SOFTWARE ENGINEERING SERVICES

1.0

Procedure Writer's Guide

60460270 01

REVISION RECORD

REVISION		DESCRIPTION
01	(02-17-84)	Preliminary manual released.

Address comments concerning this manual to:

Control Data Corporation
Software Engineering Services
4201 North Lexington Avenue
St. Paul, Minnesota 55112

60460270 01

Table of Contents

1.0 INTRODUCTION

SES is a NOS utility whose major function is to locate and process PROCEDURES so as to generate streams of CCL to the system control statement file.

PROCEDURES are text records which contain CCL interspersed with directives to the SES processor itself. The SES directives can cause CCL to be generated according to specified conditions.

SES is invoked by an SES control statement, either from a terminal session or from a batch job. The SES control statement specifies the name of the procedure to be processed, and optionally, parameters for that procedure. SES locates the procedure, processes it, and generates the appropriate CCL stream to the control statement file.

This document is intended as a guide to those who wish to write procedures to be processed by SES.

2.0 FEATURES OF SES

This section provides an overview of the features and facilities available to the user of SES. The features fall into two related classes, the first class being the ways in which parameters may be specified on the SES control statement, and the second class being the features provided by the SES processor. The two classes are related, since in general, for each way that a parameter can be written on the SES control statement, there is a corresponding SES directive or function available to process that particular aspect of the parameter.

2.1 CONTROL STATEMENT FEATURES

This section looks briefly at the way in which an SES control statement and parameters may be written.

- o Continuation Lines. SES procedure calls are not limited to one control statement line. Continuation lines may be input, whether from a terminal or in batch. The total length of a statement, including continuation lines, is limited to 2000 characters.
- o Procedure parameters may be specified by keyword, or positionally, or by a combination of both methods.
- o Parameters of a procedure may have multiple values.
- o Parameters of a procedure may be coded solely as a keyword with no values, in which case the keyword may be used to specify options.
- o A parameter keyword may have multiple synonyms.
- o Parameter values may be coded as arbitrary character strings.
- o The user may indicate on the SES control statement that a particular user's catalog is to be searched when locating a procedure.
- o Users may establish procedure library search order and other

default information in a PROFILE, which SES accesses at call time.

2.2 PROCEDURE PROCESSING

This section provides a brief look at the features available to the SES procedure writer.

- o Values and defaults established in a user's PROFILE may be accessed.
- o The names of parameters, their possible types, and the number of values that may be coded for them, are predefined within the procedure.
- o SES provides functions to test for the type, number of values, and existence of a parameter.
- o SES provides a function to index along a multiple valued parameter.
- o The procedure writer may define variables to hold values during procedure processing.
- o CCL statements may be generated conditionally or iteratively via IF and WHILE directives.
- o Expression evaluation and string manipulation facilities.
- o Generation of unique strings for names and labels.
- o Text from within the body of a procedure may be ROUT'ed to any specified file.
- o Text may be INCLUDE'd into the body of the procedure from any specified file, or from any specified procedure of any specified plib library.
- o Local files may be tested for attributes, similar to the FILE function provided by the operating system.
- o The user's environment at procedure call time can be restored at procedure end.

2.3 LAYOUT OF THIS GUIDE

Rather than supplying an alphabetical list of directives and functions, the features are going to be introduced in related chunks, mostly illustrated by examples. As far as possible, the examples given are taken from real live SES procedures, to avoid creating artificial examples. The general layout of the guide is in this order.

- o Basic SES concepts, processing and syntax.
- o Expression evaluator.
- o Functions.
- o SES directives.
- o Parameter definition and processing.
- o File system directives.
- o Various summaries in appendices.

3.0 BASIC SES PROCESSING

This section is going to show the major aspects of how SES performs its processing. Topics covered in this section are.

- o procedure call format, showing the basic format of an SES control statement.
- o what a procedure looks like.
- o The mechanism for substitution of parameters and names.
- o SES directives within procedure files.
- o Profiles and the SEARCH directive.
- o Locating a procedure. Explains the search method that SES uses to locate a procedure.
- o Processing a procedure. Explains what happens to each line of text in an SES procedure.

3.1 SES PROCEDURE CALL FORMAT

The basic form of an SES procedure call is.

SES.procedure_name list_of_parameters

where "procedure_name" is the name of the procedure to be processed, and "list_of_parameters" is the (optional) list of parameters for the procedure. The list is separated from the procedure name by a comma or by space(s) or both. Elements in the parameter list are separated from each other by commas or space(s) or both. The parameter list is terminated by an end of line, a period, or a semicolon.

Parameters are generally written in the form of

keyword=value

this is only a part of the story however, and later in the document we'll get to specific definitions of the manner in which parameters may be coded.

3.1.1 SES PROCEDURE LAYOUT

The general layout of an SES procedure is:

PROCNAME	where PROCNAME is the name of the procedure.
\ PARM	
\ PARM	
\ PARM	there are zero to many of these PARM directives. They are used to define the exact format of the parameters in the list. The form of a PARM directive will be defined in a later section.
\ PARMEND	this indicates the end of the PARM directives, and is always neccessary even when there are no PARM directives.
BODY OF PROCEDURE	the procedure body contains CCL which gets written to the control statement file, and SES directives which are processed at procedure build time.
\ blah blah	any line which starts with the directive character, which is a reverse slash (\) by default, is taken to be an SES directive.

A procedure of name PROCNAME may be a local file, or a file in the current user's catalog, or it may be a record in a PLIB. But, no matter where the procedure comes from, the first line of the procedure must be the name of the file or record in which the procedure resides.

3.2 SES SYNTAX

This section provides a short and informal introduction to the syntax of the SES processor. A more formal and complete syntax definition is provided at the end of the document.

The discussions on syntax use the characters [and] to indicate that an item is optional.

3.2.1 DIRECTIVES

To determine if a line of a procedure is a directive, the SES processor goes through the following steps:

1. Any leading spaces on the line are ignored.

2. SES looks for a variable called **DIRCHAR** (for **DIR**ective **CHAR**acter) in its tables (we'll discuss variables later). If DIRCHAR is undefined, or if DIRCHAR is defined but contains a value other than a single character which is a "visible delimiter character" (space is not considered a visible delimiter), then SES will use the reverse slash (\) as the directive character, otherwise SES will use the character in DIRCHAR as the directive character.

2. If the (now) first character of the line is equal to the directive character, then the line is assumed to be either a directive or an assignment statement, and is processed accordingly.

3.2.2 VARIABLES

Variables are one thru thirty-one characters in length, must start with a letter, and may contain only letters, digits, or the characters _, $, @, or #.

3.2.3 NUMBERS

SES only handles integers, there are no reals. Integers are represented internally by 48 bit quantities. Thus integers range between -2**47-1 and 2**47-1.

Numbers are normally assumed to be decimal, but bases other than decimal may be represented by appending a base specification to the string of digits. The base may be any base between 2 and 16, but generally, the useful bases are 2, 8, 10 and 16, and any others are sort of weird. For example:

```
4975 is a decimal number
377475(8) is an octal number
9A46(16) is a hexadecimal number
```

note that hexadecimal numbers (and in fact any base requiring use of the letters A thru F) must start with a decimal digit (even if it's zero), to avoid confusion with names.

3.2.4 STRINGS

Strings are arbitrary strings of characters enclosed in single quote marks, for example:

'Just the place for a Snark, the Bellman cried.'

to represent a string quote inside a string, you must code it as two string quotes:

'The time is Seven O''Clock'

two juxtaposed string quotes, that is, '', represent a null, or empty string.

3.2.5 BOOLEANS

Strictly speaking, there aren't really booleans in SES. However, SES has the predefined variables TRUE, YES, FALSE and NO. The first two represent the value TRUE, and the second two represent the value FALSE. They are conformable with integers, in that TRUE or YES are equal to one (1), and FALSE and NO are equal to zero (0).

Otherwise, any non zero value is assumed to be TRUE, and a zero value is assumed to be FALSE.

3.2.6 FUNCTIONS

SES provides many built in functions. A function reference follows the standard form, that is:

function_name (list_of_arguments)

where "function_name" is the name of the function to be referenced, and "list_of_arguments" is the argument(s) to the function. Elements of an argument list are separated from each other by commas or space(s) or both.

3.2.7 EXPRESSION EVALUATION

SES can evaluate expressions containing mixed mode integer, string, boolean and function references. Implicit type conversion is performed as required.

3.2.8 COMMENTS

A comment is any arbitrary string of characters enclosed between double quote marks ("). The comment may not itself contain comment quotes. Comments may appear anywhere that a space may appear, and in fact is syntactically equivalent to a space.

Comments may not appear before the directive character of an SES directive line, nor after the continuation signal on lines which are being continued.

3.2.9 CONTINUATION LINES

Any SES directive or call line may be continued by placing a continuation signal (..) at the end of the line to be continued. A continuation signal is defined to be two or more contiguous periods. The total length of an SES call line may not exceed 2000 characters, while the length of a directive line may not exceed 256 characters.

3.2.10 SUBSTITUTION

A major function of SES is to substitute parameters into procedures. In actual fact, SES can substitute elements other than parameters, and this latter aspect is covered first.

The basic substitution mechanism when processing a line of a procedure is this:

1. SES looks for a variable SUBCHAR (for SUBstitution CHARacter) in its tables (we'll discuss variables later). If SUBCHAR is undefined, or if SUBCHAR is defined but contains a value other than a single character which is a "visible delimiter character" (space is not considered a visible delimiter), then SES will use the ampersand (&) character as the substitution character, otherwise SES will use the character in SUBCHAR as the substitution character.

2. If SES finds on a line, the substitution character followed by a name followed by the substitution character, then SES follows the procedure below:

   a) SES first searches for a parameter of the specified name, and if such a parameter is not found, then SES searches for a variable of the specified name. If the parameter or variable is defined, then the value of the variable, or the value of the parameter is inserted into the output text at that point, without the substitution characters.

   b) If SES finds neither a parameter of the specified name, nor a variable of the specified name, then the substitution characters are stripped off and the literal character string which comprises the name is inserted into the output text.

3.0 BASIC SES PROCESSING
3.2.10 SUBSTITUTION

For example, supposing that the substitution character is &, and that the name YIN is associated with the value YANG.

Input	Output	Explanation
REWIND(&YIN&)	REWIND(YANG)	this example is straightforward. The value YANG is simply substituted for the name YIN.
REWIND(&MIN&)	REWIND(MIN)	since MIN wasn't defined, then the substitution characters are simply removed.

3.3 PROFILES

A PROFILE is an important, albeit optional, component of the SES system. Any user may choose to establish a PROFILE in their catalog. PROFILE follows the same rules as any SES procedure, that is, the name of the file must be PROFILE, and the first line of the profile must be the word PROFILE. From there on, the profile may contain just about any SES command. The most important aspect of the profile is the SEARCH directive, explained in the next section.

Typically, the types of things that a user may place in the profile would be:

- a command to set a variable called PASSWOR to the user's password. Procedures which optionally run as batch jobs can then get the user's password without having to be told it on the SES control statement.

- commands to establish defaults for library names (for the source code and library maintenance procedures), and other data for various procedures.

- SEARCH directives to establish a search order for procedures.

It is possible for a user to have more than one PROFILE, and select which one to use by coding the PN or P parameter on the SES control statement, for example.

```
SES,PN=alternate_profile.procname list_of_parameters
```

allows the user to use the file "alternate_profile" as the PROFILE for the duration of that procedure call. Also, a user may use someone else's profile by coding the PUN or PU parameter, for example:

```
SES,PUN=profile_owner.procname list_of_parameters
```

allows the user to access the profile belonging to "profile_owner". Of course, the PN and PUN parameters may be used together.

3.3.1 SEARCH DIRECTIVE - ESTABLISH LIBRARY SEARCH ORDER

The SEARCH directive allows a user to establish, within PROFILE, the names of libraries to search when locating a procedure, and also the user names in whose catalogs those procedure libraries reside. The general form of SEARCH is:

```
\ SEARCH search_spec, search_spec.......
```

where "search_spec" is in the form:

```
user_name

or

(library_name, library_name....., user_name)
```

The first form indicates that the library name contained in the predefined variable **SESLNAM** is to be searched for in the catalog of the user specified by "user_name". The second form gives a list of library names, with the last item in the list being the user name in whose catalog those libraries may be found.

3.4 LOCATING A PROCEDURE

SES performs its search for a given procedure according to well defined and consistent rules. Basically SES has three methods of specifying how a procedure is to be located. SES has an internal table which contains the following data:

library_name	user_name
library_name	user_name
etc.	
etc.	
etc.	

Given that the table may be set up by one of three different methods which are explained in more detail in the sections following, the procedure that SES follows to locate a procedure is:

1. If there is a local file of the "procedure_name", whose first line is "procedure_name", then SES uses that file as the procedure.

2. SES searches the catalog of the user whose user name appears as the first entry in the table, for a file of name "procedure_name", whose first line is "procedure_name". If such a file is found, then SES uses that file as the procedure.

3. For each entry in the search table, SES searches for a library of name "library_name" in the catalog of the corresponding "user_name", and searches that library (which must have a directory) for a TEXT record of name "procedure_name". If SES eventually finds such a record, then SES uses that record as the procedure.

4. If the search is unsuccessful, then SES issues an error message

   procedure_name NOT FOUND

The next three sections provide a more detailed explanation of the methods by which SES has its search table set up. The methods are basically the default, the user name specified on the SES control statement, and the SEARCH directive.

3.4.1 DEFAULT ORDER OF SEARCH

When SES is called, it sets up the following data in its search table:

```
+-----------+-----------+
| &SESLNAM& | user_name |
|           |           |
| &SESLNAM& | &SESUNAM& |
+-----------+-----------+
```

This table is the normal default for SES. "user_name" is the user name of the currently logged in user.

"SESLNAM" is a predefined variable which contains the name of the SES procedure Library NAMe. "SESUNAM" is a predefined variable which contains the SES User NAMe. There will be a more detailed section on predefined variables later in the document.

3.4.2 SEARCH SPECIFIED ON CONTROL STATEMENT

When the user types the SES control statement, he may specify via the UN or U parameter of the SES program, which user's catalog to look in for the procedure specified by the call. For example:

```
SES,UN=user_name.procedure_name list_of_parameters
```

specifies that the procedure "procedure_name" is to be searched for only in the catalog of the user "user_name" (if the procedure is not already local). In this case SES modifies its search table to contain only the following data:

```
+----------+----------+
| &SESLNAM& | user_name |
+----------+----------+
```

where "SESLNAM" contains the SES Library NAMe as before, and "user_name" is the user name specified on the SES control statement. It is also possible to tell SES, via the LIBPFN or LPFN parameter, the name of the library to be searched for the procedure. For example:

```
SES,LPFN=lib_name.procedure_name list_of_parameters
```

specifies that the procedure "procedure_name" is to be searched for only in the library "lib_name". In this case SES modifies its search table to reflect the following data:

```
+----------+----------+
| lib_name  | user_name |
+----------+----------+
```

where "user_name" is the user name of the current user, and "lib_name" is the library name specified on the SES control statement. Of course, the UN and LIBPFN parameters may be used together.

