PSS®E 33.4

RELEASE 19.0

IPLAN PROGRAM MANUAL

March, 2013

SIEMENS

Siemens Industry, Inc.
Siemens Power Technologies International
400 State Street, PO Box 1058
Schenectady, NY 12301-1058 USA
+1 518-395-5000
www.siemens.com/power-technologies

SIEMENS

Table of Contents

Chapter 1 - Introduction

1.1 Document Conventions 1-1

Chapter 2 - Language Reference

2.1 Descriptive Terms 2-1

2.2 Single Line Statements 2-3

2.3 Multiline Statements 2-5

2.4 Reserved Words 2-6

Chapter 3 - Language Definitions

3.1 Program Format 3-1

3.2 Variable Names and Labels 3-1
- 3.2.1 Character Set 3-1
- 3.2.2 Statement Format 3-2

3.3 Program Statement 3-2

3.4 Comment Lines 3-2
- 3.4.1 General Comments 3-2
- 3.4.2 In-Line Comments 3-3

3.5 Statement Labels 3-3

3.6 Constants 3-3
- 3.6.1 String Constants 3-3
- 3.6.2 Integer Constants 3-3
- 3.6.3 Real Constants 3-4
- 3.6.4 Logical Constants 3-4

3.7 Declaration Statements 3-4

3.8 Arrays 3-5

3.9 Assignment Statements 3-5

3.10 Expressions 3-5
- 3.10.1 Arithmetic Expressions 3-6
- 3.10.2 Logical Expressions 3-7
- 3.10.3 String Expressions 3-8
- 3.10.4 Relational Expressions 3-10

3.11 Program Termination 3-11

3.12 Program Pauses 3-12
- 3.12.1 PAUSE 3-12

3.12.2 PAUSE UNTIL . . . 3-12
3.12.3 PAUSE READ . . . 3-13
3.12.4 PAUSE WHILE . . . 3-13

3.13 Terminal Input/Output . . . 3-13
3.13.1 Terminal Input . . . 3-14
3.13.2 Terminal Output, Temporary Information: PRINT and PRINTX . . . 3-14
3.13.3 Terminal Output, Permanent Information: REPORT and REPORTX . . . 3-15
3.13.4 Terminal Output, Prompts: ASK and ASKX . . . 3-16

3.14 File Input/Output . . . 3-18
3.14.1 OPEN Statement . . . 3-18
3.14.2 File Input . . . 3-19
3.14.3 File Output . . . 3-20
3.14.4 CLOSE Statement . . . 3-20
3.14.5 REWIND Statement . . . 3-21
3.14.6 DELETE Statement . . . 3-21

3.15 Input/Output Data Conversions . . . 3-21
3.15.1 Unformatted Output . . . 3-21
3.15.2 Formatted Output . . . 3-22
3.15.3 Unformatted Input . . . 3-23
3.15.4 Formatted Input . . . 3-23
3.15.5 TAB Function . . . 3-23
3.15.6 NEWREPORT Statement . . . 3-24

3.16 OPTION Statement . . . 3-24

3.17 Simple Control Flow . . . 3-25

3.18 Procedures . . . 3-25

3.19 Subroutines . . . 3-26

3.20 Condition/Exception Handling . . . 3-27

3.21 Advanced Control Flow . . . 3-28
3.21.1 IF Statements . . . 3-28
3.21.2 LOOP Statements . . . 3-29
3.21.3 WHILE and UNTIL Statements . . . 3-29
3.21.4 CONDITIONAL Statements . . . 3-30
3.21.5 Control Modifiers . . . 3-30

3.22 INCLUDE Statement . . . 3-31

3.23 Miscellaneous Commands . . . 3-31

3.24 Program Debugging . . . 3-31

3.25 Communication With Application Program . . . 3-32
3.25.1 Transmitting to the Application: PUSH and PUSHX . . . 3-32
3.25.2 Transmitting to the Application: QPUSH and QPUSHX . . . 3-32
3.25.3 Receiving from the Application . . . 3-32

3.26 SET Statement . . . 3-33
3.26.1 Transmitting to the Application on Termination . . . 3-33
3.26.2 Controlling New Output Buffer Creation . . . 3-33

3.27 Program Arguments . . . 3-34

3.28 Graphics 3-35
3.29 User-Defined Routines 3-35

Chapter 4 - Developing IPLAN Programs

4.1 Editing 4-1
4.2 Compilation 4-1
4.3 Options 4-1
4.3.1 LIST/NOLIST Options 4-2
4.3.2 STAT/NOSTAT Options 4-2
4.3.3 COMP, RUN, and CRUN Options 4-3
4.3.4 LOOP Option 4-3
4.3.5 INC/NOINC Options 4-3
4.3.6 SEARCH Option 4-3
4.3.7 ARG/NOARG Options 4-4
4.4 File Types 4-4
4.4.1 Carriage Control Characters 4-4
4.4.2 Fortran Carriage Control on UNIX 4-4
4.4.3 Fortran Carriage Control on MS Windows 4-5
4.4.4 Fortran Carriage Control on Other Systems 4-6
4.1 Stand-Alone Simulator

Chapter 5 - IPLAN Internal Functions

5.1 Functional Summary 5-1
5.2 Alphabetic Listing and Reference 5-4

Chapter 6 - Program Limitations

Chapter 7 - User-Defined Routines

7.1 Usage 7-1
7.2 Source Language 7-1
7.3 Program Arguments 7-1
7.4 Terminal I/O in User-Defined Routines 7-2
7.5 File I/O in User-Defined Routines 7-2
7.6 Compilation and Execution of User-Defined Routines 7-2

Chapter 8 - Executing IPLAN Programs

8.1 How IPLAN Interacts With the Host Application 8-1
8.2 Interrupting IPLAN Programs 8-1
8.3 Specifying Arguments 8-2
8.4 Parsing 8-3
8.4.1 Definition 8-3
8.4.2 Application 8-3
8.4.3 Position and Delimiters 8-3

8.4.4 Considerations for Specific Data Types 8-4

Appendix A - PSS®E Application Notes

A.1 Invoking an IPLAN Program from Within PSS®E A-1
A.1.1 Line Mode A-1
A.1.2 GUI A-1
A.1.3 Python A-1
A.2 PSS®E Subroutines: Functional Summary A-2
A.3 IPLAN and Response Files A-11
A.3.1 How IPLAN and Response Files Control PSS®E Execution A-14
A.3.2 Common Questions A-14
A.4 Warning Concerning PSS®E Activity ECHO A-16
A.5 Interaction of Prompts from IPLAN and PSS®E A-17
A.5.1 Common Questions A-17
A.6 Interaction of IPLAN and PSS®E activities PSEB and PSAS A-19

Appendix B - IPLAN Graphics

B.1 Introduction B-1
B.2 Graphic Conventions B-1
B.3 Using the IPLAN Graphics Package B-2
B.4 Restrictions B-2
B.5 Requirements B-2
B.6 Graphic Subroutines B-2
B.7 Sample Graphics Program B-19
B.8 Common Questions B-21

Appendix C - Character Sets

SIEMENS

Chapter 1

Introduction

IPLAN is a programming language designed to be utilized as a stand-alone product or as an enhancement to existing application programs such as PSS®E (Power System Simulator for Engineering). IPLAN offers modern control flow along with a rich set of expression operators.

This manual is meant to help the reader learn how to utilize IPLAN. Example programs, usually complete rather than program fragments, are included as part of each command description to aid in the understanding of structural concepts.

It is assumed that the reader is familiar with at least one programming language and standard programming concepts such as assignment statements, looping, and subroutine calls.

1.1 Document Conventions

All keywords, function names, etc., that appear in this manual are shown exactly as they must appear in a program and will be shown in uppercase (even though lower and mixed case are permitted). Single items that must be substituted by actual items will be shown in lowercase (examples may show variable names in uppercase where they are referred to explicitly in the surrounding text). Sets of such items will generally be delimited by the symbols < >.

This page intentionally left blank.

SIEMENS

Chapter 2

Language Reference

This chapter is a quick reference to all the statements in IPLAN. Each entry shows the statement syntax, followed by a short description of the statement's function. Refer to Chapter 3, Language Definition, for a more complete explanation of statements.

