by one of the other workstation output nosls.

n$Line25 constant

n$Line25 is a constant that is available for you to use.

Constant Definition	*Description*
n$Line25 n$Stream;	Predefined Line25 output stream.

Example:

```
function () :=
{
   n$format(n$Line25, 'Waiting for a message', LN);
```

Line25 nosl n$open

This function diagram describes the parameterization of the n$open for the Line25 nosl.

Note: This function will return a stream with the same value as the n$Line25 constant described above.

Function Syntax	*Input*	*Output*
n$open (Line25,	Pseudo 25th line output stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData]	Address of error data.	
) n$Stream;		Opened stream.

Example:

```
out n$Stream;

function () :=
{
   out := n$open(Line25, &errorF);
   n$format(out, 'Waiting for a message', LN);
```

LogOut nosl

The LogOut nosl outputs to the workstation screen and log file if logging is active.

n$LogOut constant

n$LogOut is a constant that is available for you to use.

Constant Definition	*Description*
n$LogOut n$Stream;	Predefined LogOut output stream.

Example:

```
function () :=
{
   n$format(n$LogOut, 'Processing labels', LN);
```

LogOut nosl n$open

This function diagram describes the parameterization of the n$open for the LogOut nosl.

Note: This function will return a stream with the same value as the n$LogOut constant described above.

Function Syntax	*Input*	*Output*

`n$open (`
`LogOut,`
	Log output stream indicator.	

`[ eF ∧ n$ErrorFunc ],`	Address of error function.	
`[ eD ∧ n$ErrorData ]`	Address of error data.	
`) n$Stream;`		Opened stream.

Example:

```
out n$Stream;

function () :=
{
   out := n$open(LogOut, &errorF);
   n$format(out, 'Processing labels', LN);
```

Print NOSL

Introduction

NOSL provides the Print nosl to output data to a printer.

NOSL function support

The Print nosl supports all of the NOSL functions except

- n$read,
- n$tell,
- n$seek,
- n$size, and
- n$trunc.

Print nosl n$open

This function diagram describes the parameterization of the n$open for the Print nosl.

Function Syntax	*Input*	*Output*
n$open (Print,	Printer output stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
sfentP ∧ $SFENT,	Address of file entry table to open.	
[bufSize NUMBER],	Size of buffer to use in bytes. Note: Defaults to optimum size declared by the printer.	
[offLineF ∧ D$CALLF]	Address of a function to call if the printer is offline or out of paper. Note: Will display "Printer not ready" through Line25 nosl if function not specified.	
) n$Stream;		Opened stream.

Print nosl n$open (continued)

Example:

```
prt n$Stream;
prtSFT $SFENT := {
   'PRT     ', '$LP           ', 'PRT ', 'L         '
   };

function () :=
{
   prt := n$open(Print,,, &prtSFT);
```

Pipe NOSL

Introduction

NOSL provides the Pipe nosl to input and output data to a pipe.

NOSL function support

The Pipe nosl supports all of the NOSL functions except

- n$tell,
- n$seek,
- n$size, and
- n$trunc.

NOSL function results

The following NOSL functions have unique results when used with the Pipe nosl.

Function	*Results*
n$read	• causes a flush of any pending write data. • stops after the last character in the pipe message or the maximum number of characters is read. If there is no current pipe message when a n$read is performed, an RMS physical read is performed on the pipe.
n$write n$format	discards any remaining read data.
n$status	• returns a n$EOR status when an entire pipe message has been read. • There is no n$EOF status for pipes.

Pipe nosl n$open

This function diagram describes the parameterization of the n$open for the Pipe nosl.

Function Syntax	*Input*	*Output*
n$open (Pipe,	Pipe I/O stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
sfentP ∧ $SFENT,	Address of file entry table to open.	
[bufSize NUMBER],	Size of buffer to use in bytes. <u>Note</u>: Defaults to 256.	
[timeout BYTE]	Number of seconds before a pipe operation will time out. <u>Note</u>: Default is $FOREVER.	
) n$Stream;		Opened stream.