3.4.3 SEARCH ORDER SPECIFIED VIA SEARCH DIRECTIVES

The third method of specifying the order in which to look for the procedure is via SEARCH directives in the user's PROFILE. For example, supposing that the user's PROFILE contains the following SEARCH directive:

```
\ SEARCH (HOLMLIB,JIMLIB,HG74), AM74, JF03, (ANDYLIB,ED73)
```

in this case SES would modify its search table to look like this:

```
+-----------+------+
| HOLMLIB   | HG74 |
|           |      |
| JIMLIB    | HG74 |
|           |      |
| &SESLNAM& | AM74 |
|           |      |
| &SESLNAM& | JF03 |
|           |      |
| ANDYLIB   | ED73 |
+-----------+------+
```

Note that SEARCH directives are ignored if the UN or LIBPFN parameters were specified on the SES control statement.

3.5 PROCESSING A PROCEDURE

Now assuming that SES is able to locate the required procedure, then the procedure is processed, at least in principle, on a one pass, line by line basis--we say "in principle", since in fact because of WHILE (looping) directives, a given line may be processed many times. Also each line may be scanned twice. Leaving all that aside for the nonce, the processing for each line of the procedure goes like this:

1. The line is scanned by the substitution processor. Any substitutable elements are processed at this stage, and the replacement text inserted into the line at that point. This process continues until the whole line is scanned.

2. The line is then examined to see if it is an SES directive (or assignment statement), and if so it is processed accordingly.

3. If the line is not an SES directive, then that line is written to the output stream, whatever that happens to be at the time.

4.0 EXPRESSION EVALUATION

Although, as we said before, the principal function of SES is to substitute parameters into procedures, the expression evaluator of SES is a sufficiently important aspect of processing SES procedures that it and its related topics are covered first, before we ever get to explaining parameters and parameter substitution. By starting with the expression evaluator, you'll find it easier to understand parameters when we get to them.

4.1 ASSIGNMENT OF EXPRESSIONS TO VARIABLES

Within the body of an SES procedure it is possible to have variables. Variables are used for many purposes, such as control variables in WHILE loops, building character strings, etc, etc.

If you assign a value to a variable which was previously undefined, then SES defines the variable for you, and initializes it to the value of the expression to the right of the equal sign. If the variable was already defined, then its new value becomes the value of the assignment expression.

Variables within SES may be of type NAME, INTEGER, STRING, or BOOLEAN. When a variable is initialized, it takes the type of the initialization expression. Upon subsequent assignment to the variable, it takes the type of the expression to the right of the equal sign. For example, here are four variables being declared:

```
\  stringy = 'MOZZARELLA CHEESE'

\  number = 547(8)

\  logical = TRUE

\  aname = fred
```

In the example, the first variable is of type STRing; the second is of type NUMber (there is no type REAL); the third is of type BOOLEAN; and the fourth is of type NAMe (it is assumed that fred was not previously defined as a variable). Generally speaking, the expression evaluator performs implicit type conversion, so that variables of different types may be mixed within an expression.

4.2 OPERATORS IN EXPRESSION EVALUATION

Before we go further into expression evaluation, we'll show the operators that may be used in expressions. They fall into the classes of arithmetic, string, relational and logical operators. The table below also indicates the relative priority of the operators.

Operator Class	Precedence	Operator	Comments
Arithmetic	1	**	Exponentiation
	2	*	Multiply
		/	Divide
		//	Modulo or Remainder
	3	+	Add or Monadic Plus
		-	Subtract or Monadic Minus
String	4	++	String Concatenation
Relational	5	=	Equal To
		/=	Not Equal To
		<>	Not Equal To
		>	Greater Than
		>=	Greater Than or Equal To
		<	Less Than
		<=	Less Than or Equal To
Logical	6	NOT	Logical NOT or Negation
	7	AND	Logical AND
	8	OR	Logical OR
		XOR	Logical Exclusive OR

Notes:

- o Operators at the same precedence level are processed from left to right.
- o The right operand of the exponentiation operator must be greater than or equal to zero.

- The right operand of the division and modulo operators must not be zero.

- Processing of relational operators is as follows:

  - If both operands of a relational operator can be converted to integers, they are so converted and then compared; otherwise both operands are converted to strings (if necessary) and then compared.

4.3 EXAMPLES OF ASSIGNMENT STATEMENTS

In this section we're going to look at an example of the use of the expression evaluator, showing how substitution of names works in conjunction with assignment. This example is from the SES MATH procedure, which acts as a quick and dirty desk calculator. At the start of the MATH procedure, the following chunk of code may be found.

```
\  curnamq = 'VALUE'                    " 1 "
\  &curnamq& = 0                        " 2 "
   .
     .
       .
\  MSG '&curnamq& = ' ++ &curnamq&      " 3 "
     etc.
        etc.
           etc.
```

Note how we made use of comment quotes in the example in order to number the lines of interest to the discussion. Now the way this works is that line 1 sets a variable "curnamq" to the character string 'VALUE'. When line 2 is processed, the substitutor looks for something called "curnamq", and finds the string 'VALUE', so that by the time the assignment statement is processed, the line will actually read

```
\  VALUE = 0
```

so the variable VALUE gets initialized to zero. Now line 3 is scanned by the substitutor, and when substitution is finished, the line will look like

```
\  MSG 'VALUE = ' ++ VALUE
```

now the expression evaluator is called into play to process the argument to the MSG directive. MSG wants its final argument in the form of a string. The expression evaluator finds that the first part of the expression is indeed a string. Then it finds that the second part of the expression calls for a string concatenation of whatever is in the variable VALUE. Name lookup finds that the variable contains the value 0. The expression evaluator converts the 0 to a string and concatenates it to the previous string in the expression. Finally the MSG directive outputs to the user a message that says

```
VALUE = 0
```

5.0 FUNCTIONS

SES has a number of functions for use by the expression evaluator. These functions are explained in detail in the following sections. First there's a brief overview of the functions.

UNIQUE	generates unique seven character strings in the form of labels or filenames.
Attribute Testing	There are functions that test whether a variable is a name, number, string or operator, or whether an arithmetic expression is legal.
String Handling	string handling functions are provided by SUBSTR, which returns a substring of a larger string, STRLEN, which returns the length of a string, GENSTR, which restores a parsed string to its original format, and functions to raise or lower the case of alphabetic characters.
Number Conversion	the functions OCT, DEC and HEX perform integer to string conversion.
DATE, CLOCK and TIME	these functions returns the date and time in various formats as specified by their arguments.
VALEXPR	this function can be used to VALidate and/or eVALuate an EXPRession contained within a string variable.
TOKEN	this function reads the next valid SES token (syntactic unit) from a string variable.

5.1 UNIQUE - GENERATE UNIQUE NAMES OR LABELS

This function is used by most of the SES procedures to generate unique names for intermediate scratch files, unique names for programs invoked by the SES procedures, and unique labels (in those rare cases where labels are needed). They are explained in detail below.

UNIQUE(NAME) returns as a value a seven character alphanumeric string, starting with the letters ZQ. The name is guaranteed to be different from the name of any file currently assigned to the running job from which this SES procedure is being called.

UNIQUE(LABEL) returns as a value a seven character alphanumeric string starting with the characters 9Q.

The UNIQUE function repeats about every seventeen hours.

As an example of how this is used, the following is a short extract from the SES COPYACR procedure.

```
\  copyacr = UNIQUE(NAME)
\  library = UNIQUE(NAME)
.
   .
     .
EXTRACT(&copyacr&=COPYACR/T=ABS,LFN=&library&,L=PROGLIB,UN=&SESUNAM&)
&copyacr&(HERE,THERE)
.
  .
    .
```

The two variables at the top are initialized to unique names, so that when those names are used, they will not conflict with any file that the user may have assigned to the job.

It is good practice to use unique names for files and programs wherever possible, because then the user does not have to remember which procedures use which filenames.

5.2 TESTING ATTRIBUTES OF EXPRESSIONS

The functions described below are mainly used to test the type of an expression. NAM, NUM, and STR return true if the argument is of type NAMe, NUMber or STRing, respectively. DEF returns true if its argument is DEFined, DEFF returns true if its argument is a DEFined Function or a symbolic operator. OPR returns true if its argument is an OPeRator. VALEXPR checks and computes a VALid EXPRession.

5.2.1 NAM - TEST FOR NAME

The NAM function returns true if its argument is a NAMe. The general form of NAM is:

NAM (expression)

if "expression" evaluates to something that can be converted to a name, then the NAM function returns TRUE, otherwise it returns FALSE. For example:

```
\  test = NAM (FRED)
\  test = NAM ('ABC' ++ 'DEF')
\  test = NAM ('JUNK' ++ TRUE)
```

a return the value TRUE. In the first example, FRED is definitely a name, in the second example, the result of concatenating the two strings results in a value which can be converted to a name, and in the third example, the result of the expression is the string 'JUNK1', which can also be converted to a name. So in each case, the value of variable "test" is TRUE. However, the tests:

```
\  test = NAM (12345)
\  test = NAM (TRUE)
\  test = NAM ('123ABC')
```

all fail, since 12345 is not a name but a number, TRUE converts to the value 1, which is also not a name, and '123ABC' is a string which cannot be converted to a valid name.

5.2.2 NUM - TEST FOR NUMBER

The NUM function returns true if its argument is a NUMber. The general form of NUM is:

NUM (expression)

if "expression" evaluates to something that can be converted to a number, then the NUM function returns TRUE, otherwise it returns FALSE. For example:

```
\ test = NUM (497500)
\ test = NUM (377 ++ '(8)')
\ test = NUM (0ABC ++ TRUE ++ '(16)')
```

all return the value TRUE. In the first example, 497500 is definitely a number, in the second example, the result of concatenating the two strings results in a value which can be converted to a number, and in the third example, the result of the expression is the string '0ABC1(16)', which can also be converted to a number. So in each case, the value of variable "test" is TRUE. However, the tests:

```
\ test = NUM (FILENAM)
\ test = NUM ('Haddocks Eyes')
```

both fail, since FILENAM is not a number but a name and the character string 'Haddocks Eyes' cannot be converted to a valid number.

5.2.3 STR - TEST FOR STRING

The STR function returns true if its argument is a STRing. The general form of STR is:

STR (expression)

if "expression" evaluates to something that can be converted to a string, then the STR function returns TRUE, otherwise it returns FALSE. For example:

```
\ test = STR (THROCKS)
\ test = STR (735725(8))
```

\ test = STR ('Nurdle yer Cordwangler')

all return the value TRUE, since any of those things, names, numbers and strings can indeed be converted to a string. In fact it looks as if you can convert anything at all to a string, and if this is the case, what's the use of the STR function? Well as you've probably guessed, life's not as simple as all that, and there is in fact one thing that cannot be converted to a string, and that is an omitted value. We'll talk about this a bit more when we describe parameters later in the guide.

5.2.4 DEF - TEST FOR DEFINED VARIABLE

DEF stands for DEFined, and its aim in life is to return TRUE if the name specified as its argument is deined as a variable. The general form of DEF is:

DEF(name)

where "name" is the name of the thing that you want to know about. The "name" argument to DEF may not be an expression, only a name.

Note: that while SES is in operation a vast quantity of variables get defined, other than those that the procedure writer may define. The list of predefined variables is given at the end of this document.

5.2.5 DEFF - TEST FOR DEFINED FUNCTION OR OPERATOR

DEFF stands for DEFined Function, and it returns a true value if the name given as its argument is any of the SES function names or mnemonic operators. The general form of DEFF is:

DEFF(name)

where "name" is the function or operator name that you want to test for. The argument to DEFF may not be an expression, only a name.

5.2.6 OPR - TEST FOR OPERATOR

OPR tests for its argument to be an OPeRator or delimiter. The general form of OPR is:

OPR(string_expression)

OPR reads the first token from the string given by "string_expression", and, if the token is a valid SES token, then OPR returns TRUE if the token is an operator or a delimiter other than an operator. A list of the valid SES tokens is given in the appendix on SES syntax.

5.2.7 VALEXPR - CHECK AND COMPUTE EXPRESSION

The purpose of VALEXPR is twofold. Firstly it is intended to VALidate an EXPRession, to see if it can be evaluated, and secondly, if the expression is computable, then VALEXPR eVALuates the EXPRession. The raison d'etre of VALEXPR is that it is possible to read data from a file into a string variable, and then evaluate that string as an expression. The general format of VALEXPR is:

VALEXPR (result_variable, input_string_variable)

where "result_variable" is the name of a variable to receive the result of the expression specified by the string in the variable of name "input_string_variable". VALEXPR returns a value which is either the null string, which indicates that the expression was valid, in which case the result of the expression is in "result_variable", or else the function value is a character string which is the SES error message indicating what was wrong with the expression.

As an example of VALEXPR, we'll show another extract from the SES MATH procedure. The relevant pieces of the procedure are given here, with the interesting line numbers in comments.

```
\  ACCEPT INTO='stringq',PROMPT='&curnamq&='++&curnamq&  " 1 "
.
   .
     .
\  tokstsq = VALEXPR (resultq, stringq)                   " 2 "
\  IF tokstsq /= '' THEN                                  " 3 "
\    MSG tokstsq                                          " 4 "
etc.
   etc.
     etc.
```

Line 1 reads a string from the user into the string variable "stringq". Further down in the procedure, after a lot of other belt and braces checking, the VALEXPR call at line 2 places the result of the expression evaluation in "resultq" and returns as a function value a character string which is checked at line 3 to see if it's the null string. If it isn't, then the string in "tokstsq" represents an error message which is output to the user at line 4.

5.3 STRING HANDLING

The functions described below allow you to massage strings. The functions described are STRLEN, to find the length of a string, SUBSTR, to extract a part of a string, GENSTR, to regenerate a string, and GENUPR and GENLOWR to raise or lower the case of alphabetic characters in a string.

5.3.1 STRLEN - DETERMINE LENGTH OF STRING

STRLEN stands for STRing LENgth, and it returns as a function value, the length of its argument. The general form of STRLEN is:

```
STRLEN(string_expression)
```

where "string_expression" is the character string of which you want to find the length. For example,

```
\  game = 'DWILE FLONKING'
 .
  .
\  size = STRLEN (game ++ ' AND NURDLING')
```

the STRLEN function call has as its argument a string expression which should result in a string having the value

'DWILE FLONKING AND NURDLING'

and after evaluation is complete, the variable "size" should contain the value 27.

5.3.2 SUBSTR - EXTRACT SUBSTRING FROM CHARACTER STRING

SUBSTR, for SUB STRing, returns a part of a string from a string variable. The general format of SUBSTR is:

SUBSTR(string_exp, integer_exp, integer_exp)

where "exp" stands for "expression". The first parameter of SUBSTR is the string from which you wish to extract a substring. The second parameter is the character number (starting from one) at which the substring is to start, and the third parameter is the number of characters to be extracted from the string. For example.

```
\  this = 'MONEY FOR OLD ROPE'
.
.
\  other = SUBSTR(this, 11, 8)
```

After the substring function has been evaluated, the value of variable "other" is 'OLD ROPE', and STRLEN(other) returns the value 8.