In the following paragraphs, keywords are shown in capital letters. Variables, expressions, and descriptions are shown in lowercase. It is assumed that constants and variable names do not need explanations. Also not defined here are expression operators, relational operators, format specifications, and the TAB function. References to arrays are also omitted. The preceding notwithstanding, the following nonkeywords either appear in the syntax descriptions or are used in the definitions of those nonkeywords.

2.1 Descriptive Terms

constant	Any of the following: integer constant, real constant, logical constant, string constant.
integer-variable	A variable name declared on an INTEGER statement.
real-variable	A variable name declared on a REAL statement.
logical-variable	A variable name declared on a LOGICAL statement.
string-variable	A variable name declared on a STRING statement variable any of the following: integer-variable, real-variable, logical-variable, string-variable.
integer	Integer constant or integer-variable.
real	Real constant or real-variable.
logical	Logical constant or logical-variable.
string	String constant or string-variable.
integer-expression	An integer or a function-reference (that returns an integer) or combinations of them using expression operators.
real-expression	A real or a function-reference (that returns a real) or combinations of them using expression operators.
logical-expression	A logical, or a function-reference (that returns a logical); integer-expressions combined by relational operators; real-expressions combined by relational operators; string-expressions combined by relational operators; or combinations of any of the above using expression operators.

string-expression	A string or a function-reference (that returns a string or combinations of them using expression operators.
expression	Any of the following: integer-expression, real-expression, logical-expression, or string-expression.
label	A variable name, defined by use in the PROGRAM statement, in a PROCEDURE statement, or in a statement label.
function-reference	Value returned from a call to an IPLAN internal function (not TAB).
input-element	A variable which may, optionally, have a format specification, or a TAB function.
output-element	An expression which may, optionally, have a format specification, or a TAB function.
single-line-statement	Defined below.
subroutine-name	Name of any of the subroutines defined in Appendix A or Appendix B.
argument-list	An appropriate list of expressions.

The following notation is also used:

{ }

A list; one element of the list must be used, separated by |.

[ ]

Optional; the bracketed portion of the definition may or may not be used.

-->

A reference to information later in this section.

Neither braces or brackets are used in actual IPLAN programs.

2.2 Single Line Statements

Statement	Section Reference
ARGUMENT integer [, variable [, ...]]	3.27
ASK [output-element [, output-element [, ...]]]	3.13.4
ASKX [output-element [, output-element [, ...]]]	3.13.4
BEEP	3.23
CALL subroutine-name (argument-list)	3.19
CLOSE integer	3.14.4
--> CONDITIONAL, see multiline statements, below	3.21.4
DELETE string	3.14.6
--> ELSE, see multiline statements, below	3.21.1
END	3.11
--> ENDCOND, see multiline statements, below	3.21.4
--> ENDLOOP, see multiline statements, below	3.21.2
--> ENDIF, see multiline statements, below	3.21.1
EXITCOND	3.21.5
EXITLOOP [integer-expression]	3.21.5
--> IF logical-expression THEN, see multiline statements, below	3.21.1
IF logical-expression THEN single-line-statement [ELSE single-line-statement] ENDIF	3.21.1
INCLUDE string-constant	3.22
INPUT input-element [, input-element [, ...]]	3.13.1
INPUTLN string-variable	3.13.1
INTEGER integer-variable [, integer-variable [, ...]]	3.7
GOTO label	3.17
LOGICAL logical-variable [, logical-variable [, ...]]	3.7
--> LOOP, see multiline statements, below	3.21.2
NEWREPORT	3.15.6
NEXTCOND	3.21.5
NEXTLOOP [integer-constant]	3.21.5
ON {EOF | ERR} (integer) {GOTO | PERFORM} label	3.20
OPEN string ON integer FOR '{W | R | RW | A}[F]'	3.14.1
OPTION {ZERO | NOZERO | PRINTX | NOPRINTX}	3.16

Statement	Section Reference
--> (OTHERWISE), see multiline statements, below	3.21.4
PAUSE [{UNTIL {string | string-variable} | READ | WHILE}]	3.12
PRINT [output-element [, output-element [, ...]]]	3.13.2
PRINTX [output-element [, output-element [, ...]]]	3.13.2
PERFORM label	3.18
PROCEDURE label	3.18
PROGRAM label	3.3
PUSH output-element [, output-element [, ...]]	3.25.1
PUSHX output-element [, output-element [, ...]]	3.25.1
QPUSH output-element [, output-element [, ...]]	3.25.2
QPUSHX output-element [, output-element [, ...]]	3.25.2
READ integer;input-element [, input-element [, ...]]	3.14.2
READLN integer;string-variable	3.14.2
REAL real-variable [, real-variable [, ...]]	3.7
REPORT [output-element [, output-element [, ...]]]	3.13.3
REPORTX [output-element [, output-element [, ...]]]	3.13.3
RETURN	3.18
REWIND integer	3.14.5
SET {END|INTERRUPT|STOP|PRINT|WRITE|REPORT|ASK|PUSH} expression	3.26
SLEEP integer	3.23
STOP	3.11
STRING[*integer-constant] string-variable [,string-variable [, ...]]	3.7
TRACE {ON | OFF}	3.24
--> UNTIL, see multiline statements, below	3.21.3
--> WHILE, see multiline statements, below	3.21.3
WRITE integer;output-element [, output-element [, ...]]	3.14.3
WRITEX integer;output-element [, output-element [, ...]]	3.14.3

2.3 Multiline Statements

```
<statement list> means one or more single line statements or
multiline statements.

IF logical-expression THEN
   <statement list>
   ENDIF

IF logical-expression THEN
   <statement list>
   ELSE
   <statement list>
   ENDIF

LOOP integer-variable=integer-expression {TO|,|DOWNTO} integer-
expression [{BY|STEP} integer-expression]
   <statement list>
   ENDLOOP

LOOP {WHILE|UNTIL} (logical-expression)
   <statement list>
   ENDLOOP

LOOP
   <statement list>
   {WHILE|UNTIL} (logical-expression)

CONDITIONAL
   (logical-expression)
   <statement list>
  [(logical-expression)
   <statement list>]
  [ ... ]
   (OTHERWISE)
   <statement list>
   ENDCOND
```

2.4 Reserved Words

Reserved words cannot be used as variables. A list of these words is given below.

AND	ENDLOOP	LOOP	PRINT	SET
ARGUMENT	EOF	NEWREPORT	PRINTX	SLEEP
ASK	ERR	NEXTCOND	PROCEDURE	STEP
ASKX	EXITCOND	NEXTLOOP	PROGRAM	STOP
BEEP	EXITLOOP	NOPRINTX	PUSH	STRING
BY	FALSE	NOT	PUSHX	THEN
CALL	FOR	NOZERO	QPUSH	TO
CLOSE	GOTO	OFF	QPUSHX	TRACE
CONDITIONAL	IF	ON	READ	TRUE
DELETE	INCLUDE	OPEN	READLN	UNTIL
DOWNTO	INPUT	OPTION	REAL	WHILE
ELSE	INPUTLN	OR	REPORT	WRITE
END	INTEGER	OTHERWISE	REPORTX	WRITEX
ENDCOND	INTERRUPT	PAUSE	RETURN	ZERO
ENDIF	LOGICAL	PERFORM	REWIND	

SIEMENS

Chapter 3

Language Definitions

3.1 Program Format

An IPLAN program is made up of a main program followed by any number of procedures. The main program takes the following form:

```
PROGRAM program-name
global variable declarations
program-body
END
```

The form of the procedures is similar:

```
PROCEDURE procedure-name
local variable declarations
procedure-body
END
```

3.2 Variable Names and Labels

Program names, procedure names, statement labels, and variable names may be up to eighty alphanumeric characters. A legal name is an uppercase alphabetic letter followed by zero or more uppercase letters, numeric digits, and/or underscores. The following are valid names:

```
A
TEMP1
A2
NEW_VAR
```

Legal names may not duplicate IPLAN keywords (note that all keywords in IPLAN are reserved words; see Section 2.4 Reserved Words), IPLAN function names (listed in Chapter 5), or IPLAN CALL routines (application calls are listed in Appendix A, graphics calls in Appendix B, and user-defined routines in Section 3.29 User-Defined Routines).

3.2.1 Character Set

IPLAN does not differentiate between upper and lowercase except within a quoted string. For example, the following variable names would all have the same meaning: ABC, abc, aBc, Abc. This is true for reserved keywords as well.