<u>Example</u>:

```
pipe n$Stream;
pipeSFT $SFENT := {
   'PIPE    ', 'MY_PIPE_NAME', 'PIPE', '        '
   };

function () :=
{
   pipe := n$open(Pipe,,, &pipeSFT);
```

Split NOSL

Introduction

NOSL provides the Split nosl for duplicating output to two other nosls.

NOSL function support

The Split nosl supports all of the NOSL functions except

- n$read,
- n$tell,
- n$seek, and
- n$size.

NOSL function results

The following NOSL functions have unique results when used with the Split nosl.

Function	*Results*
n$trunc	performs n$trunc on both "under" nosls.
n$status	always returns 0.

Split nosl n$open

This function diagram describes the parameterization of the n$open for the Split nosl.

Function Syntax	*Input*	*Output*
n$open (Split,	Output splitter stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
underl n$Stream,	First stream to split the output data to.	
under2 n$Stream	Second stream to split the output data to.	
) n$Stream;		Opened stream.

Example: The following program segment opens a split nosl that will duplicate output to a disk file and the workstation screen.

```
outD n$Stream;
out n$Stream;

fileSpk $FILESPK := {
   { 'OUT     ', '             ', '    ', '        ' },
     $FILNAMR,    $NOADR,       &'TEXT',&'         '
   };

function () :=
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   outD := n$open(Disk,,, &fileSpk.$FSOSFT,, $OMPREP);
   out := n$open(Split,,, n$WSOut, outD);
```

Mem NOSL

Introduction

NOSL provides the Mem nosl for input and output to a block of memory.

Uses for the Mem nosl

The Mem nosl is primarily useful in conjunction with n$format for producing strings of text, as well as decimal, octal, hexadecimal, and floating numbers in memory.

NOSL function support

The Mem nosl supports all of the NOSL functions.

Mem nosl n$open

This function diagram describes the parameterization of the n$open for the Mem nosl.

Function Syntax	*Input*	*Output*
n$open (Mem,	Memory I/O stream indicator.	
[eF ^ n$ErrorFunc],	Address of error function.	
[eD ^ n$ErrorData],	Address of error data.	
buffP ^ BYTE,	Address of the first byte of memory to use.	
buffSize UNSIGNED	Number of bytes in the memory buffer.	
) n$Stream;		Opened stream.

Example:

```
mem n$Stream;
buffer [80] BYTE;

function () :=
{
   mem := n$open(Mem,,, &buffer[0], SIZEOF buffer);
```

Null NOSL

Introduction

NOSL provides a nosl that will ignore all nosl operations.

The Null nosl is a "do nothing" nosl. All operations to the Null nosl are ignored.

Uses for the Null nosl

The null nosl may be used as a replacement of an optional disk file or printer nosl.

Example: A program that creates an output file if a certain option is specified on the command line can use the Null nosl if the option is not specified. This will alleviate having to check whether the option was set or not before every write, flush, etc...

Predefined stream constant

The Null nosl has a corresponding predefined stream constant. This constant may be used as the stream parameter to the NOSL functions.

The only reason to actually "open" the Null nosl, instead of using this stream constant, is to specify an optional parameter with a value other than the defaults.

NOSL function support

The Null nosl supports all of the NOSL functions.

NOSL function results

The following NOSL functions have unique results when used with the Null nosl.

Function	*Results*
n$open n$write n$trunc n$flush n$close n$format	does nothing.
n$read n$tell n$seek n$size	returns 0.
n$status	returns status n$EOR and n$EOF.

n$Null constant

n$Null is a constant that is available for you to use.

Constant Definition	*Description*
n$Null n$Stream;	Predefined Null input stream.

Example:

```
function () :=
VAR    n BYTE;
       s [81] CHAR;
{
   n := n$read(n$Null, &s[0], SIZEOF s);
```

Null nosl n$open

This function diagram describes the parameterization of the n$open for the Null nosl.