If you omit the third parameter from the SUBSTR function, then it returns one character from the position designated by the second parameter.

If you omit the second and third parameters, then the SUBSTR function returns the entire string. This doesn't seem to be a lot of use, and it's a whole lot quicker to just assign the string to another one.

It is not possible to omit the second parameter and code the third. If you do such an antisocial thing, you'll get the error message EXPECTING NUMBER.

If the starting index parameter is given as less than one, it is (internally) set to one; or if the starting index is given as greater than the length of the string, the starting index is (internally) set to the larger of the length of the string or one. The default starting index is one.

If the length is given as less than zero, it is (internally) set to zero; or if the length is given as greater than the maximum (80), it is (internally) set to the maximum. The default length is the length of the original string.

Once the starting index and length have been determined, the requested number of characters is returned as the function value (padding on the right with spaces if neccessary).

5.3.3 GENSTR - REGENERATE A STRING

The GENSTR function is used to restore a string. When a string value is coded for a parameter or in a string assignment, the string is initially enclosed in single quote marks, and a single quote mark within the string is represented by a pair of quotes. For example,

```
\  time = 'Thirteen O''Clock'
```

will set the variable time to the string shown. When SES processes this, the outer quotes are removed, and pairs of quotes replaced by single ones. However, if this string was to be passed on to the call statement of another SES procedure, then it must be restored to its original form, so that eventually SES can crunch it again. So the function GENSTR, for GENerate STRing is used. The general form of GENSTR is

```
GENSTR(string_expression)
```

5.3.4 GENUPR - RAISE CASE OF ALPHABETICS

GENUPR is used to raise the case of alphabetic characters in a string variable. The format of the GENUPR function is

GENUPR (string_expression)

where "string_expression" is the string you want to process. An example of GENUPR is shown here in this extract from the SES system library procedure MATH. The procedure reads a string from the input file into a string variable "stingq"

```
\  IF GENUPR(stringq) = 'END' OR GENUPR(stringq) = 'BYE' THEN
\    STOP
etc.
   etc.
```

the MATH procedure allows you to type END or BYE to exit from the procedure, and that is what is being tested for in the example above. Since the user may be logged in in ASCII mode, it's possible for the input to be in a mixture of upper and lower case, so we use GENUPR to raise the case of the alphabetic characters.

5.3.5 GENLOWR - LOWER CASE OF ALPHABETICS

GENLOWR is used to lower the case of alphabetic characters in a string variable. The format of the GENLOWR function is

GENLOWR (string_expression)

where "string_expression" is the string you want to process.

5.4 CHARACTER HANDLING FUNCTIONS

The functions in this category can be used to manipulate ASCII characters which do not have a graphic representation. The functions are CHARREP which return the character represented by its integer argument; and INTREP which returns the integer representation of its character argument.

5.4.1 CHARREP - CHARACTER REPRESENTATION

This function returns as its value, the ASCII character corresponding to its argument. The general form of CHARREP is:

CHARREP (integer_expression)

where "integer_expression" must evaluate to an integer between 0 (zero) and 255. For example: CHARREP(15(8)) is the ASCII character for "carriage return".

If the "integer_expression" has a value of 128, it will be translated to a colon. If the value is between 129 and 255, and the value of CHARREP with this argument is written to a file, it will be translated to an asterisk.

5.4.2 INTREP - INTEGER REPRESENTAION OF CHARACTERS

This function will return as its value, the integer representation of its character argument. The general form of INTREP is:

INTREP (string_expression)

where STRLEN(string_expression) must be equal to 1 (one). For example:

INTREP ('2')

has the value 50 or 62(8) or 32(16); and

INTREP (CHARREP(10))

has the value 10 (i.e. the integer representation of the ASCII line feed character.

5.5 INTEGER EXPRESSION TO STRING CONVERSION

The functions described below are for converting integers to strings. The functions are OCT, DEC and HEX, which convert integers to their OCTal, DECimal and HEXadecimal representations respectively. None of the functions append any base designators, that's up to you and your particular application.

5.5.1 OCT - INTEGER TO OCTAL STRING CONVERSION

This function converts an integer to its OCTal string representation. The form of the function is:

```
OCT(integer_expression)
```

For example, if the variable "titus" has the value 795, then the assignment statement

```
\  groan = OCT (titus + 4)
```

results in the variable "groan" being set to the string '1437', this being the octal representation of the decimal integer 799.

5.5.2 DEC - INTEGER TO DECIMAL STRING CONVERSION

This function converts an integer to its DECimal string representation. The form of the function is:

```
DEC(integer_expression)
```

For example, if the variable "mortice" has the value 497, then the assignment statement

```
\  tenon = DEC (mortice + 29)
```

results in the variable "tenon" being set to the string '526'.

5.5.3 HEX - INTEGER TO HEXADECIMAL STRING CONVERSION

This function converts an integer to its HEXadecimal string representation. The form of the function is:

HEX(integer_expression)

For example, if the variable "easter" has the value 10138, then the assignment statement

```
\  bunny = HEX (easter + 1311)
```

results in the variable "bunny" being set to the string '2CB9', this being the hexadecimal representation of the decimal integer 11449.

The HEX function always guarantees that there is a decimal digit at the start of the character string, since the syntax of hexadecimal numbers within SES requires that they start with a decimal digit. If the first character of the resultant string is not a decimal digit, then SES will place a 0 (zero) at the start of the string.

5.6 DATE, CLOCK AND TIME FUNCTIONS

These functions return character strings as values. DATE returns the current date in a number of formats determined by its argument, CLOCK returns the time of day in various formats, and TIME returns information about job and system time.

5.6.1 DATE - CURRENT DATE FUNCTION

DATE returns the current date in any one of a variety of formats. The form of the DATE function call is:

DATE(format)

where "format" may be specified in one of these ways:

YMD returns the date in the form 76/09/08 (AD 1976, month of September, day 8)

DMY returns the date in the form 08/09/76, the reverse of the way just above.

MDY returns the date in the form 09/08/76 (American style - month first).

DMONY returns the date in the form 8 SEP 76

MONDY returns the date in the form SEP 8, 1976

JULIAN returns the Julian date, 76252 for September 8.

5.6.2 CLOCK - TIME OF DAY FUNCTION

The CLOCK function returns the time, so that it is possible to get NOS to give you the time of day. The general form of CLOCK is

CLOCK(format)

where "format" is one of the following.

HMS returns Hours, Minutes and Seconds, in the form 16:40:19.

AMPM returns the above time in the form 4:40 PM

5.6.3 TIME - SYSTEM AND JOB TIME FUNCTION

The TIME function returns information about the system time. The general form of TIME is:

TIME(format)

where the "format" parameter is one of the following.

SYS elapsed time in seconds since deadstart.

SYSMS elapsed time in milliseconds since deadstart.

JOB processing time in seconds since the start of this job or terminal session

JOBMS processing time in milliseconds since the start of this job or terminal session.

5.7 TOKEN - READ SES TOKEN FROM A STRING

TOKEN makes the internal lexical scanner of the SES processor available to the procedure writer. TOKEN reads the next SES token (syntactic unit) from a string variable. The calling format of TOKEN is:

TOKEN(source_string, token_string)

TOKEN reads the next available SES token from the variable "source_string", and places that token into the variable "token_string". TOKEN returns as a value one of the following:

- o If the token read from the string is a valid SES token, TOKEN returns a null (empty) string to indicate that all is well. Note that in this case, the token is deleted from "source_string", so that you can place calls on TOKEN into a loop, and get successive tokens from the source string.

- o If the next token in the string is not a valid SES token, TOKEN returns as a value a character string consisting of an error message indicating what is wrong with the token.

5.8 EXAMPLE - TIME, TOKEN AND EXPRESSION EVALUATOR

The example below is taken from an old version of the SES TIME procedure. TIME does not give you the time to the exact second, rather it gives you the time in words to the nearest five minutes. For example if you type SES.TIME, and the time was 11:17:43, SES would output the time in the form

* QUARTER PAST ELEVEN

Part of the procedure to accomplish this is:

```
\  sect0  = ' O''CLOCK'
\  sect1  = 'FIVE PAST'
\  sect2  = 'TEN PAST'
\  sect3  = 'QUARTER PAST'
\  sect4  = 'TWENTY PAST'
\  sect5  = 'TWENTY FIVE PAST'
\  sect6  = 'HALF PAST'
\  sect7  = 'TWENTY FIVE TO'
\  sect8  = 'TWENTY TO'
\  sect9  = 'QUARTER TO'
\  sect10 = 'TEN TO'
\  sect11 = 'FIVE TO'

\  h1 = ' ONE'
\  h2 = ' TWO'
\  h3 = ' THREE'
\  h4 = ' FOUR'
\  h5 = ' FIVE'
\  h6 = ' SIX'
\  h7 = ' SEVEN'
\  h8 = ' EIGHT'
\  h9 = ' NINE'
\  h10 = ' TEN'
\  h11 = ' ELEVEN'
\  h12 = ' TWELVE '

\  tiktok  = CLOCK(AMPM)                                  " 1 "
\  junk    = TOKEN(tiktok, hours)                         " 2 "
\  minutes = SUBSTR(tiktok, 2, 2)                         " 3 "

\  hours = (hours+((33<=minutes) AND (minutes<=59)))//12  " 4 "
\  hours = 'h'++hours+(hours = 0)*12                      " 5 "

\  sector = (minutes/5+(minutes//5>2))//12                " 6 "
\  sectnam = 'sect'++'&sector&'                           " 7 "
```

5.0 FUNCTIONS
5.8 EXAMPLE - TIME, TOKEN AND EXPRESSION EVALUATOR

This simple little procedure illustrates some of the more esoteric uses of the expression evaluator. The numbers that appear in comment quotes are for reference in the discussion that follows.

-- The assignment statements at the beginning are just initializing a bunch of strings which form parts of the eventual output.

line 1 calls the CLOCK function which returns the time. Let us suppose that the time is 4:20 pm. The variable "tiktok" will contain the string '4:20 PM'.

line 2 calls the TOKEN function which sets the value 4 into the variable "hours". The result of TOKEN is being placed in the variable "junk", because that's what it is in this application.

line 3 uses SUBSTR to get the "minutes" field out of "tiktok". We can't use TOKEN to get rid of the colon because colon is not a valid SES token. So we use SUBSTR to get the second and third characters of the string and place that in the variable "minutes".

line 4 is incrementing the "hours" counter if "minutes" lies anywhere between 33 and 59 minutes past the hour. The boolean expression ((33<=minutes) AND (minutes<=59)) will evaluate to either TRUE or FALSE, which is convertible to 1 or 0. Then we assign the whole expression modulo 12 to "hours".

line 5 is setting "hours" to one of the "hxx" variables defined at the start. The expression has to generate the string 'H12' if the value of "hours", set in line 4, turned out zero because of the modulo operator.

line 6 computes the "sector", that is, the five minute slot on the face of the dial. The expression will set the sector to "minutes"/5. But the expression also says that if it's 3,4,5,6 or 7 minutes past the hour, then we'll set it to five past the hour

line 7 builds a name "sectxx". The names "hxx" and "sectxx" can be accessed by substituting. For example, if the time is 4:43 pm, then "hours" will eventually contain the string 'H5', and "sectnam" will contain the string 'SECT9'. Then the substitution &hours& will give the string 'FIVE', and &sectnam& will give the string 'QUARTER TO'.

6.0 SES DIRECTIVES

In this section we'll look at all the SES directives that are not directly concerned with processing of parameters from the control statement. Again we'll provide a brief summary of the available commands, and go into more detail later on. The commands we're going to discuss in this section are.

Conditional Processing	The IF-ORIF-ELSE-IFEND commands provide a means to process the procedure conditionally.
Iterative Processing	WHILE-WHILEND provide a means of repeating a group of statements while some condition remains true.
Other Control Statements	CYCLE provides the means to go to the beginning of a WHILE loop. EXIT terminates its immediately enclosing structure, STOP terminates procedure processing and starts execution of the generated control statement file, while ABORT terminates procedure processing and returns control to the user.
Alternate File Creation	ROUT provides a capability to direct text from the body of a procedure to a designated file.
File Inclusion	INCLUDE can insert into the body of the procedure, the text of any other designated file.
User Interaction	The MSG command can send messages to any designated file. The ACCEPT command can read lines from any designated file. These two commands are most useful for making interactive procedures which may talk to the user. DAYFMSG allows messages to be written to the job dayfile during procedure processing.

6.1 IF - ORIF - ELSE - IFEND CONDITIONAL PROCESSING

The SES processor provides a method whereby a block of statements can be processed conditionally. The general form of the complete IF gang is laid out below.

```
\  IF some condition THEN              "  One of these  "
     blah
       blah
\  ORIF another condition THEN         "  There may be zero to  "
     mumble
       mumble
\  ORIF yet another condition THEN     "  many ORIF statements  "
     rhubarb
       rhubarb
\  ELSE                                "  zero or one of these  "
     yakk
       yakkity yakk yakk
\  IFEND                               "  Terminates the lot  "
```

To illustrate the use of IF, we'll look at the last few lines of the TIME procedure that was shown previously. Remember that we had the variables "sector", "sectnam" and "hours" set up. The small piece of conditional code in TIME is so that the time always comes out in the form of SOMETHING TO/PAST SOMETIME, <u>except</u> when it is the hour itself, in which case SOMETIME O'CLOCK will be output. The piece of code that does this is:

```
\  IF sector = 0 THEN
\    lastwrd = &hours& ++ &sectnam&
\  ELSE
\    lastwrd = &sectnam& ++ &hours&
\  IFEND

*    &lastwrd&
```

As you can see from the code, the two halves of the time string are arranged in a different order depending whether it's the hour or not.

6.2 WHILE - WHILEND REPETITIVE CODE PROCESSING

The WHILE command allows a section of code to be processed over and over as long as a condition is true. The general form of the WHILE command is:

```
\  WHILE condition is true DO
     bunches of
         procedure statements
\  WHILEND
```

6.3 CONTROL STATEMENTS

This section describes directives in addition to IF and WHILE that control which statements will be processed by SES. These directives are: STOP, ABORT, EXIT, and CYCLE.

6.3.1 STOP - STOP PROCEDURE PROCESSING

The STOP directive is used to prematurely terminate processing of the procedure. Its effect is equivalent to surrounding the part of the procedure that followed it with an IF/IFEND for which the condition is FALSE. The general form of the STOP directive is:

```
\ STOP [ string_expression ]
```

where string_expression specifies an optional message to be sent to the dayfile.