IPLAN will recognize any displayable character, but not all are meaningful. Error messages will indicate unexpected characters or sequences of characters and cause compilation to fail. Any ASCII value may be used in a quoted string or a comment.

3.2.2 Statement Format

A statement may begin in any column including column 1. With a few exceptions, which will be noted where applicable, only one statement may appear on a line.

The length of a line of input into IPLAN is limited (see *Program Limitations*). Statements may be continued by terminating a line with an ampersand (&). The & can appear anywhere another valid language element could appear.

Unlike some languages such as Fortran, spaces are significant when writing IPLAN programs. That is, keywords, variable names, and constants may not contain embedded spaces (with the exception of quoted string constants) and they must be separated by a space or the appropriate delimiter. As an example:

```
WRITE5;X or WRI TE 5;X
```

are not valid IPLAN statements, while the following statement is valid:

```
WRITE 5;X
```

Consecutive spaces are treated as a single space except when part of a quoted string.

3.3 Program Statement

Every program must begin with a PROGRAM statement. The format of the statement is:

```
PROGRAM program-name
```

3.4 Comment Lines

3.4.1 General Comments

IPLAN recognizes the sequence "/*...*/" as a comment stream where the delimiter "/*" initiates the comment and "*/" terminates the comment. The comments delimited in this way may be contained on a single line or may span several lines. In addition, it is possible to have the comments "nested" (up to a depth of 8) by placing the "/*...*/" delimiters around another set of delimiters. It is very important to keep the delimiter pairs balanced or compilation errors will occur. The asterisks within the text, as shown in the following example, are not required, but do enhance readability. The following is a sample of a multiline comment:

```
/*
 * This is a comment
 * which stretches across
 * several lines.
*/
```

3.4.2 In-Line Comments

In addition to the general comments described in the preceding section, IPLAN recognizes in-line comments which are initiated by an exclamation mark (!). In-line comments may follow any valid statement and are terminated by the end of the line.

3.5 Statement Labels

A statement label may appear on a line by itself or it may be followed by a valid IPLAN statement. Labels must be followed by a colon.

For example:

```
LABEL:
   A = A+1
```

and:

```
LABEL:  A = A+1
```

are both valid and produce the same results.

3.6 Constants

IPLAN recognizes four types of constants:

String (e.g., ‘HELLO’).

Integer (e.g., 346).

Real (e.g., 17.38).

Logical (e.g., TRUE).

3.6.1 String Constants

String constants consist of characters contained within apostrophes. An apostrophe can be embedded within the string by coding two consecutive apostrophes:

```
STR1 = 'HOW''S THIS'       STR1 is equal to "HOW'S THIS"
STR2 = ''''                STR2 is equal to a single apostrophe
STR3 = 'THIS WON'T WORK'   this line will produce a syntax error
```

String constants may contain lowercase characters. The minimum length of a string constant is one (a null string will be accepted by the computer, but it will be treated as a length one blank string). The maximum length is documented in *Program Limitations*.

3.6.2 Integer Constants

Integer constants consist of an optional sign (-) followed by one or more digits (0-9). The following are valid integer constants:

```
0, 76, -37
```

Integer constants may range between $-(2^{31})$ and $(2^{31})-1$.

3.6.3 Real Constants

Real constants consist of an optional sign (-) followed by zero or more digits and a decimal. Optionally, zero or more digits may follow the decimal. Hence, the following are valid real constants:

```
0., 7.6, -30.0, .002
```

3.6.4 Logical Constants

Logical constants take on one of two values: true or false. True is designated by the keyword TRUE and false is designated by the keyword FALSE.

3.7 Declaration Statements

All variables must be declared before they can be used in an IPLAN program. Declaration statements must precede any executable statements. The format of a declaration statement is:

```
type var1, var2, var3,...
```

where "type" is one of the four IPLAN types "STRING," "INTEGER," "REAL," or "LOGICAL" and "var1," "var2," etc. are scalar or variable names. For example:

```
INTEGER IVAR
LOGICAL FLAG
REAL AMOUNT
```

declares a variable named IVAR which is type integer, a variable FLAG which is type logical and a variable AMOUNT which is type REAL. Variables which are declared in the main program are global. That is, they are accessible both in the main program and in any procedures which are called. Variables which are declared in a procedure are local and cannot be referenced outside that procedure. If a variable with the same name is declared both in the main program and in a procedure, the procedure definition is used when the procedure code is executed.

Strings may be declared to be of a specified length. The syntax is:

```
STRING*i varname
```

where "i" is an integer constant between 1 and the maximum string length (see *Program Limitations*). The above statement declares "varname" to be a string of length "i". "Varname" may be a scalar or an array. For example:

```
STRING*2      TEMP
STRING*60     TEMP1(10)
```

declares TEMP to be a string of length 2 and declares TEMP1 to be an array of 10 elements each of which is a string of length 60.

If length is not specified, the string defaults to a length of 132. For example:

```
STRING        TEMP2
```

declares TEMP2 to be a string of 132 characters in length.

3.8 Arrays

Variables may be either single or double dimensioned arrays. Arrays are stored in column major order, as with Fortran, i.e., the first subscript moves faster than the second subscript. For example:

```
LOGICAL B(10)
REAL X(3,11)
```

declares B to be a singly subscripted array of 10 elements and X to be a doubly subscripted array of 3 rows and 11 columns. The elements of X would be stored in the following order: X(1,1),X(2,1),X(3,1),X(1,2),X(2,2),...

3.9 Assignment Statements

One of the most basic commands in any language is the assignment statement. For IPLAN, the syntax of the assignment statement is:

```
var = expression
```

where "var" is any declared program variable scalar or array element and "expression" is one or more variables, functions, or constants along with legal operators. Section 3.10 Expressions describes expressions in more detail.

Examples of simple assignments follow:

```
   PROGRAM SHOWASSIGN

/*
 * Show the assignment statement
 */

   INTEGER  ITEMP
   REAL     RTEMP
   LOGICAL  LTEMP
   STRING   STEMP

   ITEMP = 10 ! Sets ITEMP to 10
   ITEMP = -100 ! Sets ITEMP to -100
   RTEMP = 67.0 ! Sets RTEMP to 67.0
   RTEMP = -6.987 ! Sets RTEMP to -6.987
   LTEMP = TRUE ! Sets LTEMP to true
   LTEMP = FALSE ! Sets LTEMP to false
   STEMP = 'THIS IS A TEST' ! Sets STEMP to the string 'THIS IS A  TEST'

   END
```

3.10 Expressions

Expressions may be numeric, logical, or string expressions. It is not permissible to mix variable types in expressions. Table 3-1 IPLAN Expression Operators lists the expression operators and their meaning.

Table 3-1. IPLAN Expression Operators

Expression Operator[1]	Type	Meaning of Operator with Numeric Operands[2]	Meaning of Operator with Logical Operands[3]	Meaning of Operator with String Operands[4]	Precedence[5]
+	binary	Addition	inclusive OR[6]	Append	5
-	binary	Difference	if operand 1 true and operand 2 false[6]	Remove from left	5
*	binary	Multiplication	logical AND[6]	Remove from right	4
**	binary	Exponentiation	logical NAND[6] (NOT AND)	Repeat	1
/	binary	Division	if operand 1 true and operand 2 false[6]	Best match	4
%	binary	Modulo Addition	exclusive OR	Overlay	3
-	unary	Negation	logical negation[6] (NOT)	Upcase	2
AND	binary	Addition[6]	logical AND	Append[6]	8
OR	binary	Multiplication[6]	inclusive OR	Append[6]	9
NOT	unary	Negation[6]	logical negation (NOT)	Lowcase	7

1 Operators require two operands, except those indicated as unary.
2 See Section 3.10.1 Arithmetic Expressions for examples of arithmetic expressions.
3 See Section 3.10.2 Logical Expressions for examples of logical expressions.
4 See Section 3.10.3 String Expressions for examples of string expressions.
5 Precedence defines the order in which expressions are evaluated. The lower the precedence number, the earlier the evaluation occurs within the expression. Operations of equal precedence are executed left to right. Parentheses may be used within an expression to change the order of evaluation.
6 Use of these operators is not recommended.

IPLAN supports a number of functions which can be used within expressions. The functions are described in Chapter 5.