Function Syntax	*Input*	*Output*
n$open (Null,	Null stream indicator.	
[eF ∧ n$ErrorFunc],	Address of error function.	
[eD ∧ n$ErrorData],	Address of error data.	
) n$Stream;		Opened stream.

Example:

```
prt n$Stream;
prtSFT $SFENT := {
   'PRT     ', '$LP         ', 'PRT ', 'L       '
   };

optP $OPTION  := { 'PRINT   ', 0, $OPTVCLR, 0 };
optT $OPTTAIL := { $OPTTERM, 0 };

function () :=
{
   IF $SCANOS(&optP) && D$CFLAG THEN $ERMSG();
   prt := optP.$OPTFLG && $OPTFDEF
     ? n$open(Print,,, &prtSFT)
     : n$open(Null);
```

Chapter 5. NUMERIC SCANNERS

OVERVIEW

Introduction

In addition to the actual NOSL I/O functions, several numeric scanning functions are provided in the NOSL library.

Operate on strings in memory

For reasons of both efficiency and flexibility, the numeric scanning functions deal with strings in memory rather than directly with streams.

String and field scanners

The numeric scanning functions are divided into two categories,

- string scanners, and
- field scanners.

The following pages contain a detailed description of the numeric string and field scanners.

STRING SCANNERS

Description

The string scanners parse a string in memory looking for a number.

Numeric string format

The numeric string may contain leading

- blanks,
- minus sign, and
- in the case of hexadecimal, the characters "0X" or "0x".

The number must be terminated by a

- space,
- comma, or
- $LEOR.

String scanning functions

Use the following table to determine which string scanning function to use.

If you want to...	*then use...*	*page*
scan a decimal string	n$scanD.	5-3
scan an octal string	n$scanO.	5-4
scan a hexadecimal string	n$scanH.	5-5
scan a real string	n$scanR.	5-6

n$scanD function

n$scanD converts a decimal string number into its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanD (ppCh ^^ CHAR,	Address of a pointer to the first character in the string.	Points to the terminator if a number was found. Unchanged if no number.
pStatus ^ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) LONG;		Decimal number scanned. Note: 0 if number was not found.

Example:

```
function (pC ^ CHAR) :=
VAR   status BOOLEAN;
      d LONG;
{
   d := n$scanD(&pC, &status);
```

n$scanO function

n$scanO converts an octal string number into its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanO (ppCh ^^ CHAR,	Address of a pointer to the first character in the string.	Points to the terminator if a number was found. Unchanged if no number.
pStatus ^ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) LONG;		Octal number scanned. Note: 0 if number was not found.

Example:

```
function (pC ^ CHAR) :=
VAR    status BOOLEAN;
       o LONG;
{
   o := n$scanO(&pC, &status);
```

n$scanH function

n$scanH converts a hexadecimal string number into its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanH (ppCh ∧∧ CHAR,	Address of a pointer to the first character in the string.	Points to the terminator if a number was found. Unchanged if no number.
pStatus ∧ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) LONG;		Hexadecimal number scanned. Note: 0 if number was not found.

Example:

```
function (pC ∧ CHAR) :=
VAR   status BOOLEAN;
      h LONG;
{
   h := n$scanH(&pC, &status);
```

n$scanR function

n$scanR converts a real string number into its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanR (ppCh ^^ CHAR,	Address of a pointer to the first character in the string.	Points to the terminator if a number was found. Unchanged if no number.
pStatus ^ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) DOUBLE;		Floating point number scanned. Note: 0 if number was not found.

Example:

```
function (pC ^ CHAR) :=
VAR   status BOOLEAN;
      r DOUBLE;
{
   r := n$scanR(&pC, &status);
```

FIELD SCANNERS

Description

The field scanners parse a fixed field in memory looking for a number.

They are actually low level routines called by the string scanners.

Numeric field format

The numeric field may contain a leading

- minus sign, and
- in the case of hexadecimal, the characters "0X" or "0x".

The number must completely fill the field. No trailing spaces are allowed.

Field scanning functions

Use the following table to determine which field scanning function to use.