6.3.2 ABORT - ABORT PROCEDURE PROCESSING

The ABORT directive is similar to the STOP directive with the exception that the SES program will abort instead of executing the generated procedure. It can be used, for instance, when parameters to a procedure were not specified correctly. The general form of the ABORT directive is:

```
\  ABORT [ string_expression ]
```

6.3.3 EXIT - EXIT STRUCTURE

The EXIT directive is used to conditionally or unconditionally exit from the "immediately containing structure". This "structure" may be an IF "statement", WHILE "statement", or INCLUDEd procedure "segment". If the EXIT directive is not contained within any of these "structures", it acts like a conditional or unconditional STOP directive. The general form of the EXIT directive is:

```
\  EXIT [ WHEN boolean_expression ]
```

The exit is taken if the WHEN boolean_expression is omitted or if it is given and evaluates to TRUE. For example, the following are equivalent:

```
\  IF cond1 THEN                    \  IF cond1 THEN
     stuf and junk 1                     stuf and junk 1
\    EXIT WHEN cond2                \    IF NOT cond2 THEN
     stuf and junk 2                       stuf and junk 2
                                    \    IFEND
\  IFEND                            \  IFEND
```

6.3.4 CYCLE - NEXT ITERATION OF WHILE

The CYCLE directive can be used to proceed to the next iteration of the innermost WHILE "statement" that contains the CYCLE directive either conditionally or unconditionally. The general form of the CYCLE directive is:

```
\  CYCLE [ WHEN boolean_expression ]
```

The cycle is taken if the **WHEN** boolean_expression is omitted or if it is given and evaluates to TRUE. For example, the following are equivalent:

```
\  WHILE cond1 DO                  \  WHILE cond1 DO
     stuf and junk 1                    stuf and junk 1
\     CYCLE WHEN cond2             \     IF NOT cond2 THEN
     stuf and junk 2                      stuf and junk 2
                                   \     IFEND
\  WHILEND                         \  WHILEND
```

6.4 ROUT - ROUTEND ROUT TEXT TO A NAMED FILE

ROUT provides the ability to divert text from within the body of an SES procedure to a specified file. The form of the ROUT command is:

```
\  ROUT [FA=] file_name
  text
    to
      be
        routed to another file
\  ROUTEND [NOEOR=]  [file_name]
```

All the text within the ROUT - ROUTEND bracket is sent to the named file, with the proviso that any directive lines within the block are processed as they are encountered.

"file_name" is the file to which the text is to be routed. The optional FA keyword on the ROUT command specifies that the text is to be output in Full ASCII, that is, blank lines are output and lower case letters are not folded to upper case. If the FA parameter is not coded, then the output text has lower case letters folded to upper case, and blank lines are discarded on output.

When the ROUTEND command is encountered, SES normally writes an end of record on the file at that point. If the optional NOEOR parameter is coded on the ROUTEND, then the end of record is not written. This provides a useful facility to ROUT many sections of text to the same file in a disjointed fashion.

The "file_name" on the ROUTEND is optional. Its use is encouraged, since it makes the procedure easier to read, however, if the name on the ROUTEND doesn't match the name of the file at the top of the output control stack, then the ROUTEND directive is ignored.

It is possible to nest ROUT - ROUTEND blocks within other ROUT - ROUTEND blocks, as long as the inner ROUT's don't reference the same filename as the other ROUT's. If a ROUT directive does try to reference a file which is already being ROUT'ed to, then that ROUT directive is ignored.

ROUTing is particularly useful for creating a file, within a procedure, which is to be submitted as a batch job. For example:

```
\  jobfile = UNIQUE(NAME)
$SUBMIT(&jobfile&,B)
$RETURN(&jobfile&)
*    JOB &procnam& SUBMITTED

\  ROUT  jobfile
&user&,T2000.   *** &procnam& ***
$USER(&user&,&passwor&)
$CHARGE(&charge&,&project&)
  job
    control
      statements
\  ROUTEND  jobfile
```

could be used by a procedure to SUBMIT a batch job.

As a convention, procedures that ROUT stuff to files have the ROUT'ed text blocks at the end of the procedure. This makes the main body of the procedure easier to read, without it being cluttered up with all the ROUT'ed material.

6.5 INCLUDE - SWITCH INPUT TO A NAMED FILE

INCLUDE allows you to read text from a file other than the body of the current procedure file. The effect is as if the INCLUDE'd file was physically inserted in the procedure file body at the point of the INCLUDE command. The most primitive form of INCLUDE is:

```
\  INCLUDE F=file_name [,UN=user_name]
```

"file_name" is the name of the file to be INCLUDE'd, and the (optional) "user_name" specifies the catalog where the file is to be found. If the file is already local to the running job, then the local copy of the file is used.

The general, and probably more useful form of INCLUDE looks like this:

```
\  INCLUDE F=file_name, L=local_lib, LPFN=library_name, UN=user
```

In this format, "file_name" is still the name of the file to be INCLUDE'd, but now it refers to a procedure record in a PLIB library of name "library_name" in the catalog of the user given by "user". "local_lib" is the name of the LFN or Local File Name by which the library is known when SES ACQUIRE's the library. It is always, always, but always good practice to use a local file name for the library because INCLUDE's may be nested within INCLUDE's, and as long as the local file names are different, NOS is quite happy to let you read from different positions of the same file.

To illustrate how INCLUDE works, we'll show a section from the SES REPMOD procedure. This same INCLUDE file is used by all SES procedures which update libraries.

```
\  rewriti = '&intbase&'
\  rewrito = 'nb&'
\  INCLUDE 'REWRITE', L=UNIQUE(NAME), LPFN=SESLNAM, UN=SESUNAM
```

The procedure section shown above sets the input and output file names for REWRITE ("rewriti" and "rewrito"), and then INCLUDE's the REWRITE procedure. In actual fact, REWRITE is a stand alone procedure in its own right, and it is possible to simply use REWRITE as a standard SES procedure, such as.

```
SES.REWRITE I=&intbase&, O=&nb&
```

Why didn't we do it that way? Well the main reason is that the complete REPMOD procedure run (from procedure call to finishing the job) goes faster by INCLUD'ing the REWRITE procedure. If we'd written SES.REWRITE, then during the processing of REPMOD, we'dve hit the procedure call, searched the procedure libraries, cracked the control statement, etc, etc. The whole thing goes a lot faster for INCLUD'ing the inner procedure.

6.6 USER INTERFACE DIRECTIVES

The directives in this group can be used to "talk" to the user of a procedure and to let the user "talk" back. The directives are: DAYFMSG, MSG, and ACCEPT.

6.6.1 DAYFMSG - SEND MESSAGE TO DAYFILE

This directive can be used to place a message in the user's dayfile. The general form of the DAYFMSG directive is:

```
\ DAYFMSG string_expression
```

where string_expression defines the message to be sent.

6.6.2 MSG - WRITE MESSAGE TO FILE

This directive is used to write messages to a specified file. Its general form is:

```
\  MSG M=string_expression [ TO=file_name ]
```

where **string_expression** defines the message to be written and file_name specifies the name of the file to receive the message.

The default for **file_name** is OUTPUT. Note: that if **file_name** is omitted and file OUTPUT is not assigned to a terminal, the message is not written.

6.6.3 ACCEPT - READ 1 LINE FROM A FILE

The ACCEPT directive reads 1 line from a specified file into a specified variable, optionally preceeding the read request with a prompting message to another file. The general form of the ACCEPT directive is:

```
\  ACCEPT INTO=var_name [FROM=infile] [PROMPT=mesg] [TO=outfile]
```

where var_name is the name of the variable which will receive the line from file infile. The PROMPT and TO parameters are equivalent to the M and TO parameters, respectively, of the MSG directive (see above). The default for infile is INPUT.

Note: all parameters on directives are expressions; therefore it is strongly recommended that parameters which are to be names, be given as strings. For example:

```
\  ACCEPT INTO='var1' PROMPT=msg1
```

is generally much safer than:

```
\  ACCEPT INTO=var1 PROMPT=msg1
```

since in the second case, if var1 already has a value, that value will be used for the INTO parameter.

6.7 SETRFL - PROCEDURE FIELD LENGTH CONTROL

When a procedure is running, it is always good pratice to keep central memory field length to a minimum. This helps to provide better response time for all users (including you!), by reducing swap time. However, it is also a nice touch to restore the user's field length at procedure end to what it was when the procedure was called. The SETRFL directive provides the ability to do this. The format of the SETRFL directive is:

```
\  SETRFL min[..max]
```

the action of SETRFL is best explained in this set of SES code.

```
\  IF FL < min OR FL > max THEN
$RFL(&min&)
\    RFLLINE = '$RFL(&FL&)'
\  ELSE
\    RFLLINE = ' '
\  IFEND
```

in other words, if the current user's field length, given by the predefined variable FL is outside the limits specified by "min" and "max", then we generate a control statement to the control statement file to RFL to "min", and we then set the predefined variable "RFLLINE" to the control statement needed to restore the user's field length. Typically, we would then place an &RFLLINE& line at the end of the procedure. The "max" part of the SETRFL directive is optional, and if omitted, is the same as "min". In that case, the $RFL statement is generated if the current FL is unequal to that specified by "min".

7.0 PARAMETER DEFINITION AND PROCESSING

Now at last we come to the real purpose of SES, that is, reading parameters from the SES control statement and substituting them into the CCL statement file.

The topics discussed in this section should now be fairly straightforward. They are basically concerned with parameter definition via the PARM-PARMEND directives, and the various facilities for accessing parameter attributes and values.

7.1 PARM - PARMEND DEFINING PARAMETER LISTS

When we discussed the general layout of a procedure in the section on basic SES processing, we saw that a procedure may have zero to many PARM directives, terminated with a PARMEND directive.

PARM stands for PARaMeter, and it is the basic SES directive which defines what a procedure parameter may look like. PARM allows you to define the following things about a parameter.

- o The keyword or keywords used to define that parameter.
- o number of values which that parameter may be given when it is coded.
- o whether the parameter is required to be specified in the procedure call line.
- o the allowed type of the parameter.

The general form of the PARM directive is:

```
\ PARM KEY=keywords, [NVALS=xx], [type] [REQ] [RNG]
```

KEY identifies the keyword or keywords that may be used to specify this parameter when coding it on the control statement.

NVALS specifies the minimum and maximum Number of VALueS that may be coded for this parameter. Default is 1 (one).

type identifies the allowed type of the parameter. "type" may be coded as one of the following:

NAM specifies that the parameter must be a NAMe. That is, a one to seven character alphanumeric string starting with a letter.

NUM specifies that the parameter must be a NUMber, that is, a pure numeric string, with an optional base.

STR specifies that the parameter must be a STRing.

FGN designates that the parameter may be a ForeiGN text parameter. This type of parameter has the format of an expression (or parameter specification) but it is not evaluated when encountered, rather it is "passed on" essentially unmodified. Foreign text parameters are normally only used when value sub-lists are required for a parameter, and it then becomes the responsibility of the procedure to check the validity of the parameter. Details of the format (and restrictions) of foreign text values can be found in the subsection "Foreign Text" in the appendix on syntax.

REQ is a keyword that specifies that the parameter is REQuired to be stated when calling the procedure.

RNG states that the parameter may be coded as a RaNGe. That is, the parameter may be coded in the form of x..y, for example cols=2..75

The basis of all this definition is that SES checks the parameters given on the control statement to see if they are actually as you said they should be. If they are not, then SES outputs an error message at the time of processing the procedure, saving a massive amount of playing about in the body of the procedure itself.

Note that on an SES procedure call, a value can be omitted from a parameter's value list only if that parameter was not declared with an explicit type specifier on its PARM directive.

7.2 PARAMETER ATTRIBUTE TESTING

SES provides a number of functions to test the attributes of parameters as defined in the PARM directives. A short summary of the functions is provided below, and more detailed explanations follow.

In the discussions that follow, we use the convention that "parameter_name" means any of the keywords used to specify a particular parameter, and "keyword_name" to mean a specific keyword out of the set of possible keywords for a parameter. For example, if we'd coded the following PARM directive:

```
\  PARM KEY = ('i', 'f', 'input', 'file') etc.
```

then "parameter_name" means 'i' or 'f' or 'input' or 'file', whereas "keyword_name" means only one of those, say 'input'.

Function	Explanation
DEFP(parameter_name)	returns a true value if the parameter specified by "parameter_name" was actually coded on the control statement.
DEFK(keyword_name)	returns a true value if the keyword specified by "keyword_name" was actually coded on the control statement.
KEYVAL(parameter_name)	returns the keyword that was actually used to define the parameter specified by "parameter_name".
VCNT(parameter_name)	returns the number of values actually coded for the parameter specified by "parameter_name". VCNT is described in detail in the subsection on "Accessing Parameter Values".

7.2.1 DEFP - TEST FOR THE PRESENCE OF A PARAMETER

The DEFP function allows you to test if a parameter was actually defined. DEFP stands for DEFined Parameter. The general form of DEFP is:

```
DEFP(parameter_name)
```

where "parameter_name" is any of the keywords for the parameter in question. For example, there are many of the SES library procedures (or filters), that take one input file and produce one output file. These procedures are geared up so that if you only specify one filename, then when the procedure is finished it will write the output file over the input file. The piece of SES procedure code that would achieve this is:

```
\  IF NOT DEFP(o) THEN
\    o = '&i&'
\  IFEND
```

Later on in the procedure, we would use the fact that the 'i' and 'o' parameters are either equal to each other or not.

Note that if "parameter_name" is not the name of a parameter to the procedure, this function will return FALSE as its value.

7.2.2 DEFK - TEST FOR PRESENCE OF SPECIFIC KEYWORD

The DEFK function stands for DEFined Keyword, and its function in life is to test whether, when a parameter was coded, a specific keyword was used to define that parameter. The general form of DEFK is:

```
DEFK(keyword_name)
```

where "keyword_name" is the keyword for which we want to test. To illustrate the use of DEFK, we'll show a short extract from the FORMAT procedure. Most of the SES system library procedures which can run as batch jobs contain this particular section of code. The idea of the bit of code is to dump a dayfile to the user's catalog if there were any errors in the job. The dayfile parameter is defined via the following PARM directive.

```
\  PARM KEY=('nodayf','dayfile','df') NVALS=0..1 NAM
```

the 'nodayf' keyword specifies that no dayfile is required at all, in any event.

In the FORMAT job, the following piece of SES code may be found to process the parameter.

```
\  IF NOT DEFK(nodayf) THEN
EXIT.
$DAYFILE(&dayfile&)
$REPLACE(&dayfile&)
\  IFEND
```

To explain how this works, if the user coded nodayf as an option on the procedure call, then the test at the IF statement would fail, and none of the statements between the IF and the IFEND would be processed or output to the control statement file. However if the user coded df=some_file, or dayfile=some_file, or omitted the parameter altogether, then the test would succeed, and the EXIT, $DAYFILE and $REPLACE control statements would be processed to refer to whatever "some_file" happened to be, or would refer to "DAYFILE" if the parameter had not been coded.

<u>Note</u> that if "keyword_name" is not the name of a keyword to the procedure, this function will return FALSE as its value.