3.10.1 Arithmetic Expressions

Arithmetic expressions are integer or real variables, constants, or functions, or combinations using the expression operators in Table 3-1 IPLAN Expression Operators. It is not permissible to mix integer and real types within an expression except by use of the functions TOREAL or TOINT. For example:

```
I = 5                          value of I is 5
R = 3.0 * TOREAL(I)/3.3        value of R is 4.545
```

3.10.2 Logical Expressions

Logical expressions are logical variables, logical constants, functions returning logical values, relational expressions, or combinations using the expression operators in Table 3-1 IPLAN Expression Operators. Another way to look at binary logical operators is that there are only sixteen sets of results that are theorectically possible for any such operator. They are listed in Table 3-2 . Not all have meaningful names.

Table 3-2.

A	0	0	1	1	Non-trivial Binary Operations and Their Inverses	Trivial Cases	IPLAN Operators
B	0	1	0	1			
A op B (truth tables for all possible binary operations)	0	0	0	0		[equivalent to IF (FALSE)]	
	1	0	0	0	inverse inclusive or		
	0	1	0	0	(inverse reverse implication)[1]		/
	0	0	1	0	inverse implication[2]		-
	0	0	0	1	logical and		AND, *
	1	1	0	0		[equivalent to IF (NOT A)]	
	1	0	1	0		[equivalent to IF (NOT B)]	
	1	0	0	1	inverse exclusive or[3]		
	0	1	1	0	exclusive or		%
	0	1	0	1		[equivalent to IF (B)]	
	0	0	1	1		[equivalent to IF (A)]	
	1	1	1	0	inverse logical and		**
	1	1	0	1	logical implication[4]		
	1	0	1	1	(reverse implication)		
	0	1	1	1	inclusive or		OR, +
	1	1	1	1		[equivalent to IF (TRUE)]	

1 also called "B only"
2 also called "A only"
3 also called bi-conditional; also called equivalent
4 also called conditional

For example, if I is an integer whose value is -1, and L is a logical whose value is TRUE, and all the other variables used below are declared LOGICAL:

A = I > 0	Value of A is FALSE.
B = L OR A	Value of B is TRUE.
C = K OR NOT (I > 0)	Value of C is TRUE.
D = L AND C	Value of D is TRUE.
E = L AND A	Value of E is FALSE.
F = B% C	Value of F is FALSE.
G = B ** C	Value of G is FALSE.

H = D - E Value of H is TRUE.

3.10.3 String Expressions

String expressions are used to manipulate one or more strings to obtain the desired results. For example:

```
STR1 = 'HELLO'                  STR1 is equal to 'HELLO'
STR2 = STR1 + ' THERE'          STR2 is equal to the string 'HELLO THERE'
STR3 = 'ABCABC' - 'BC'          STR3 is equal to the string 'AABC'
STR4 = 'HELLO' / 'HELP'         STR4 is equal to the string 'HEL'
STR5 = 'ABCABC' * 'BC'          STR5 is equal to the string 'ABCA'
STR6 = 'AB' ** 3                STR6 is equal to the string 'ABABAB'
STR7 = 'HOWZIT'%'WHO'           STR7 is equal to the string 'WHOZIT'
```

Handling of Trailing Blanks

Trailing blanks are not stored, even for intermediate operations. For example:

```
STR1 = 'WATCH' + ' ' + 'THIS'    STR1 is equal to 'WATCHTHIS'
STR2 = 'WATCH' + (' ' + 'THIS')  STR2 is equal to 'WATCH THIS'
```

The above example demonstrates one solution to a situation where the elimination of trailing blanks can cause difficulty; namely, where the trailing blanks are really embedded blanks at an intermediate form of construction. Where this occurs with output commands, it can usually be handled by using formats or the TAB function. Where this occurs in a string expression, it can be handled as above by reordering the sequence of operations (via parentheses) to cause the blanks to appear preceding a nonblank string. Preceding and embedded blanks are stored.

Trailing blanks will be recognized in string constants, but not in intermediate values computed with them. An all-blank variable will be treated as a single blank. The following are some examples demonstrating how results might vary depending on whether a variable or a constant is used. In each example there are a pair of identical expressions producing different results. In each case, the first expression has been evaluated exactly as shown, and the second expression has been evaluated by using STRING*10 variables containing the values in the expression. The result (stored in a STRING*10 variable) is shown in quotes after the symbol ->.

```
'xx  '+'yy'->"xx  yy"
'xx  '+'yy'->"xxyy"

'  '+'xyz'->"  xyz"
'  '+'xyz'->" xyz"

'a '+'a '+'a '->"a aa"
'a '+'a '+'a '->"aaa"

' ab ab'-'b '->" aab"
' ab ab'-'b '->" a ab"

' abcab '-'b '->" abca"
' abcab '-'b '->" acab"

repeat('a ')->"a a a a a"
repeat('a ')->"aaaaaaaaaa"

right(' thisone  ',3)->"e"
right(' thisone  ',3)->"one"

index(' abca  ','a ')-> 5
index(' abca  ','a ')-> 2

index(' abca  ','  ')-> 6
index(' abca  ','  ')-> 1

len('  ')-> 2
len('  ')-> 1

len(' a b c ')-> 7
len(' a b c ')-> 6

' abcab '*' '->" abcab"
' abcab '*' '->"abcab"

' ab ab'*'b '->" aab"
' ab ab'*'b '->" ab a"

'a '**3->"a a a"
'a '**3->"aaa"

scan(' abca  ','a ',F)-> 1
scan(' abca  ','a ',F)-> 2

scan(' abca  ','a ',T)-> 7
scan(' abca  ','a ',T)-> 5

verify(' abca  ','a',T)-> 7
verify(' abca  ','a',T)-> 4

verify(' abca  ','a ',F)-> 3
```

```
verify(' abca  ','a ',F)-> 1

substr(' abca  ','a ','new ')->" abcnew"
substr(' abca  ','a ','new ')->" newbca"

substr(' abca  ','b',' ')->" a  ca"
substr(' abca  ','b',' ')->" a ca"

bufstr('a ') X3 ->"a a a"
bufstr('a ') X3 ->"aaa"
(i.e. execute v=bufstr('a ') 3 times)

bufstr('  ') X3, bufstr('xyz')->"      xyz"
bufstr('  ') X3, bufstr('xyz')->"   xyz"
(i.e. execute v=bufstr('a ') 3 times, then v=bufstr('xyz') )
```

3.10.4 Relational Expressions

Relational expressions may be either simple or compound. Simple expressions may be logical variables, constants or functions, or the result of comparing other like types using the relational operators in Table 3-3 IPLAN Relational Operators. For example:

```
(A > 5)
(STRING == 'Hi there')
```

Compound expressions are one or more simple expressions combined with the expression operators AND, OR, or NOT. Compound expressions have the form:

```
((A > 5) OR (STRING == 'Hi there'))
(NOT (A > B))
(NOT ((A == 67) OR (B < 89)))
```

Table 3-3 IPLAN Relational Operators lists the relational operators along with their meaning.

Table 3-3. IPLAN Relational Operators

Operator	Function	Precedence[1]
==	Equals	6
>	Greater than	6
>=	Greater than or equal	6
<	Less than	6
<=	Less than or equal	6
<>	Not equal	6

1 Precedence relates to Table 3-1 IPLAN Expression Operators

All possible sets of results of a pair of logical values can be generated using the relational and expression operators. This is demonstrated in the following program:

```
PROGRAM LOGTST
LOGICAL A,B,C(16)
A=FALSE
B=FALSE
PERFORM SHOW_C
B=TRUE
PERFORM SHOW_C
A=TRUE
B=FALSE
PERFORM SHOW_C
B=TRUE
PERFORM SHOW_C
STOP
END                          ! equivalent expressions
PROCEDURE SHOW_C             ! ----------------------
C(1) = FALSE
C(2) = A AND B               ! NOT (A ** B)
C(3) = A  -  B               ! B / A
C(4) = A  /  B               ! B - A
C(5) = NOT( A OR  B)
C(6) = A
C(7) = B
C(8) = NOT( A  %  B )
C(9) = A  %  B
C(10)= NOT B
C(11)= NOT A
C(12)= A OR  B
C(13)= NOT( A  /  B )        ! NOT (B - A)
C(14)= NOT( A  -  B )        ! NOT (B / A)
C(15)= A **  B               ! NOT (A AND B)
C(16)= TRUE
PRINTX A:1,B:1,'  ',C(1):1,' ',C(2):1,' ',C(3):1,' ',C(4):1,' ',C(5):1,' '
PRINTX C(6):1,' ',C(7):1,' ',C(8):1,' ',C(9):1,' ',C(10):1,' ',C(11):1,' '
PRINT  C(12):1,' ',C(13):1,' ',C(14):1,' ',C(15):1,' ',C(16):1,' '
RETURN
END

IPLAN Language Simulator 9.0

 FF  F F F F T F F T F T T F T T T T
 FT  F F F T F F T F T F T T F T T T
 TF  F F T F F T F F T T F T T F T T
 TT  F T F F F T T T F F F T T T F T
```

3.11 Program Termination

There are two program statements which cause an IPLAN program to terminate when executed. The first statement is the "END" statement, which must be the last statement of a program.

```
   PROGRAM SHOWEND
/*
 * Demonstrate use of the END statement
```

```
* to terminate a program.
*/
  INTEGER I
  I = 73
  PRINT I
  END
```

The above program causes the value "73" to be printed at the user's terminal and the program to terminate.