If you want to...	*then use...*	*page*
scan a decimal field	$scanD.	5-8
scan an octal field	$scanO.	5-9
scan a hexadecimal field	$scanH.	5-10
scan a real field	$scanR.	5-11

$scanD function

$scanD converts a fixed field, decimal string number to its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanD (pCh ∧ CHAR,	Address of the first character in the string.	
size UNSIGNED,	Width of the field size.	
pStatus ∧ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) LONG;		Decimal number scanned. Note: Undefined if number was not found.

Example:

```
function () :=
VAR    n BYTE;
       field [40] CHAR;
       d LONG;
       status BOOLEAN;
{
   n := n$read(n$WSIn, &field[0], SIZEOF field);
   d := $scanD(&field[0], n, &status);
```

$scanO function

$scanO converts a fixed field, octal string number to its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanO (pCh ∧ CHAR,	Address of the first character in the string.	
size UNSIGNED,	Width of the field size.	
pStatus ∧ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) LONG;		Octal number scanned. Note: Undefined if number was not found.

Example:

```
function () :=
VAR   n BYTE;
      field [40] CHAR;
      o LONG;
      status BOOLEAN;
{
   n := n$read(n$WSIn, &field[0], SIZEOF field);
   o := $scanO(&field[0], n, &status);
```

$scanH function

$scanH converts a fixed field, hexadecimal string number to its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanH (pCh ∧ CHAR,	Address of the first character in the string.	
size UNSIGNED,	Width of the field size.	
pStatus ∧ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) LONG;		Hexadecimal number scanned. <u>Note</u>: Undefined if number was not found.

<u>Example</u>:

```
function () :=
VAR    n BYTE;
       field [40] CHAR;
       h LONG;
       status BOOLEAN;
{
   n := n$read(n$WSIn, &field[0], SIZEOF field);
   h := $scanH(&field[0], n, &status);
```

$scanR function

$scanR converts a fixed field, real string number to its numerical equivalent.

Function Syntax	*Input*	*Output*
n$scanR (pCh ∧ CHAR,	Address of the first character in the string.	
size UNSIGNED,	Width of the field size.	
pStatus ∧ BOOLEAN	Address of a boolean.	TRUE if a number was found. FALSE otherwise
) DOUBLE;		Floating point number scanned. Note: Undefined if number was not found.

Example:

```
function () :=
VAR    n BYTE;
       field [40] CHAR;
       r DOUBLE;
       status BOOLEAN;
{
   n := n$read(n$WSIn, &field[0], SIZEOF field);
   r := $scanR(&field[0], n, &status);
```

Chapter 6.
MEMORY MANAGEMENT
OVERVIEW

Introduction

NOSL has an internal memory manager for managing the buffers, internal control blocks, and data space used by the nosls.

This chapter contains information about the memory manager and important considerations when using virtual files and overlays.

Uses $BUDDY and $BUFFER CUFs

NOSL does most of its memory management through the $BUDDY memory management CUF. The virtual file nosl, VirFil, is the only exception as it manages memory directly through the $$MEMGET and $$MEMREL UFRs.

The $BUFFER CUF, which calls the $BUDDY CUF, is also used for memory management.

Incompatibility with the memory management UFRs

Because the memory management UFRs allocate all available logical memory when initialized, they are not compatible with NOSL.

The $ALLOC and $BUDDY CUFs are compatible with NOSL and may be used instead.

VIRTUAL FILES

Introduction

The virtual file nosl, VirFil, will allocate as much memory as is needed to maintain the file in memory.

Memory limited by the PCR

The memory allocated by VirFil is limited by the maximum task memory limit set in the program communications region, PCR.

Caution when using virtual files

Because there is the potential for VirFil to allocate all of the available memory, you should insure that all of the streams you are going to use in a program are open before writing to a virtual file. Otherwise, NOSL may not have enough memory available to open the streams.

OVERLAYS

Introduction

DASL programs with overlays may use NOSL but should be careful when overlaying code.

Indirect calls through pointers

All of the NOSL functions, except n$open, do indirect calls through pointers. This means that the functions that are called indirectly must be accessible whenever the nosl is accessed.