7.2.3 KEYVAL - ACCESS ACTUAL KEYWORD OF PARAMETER

KEYVAL stands for KEYword VALue, and its function is to let you know which keyword was actually used when specifying a parameter. The general form of KEYVAL is:

```
KEYVAL(parameter_name)
```

where "parameter_name" is <u>any</u> of the keywords that can be used to specify the particular parameter that you are interested in. The KEYVAL function returns as a string, the keyword that was actually used when the parameter was coded.

If no keyword was used to define the parameter, that is, the parameter was specified positionally, then KEYVAL returns the null string.

<u>Note</u> that if "parameter_name" is not the name of a parameter to the procedure, then this function returns the null string as its value.

7.3 ACCESSING PARAMETER VALUES

There are essentially two ways of getting at the value of a parameter: by substitution, and by using one of the functions described in the following subsections.

It is not possible to directly reference a parameter value in an expression, rather, one of the methods described above must be used. This is to allow a keyword for a procedure to have the same name as, for instance, one of the predefined variables, and yet within the procedure, to access both the parameter and the variable.

The substitution mechanism of SES always first checks for the name as being the name of a parameter, and only if this check fails does it look for a variable to substitute. This priority is also followed when assignment takes place, either explicitly via the assignment statement, or implictly via functions such as TOKEN and VALEXPR, discussed previously.

Substituting a parameter can be represented by the following SES code:

```
\  dummy = VALS(param, 1, LOV)
-----------  &dummy&  --------
```

where "dummy" is some temporary variable. In other words, the LOw Value, of value 1 for parameter "param" is substituted. You would actually code such a substitution as:

```
-----------  &param&  --------
```

The functions described in the following subsections may also be applied to variables with string values (in addition to parameters). When used for this purpose the string value must be in the format of a value list (see the Appendix on Syntax for a detailed description of the format of a value list). In particular, the interpretation of the string is that it contains a value list for a "parameter" defined by:

```
\  PARM NVALS=0..maxvals, FGN, RNG
```

The descriptions of the functions: VCNT, VALS, and GENLIST which follow only discuss their use with parameters in order to keep the description as simple as possible, however, the first argument to all these functions may also be the name of a variable whose value has the properties discussed above.

7.3.1 VCNT - NUMBER OF VALUES OF A PARAMETER

VCNT stands for Value CouNT, and its function is to determine the number of values coded for a parameter. The form of VCNT is:

```
VCNT(parameter_name)
```

where "parameter_name" is the name of the parameter for which you want the value count.

For example, the SES WIPEMEM procedure has one of its parameters defined via the following PARM directive:

```
\  PARM KEY = 'text' NVALS = 1..maxvals NAM
```

so that it's possible for the user to code a call on WIPEMEM which looks something like this:

```
SES.WIPEMEM text=(glug,grog,berk,clag)
```

so that within the WIPEMEM procedure, the assignment statement

```
\  memcoun = VCNT (text)
```

would set the variable "memcoun" to 4 in this case. The way that this is actually used is in a WHILE loop, something along the lines of:

```
\  index = 1
\  WHILE index <= VCNT (text) DO
     blah
       blah
         blah
\      index = index + 1
\  WHILEND
```

7.3.2 VALS - EXTRACT PARAMETER VALUE FROM A VALUE LIST

VALS is probably about the most useful function available to the writer of SES procedures. VALS stands for VALueS, and its function is to extract a value from a list of values which may be coded for a specific parameter. The general form of the VALS function is:

VALS(parameter_name, index, LOV/HIV)

where "parameter_name" is the parameter for which the value is to be extracted. "index" is an integer expression which determines which value out of the value list is to be extracted. The last parameter is LOV which stands for LOw Value, or HIV which stands for HIgh Value. This indicates whether the low or the high side of a range is to be extracted. As an example, the COLS parameter of the COPYACR procedure can be coded as

COLS=xx..yy

where "xx" is the low side of the range and "yy" is the high side of the range. The appropriate VALS functions is something like the following:

```
\  loside = VALS(cols, 1, LOV)

\  hiside = VALS(COLS, 1, HIV)
```

If the LOV or HIV parameter is omitted, then VALS takes the LOV as default. If the "index" parameter is omitted, then VALS uses 1 as default. So for instance the VALS function:

VALS (parameter_name)

is equivalent to the VALS function:

VALS (parameter_name, 1, LOV)

If the "index" parameter is given as less than or equal to zero, or greater than the maximum values allowed for any parameter (50) the error message **VALUE OUT OF RANGE** is given; or if "index" is greater than the actual number of values coded for the parameter, the function returns an "<u>undefined</u>" value.

If HIV was specified on the call to VALS but the value was not coded as a range (or was not allowed to have a range) the corresponding LOw Value is returned.

7.3.3 GENLIST - GENERATE LIST FROM PARAMETER LIST

The GENLIST function, for GENerate LIST, allows you to build up a string from a parameter list supplied to the SES procedure. The general form of GENLIST is:

```
GENLIST(parm_name, coun, line_lim, max_coun, range_sep, value_sep)
```

This looks complicated but since most of the arguments can be omitted, it's actually a lot simpler than it looks. The meaning of the various parameters of GENLIST are:

parm_name is the name of the parameter that you want to access.

coun is an index which indicates at which value in the list you want to start accessing. If "coun" is undefined when GENLIST is invoked, then GENLIST defines it for you, and initializes it to one (1). When GENLIST has processed the list, "coun" will be set to one greater than the last value processed in the list. If "coun" has a value (when GENLIST is invoked) which is less than one or greater than "max_coun", the error message VALUE OUT OF RANGE is given.

line_lim is the maximum number of characters you want to go in the generated list. The default is 80.

max_coun is the highest index that GENLIST is to process up to. The default is the actual number of values coded for the parameter.

range_sep is the character used to separate the low side and high side of a range value. If "range_sep" is omitted, it defaults to the SES range separator (..)

value_sep is the character used to separate values in the list. If "value_sep" is omitted, it defaults to comma (,)

To illustrate how GENLIST works, we'll look at a section of the SES WIPEMEM procedure. One of WIPEMEM's parameters is defined as follows.

```
\  PARM KEY = 'text' NVALS = 1..maxvals NAM RNG
```

The WIPEMEM procedure invokes LIBEDIT to actually delete the member records from the library. It is possible to give LIBEDIT a directive of the form.

```
*DELETE TEXT/GRAB,HOLD,HERE-THERE,JUNK-YUKK etc etc
```

Such a deletion could be coded on the SES control statement in the following manner:

```
SES.WIPEMEM L=MYLIB TEXT=(GRAB,HOLD,HERE..THERE,JUNK..YUKK)
```

The following section of code is taken from the WIPEMEM procedure, showing how GENLIST is used to generate the LIBEDIT directives.

```
\  memtyps = '(TEXT,OPLC,OPL,REL,OVL,ABS,PPU,PP,COS)'        " 1 "
\  dirfile = UNIQUE (NAME)                                   " 2 "
 .
   .
\  ROUT dirfile                                              " 3 "
\  typcoun = 1                                               " 4 "
\  WHILE typcoun <= 9 DO                                     " 5 "
\    memtype = VALS (memtyps, typcoun, LOV)                  " 6 "
\    IF DEFP (&memtype&) THEN                                " 7 "
\      memcoun = 1                                           " 8 "
\      WHILE memcoun <= VCNT (&memtype&) DO                  " 9 "
\        comd = GENLIST(&memtype&,memcoun,64,VCNT(&memtype&),'-')
*DELETE,&memtype&/&comd&                                     " 11 "
\      WHILEND                                               " 12 "
\    IFEND                                                   " 13 "
\    typcoun = typcoun + 1                                   " 14 "
\  WHILEND                                                   " 15 "
\  ROUTEND dirfile                                           " 16 "
```

line 1 defines the list of parameters for which LIBEDIT directives may be generated.

line 2 defines the name of the file to receive the directives.

line 3 initiates the directives file.

line 5 starts a loop which cycles through all of the directive generating parameters defined on line 1.

line 6 sets the variable "memtype" to the name of the next parameter for which directives may be generated.

line 7 checks whether the current parameter was specified when the procedure was called.

line 9 starts a loop which cycles through all values supplied for the current parameter when the procedure was called.

line 10 uses the GENLIST function to extract member names from the value list of the current parameter, and format them into a LIBEDIT directive.

line 11 causes the directive to be written to the directives file.

The remaining lines handle the cycling of the loops and the finishing off of the directives file.

7.4 DEFINING PARAMETER DEFAULTS

In this section we will describe some of the methods for defining default values for parameters (and keywords). The simplest way of setting a default for a single valued NAM parameter is to declare the default as one of the keywords for the parameter; then use that keyword throughout the procedure to refer to that parameter. For example, the SES REPMEM procedure contains:

```
\ PARM KEY=('g', 'group'), NVALS=1, NAM
```

which defines the parameter for the 'group' (of members) file. All through the procedure, this parameter is referred to via the 'group' keyword, thus if the parameter is not specified on the call REPMEM, GROUP will be substituted anywhere &group& appears.

When this method is not appropriate, one of the functions described in the following subsections could be useful. The purpose of these functions is to determine if a parameter was given a value (SETVAL) or if a keyword was used to specify the parameter (SETKEY). If this condition is true, then the SETVAL function is treated like the VALS function, and SETKEY is treated like KEYVAL. If the condition is false, the remaining processing done by both functions is the same. If the specified variable is defined (usually in the user's PROFILE) then its value is returned by the function, otherwise the specified default value is returned.

7.4.1 SETVAL - SET DEFAULT VALUE

The purpose of this function is to return a value for a parameter, much like the VALS function described earlier, in fact this function's last three parameters are treated just like the three parameters for the VALS function. The general form of the SETVAL function is:

```
SETVAL(default_value,var_name,parameter_name,index,LOV/HIV)
```

where "index" and "LOV/HIV" are the parameter value indices and are handled in the same manner as in the VALS function, "parameter_name" is any of the keywords for the parameter you are interested in, "var_name" is a variable name, and "default_value" is an expression.

The processing of the SETVAL function can best be explained in terms of the following pseudo SES code:

```
\ IF STR(VALS(parameter_name, index, LOV/HIV)) THEN
\   SETVAL = VALS(parameter_name, index, LOV/HIV)
\ ORIF DEF(var_name) THEN
\   SETVAL = var_name
\ ELSE
\   SETVAL = default_value
\ IFEND
```

7.4.2 SETKEY - SET DEFAULT KEYWORD

The purpose of this function is to establish a value for the keyword of a parameter. The general form of the SETKEY function is:

```
SETKEY(default_value,var_name,parameter_name)
```

where "parameter_name" is any of the keywords for the parameter you are interested in, "var_name" is a variable name, and "default_value" is an expression. The processing done by the SETKEY function can best be described by the following pseudo SES code:

```
\  IF KEYVAL(parameter_name) /= '' THEN
\    SETKEY = KEYVAL(parameter_name)
\  ORIF DEF(var_name) THEN
\    SETKEY = var_name
\  ELSE
\    SETKEY = default_value
\  IFEND
```

8.0 FILE SYSTEM DIRECTIVES

SES provides directives which allow you to issue file system commands directly from the body of an SES procedure. The commands fall into the groups of

- o File attribute testing similar to the NOS FILE function.
- o Rewinding and Returning Files
- o ACQUIRE and EXTRACT directives similar in function to the ACQUIRE and EXTRACT control statements (the latter are described in appendices to this document).

8.1 FILE - TESTING FILE ATTRIBUTES

SES implements the NOS FILE function which allows you to ask various questions about files. The tests that may be performed are described in the NOS reference manuals. The general form of the FILE function is:

FILE (file_name, expression)

where "file_name" is the name of the file to be tested, and "expression" is the test to be performed.

Don't forget that the FILE function implemented by SES tests the file attributes at the time the procedure body is being processed, and not when the generated control statements are actually being executed.

8.2 REWIND FILES

SES allows files to be rewound during SES processing. The format is:

REWIND F=list_of_file_names

where list_of_file_names is the name(s) of the file(s) to be rewound.

8.3 RETURN FILES

Files may also be returned during SES processing. The format is similar to the Rewind directive:

RETURN F=list_of_file_names

where list_of_file_names is the name(s) of the file(s) to be returned.

8.4 ACQUIRE DIRECTIVE

SES supports the ACQUIRE directive from inside the SES processor. The SES ACQUIRE directive works in the same manner as the ACQUIRE control statement. The general form of the ACQUIRE directive is:

```
ACQUIRE FN/F=local_file_name, PFN=permanent_file_name, UN=user_name
```

where "local_file_name" is the local file name when the file is ACQUIRE'd, "permanent_file_name" is the permanent file name of the file in the file system, and "user_name" is the name of the user who owns the file.

A complete description of the ACQUIRE control statement can be found in an appendix to this document.

8.5 EXTRACT DIRECTIVE

SES also supports the EXTRACT directive, which functions at procedure build time in the same manner as the EXTRACT control statement functions at procedure run time. The general form of the EXTRACT directive is:

```
EXTRACT F=lfn,R=rn,L=l,LPFN=lpfn,U=un,T=type
```

where the parameters of the EXTRACT directive have the following meaning.

F or FN	specifies the local File Name for the record when it has been EXTRACT'ed.
R or RN	specifies the Record Name of the record in the library.
L or LIB	specifies the Local file name of the LIBrary when the EXTRACT directive ACQUIRE's the library for processing.
LPFN or LIBPFN	specifies the LIBrary Permanent File Name of the library in the permanent file system.
U or UN	specifies the User NAMe of the file's owner.
T or RT	specifies the Record Type. The record type may be specified as TXT, TEXT, PP, ULIB, REL, OVL, ABS, OPL, OPLC, OPLD, PPU. If this parameter is omitted from the directive, then only the record name is used when searching the library, and the first record of that name is EXTRACT'ed.

A complete description of the EXTRACT control statement can be found in an appendix to this document.

9.0 PREDEFINED VARIABLES

When SES is called it sets up a number of variables which are available for use by the procedure writer to control the flow of procedures. Most of these predefined variables are a record of the user's environment at the time that SES was called.

9.1 SES SYSTEM DEFAULT VARIABLES

MAXVALS defines the MAXimum number of VALueS that may be coded for a parameter. It is set to 255.

LINELEN defines the maximum LINE LENgth. It is currently set to 80.

SESLNAM defines the default name for the SES Library NAMe. It is currently set to SESPLIB.

SESUNAM defines the default name for the SES User NAMe.

PRCLNAM defines the name of the file (library) from which the current procedure is being read.

PRCUNAM defines the user name for the owner of the file (library) from which the current procedure is being read. If the current procedure is being read from a local file, then this variable is set to the name of the current user.

HLPLNAM defines the default name for the SES HeLP Library NAMe. This library contains help documentation for standard SES procedures. It is set to SESHLIB.

HLPUNAM defines the user name for the owner of the help library. It is currently set to SES.

STALNAM defines the default name for the SES STAtus Library NAMe. This library contains status information for the standard SES procedures. It is set to SESSLIB.

STAUNAM defines the default name for the owner of the status library. It is currently set to SES.

PROCNAM contains the name of the procedure which is currently being processed.

PRIMOUT contains the name of the current PRIMary OUTput file.