The second program statement which causes program termination is the "STOP" statement.

```
   PROGRAM SHOWSTOP
/*
 * Demonstrate use of the STOP statement
 * to terminate a program
 */
   INTEGER I
   I=73
   IF (I == 73) THEN STOP ENDIF
   PRINT I
   END
```

Because the value of I is 73 in the above program, the "IF" condition will be true and the STOP statement will be executed. The value of I will not be printed.

3.12 Program Pauses

The PAUSE statement causes program execution to be suspended. Input is taken from the normal application source (e.g., user input from the terminal). Execution of the IPLAN program will resume at the statement following the PAUSE statement. Use of the PAUSE statement causes any buffer built up by PRINTX commands to be displayed, and any buffer built up by PUSHX (or QPUSHX) to be destroyed.

3.12.1 PAUSE

The simplest form of the PAUSE command is:

```
PAUSE
```

The IPLAN program is suspended for a single input request, and resumes at the next request for input by the application.

3.12.2 PAUSE UNTIL

The format of the PAUSE UNTIL command is as follows:

```
PAUSE UNTIL string
```

where string can be any string (i.e., a literal or a variable). The IPLAN program is suspended until an input request is satisfied by a value that matches "string". That input value is ignored, and the IPLAN program is resumed. The length of the string that can be used is specified by the pause until string limit in *Program Limitations*. Characters beyond that limit are ignored.

3.12.3 PAUSE READ

The format of the PAUSE READ command is as follows:

```
PAUSE READ
```

The IPLAN program is suspended for as long as input is directed to a file. Many applications provide for the ability to redirect normal terminal input to a file. When the application is in this state, the PAUSE READ command will suspend the IPLAN program until input returns to the terminal. If input is currently directed to the terminal, the PAUSE READ command has no effect.

3.12.4 PAUSE WHILE

The format of the PAUSE WHILE command is as follows:

```
PAUSE WHILE
```

The IPLAN program is suspended for as long as input is directed to a file. The difference between PAUSE WHILE and PAUSE READ has to do with nested file redirection. In situations where input can be redirected to a file, and then redirected to another file with input returning to the first when all data in the second is exhausted, PAUSE READ will suspend execution until all such files are closed. PAUSE WHILE will return control to the IPLAN program when the redirected input file that was being used at the time the PAUSE WHILE is executed is closed, even if input is still redirected to a file at a "higher" level.

For example, say an application program uses input file redirection for automation, and that file, say, file1, contains a command to invoke an IPLAN program. That IPLAN program may redirect input itself (using, say, PUSH '@input file2') and then PAUSE WHILE. The IPLAN program will pause while the commands in file2 are read, but then return control to the IPLAN program without reading any commands from file1. Using PAUSE READ, the remaining commands in file1 would have been read before the IPLAN program resumed.

3.13 Terminal Input/Output

Terminal input means information entered into the IPLAN program in dialog with the user, usually from a computer keyboard or terminal. Terminal output means information normally displayed directly on a computer display.

Terminal input is handled via the INPUT and INPUTLN statements (see Section 3.13.1 Terminal Input), and can be modified by the OPTION statement keywords ZERO and NOZERO (see Section 3.16 OPTION Statement).

Terminal output can be roughly divided into three categories, as follows: 1) temporary information (e.g., intermediate values of a calculation or a warning such as a default value being used for missing data); 2) permanent information (i.e., reports or information designed to be kept and used later); and 3) information that requests a response from the user (i.e., prompts or questions). These are handled by the statements PRINT, REPORT, and ASK, respectively; all three statements also have a variant that buffers the output, called PRINTX, REPORTX, and ASKX.

The distinction between the three types of terminal output is not really appreciable when running an application in line mode (except for the timing of prompts; see Section 3.13.4 Terminal Output, Prompts: ASK and ASKX for an example). When running with a GUI, however, each of the three types will usually appear in a different window.

Program arguments are another form of program input. They can be considered terminal input in the sense that they form part of the dialog between the user and the program (see Section 3.27 Program Arguments).

3.13.1 Terminal Input

Input from the user's terminal is accomplished with the INPUT and INPUTLN statements. The format of the INPUT statement is:

```
INPUT iolist
```

where "iolist" is a list of one or more variables of any type into which values are to be placed. The input values are free form. That is, it is not necessary for values to be in specific columns. Each input value is separated by a comma or one or more spaces. Leading spaces are ignored when reading. Any variable which does not have a corresponding input value is left unchanged.

The format of the INPUTLN statement is:

```
INPUTLN buffer
```

where "buffer" is a string variable into which an entire input line is read as entered.

INPUT and INPUTLN do not convert input characters to uppercase. Note that if no prompt is issued, input will be requested with a default prompt of '?'.

3.13.2 Terminal Output, Temporary Information: PRINT and PRINTX

The PRINT (or PRINTX) statement should be used for this purpose. They have the following format:

```
PRINT iolist
```

or:

```
PRINTX iolist
```

where "iolist" is a list of zero or more expressions (expressions are defined in Section 3.10 Expressions). Each expression is separated by a comma, and can be of any type.

For example:

```
PROGRAM PRINTEXAMPLE
STRING*2 ORDINATION
INTEGER COUNTER
REAL VALUE
COUNTER = 0
START:
COUNTER = COUNTER+1
ASK 'Enter value:'
INPUT VALUE
ORDINATION = 'th'
IF (VALUE==1) THEN ORDINATION = 'st' ENDIF
IF (VALUE==2) THEN ORDINATION = 'nd' ENDIF
IF (VALUE==3) THEN ORDINATION = 'rd' ENDIF
PRINT 'The ',COUNTER,ORDINATION,' number is ',VALUE
/* this works OK up to 20, ... */
```

```
IF (VALUE <> 0.0) THEN GOTO START ENDIF
END
```

(The ASK command is documented in Section 3.13.4 Terminal Output, Prompts: ASK and ASKX; the IF command is documented in Section 3.21.1 IF Statements).

The PRINTX command has exactly the same form as the PRINT command, but operates with one difference. Instead of being immediately displayed, the output is held in a buffer. Subsequent PRINTX commands append to that buffer. The buffer is displayed when one of the following events occurs:

1. A PRINT command is executed (the contents, if any, of the PRINT command are first appended to the contents of the PRINTX buffer, if any).
2. A STOP or END command is executed.
3. An INPUT or INPUTLN command is executed.
4. A PRINT/PRINTX command would cause the length of the buffer to exceed the buffer wrapping limit (see Section 3.26.2 Controlling New Output Buffer Creation), in which case that command starts a new buffer beginning with the element of the iolist that would have overflowed the limit.
5. A PAUSE command is executed (any form of the PAUSE command).

In addition, if OPTION PRINTX is in effect, then an INPUT or INPUTLN command causes the buffer to be displayed as a prompt (see Section 3.16 OPTION Statement).