Accessing a nosl in multiple overlays

If you have a nosl that is opened, and then overlayed with code that accesses the nosl, you must insure that the NOSL code is LINKed into an area of memory that is present during the execution of both overlays.

Chapter 7. ERROR HANDLING

OVERVIEW

Introduction

NOSL contains internal error handling routines as well as a provision for customizing error handling.

This chapter describes the NOSL error handler and contains instructions on how to customize error handling.

When NOSL uses the error handler

NOSL calls the error handler from any NOSL function that detects a fatal error when it is processing a stream.

The default error handling routines

If you do not customize error handling, and a fatal error occurs, the nosl functions will call the default error handling routines.

The default error handling routines call $ERMSG on any error.

Major reason to customize error handling

The major reason for customizing error handling is to recover from errors that may not be considered fatal by your program.

Example: n$open will generate a fatal error if an open is performed on a nonexistent file. Quite often, this is not considered a fatal error.

CUSTOMIZING ERROR HANDLING

Overview

This section contains the information needed to customize error handling.

Error codes

NOSL updates the standard RMS error code, $ERRC, which may be used by your error routines.

- $ERRC.$FUNC contains
  - the system code number for RMS system call errors,
  - the UFR class for UFR errors, and
  - the NOSL function code for NOSL errors.
- $ERRC.$CODE contains the error number.

Symbolic values for $ERRC.$FUNC may be found in D$ERRNUM/TEXT. Symbolic values for $ERRC.$CODE may be found in D$ERRCODE/TEXT.

State of a stream

Determining whether an error is fatal or not should partly depend on the state of the stream after the error. A stream will be left in a defined state following a fatal error if

- a n$open failed because of a nonexistent file, or
- an illegal NOSL operation was attempted, such as a n$read on a Print nosl.

Otherwise, the stream is left in an undefined state.

n$ErrorData type

The n$ErrorData type is a type that must be defined by you. Variables of this type may be used to store information that will be passed into the error handling routines when a fatal error occurs.

Example: This example uses n$ErrorData to store a pointer to the filename related to each stream. This filename can later be displayed by the error handling routines on a fatal error.

```
TYPDEF n$ErrorData ^ $NAMEEXTENV;
```

n$ErrorFunc type

Use the n$ErrorFunc to declare functions that will be called by the NOSL functions when an error occurs.

Note: Do not call any NOSL functions from the error function.

n$ErrorFunc is defined as follows:

Type Definition	*Description*
`TYPDEF n$ErrorFunc (`	
`eD ^ n$ErrorData`	Address of data that you define. The address of the data passed to n$open when a stream is opened will be passed as this parameter if an error occurs.
`);`	

Example: The following program segment defines an error function that will call $ERMSG on anything other than an open error.

```
function n$ErrorFunc :=
{
   IF $ERRC.$FUNC ~= SC$OPENENV
     & $ERRC.$CODE = $UECOPN1 THEN $ERMSG();
   };
```

Specifying error parameters to n$open

The NOSL function n$open, may be passed the address of an error function and error data. If a fatal error occurs on that nosl during the nosl open or any other NOSL function, the error function will be called and the address of the error data will be passed as a parameter to the error function.

Example:

```
out n$Stream;

function () :=
{
   out := n$open(WSOut, &errorF, &errorD);
```

Specifying error parameters to n$fKind

The NOSL function n$fKind also accepts the address of an error function and error data as parameters. If a fatal error occurs during n$fKind, the error function will be called and the address of the error data will be passed as a parameter to the error function.

Example:

```
fileSpk $FILESPK := {
   { 'IN      ', '            ', '   ', '        ' },
     $FILNAMR,   $NOADR,       &'TEXT',&'        '
   };

function () :=
VAR   dK BYTE;
{
   IF $SCANFLS(&fileSpk, 1) && D$CFLAG THEN $ERMSG();
   dK := n$fKind(&fileSpk.$FSOSFT,, &errorF, &errorD);
```