USER contains the user name of the currently logged in user.

JOBNAME defines the name of the currently running job.

CSET contains the current character set of the user terminal. CSET may be either ASCII (1) or NORMAL (0). In batch mode, CSET contains NORMAL.

MODE defines the current mode of the procedure(s) being processed. MODE contains one of RUN, meaning that the procedure is being processed for execution in the control statement file, TEST, meaning that the procedure is being run in test mode, HELP, which means that the user wants help with the procedure, or STATUS, which means that the user wants the current status of the procedure.

SES_PROC_ERROR defines a numeric value that can be used to set the EF indicating an error was detected by the procedure, not the operating system. It is currently set to 60.

9.2 USER ENVIRONMENT VARIABLES

When the user makes an SES call, SES records information about the users environment at call time, so that a procedure writer may, if so desired, restore the user's environment at the end of the procedure. The data that is recorded is:

R1 thru R3 job control registers.

R1G global job control register.

EF error flag

EFG global error flag

SW1-SW6 sense switches 1 to 6.

FL field length at procedure call time.

RFLLINE — until otherwise changed (see the SETRFL directive), RFLLINE contains the character string '$RFL(&FL&)', where FL is as defined above.

ABL — account block limit.

JSL — job step limit

OT — origin type.

SS — sub system

TL — time limit.

Note, that because of the large number of built-in functions and pre-defined variables available to the SES procedure writer, there could be some confusion on the part of the procedure user when he/she chooses a name (for a file, etc.) which conflicts with one of the "built-ins". To avoid such confusion, SES will recognize only the names: **TRUE**, **FALSE**, **YES**, and **NO** when it scans the parameters on the control statement which calls a procedure.

A1.0 USEFUL PROCEDURE SEGMENTS

This appendix contains descriptions of some procedure segments that could be useful when writing SES procedures. These segments may be included in a procedure by means of the following directive:

```
\  INCLUDE 'segname', L=UNIQUE(NAME), LPFN=SESLNAM, UN=SESUNAM
```

where "segname" is the name of the desired procedure segment.

A1.1 CALPROC - CALL SES PROCEDURE

This procedure segment allows for easy calling by one procedure of another, when the calling procedure wishes to pass to the called procedure, parameters from its own parameter list. The procedure segment itself gives further details.

```
CALPROC
" August 31, 1981 "
      "  \  PARMEND  "

\  DIRCHAR = '!'
!  IF MODE = HELP THEN
\    INCLUDE 'CALPROCHLP' L=UNIQUE(NAME) LPFN=HLPLNAM UN=PRCUNAM
!  IFEND
!  DIRCHAR = '\'

 " CALPROC_COMMON "

\  calindx = 1
\  WHILE calindx <= VCNT(&calparm&) DO
\    IF (',' /= SUBSTR(calline, STRLEN(calline))) AND ..
        (SUBSTR(calline, STRLEN(calline)) /= '(') THEN
\      calline = calline ++ ','
\    IFEND
\    calline = calline ++ ..
        GENLIST(&calparm&, calindx, LINELEN-5-STRLEN(calline))
\    EXIT WHEN calindx > VCNT(&calparm&)
\    IF STRLEN(calline) <= STRLEN(' "." ,') THEN
\      ABORT '&calparm& PARAMETER VALUE TOO LONG'
\    IFEND
&calline& ..
\    calline = ' "." '
\  WHILEND
\  caltrlr = SETVAL('', caltrlr)
\  IF   STRLEN(caltrlr) + STRLEN(' "." ') > LINELEN THEN
\    ABORT '&calparm& TRAILER VALUE TOO LONG'
\  ORIF STRLEN(calline) + STRLEN(caltrlr) > LINELEN THEN
&calline& ..
 "." &caltrlr&
\    caltrlr = ''
\  ELSE
&calline&&caltrlr&
\    caltrlr = ''
\  IFEND

 " End of CALPROC_COMMON "
```

A1.2 JOBPARM - DEFINE PARAMETERS FOR BATCH JOBS

This procedure segment contains the PARM directives which define all the "standard" parameters used in procedures which may run as batch jobs. This procedure should be INCLUDEd in any SES procedure which will handle batch processing.

```
JOBPARM
" August 28, 1981 "
\  IF MODE = HELP THEN
\    INCLUDE 'JOBPARMHLP' L=UNIQUE(NAME) LPFN=HLPLNAM UN=PRCUNAM
\  IFEND

 " JOBPARM_COMMON "

\  PARM  KEY = 'jobun'     " user name "            NVALS = 1    STR
\  PARM  KEY = 'jobpw'     " password "             NVALS = 1    STR
\  PARM  KEY = 'jobfmly'   " family "               NVALS = 1    STR
\  PARM  KEY = 'jobcn'     " charge number "        NVALS = 1    STR
\  PARM  KEY = 'jobpn'     " project number "       NVALS = 1    STR
\  PARM  KEY = 'jobfl'     " field length "         NVALS = 1    NUM
\  PARM  KEY = 'jobtl'     " time limit "           NVALS = 1    NUM
\  PARM  KEY = 'jobpr'     " job priority "         NVALS = 1    NUM
\  PARM  KEY = ('local' 'batch' 'batchn' 'defer')   NVALS = 0
\  PARM  KEY = ('nodayf', 'dayfile', 'df')          NVALS = 0..1 NAM
      "  \  PARMEND  "

 " End of JOBPARM_COMMON "
```

A1.3 JOBHDR1 - PROCESS JOB PARAMETERS

This procedure segment will process the parameters for a batch job, setting up defaults, etc. Details of its function are described in the procedure segment itself.

```
JOBHDR1
" August 28, 1981 "
     "  \  PARMEND  "

\  IF MODE = HELP THEN
\    INCLUDE 'JOBHDR1HLP' L=UNIQUE(NAME) LPFN=HLPLNAM UN=PRCUNAM
\  IFEND

 " JOBHDR1_COMMON "

\  jobun    = SETVAL('USER', USER, jobun)
\  jobmode = SETKEY('LOCAL', jobmode, batch)

\  IF jobmode /= 'LOCAL' THEN

\    IF NOT DEF(PASSWOR) AND NOT DEFP(jobpw) THEN
\      IF FILE(INPUT NOT TT) OR FILE(OUTPUT NOT TT) THEN
\        ABORT 'PASSWORD NOT GIVEN'
\      IFEND
\      PARTIAL_PASSWOR = CHARREP(128)++'I ENTER PASSWORD ' ..
           ++CHARREP(13)++CHARREP(0) ..
           ++CHARREP(0)++CHARREP(10)++CHARREP(0)++'  HHHHHHHH'
\      PASSWOR = PARTIAL_PASSWOR ..
           ++CHARREP(13)++CHARREP(0)++CHARREP(0)++'  IIIIIIII' ..
           ++CHARREP(13)++CHARREP(0)++CHARREP(0)++'  ########' ..
           ++CHARREP(13)++CHARREP(128)++'A'
\      ACCEPT PROMPT PASSWOR TO 'OUTPUT' FROM 'INPUT' INTO 'PASSWOR'
\    IFEND
\    jobpw = SETVAL(notused, PASSWOR, jobpw)

\    IF NOT DEF(CHARGE) AND NOT DEFP(jobcn) THEN
\      IF FILE(INPUT NOT TT) OR FILE(OUTPUT NOT TT) THEN
\        ABORT 'CHARGE NUMBER NOT GIVEN'
\      IFEND
\      CHARGE = ' ENTER  CHARGE NUMBER ' ++ CHARREP(128) ++ 'A'
\      ACCEPT PROMPT CHARGE TO 'OUTPUT' FROM 'INPUT' INTO 'CHARGE'
\    IFEND
\    jobcn = SETVAL(notused, CHARGE, jobcn)

\    IF NOT DEF(PROJECT) AND NOT DEFP(jobpn) THEN
\      IF FILE(INPUT NOT TT) OR FILE(OUTPUT NOT TT) THEN
```

A1.0 USEFUL PROCEDURE SEGMENTS
A1.3 JOBHDR1 - PROCESS JOB PARAMETERS

```
\         ABORT 'PROJECT NUMBER NOT GIVEN'
\       IFEND
\       PROJECT = ' ENTER PROJECT NUMBER ' ++ CHARREP(128) ++ 'A'
\       ACCEPT PROMPT PROJECT TO 'OUTPUT' FROM 'INPUT' INTO 'PROJECT'
\     IFEND
\     jobpn = SETVAL(notused, PROJECT, jobpn)

\     jobfmly = SETVAL('', FAMILY, jobfmly)

\   IFEND

\   IF DEFP(jobfl) THEN
\     IF VALS(jobfl) < 70000(8) THEN
\       jobfl = 70000(8)
\     IFEND
\     jobfl = ',CM' ++ OCT(VALS(jobfl))
\   ELSE
\     jobfl = ''
\   IFEND

\   jobtl   = ',T' ++ OCT(SETVAL(2000(8), defjbtl, jobtl))

\   IF DEFP(jobpr) THEN
\     jobpr = ',P' ++ VALS(jobpr)
\   ELSE
\     jobpr = ''
\   IFEND

\   jobfile = UNIQUE(NAME)

 " End of JOBHDR1_COMMON "
```

A1.4 JOBHDR2 - PROCESS START OF JOB FILE

This procedure segment conditionally generates the code necessary to submit the procedure for batch processing. Details of its function are described within the procedure segment itself.

```
JOBHDR2
" August 28, 1981 "
      "  \  PARMEND  "

\  IF MODE = HELP THEN
\    INCLUDE 'JOBHDR2HLP' L=UNIQUE(NAME) LPFN=HLPLNAM UN=PRCUNAM
\  IFEND

 " JOBHDR2_COMMON "


\  IF STRLEN(PROCNAM) > 7 THEN
\    PROC_JOBNAME = SUBSTR(PROCNAM, 1, 7)
\  ELSE
\    PROC_JOBNAME = PROCNAM
\  IFEND

\  IF jobmode /= 'LOCAL' THEN
\    IF jobmode = 'BATCHN' THEN
$SUBMIT(&jobfile&,N)
\    ELSE
$SUBMIT(&jobfile&,B)
\    IFEND
$RETURN(&jobfile&)
REVERT.    JOB  &PROC_JOBNAME&  SUBMITTED
\    IF MODE = TEST THEN
&jobfile&
\    ELSE
\      ROUT  jobfile
\    IFEND
&PROC_JOBNAME&&jobfl&&jobtl&&jobpr&.   ***  &PROCNAM&  ***
\    IF VALS(jobfmly) ++ VALS(jobpw) = '' THEN
$USER(&jobun&)
\    ORIF  VALS(jobfmly) = ''  THEN
$USER(&jobun&,&jobpw&)
\    ELSE
$USER(&jobun&,&jobpw&,&jobfmly&)
\    IFEND
\    IF VALS(jobcn) ++ VALS(jobpn) /= '' THEN
$CHARGE(&jobcn&,&jobpn&)
\    IFEND
```

A1.0 USEFUL PROCEDURE SEGMENTS
A1.4 JOBHDR2 - PROCESS START OF JOB FILE

```
\    IF jobmode = 'DEFER' THEN
$CHEAP.
\      IF VALS(jobcn) ++ VALS(jobpn) /= '' THEN
$CHARGE(&jobcn&,&jobpn&)
\      IFEND
\    IFEND
&RFLLINE&
SESMODE,NEW.
\    EXIT WHEN  FILE('SES', NOT AS)
$GET(XSES/UN=&SESUNAM&)
$BEGIN(XSES,XSES)
\  IFEND

\  IF DEFP(jobfl) THEN
\    jobfl = SUBSTR(VALS(jobfl), 4, STRLEN(VALS(jobfl)) - 3)
\  ORIF DEF(defjbfl) THEN
\    jobfl = OCT(defjbfl)
\  ELSE
\    jobfl = '70000'
\  IFEND
\  IF NOT DEF(minjbfl) OR minjbfl < 70000(8) THEN
\    minjbfl = 70000(8)
\  IFEND
\  IF minjbfl > &jobfl&(8) THEN
\    jobfl = OCT(minjbfl)
\  IFEND
\  jobtl = SUBSTR(VALS(jobtl), 3, STRLEN(VALS(jobtl)) - 2)
\  IF DEFP(jobpr) THEN
\    jobpr = SUBSTR(VALS(jobpr), 3, STRLEN(VALS(jobpr)) - 2)
\  IFEND

 " End of JOBHDR2_COMMON "
```

A1.5 MSGCTRL - HANDLE MSG / NOMSG PARAMETER

This procedure segment will process the msg/nomsg keyword parameter used by many of the "standard" SES procedures. Details of its function are described in the procedure segment itself.

```
MSGCTRL
" August 31, 1981 "
      "  \  PARMEND  "

\  IF MODE = HELP THEN
\    INCLUDE 'MSGCTRLHLP' L=UNIQUE(NAME) LPFN=HLPLNAM UN=PRCUNAM
\  IFEND

 " MSGCTRL_COMMON "

\  IF DEFK(nomsg) OR (DEF(jobmode) AND jobmode /= 'LOCAL') THEN
\    sesmsg = '*'
\  ORIF NOT DEFP(msg) AND DEF(MSGCTRL) THEN
\    sesmsg = MSGCTRL
\  ELSE
\    sesmsg = 'SESMSG.*'
\  IFEND

 " End of MSGCTRL_COMMON "
```

A1.6 REWRITE - OVER-WRITE OR CREATE PERMANENT FILE

This procedure (segment) can be used to over-write or create a permanent file (if create, the file is defined as a direct access private, read-only permanent file. If the procedure is used as a procedure segment (i.e. INCLUDEd) the variables "rewriti" and "rewrito" must have been defined by the INCLUDing procedure.

```
REWRITE
" August 31, 1981 "

\  IF  MODE = HELP  THEN
\    INCLUDE 'REWRITEHLP' L=UNIQUE(NAME) LPFN=HLPLNAM UN=PRCUNAM
\  IFEND
\  PARM    KEY = ('i', 'rewriti')         NVALS = 1   NAM   REQ
\  PARM    KEY = ('o', 'rewrito')         NVALS = 1   NAM   REQ
\  PARM    KEY = ('un', 'rewritu')        NVALS = 1   STR
\  PARM    KEY = ('status', 'sts')        NVALS = 0
\  PARM    KEY = ('msg', 'nomsg')         NVALS = 0
\  PARMEND

 " REWRITE_COMMON "

\  rewritu = SETVAL(USER, rewritu, rewritu)
\  retryrw = UNIQUE(NAME)
\  donerw  = UNIQUE(NAME)
\  skiprw  = UNIQUE(NAME)
\  label1  = UNIQUE(NAME)
\  label3  = UNIQUE(NAME)
\  label4  = UNIQUE(NAME)
\  label5  = UNIQUE(NAME)
\  label6  = UNIQUE(NAME)
\  label7  = UNIQUE(NAME)
\  label8  = UNIQUE(NAME)
\  label9  = UNIQUE(NAME)
\  label10 = UNIQUE(NAME)
\  label11 = UNIQUE(NAME)
\  label12 = UNIQUE(NAME)
\  exittag = UNIQUE(NAME)
\  samerw  = ('&rewriti&' = '&rewrito&')

\  IF (DEFP(status)) OR (DEF(status)) THEN
\    rwfaild = '$SKIP(&exittag&)'
\  ELSE
\    rwfaild = 'EXIT. *** REWRITE FAILED ***'
\  IFEND
```