3.13.3 Terminal Output, Permanent Information: REPORT and REPORTX

The REPORT (or REPORTX) command should be for this purpose. They have the same form as the PRINT and PRINTX commands, and will operate exactly the same as the PRINT and PRINTX commands, respectively, except for the following:

1. Under windows, output from REPORT/REPORTX will be displayed separately from PRINT/PRINTX.
2. REPORTX (and REPORT) will use its own buffer (separate from the PRINTX buffer).
3. The REPORTX buffer is displayed in the following cases:
   a. A REPORT command is executed (the contents, if any, of the REPORT command are first appended to the contents of the REPORTX buffer, if any).
   b. A STOP, or END command is executed.
   c. A PUSH command is executed (any form of the PUSH command).
   d. A REPORT/REPORTX command would cause the length of the buffer to exceed the buffer wrapping limit (see Section 3.26.2 Controlling New Output Buffer Creation), in which case that command starts a new buffer beginning with the element of the iolist that would have overflowed the limit.

It is possible, depending on the application and the operating system, that the window containing output issued through REPORT/REPORTX may not be updated with the same frequency as the one containing PRINT/PRINTX output.

3.13.4 Terminal Output, Prompts: ASK and ASKX

The ASK (or ASKX) command should be used for this purpose. While the format and use of ASK/ASKX is the same as PRINT/PRINTX or REPORT/REPORTX, their operation is different. Prompts are always tied to the next input request. They can be added to, modified, or deleted anytime before that. The output is always buffered in the sense that it is not displayed until the next input request; there are many prompt buffers (as opposed to 1 PRINT buffer and 1 REPORT buffer).

The difference between ASK and ASKX is that ASK completes use of that buffer, whereas after an ASKX command, a subsequent ASK/ASKX command will append to the current buffer. If an ASK/ASKX command would cause the length of the buffer to exceed the buffer wrapping limit (see Section 3.26.2 Controlling New Output Buffer Creation), then that command starts a new buffer beginning with the element of the iolist that would have overflowed the limit. Please see *Program Limitations* for the maximum output buffer length. Prompt buffers that exceed the prompt length limit will be truncated on output. The PROMPT and PROMPTR functions are affected by these limitations as well.

The PUSH command will cause the prompt buffers to be displayed. Otherwise, only an input request causes the prompt buffers to be displayed, and not even always then. The exception occurs because the application can also access the same buffers and could modify or delete them before they are displayed (QPUSH will direct the application to clear the prompts without displaying them). Prompts that are not intended to be used by an IPLAN INPUT or INPUTLN statement require a detailed knowledge of the application that IPLAN is running with. (See Section 3.25 Communication With Application Program for more information on PUSH and QPUSH.)

Let us first consider the case where there is no interaction with the application program, using the following program:

```
PROGRAM LOOKEHERE
INTEGER I
STRING BUFFER
PRINT 'LINE 1'
ASK 'LINE 2'
PRINT 'LINE 3'
INPUTLN BUFFER
END
```

When not running with a GUI, this output is produced:

```
LINE 1
LINE 3
LINE 2 input
```

(where ‘input’ was entered at the keyboard and read into BUFFER). If the line ASK ‘LINE 2’ was removed from the program, the following output would be produced:

```
LINE 1
LINE 3
? input
```

When running with a GUI, and the original program, the following would be written in the "progress area" of the main program window:

```
LINE 1
LINE 3
```

and then a separate input window would be created containing the prompt:

```
LINE 2
```

If the line ASK 'LINE 2' was removed from the program, the input window would contain:

```
INPUT?
```

The application program with which IPLAN is interacting (via PUSH and QPUSH) can also issue prompts; in fact, it would be unusual for it not to. When the IPLAN program resumes control, those prompts will have been stored, but not yet displayed. Functions are provided to access these prompts (N.B., an application program running with a GUI will not display dialogs, i.e., windows, while interacting with IPLAN). When control is returned to the application (via PUSH or QPUSH), any pending prompts are cleared (see PROMPTC below).

The following functions are provided for dealing with prompts:

```
ICODE=PROMPT(BUFFER,CNTL   ◄— Adds a prompt to the set of pending prompt lines
NBUF=PROMPTI               ◄— Returns the number of pending prompt lines
BUFFER=PROMPTR(IBUF)       ◄— Returns pending prompt line # IBUF
ICODE=PROMPTC              ◄— Clears all prompts
```

where:

INTEGER	ICODE	Is a return code.
INTEGER	NBUF	Is the number of pending prompts.
INTEGER	IBUF	Is the number of the prompt buffer (e.g., 2 for the 2nd line pending).
STRING	BUFFER	Is a prompt line.
INTEGER	CNTL	Is a control code (see definition of PROMPT in Chapter 5).

Please note that the PROMPT function is an obsolete feature. The ASK command is preferred.

ECHO Function

When running an application with a GUI, it is sometimes useful to have a copy of the prompts appear in a progress area, for purposes of keeping a record of the program's execution. The ECHO function is provided to facilitate this. When included in an ASK or ASKX command, the effect will be as though a PRINT was issued with the same string as the ASK command. The function takes no arguments, and is entered simply as a keyword in the output list. For example:

```
ASK astring, avar, ECHO, anothervar
```

If astring = 'A', avar = 1, another var = 2, then "A 1 2" would be used as a prompt, and also printed to the progress area.

Additionally:

- The ECHO function can appear anywhere in the output list.
- Multiple ECHO functions in an output list are equivalent to a single ECHO function.
- When not running with a GUI, the ECHO function (in an ASK/ASKX command) has no effect.

- If used with any other output command (e.g., PRINT), the ECHO function will print the string "ECHO".
- Only the command containing the ECHO function is affected. Subsequent ASK commands will not "echo" unless they also contain an ECHO function.
- When used with ASKX, only the current buffer will be affected. Current refers to that point in the output list where the ECHO function appears. Subsequently printed buffers will not be affected (e.g., say ASKX ECHO, string1, string2, string3 causes two buffers to be printed, the first will be echoed, the second will not).

3.14 File Input/Output

The following sections describe the statements which are used to manipulate disk files. A sample program is expanded as each statement is described to demonstrate the use of that statement.

3.14.1 OPEN Statement

Before input or output can be performed on a file, the file must be "connected" or "opened" with the appropriate access rights. Files can be connected with the OPEN statement which has the format:

```
OPEN filename ON unit FOR mode
```

where:

filename	Is a string constant, variable, or expression and specifies the name of the file to be opened.
unit	Is an integer constant, variable or expression, "unit" must be in the range of 1 to the maximum number of files (see *Program Limitations*).
mode	Is a string constant, variable or expression which specifies the access mode for the file, where:

R	=	Reading.
W	=	Writing.
RW	=	Reading and writing.
A	=	Append to the end of the file (writing only).
RF, WF, RWF, AF	=	Use Fortran carriage control for read, write, read/write, or append, respectively.

If there is more than one character in the mode string, the order is not significant.

The following example opens two files. It is expanded in the following sections to perform useful work.

```
PROGRAM IOSAMP
/*
 * File input/output example
 */
   INTEGER A,B,C
/* Open files */
   OPEN 'FILE1' ON 1 FOR 'R'
   OPEN 'FILE2' ON 2 FOR 'W'
/* End Program */
   END
```

Lowercase values for mode are allowed.

3.14.2 File Input

Reading from files is accomplished by the READ and READLN statements. The format and use of these statements is identical to the terminal input statements INPUT and INPUTLN except for the specification of the file unit number. The file unit number may be 0 to specify terminal input (equivalent to an INPUT/INPUTLN) or be the value of a previously opened file. The format of the READ statement is:

```
READ unit;iolist
```

where "unit" is an integer expression which identifies the file unit to be used for reading and "iolist" is a list of variables to be given values by the read statement.

The format of the READLN statement is:

```
READLN unit;buffer
```

where "unit" is the same as is described in the READ statement and "buffer" is a string variable into which an entire line of input is read.

The following example adds a READ command to the example from Section 3.14.1 OPEN Statement.

```
   PROGRAM IOSAMP
/*
 * File input/output example
 */
   INTEGER A,B,C
/* Open files */
   OPEN 'FILE1' ON 1 FOR 'R'
   OPEN 'FILE2' ON 2 FOR 'W'
/* Read from unit 1 */
   READ 1;A,B
/* End Program */
   END
```

3.14.3 File Output

The WRITE and WRITEX statements resemble the terminal output statements PRINT and PRINTX except for specification of the file unit for output. The file unit number may be 0 to specify terminal output (equivalent to PRINT/PRINTX) or be the value of a previously opened file. The format of the WRITE statement is:

```
WRITE unit;iolist
```

where "unit" is an integer expression specifying the file unit to be written to, and "iolist" is a list of variables whose values are to be written. The format of the WRITEX statement is identical to the WRITE statement.

The following example adds a WRITE command to the example from Section 3.14.2 File Input.

```
   PROGRAM IOSAMP
/*
 * File input/output example
 */
   INTEGER A,B,C
/* Open files */
   OPEN 'FILE1' ON 1 FOR 'R'
   OPEN 'FILE2' ON 2 FOR 'W'
/* Read from unit 1 */
   READ 1;A,B
/* Write sum of A and B to unit 2 */
   C=A+B
   WRITE 2; 'A+B=',C
/* End Program */
   END
```

See Section 3.15.2 Formatted Output for information on formatted output.

Write to Application File

Refer to Appendix A. Any application subroutine that returns a unit number for a file opened by the application can be written to in IPLAN using a WRITE or WRITEX statement. The value of that unit number will always be outside the range available for normal programming (see maximum number of files in *Program Limitations*). Calling the subroutine to acquire the unit number causes IPLAN to mark that unit as available for writing. Any variation of the PUSH statement (see Section 3.25 Communication With Application Program) or the PAUSE statement (see Section 3.12 Program Pauses) will cause that unit to be marked as unavailable.

3.14.4 CLOSE Statement

The CLOSE statement is used to close or disconnect a file which was previously opened with the OPEN statement. The format of the CLOSE statement is:

```
CLOSE unit
```

where "unit" is an integer expression which identifies the file to be closed. If a file was opened for writing only or if it was opened for reading and writing and the last operation was a write, the file will automatically be truncated, if necessary, when it is closed. The following example adds CLOSE statements to the example from Section 3.14.3 File Output.

```
      PROGRAM IOSAMP
/*
 * File input/output example
 */
      INTEGER A,B,C
/* Open files */
      OPEN 'FILE1' ON 1 FOR 'R'
      OPEN 'FILE2' ON 2 FOR 'W'
/* Read from unit 1 */
      READ 1;A,B
/* Write sum of A and B to unit 2 */
      C=A+B
      WRITE 2; 'A+B=',C
/* CLOSE files */
      CLOSE 1
      CLOSE 2
/* End Program */
      END
```

3.14.5 REWIND Statement

It is permissible to rewind a file using the statement:

```
REWIND unit
```

If a file is open for both reading and writing, it is not possible to read following a write unless a rewind is executed first.

3.14.6 DELETE Statement

A file can be deleted by an IPLAN program by executing the statement:

```
DELETE filename
```

where "filename" is a string variable, constant, or expression which identifies the file to be deleted.

3.15 Input/Output Data Conversions

IPLAN uses 4 kinds of internal data types: integer, real, logical, and string. These types determine how a given pattern of stored bits will be interpreted. IPLAN provides some control on how input or displayed values are converted to and from these formats. Elements in an iolist can have formatting information supplied. This is called formatted input or output. If the formatting is not supplied, and it is not required, this is called unformatted input and output.

3.15.1 Unformatted Output

Each of the data types will be described separately

Unformatted Integers

IPLAN will determine the number of digits required to display the value, and print only those digits. No leading zeros will be displayed. The number will be preceded by a single space.

Unformatted Reals

IPLAN will determine the number of digits required to display the value, and print only those digits. No leading zeros will be displayed for absolute values greater than one. For absolute values less than one, exactly one zero to the left of the decimal point will be displayed. For values of 1,000,000 or greater, no decimal digits will be displayed. For values less than 1,000,000 seven digits total will be displayed, including those to the left and those to the right of the decimal point. The number will be preceded by a single space.

Unformatted Strings

The contents of the string are displayed with no trailing spaces. When printed to the terminal (PRINT or WRITE 0;) and the string expression is the first I/O element, the string will be preceded by a single space. In all other cases, no space will precede the string.

Unformatted Logicals

Logical expressions are rendered as either the string 'TRUE' or the string 'FALSE'. They follow the rules for displaying strings.

3.15.2 Formatted Output

To specify a format, follow an iolist element immediately by a colon and a format specifier. A format specifier is either an integer, or two integers separated by a decimal point. This form can be described as w[.d], i.e. a field width (w), optionally (the []) followed by a decimal field (.d). For example:

```
PRINT A,B:3,C:5.2
```

For all data types, the meaning of the field width is that the iolist element will be displayed in the that number of places or characters. If the field width is larger than what is needed to hold the value, the displayed value will be right justified for integers and reals, and left justified for strings and logicals. The field width can be specified as a negative number, which reverses the normal left-right justification.

The meaning of the decimal field differs depending on data type. For reals it specifies the number of decimal places. A format of 5.2 will hold a number less than 100 and display two decimal places. For integers if specifies a minimum number of digits to display. The effect of this is to add leading zeros to meet this minimum number is the value is too small. A format of 5.2 would display 100 as 100 (with two leading spaces) but display 5 as 05 (with three leading spaces). The decimal field is ignored for strings and logicals.

The following program demonstrates some format specifications:

```
PROGRAM OUTSAMP
INTEGER I
STRING S
LOGICAL L
REAL R
ASK 'I,S,L,R: '
INPUT I,S,L,R
PRINT 'I:5 = ',I:5,' S:6.2 = ',S:6.2,' L:4 = ',L:4,' R:7.2 = ',R:7.2
PRINT 'I:10.5 = ',I:10.5,' S:3 = ',S:3,' L:2 = ',L:2,' R:7.3 =
',R:7.3
END
```

The following is an example of running this program:

```
I,S,L,R:
123 program true 15.603
 I:5 =   123 S:6.2 = progra L:4 = TRUE R:7.2 =   15.60
 I:10.5 =      00123 S:3 = pro L:2 = TR R:7.3 =  15.603
```

Formatted output can be used with the following IPLAN statements: PRINT, PRINTX, WRITE, WRITEX, REPORT, REPORTX, ASK, ASKX, PUSH, PUSHX, QPUSH, QPUSHX

3.15.3 Unformatted Input

All input statements in IPLAN follow the parsing rules described in section 8.4.

3.15.4 Formatted Input

Formats may also be specified for input using the READ and INPUT commands. Only field widths will be recognized, however. The following formats, although having different meanings when used for output, will all have exactly the same meaning when used for input:

```
4, -4, 4.2, -4.1.
```

3.15.5 TAB Function

The TAB function allows the user to tab to a specific column and begin I/O. The format of the TAB function is TAB(x) where x is an integer ranging from 1 to the length of the output buffer (see *Program Limitations* and Section 3.26.2 Controlling New Output Buffer Creation).

The TAB function may be used in conjunction with:

- PRINT and PRINTX.
- REPORT and REPORTX.
- ASK and ASKX.
- WRITE and WRITEX.
- PUSH and PUSHX.
- QPUSH and QPUSHX.
- READ.
- INPUT.

The format is as follows:

```
WRITE 1; S1,TAB(10),S2
```

This statement will write the variable S1 out to unit 1 starting at the first column. It will then write out the variable S2 starting at column 10 as demonstrated in the following example:

```
PROGRAM TABBE
STRING S1,S2
S1 = '12345678910'
S2 = 'ABCD'
OPEN 'TEST.DAT' ON 1 FOR 'W'
WRITE 1; S1,TAB(10),S2
```

```
CLOSE 1
END
```

When using the TAB function in conjunction with a READ or INPUT, the statement will cause reading to begin at the column specified.

```
READ 1; A,TAB(10),B
```

This command will read A starting at column 1 and then read B starting at column 10. This is useful for skipping over columns or rereading them.

If a negative value is used for a TAB function, the result is a "guarded" TAB. If the current position is less than the absolute value of the TAB argument, the TAB works as if the value were positive. If the current position is greater than the absolute value of the TAB argument, the TAB has no effect.

3.15.6 NEWREPORT Statement

The NEWREPORT statement has the form:

```
NEWREPORT
```

If the IPLAN simulator is running in an application that supports multiple report displays, the NEWREPORT command will cause a new report display to be created.

3.16 OPTION Statement

The OPTION statement has the form:

```
OPTION keyword
```

Keywords are defined as:

ZERO	IPLAN will clear variables that have no corresponding values in an INPUT or READ statement. INTEGERs are set to zero; REALs are set to zero, LOGICALs are set to FALSE and STRINGs are set to all blank (spaces).
NOZERO	IPLAN will not modify the value of variables that have no corresponding values in an INPUT or READ statement.
PRINTX	If the PRINTX buffer is not empty when an INPUT or INPUTLN statement is executed, it will be issued as a prompt. It would be as if the INPUT or INPUTLN statement were preceded by ASK <PRINTX buffer> (and then the PRINTX buffer is cleared).
NOPRINTX	PRINTX is handled per current documentation.