A1.0 USEFUL PROCEDURE SEGMENTS
A1.6 REWRITE - OVER-WRITE OR CREATE PERMANENT FILE

```
\  IF  PROCNAM = 'REWRITE' THEN

 " MSGCTRL_COMMON "

\  IF DEFK(nomsg) OR (DEF(jobmode) AND jobmode /= 'LOCAL') THEN
\    sesmsg = '*'
\  ORIF NOT DEFP(msg) AND DEF(MSGCTRL) THEN
\    sesmsg = MSGCTRL
\  ELSE
\    sesmsg = 'SESMSG.*'
\  IFEND

 " End of MSGCTRL_COMMON "

\  IFEND

$SET(EF=0)
$SET(EFG=0)
$SET(R1=1)
ACQUIRE(&rewriti&/A)
\  IF samerw THEN
\    pfnrw   = '&rewrito&'
\    rewrito = UNIQUE(NAME)
\    IF PROCNAM = 'REWRITE' THEN
$IFE(FILE(&rewriti&,PM),&label1&)
SESMSG. REWRITE NOT PERFORMED SINCE FILE
SESMSG. NAMES EQUAL AND &rewriti& PERMANENT
$ENDIF(&label1&)
\    IFEND
$IFE(FILE(&rewriti&,.NOT.PM),&skiprw&)
\  ELSE
\    pfnrw   = '&rewrito&'
\  IFEND

ACQUIRE(&rewrito&=&pfnrw&/UN=&rewritu&)

\  IF VALS(rewritu) = USER THEN
$IFE(FILE(&rewrito&,PM),&label3&)
ACQUIRE(&rewrito&=&pfnrw&/PO,M=W)
$ENDIF(&label3&)
$WHILE,TRUE,&retryrw&.
$IFE(FILE(&rewrito&,.NOT.AS),&label4&)
$DEFINE(&rewrito&=&pfnrw&/M=R)
$ENDIF(&label4&)
ACQUIRE(&rewrito&=&pfnrw&/A,M=W)
$IFE(FILE(&rewrito&,PM),&label5&)
$EVICT(&rewrito&)
$COPYEI(&rewriti&,&rewrito&,VERIFY)
```

A1.0 USEFUL PROCEDURE SEGMENTS
A1.6 REWRITE - OVER-WRITE OR CREATE PERMANENT FILE

```
$ELSE(&label5&)
$REPLACE(&rewriti&=&pfnrw&)
$ENDIF(&label5&)
\  ELSE
$IFE(FILE(&rewrito&,PM),&label6&)
ACQUIRE(&rewrito&=&pfnrw&/PO,M=W,UN=&rewritu&)
$ENDIF(&label6&)
$WHILE,TRUE,&retryrw&.
ACQUIRE(&rewrito&=&pfnrw&/M=W,UN=&rewritu&)
$IFE(FILE(&rewrito&,.NOT.AS),&label7&)
$SET(EF=1)
&rwfaild&
$ENDIF(&label7&)
$IFE(FILE(&rewrito&,PM),&label8&)
$EVICT(&rewrito&)
$COPYEI(&rewriti&,&rewrito&,VERIFY)
$ELSE(&label8&)
$REPLACE(&rewriti&=&pfnrw&/UN=&rewritu&)
$ENDIF(&label8&)
\  IFEND

$SKIP(&donerw&)
EXIT.
$IFE((EF.NE.ODE).AND.(EF.NE.TKE).AND.(EF.NE.PPE),&label9&)
&rwfaild&
$ENDIF(&label9&)
$SET(R1=R1+1)
$IFE((R1.GT.5),&label10&)
&rwfaild&
$ENDIF(&label10&)
$SET(EF=0)
$REWIND(&rewriti&,&rewrito&)
&sesmsg&    REWRITE FAILED - WAITING TO TRY AGAIN
$ROLLOUT(120)*    REWRITE FAILED - WAITING TO TRY AGAIN
$ENDW(&retryrw&)

$ENDIF(&donerw&)
\  IF samerw THEN
$ENDIF(&skiprw&)
\  IFEND

\  IF   PROCNAM /= 'REWRITE' THEN
$RETURN(&rewriti&,&rewrito&)
\  ORIF samerw THEN
$RETURN(&rewriti&,&rewrito&)
ACQUIRE(&rewriti&/A,UN=&rewritu&)
\  ELSE
$REWIND(&rewriti&)
```

A1.0 USEFUL PROCEDURE SEGMENTS

A1.6 REWRITE - OVER-WRITE OR CREATE PERMANENT FILE

```
$IFE(FILE(&rewrito&,PM),&label12&)
$RETURN(&rewrito&)
$ENDIF(&label12&)
ACQUIRE(&rewrito&/A,UN=&rewritu&)
\  IFEND
$ENDIF(&exittag&)

" End of REWRITE_COMMON "

\  IF samerw THEN
REVERT.   END &PROCNAM&  &rewriti&
\  ELSE
REVERT.   END &PROCNAM&  &rewriti& -> &rewrito&
\  IFEND
```

B1.0 OPERATING MODES OF THE SES PROCESSOR

The SES processor may process a procedure in one of four modes:

RUN This is the normal mode. The procedure is processed, presumably generating control statements, and then these control statements are executed.

TEST In this mode the procedure is processed in the normal manner, but the generated control statements are not executed, instead they are placed on a designated file for possible inspection by the user. This mode is meant as an aid in debugging new procedures.

HELP This mode is similar to test mode, however instead of generating control statements, a procedure set up for HELP mode will produce some documentation on its purpose and usage.

STATUS This mode is identical to help mode, except a procedure set up for STATUS mode will provide the current status of the procedure.

The modes are selectable by the user by means of parameters to the SES processor; and the procedure can determine in which of the modes it was called by means of predefined variables set up by the SES program. These variables are:

MODE This variable may be compared with the variables RUN, TEST, HELP, or STATUS to determine which of the modes is in effect; for example:

```
\  IF   MODE = HELP THEN
        " code for HELP mode "
\  ORIF MODE = TEST THEN
        " code for TEST mode "
\  ORIF MODE = STATUS THEN
        " code for STATUS mode "
\  ELSE
        " code for RUN mode "
\  IFEND
```

PRIMOUT This variable contains the name of the PRIMary OUTput file. In RUN mode this is the new control statement file;

in TEST mode this is the file designated by the test mode parameter on the SES call (default is SESTEST); and in HELP or STATUS mode this is the file designated by the help or status mode parameter on the SES call (default is OUTPUT). PRIMOUT is particularly useful in HELP or STATUS mode for directing the descriptive information about the procedure to the file selected by the user on the SES call. This may be accomplished as follows:

```
\  IF MODE = HELP THEN
\    ROUT FA=PRIMOUT
       " descriptive information about called procedure "
\    ROUTEND PRIMOUT
\    STOP
\  IFEND
```

Note, in HELP or STATUS mode, a PARMEND directive will be interpreted as a STOP directive, to prevent a procedure not set up for HELP or STATUS mode from doing strange or undesirable things.

B1.1 SELECTING MODE OF OPERATION

As stated above, the mode of operation for a procedure is selected by a parameter to the SES processor.

TEST mode may be selected by one of the keywords: TEST or T. For example:

```
ses,test.procedure_name list_of_parameters
```

will process procedure "procedure_name" in TEST mode, and place the generated control statements on file SESTEST; whereas:

```
ses,t=my_file.procedure_name list_of_parameters
```

will process procedure "procedure_name" in TEST mode, but places the generated control statements on file "my_file".

HELP mode may be selected by one of the keywords: HELP or H. For example:

```
ses,help.procedure_name
```

causes procedure "procedure_name" to be processed in HELP mode, and any descriptive information available will be placed on file OUTPUT ; whereas:

B1.0 OPERATING MODES OF THE SES PROCESSOR
B1.1 SELECTING MODE OF OPERATION

ses,h=my_info.procedure_name

causes procedure "procedure_name" to be processed in HELP mode, but any descriptive information available will be placed on file "my_info".

STATUS mode may be selected by one of the keywords: STATUS or S. For example:

ses,status.procedure_name

causes procedure "procedure_name" to be processed in STATUS mode, and any status information available will be placed on file OUTPUT; whereas:

ses,s=my_info.procedure_name

causes procedure "procedure_name" to be processed in STATUS mode, but any status information available will be placed on file "my_info".

Note, that when calling a procedure in HELP or STATUS mode, a list of parameters should not be given. HELP or STATUS for a group of procedures may be obtained by one call to SES, as follows:

ses,help.proc_1; proc_2; proc_3

C1.0 ERROR MESSAGES

This appendix describes the messages produced by the SES processor when errors are detected. SES error messages have been made as self-explanatory as possible. When an error is detected by SES, a message is printed in the form:

```
** E CL 11001: EXPECTING "name found integer for parameter I" ON
COMMAND STATEMENT
```

The E at the beginning of the line indicates this is an error message.

The CL is an abbreviation for the System Command Language used by SES to do syntax processing.

The number 11001 is an error code assigned to this error condition.

The text which follows the error code describes the error in detail. Appended to the end of the text is the line number of the line being processed by SES. In this first example, it is the command statement which is in error.

After the error message, SES outputs the line it was processing when the error was detected, followed by a line containing an up_arrow at the point in the line where the error was detected.

```
SES.REWRITE  I=123  O=ABC
                  ^
```

Usually the error actually occurred on the token just before the up_arrow.

Here are two more typical examples of error messages:

```
** F CL 11007: REQUIRED PARAMETER MISSING "I" ON COMMAND
STATEMENT
SES.FORMAT
          ^
```

```
** E CL 11011: UNKNOWN KEYWORD "NVLS" ON LINE # 7 OF PROC SEGMENT
MYPROC
PARM  KEY = ('group', 'g')  NVLS = 1   NAM
                                 ^
```

C1.0 ERROR MESSAGES

Other abbreviations used in SES error messages are SE, which means the error was detected in the processor itself, and UT, which means the error was detected by a utility routine called by SES.

D1.0 SEMI-FORMAL SYNTAX DESCRIPTION

This section gives a semi-formal description of the syntax used when writing procedures for and calling the SES. The description is not intended to be rigorous. First we introduce the "meta-language" used to describe the syntax; second the character set used by SES is defined; and finally the syntax description itself is given.

D1.1 THE META LANGUAGE

This section describes the symbols used in the description of the SES syntax.

Symbol — **Interpretation**

::= — This symbol should be read as "is defined to be".

| — This symbol is used to indicate alternatives, for example: **A** | B means that either **A** or B is allowed.

<item> — This group of symbols denotes that **item** is to be treated as a syntactic unit in relation to surrounding meta symbols.

[item] — This group of symbols denotes that **item** is optional, i.e. zero or one occurences of **item** are allowed.

{item} — This group of symbols denotes that **item** may be used zero or more times.

Spaces are used in the syntax description to improve its readability, however they are not part of what's being defined unless otherwise noted.

There are a few instances where some of the meta symbols themselves are part of the syntax definition, and when this occurs the meta symbol is underlined, for example: <u>|</u> means the | character and not the meta symbol. When an _ appears alone, it means itself.

D1.2 CHARACTER SET

Characters used for NAMES

A .. Z a .. z	Letters
0 .. 9	Decimal Digits
_	Underline
$	Dollar Sign
#	Pound
@	Commercial At

Characters used for INTEGER CONSTANTS

0 .. 9	Decimal Digits
A .. F a .. f	Hexadecimal Digits
(	Open Parenthesis
)	Close Parenthesis

Characters used for OPERATORS

+	Plus Sign
-	Minus Sign
*	Asterisk
/	Slash (Slant)
=	Equal Sign
>	Greater Than Sign
<	Less Than Sign

Characters used for PUNCTUATION

	Blank (Space)
,	Comma
(	Open Parenthesis
)	CLose Parenthesis
.	Period

Character used for STRING DELIMITER

'	Apostrophe (Single Quote)

Character used for COMMMENT DELIMITER

" .... (Double) Quote

(Default) Character used for SUBSTITUTION DELIMITER

& .... Ampersand

(Default) Character used for DIRECTIVE HEADER

\ .... Reverse Slash (Slant)

Note: Any ASCII character not listed in the above character set has no meaning to the SES processor. These characters may however be used in strings, comments, or as data characters.

D1.3 SYNTAX

D1.3.1 BASIC DEFINITIONS

```
<upper case letter> ::= A | B | C | D | E | F | G | H
                      | I | J | K | L | M | N | O | P
                      | Q | R | S | T | U | V | W | X
                      | Y | Z

<lower case letter> ::= a | b | c | d | e | f | g | h
                      | i | j | k | l | m | n | o | p
                      | q | r | s | t | u | v | w | x
                      | y | z

<letter> ::= <upper case letter>
           | <lower case letter>

<decimal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<hexadecimal digit> ::= A | B | C | D | E | F
                      | a | b | c | d | e | f

<digit> ::= <decimal digit>
          | <hexadecimal digit>

<base> ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
         | 11 | 12 | 13 | 14 | 15 | 16
```

D1.3.2 TOKENS

This section defines the building blocks of SES syntax, collectively referred to as tokens. The internal token scanner of the SES processor is made availble to the procedure writer by means of the built-in function TOKEN.

```
<token> ::= <name> | <number> | <string>
          | <delimiter> | <operator>

<name> ::= <alphabetic char> {<alphabetic char> | <decimal digit>

<alphabetic char> ::= <letter> | _ | $ | # | @


<upper case name> ::=
```

```
<upper case letter> {<upper case letter> | <decimal digit>}
```

All names are limited to thirty-one characters in length, except procedure names and procedure identifiers, which are limited to ten characters. With the exception of <upper case name>s, any name may be specified with either upper or lower case letters, but before a name is used all letters in it are converted to upper case. For instance the names: **ABC**, **abc**, **aBc**, and so on, are all equivalent. (This includes any of the "special" names, such as **DO**, **THEN**, **WHEN**, etc. In this description, however, these names are always spelled out in upper case letters.)

```
<variable name> ::= <name>

<function name> ::= <name>

<parameter name> ::= <name>

<directive name> ::= <name>

<assignee> ::= <parameter name> | <variable name>

<procedure name> ::= <name>

<procedure identifier> ::= <upper case name>


<number> ::= <decimal digit> {<digit>} [(<base>)]


<string character> ::= ''
     | <any ASCII character except '>

<string> ::= '{<string character>}'


<constant> ::= <string> | <number> | <name>


<delimiter> ::= , | ( | ) | = | . | ..{.}
              | <end of line>

<operator> ::= <graphic operator> | <mnemonic operator>

<graphic operator> ::= ** | * | / | // | + | - | ++
```

| = | /= | <> | ≤ | ≤= | ≥ | ≥=

<mnemonic operator> ::= AND | OR | XOR | NOT

D1.3.3 USE OF SPACES

Before discussing when and how spaces can be used we will first define the syntax of comments.

<comment> ::= "{<any ASCII character except ">}"

In almost all cases a comment is treated identically to a single blank character, and 2 or more contiguous blank characters (or comments) are treated as a single blank character. Blank characters and comments treated in this manner are known as spaces.

Spaces may be used between tokens to improve readability and in general may be used to replace commas when used as argument, value, or parameter separators. Spaces must be used to separate tokens when no <delimiter> or <graphic operator> can be used to separate them. For example the spaces between the tokens on the following line must be present:

V1 AND V2

whereas the following two expressions are equivalent:

V1 + V2
V1+V2

Further, the following value list contains 2 values:

(X, -3)

whereas the next contains only 1 value:

(X -3)

namely the value of the expression X-3.

Spaces within character strings represent themselves, and comments may not be used in front of the \ which occurs at the beginning of directive lines, nor following the continuation signal at the end of directive or call lines. Lines within procedures which are not directives or continuations of directives or lines which are read using the ACCEPT directive, are treated as unquoted strings, and therefore spaces are significant in them. Whenever a line is read by the SES processor, trailing blank characters are deleted. Also, it is legal to precede the \ of a directive line by one or more blank characters.

D1.3.4 EXPRESSIONS

```
<expr> ::= <lterm> {<or> <lterm>}
    <or> ::= OR | XOR

<lterm> ::= <lfactor> {AND <lfactor>}

<lfactor> ::= [NOT] <lprimary>

<lprimary> ::= <sterm> {<rel op> <sterm>}
    <rel op> ::= = | /= | ≤ | ≤= | ≥ | ≥=

<sterm> ::= <term> {++ <term>}

<term> ::= [<term op>] <factor> {<term op> <factor>}
    <term op> ::= + | -

<factor> ::= <primary> {<factor op> <primary>}
    <factor op> ::= * | / | //

<primary> ::= <operand> {** <operand>}

<operand> ::= <variable reference>
            | <function reference>
            | ( <expr> )
            | <constant>
            | <null>

<null> ::=

<variable reference> ::= <variable name>

<function reference> ::= <function name> <arguments>

<arguments> ::= ( [<arg> {, <arg>}] )
              | <null>

<arg> ::= <name> | <expr>


<integer expr> ::= <expr>    " must resolve to an integer "
<string  expr> ::= <expr>    " must resolve to a string   "
<boolean expr> ::= <expr>    " must resolve to an integer "
                             "   if the value is zero, it "
                             "     is taken to be FALSE   "
                             "   if non-zero, it's taken  "
                             "     to be TRUE             "
```

D1.3.5 FOREIGN TEXT

Foreign text is primarily used for parameter values which are to be in turn used as parameter lists (e.g. to secondary procedures) or simply to prevent the SES processor from evaluating an expression.

The scanning of foreign text is totally different from scanning "normal" text. The characteristics of this special scanning are

- parentheses are "balanced"
- single and double quotes are "matched"
- if not contained within parentheses, single quotes, or double quotes, the tokens: comma, period, ellipsis (..{.}), and close parenthesis will terminate the scanning (and thus the foreign text value). In addition, spaces which are used to separate names, numbers, or strings from names numbers or strings will terminate scanning; as will an "unenclosed" open parenthesis which follows a string or number (Note, that an open parenthesis following a name does not terminate scanning - this is because function references are allowed in foreign text but the foreign text scanner doesn't evaluate what it scans, and thus does not know if the name is indeed the name of a function).

Foreign text may also be described as having the general format of an expression, but the expression is not evaluated when scanned as foreign text. During scanning comments and blanks not contained within single quotes are "thrown away" and single blank characters are inserted between tokens which would otherwise not be separated.

The following example illustrates some of the idiosyncracies of foreign text:

```
\  vlist = '(a b c (d e) ''p q''''r, s'' 123(8) (x,(y+3)) )'
\  count = VCNT (vlist)                      " 2 "
\  value = VALS (vlist, 3)                   " 3 "
\  slist = GENLIST (vlist, index)            " 4 "
```

The first line defines a value list in the variable vlist. Line 2 sets the variable count to the value 6. Line 3 sets the the variable value to the value:

C(D E)

and line 4 sets the variable slist to the value:

A,B,C(D E),'p q''r, s',123(8),(X,(Y+3))

The next example illustrates how a parameter list may be passed as a foreign text parameter:

```
\  plist = '( i=infile  "columns"  cols=1..80 o=out )'
\  count = VCNT (plist)
\  low   = VALS (plist, 2, LOV)
\  high  = VALS (plist, 2, HIV)
\  slist = GENLIST (plist, index)
```

Count is set to 3; low is set to:

COLS=1

high is set to 80; and slist is set to:

I=INFILE,COLS=1..80,O=OUT

D1.3.6 PARAMETER LISTS

```
<parameter list> ::= [<parameter> {[,] <parameter>}]

<parameter> ::= [<parameter name> [=]] <value list>
              | <parameter name>
              | <null>

<value list> ::= <value>
               | ( [<value> {[,] <value>}] )

<value> ::= <value side> [..{.} <value side>]

<value side> ::= <expr> | <foreign text>
```

D1.3.7 SES PROCESSOR CALL

```
<csep> ::= [;|.] <end of line>
         | ; | .

<proc call> ::= <procedure name> [,] <parameter list> <csep>

<control statement> ::= <string>

<control statements> ::= <control statement> [<csep>]
                  {[,] [<control statement>] [<csep>]}

<call element> ::= <proc call> | <control statements>

<SES call> ::= SES [, <parameter list>] .
                   <call element> {<call element>}
```

Because of operating system restrictions, a <parameter list> following the SES (processor name) must have explicit punctuation. That is to say, commas must be used to separate parameters (and values) and equal signs must be used to separate parameter names (keywords) from their value lists.

Also, the operating system is not well acquainted with lower case letters, so only upper case should be used; however, NAM/IAF (or TELEX) and the SES processor alleviate this problem by converting lower case letters to upper case on command and continuation lines.

When <control statements> are used in a <SES call>, the SES processor insures that they are all "properly" terminated, i.e. each <control statement> string is scanned for a right parenthesis or period and if neither of these characters is found, a period will be appended at the end of the string; if however, a right parenthesis or period is found, the string will be left alone. NOTE that this is the only validity checking of the <control statement> done by the SES processor.

D1.3.8 SUBSTITUTION

```
<substitute> ::= <parameter name> | <variable name>

<default substitution character> ::= &

<alternate substitution character> ::=
      !|"|#|$|%|'|(|)|*|=|+|-|/|~|^|`|@|,|.|;|:|?
     |\|_|<|>|[|]|{|}||

<substitution char> ::=
      <default substitution character>
     | <alternate substitution character>

<substitution> ::=
    <substitution char> <substitute> <substitution char>
```

D1.3.9 PROCEDURES

```
<procedure> ::= <procedure identifier> {<procedure line>}

<procedure line> ::= {<procedure line element>}

<procedure line element> ::= <substitution>
                           | <any ASCII character>
```

The process of substitution applied to a <procedure line> yields an <object line>.

```
<object line> ::= <directive line>
                | <empty line>
                | <data line>

<default directive character> ::= \

<alternate directive character> ::=
     !|"|#|$|%|'|(|)|*|=|+|-|/|~|^|`|@|,|.|;|:|?
    |&|_|<|>|[|]|{|}||

<directive header> ::=
      <default directive character>
    | <alternate substitution character>

<directive line> ::= <directive header> <directive>

<empty line> ::=

<data line> ::= <any line which is not "empty" and does
                 not begin with a directive header>
```

Note: <empty line>s may contain comments enclosed in double quotes.

D1.3.10 DIRECTIVES

```
<directive> ::= <assignment>
    | <if while> <boolean expr> [<then do>]
    | <exit cycle> [WHEN <boolean expr>]
    | <directive name> <parameter list>

<assignment> ::= <assignee> = <expr>

<if while> ::= IF | ORIF | WHILE

<then do> ::= THEN
            | DO

<exit cycle> ::= EXIT | CYCLE
```

D1.4 LINES AND THEIR CONTINUATION

It is sometimes necessary to pass more parameters to a procedure (or give more parameters to a directive) than will fit on one line (lines are normally limited to 80 characters in length, however, TELEX further limits the command lines to about 70 characters -- for reasons known only to TELEX -- continuation lines entered at the terminal may, however, be 80 characters long). To handle this problem, SES processes continuation lines.

The effective net result of using continuation lines is to construct an unbroken line of up to 256 characters.

Continuation may only be used in conjunction with SES directives and when calling SES to process a procedure. Continuation is signalled on the line which is to be continued, not the continuation line itself. Note that the <continuation signal> is not considered to be part of the line. The mechanism for doing this is defined as follows:

```
<whole line> ::=
    <line starter> <stuf 1> [ <continuation signal>
                   <stuf 2> { <continuation signal>

                   <stuf n> } ]


<continuation signal> ::= ..{.}

<line starter> ::= <directive header> <name>
                 | SES <parameter list>

<stuf i> ::= <whatever belongs with the line starter>
      " 1 <= i <= n "
```

The effect of this is as if <whole line> had been specified as:

```
<line starter> <stuf 1> {<stuf i>}
```

Note: Syntactic units (tokens) may cross line boundaries.

E1.0 ACQUIRE UTILITY

ACQUIRE is a program that enables easy retrieval (acquisition) of permanent files.

ACQUIRE combines the functions of the NOS "ATTACH" and "GET" control statements. For each file specified ACQUIRE determines if the file is already local to the job (unless suppressed by the PO parameter, see below), if so it is rewound; if not, then for each one of a list of user names, an ATTACH is attempted (waiting, if necessary, until the file is not busy), and if that fails a GET is tried. If, after all this, the file is still not local, an appropriate dayfile message is issued.

Unless the A (abort) parameter is specified, ACQUIRE will abort only because of control statement format or argument errors, or because of a permanent file manager (PFM) detected error; and not because one (or more) of the specified files could not be found.

The control statement format for ACQUIRE is :

```
ACQUIRE(lfn1=pfn1,lfn2=pfn2,.../op1,op2,...)
```

lfni — is the (local) name of the file once it has been ACQUIREd (note that this is the name used in making the "is the file already local?" test)

pfni — is the permanent file name for the file (if =pfni is omitted, pfni is assumed to be the same as lfni)

opi — specify options used for acquiring the file(s) :

- A — specifies that if a file is not found, the program should abort
- NA — is the opposite of A (and is the default)
- PO — specifies Permanent Only, i.e. that if a file is already local, it will be returned and then the ATTACH and GET will be attempted
- UN=users — specifies a list of user names to be searched for each file (the user names are separated from each other by commas)

E1.0 ACQUIRE UTILITY

M=mode specifies the access mode desired for the file (READ or R -- the default, WRITE or W, or EXECUTE or E)

PW=pw specifies the permanent file's password

PN=pn specifies the packname for the permanent file

When ACQUIRE is attempting an ATTACH or GET, if the file is busy or if a permanent file utility is active, the following message will be issued and the request will be retried :

- WAITING FOR PFN=permanent_file_name UN=user_name

When ACQUIRE is attempting an ATTACH or GET, if an error is detected by PFM the following message is issued and the program is aborted :

- ERROR WITH PFN=permanent_file_name UN=user_name

In both of the above cases, the designated message will be preceeded by a more specific message generated by PFM.

F1.0 EXTRACT UTILITY

EXTRACT is a program that enables easy retrieval of records from permanent file (or local) libraries.

Although the program is designed primarily for use in procedure files, it can be very useful on its own.

EXTRACT is similar in function to the NOS "GTR" statement. It differs from "GTR" in the following ways:

- o EXTRACT insists that the library to be searched has a directory (this can be built using the NOS utility "LIBEDIT").
- o The record type parameter for EXTRACT, if given, applies to all records to be extracted, and if not given, only the names of the records are used when searching the library.
- o Each extracted record is copied to its own local file by EXTRACT, rather than all to the same file.
- o EXTRACT does not insist that the library to be searched be local to the job when it's called, but will ACQUIRE the library from a permanent file catalog.

The control statement format is:

```
EXTRACT(lfn1=rn1,lfn2=rn2,.../op1,op2,...)
```

lfni Is the local file name given to the record once it's extracted (lfni is REWOUND before and after the extraction takes place).

rni Is the name of the record to be extracted (if omitted, it is assumed to be the same as lfni).

opi These parameters specify options that control the extraction process :

A Specifies that if a record is not found, the program should abort.

F1.0 EXTRACT UTILITY

NA	Is the opposite of A (and is the default).
T=rt	Specifies the record type (if given, it applies to all records being extracted; if omitted, only the record names are used when searching the library).
L=libname	Specifies the name of the library to be searched for the records (if omitted, "PROCLIB" is assumed).
LFN=liblfn	Specifies the local file name for the library (if omitted, the "libname" from the L paraeter is used). Note that this is the name used to make the "is file local?" test when ACQUIRing the library.
UN=un	Specifies the user name of the permanent file catalog to be searched for "libname" if it's not already local (if omitted, the current user is assumed).
PW=pw	Specifies the library's permanent file password.
PN=pn	Specifies the library's permanent file packname.

Valid record type designators are documented under the description of the "CATALOG" control statement in the NOS Reference Manual.

In addition to these standard types, there's one more "type" processed by EXTRACT, which is designated by "TXT". This "type" is used to denote "TEXT" records that, when extracted, are to have their first line (which contains the record's name) "stripped off". This is useful if, for example, one has records containing directives for a NOS utility, in which case the name of such a record is in all likelihood an illegal directive to the utility program.

EXTRACT will abort under any of the following conditions:

- o format or argument error(s) on the control statement
- o the specified library could not be AQUIREd

F1.0 EXTRACT UTILITY

- o the library file does not have a directory as the last record before end-of-information

Note, however, that EXTRACT won't abort if it does not find any of the requested records (only an informative dayfile message is issued), unless the Abort parameter was coded on the call.

If the library file was not local to the job when EXTRACT was called, it will be RETURNed when EXTRACT terminates normally; but, if the library file was local, EXTRACT will REWIND it prior to normal termination.

G1.0 SESMSG UTILITY

SESMSG is a program which copies the comment field of its call line to a file. The control statement format is:

SESMSG,file.message

file — is the name of the file to receive the message (if omitted, OUTPUT is assumed)

message — is the message to be written to the file

The message will be written to the file only if the file is a terminal file, or if "file" was explicitly quoted on the call line.

SESMSG can be used in procedure files to inform the user about what the procedure is currently doing. It can also be used for creating files of input directives to utility programs when such directives are dependent on execution time considerations.

CORPORATE HEADQUARTERS, P.O. BOX 0, MINNEAPOLIS, MINN. 55440
SALES OFFICES AND SERVICE CENTERS IN MAJOR CITIES THROUGHOUT THE WORLD

LITHO IN U.S.A.

CONTROL DATA CORPORATION