The default is:

```
OPTION ZERO
OPTION NOPRINTX
```

An OPTION statement may contain only one keyword. Multiple OPTION statements are permitted. The OPTION statement indicates a state for the entire program, no matter where in the program it is coded. Therefore, it is recommended to place them after declarations and before executable statements. Note that the following pairs of keywords indicate opposite states. If more than one of the same pair appears in a program, the last one is in effect for the entire program.

```
ZERO    /   NOZERO
PRINTX  /   NOPRINTX
```

3.17 Simple Control Flow

In this section, we will only be concerned with a simple method of controlling the flow of the program. Our device for this will be the GOTO statement. The GOTO here is similar to the GOTO in Fortran and PASCAL. A GOTO transfers execution to the specified statement label (see Section 3.5 Statement Labels). The label must be in the same procedure as the GOTO statement. For example:

```
        PROGRAM GOSAMP
        INTEGER I
        I = 0
LABEL1: I = I + 1
        PRINT I
        IF (I == 10) THEN STOP ENDIF
        GOTO LABEL1
        END
```

3.18 Procedures

A procedure is a section of code that, although separate from the main program, has access to main program variables. Hence, no argument passing is involved. Procedures are invoked using the PERFORM command. Procedures are similar in structure to the main program. If it is desired that a procedure return to the calling procedure before ENDing, it can be done with the RETURN statement. Note that although variables declared in the main program are global between the main program and the procedure, labels are not. Hence, you cannot GOTO a label outside the procedure. Procedures can PERFORM other procedures, but only main program variables are global. That is, variables declared in procedures are not available to other procedures no matter where the PERFORM statement appears. Procedures must follow the main program END statement and must conclude with their own END statements.

The source code for a procedure can be in a separate file and be included after the END statement in your main program. The following are simple examples of procedures:

```
   PROGRAM PROCSAMP
/* MAIN PROGRAM*1/
   INTEGER I
   I = 1
   PERFORM PROC1                    ! INVOKING FIRST PROCEDURE
   PRINT 'IN MAIN I IS: ',I
   PERFORM PROC2                    ! INVOKING SECOND PROCEDURE
   PRINT 'IN MAIN AGAIN I IS: ',I
   END

/* DEFINING FIRST PROCEDURE*/

   PROCEDURE PROC1
   INTEGER I
   I = 10
   PRINT 'PROC1 I IS: ',I
   RETURN
   END
```

```
/* DEFINING SECOND PROCEDURE*/

   PROCEDURE PROC2
   INTEGER I
   I = -99
   PRINT 'PROC2 I IS: ',I
   RETURN
   END
```

Similarly, if PROC1 was in a file called FIRST.DAT and PROC2 was in a file called SECOND.DAT, the following would also work:

```
   PROGRAM PROCSAMP
/* MAIN PROGRAM*1/
   INTEGER I
   I = 1
   PERFORM PROC1                    ! INVOKING FIRST PROCEDURE
   PRINT 'IN MAIN I IS: ',I
   PERFORM PROC2                    ! INVOKING SECOND PROCEDURE
   PRINT 'IN MAIN AGAIN I IS: ',I
   END

/* DEFINING FIRST PROCEDURE*/
   INCLUDE 'FIRST.DAT'
/* DEFINING SECOND PROCEDURE*/
   INCLUDE 'SECOND.DAT'
```

Please refer to Section 3.22 INCLUDE Statement for more information on the IPLAN INCLUDE statement.

3.19 Subroutines

The IPLAN compiler will not generate unresolved references to external routines, and the simulator does not contain general linkage-editing capability. All subroutines must be known by the compiler and simulator. Therefore, IPLAN does not have general subroutine capability (i.e., you cannot write a subroutine in the IPLAN language, and IPLAN cannot reference an existing program that is not documented in this manual). There are many subroutines that are available to an IPLAN program. Built-in functions are referenced as parts of expressions. For example:

```
variable = function-name(argument-list)
variable = variable + function-name(argument-list)
```

Other subroutines are accessed by the CALL statement, which has this form:

```
CALL subroutine-name(argument-list)
```

The argument-list is a list of expressions separated by commas. The IPLAN compiler will verify the number and types of the arguments. It will not enforce input/output expectations. Any expression (of the correct type) can be used for input arguments. If an expression (or constant) is used for an output argument, or in/out argument, the subroutine is supplied with a copy of the value of that expression, and the returned value is discarded.

IPLAN built-in functions are fully described in Chapter 5.

See sections 3.25.3 and 3.29 for more about subroutines that are referenced using the CALL statement. In particular, section 3.29 discusses how a user-supplied program may be called from IPLAN.

3.20 Condition/Exception Handling

When certain file I/O errors (conditions) or errors in mathematical calculations (exceptions) occur, it is possible to trap them (i.e., to have the IPLAN take specific actions when they occur). This is done with one of the following commands:

```
ON ERR (unit) GOTO label
ON ERR (unit) PERFORM procedure-name
```

Program control will transfer to the target (label or procedure-name) when any error occurs during a READ or WRITE statement to the unit specified in the ON command.

If the unit specified is zero, then that ON command will specify action to be taken in the case of a mathematical exception (e.g., divide by zero).

End-of-file while performing a READ is an error condition that can occur during file I/O that is often treated differently than other types of errors. The ON command can specify that end-of-file errors be handled differently with one of the following commands:

```
ON EOF (unit) GOTO label
ON EOF (unit) PERFORM procedure-name
```

The ON EOF specification take precedence over an ON ERR specification for the same unit. If only ON ERR is specified for a given unit, then end-of-file conditions are treated as any other I/O condition.

If both ON ERR and ON EOF appear in the program for the same unit, their order is not important.

The ON command sets a state for the remainder of the program execution. For example, it is not necessary to ever re-execute an ON command (i.e., an identically specified one). Neither is it necessary to execute an ON command at any particular place. It is common practice to place all ON commands at the beginning of the program, except in those cases where either different actions are required for the same unit at different places in the program, or where the GOTO form is used for a label in a procedure (in which case the ON command must be in the same procedure).

When the PERFORM form of the ON statement is used, the END or RETURN statement of the procedure will transfer control to the statement following the one that produced the error. For example, this program:

```
PROGRAM T
INTEGER I
ON ERR (0) PERFORM GOTERR
PERFORM DIV0
PRINT 'HERE'
END
PROCEDURE DIV0
I = 10/0
PRINT 'NEVER'
END
```

```
PROCEDURE GOTERR
PRINT 'GOTERR'
END
```

will print:

```
GOTERR
NEVER
HERE
```

The GOTO form of the ON statement will transfer control directly to the label specified. If the GOTO form is used in a program that also uses procedures, great care must be taken as IPLAN will still treat the next END or RETURN statement as terminating the procedure in which the error occurred, unless the label is in the main program. In that case, all the active procedures are terminated and control is transferred to the label. Here are two examples:

```
PROGRAM T
INTEGER I
ON ERR (0) GOTO LAB
PERFORM DIV0
PRINT 'RIGHT'
LAB:
PRINT 'HERE'
END
PROCEDURE DIV0
I = 10/0
PRINT 'NEVER'
END
```

which prints:

```
HERE
```

```
PROGRAM C
INTEGER I
PERFORM ANOTHER
PRINT 'AFTER'
PERFORM DIV0
PRINT 'RIGHT'
END
PROCEDURE DIV0
I = 10/0
PRINT 'NEVER'
END
PROCEDURE ANOTHER
ON ERR (0) GOTO LAB
LAB:
PRINT 'HERE'
END
```

which prints:

```
HERE
AFTER
HERE
RIGHT
```

3.21 Advanced Control Flow

A high-level programming language is complemented by a set of advanced control flow commands. The commands discussed here are IF/THEN/ELSE, LOOP, WHILE, UNTIL, and CONDITIONAL.

3.21.1 IF Statements

The IF statement takes one of three forms:

```
IF (logical expression) THEN <statement> ENDIF

-- or --
```

```
IF (logical expression) THEN
<statement list>
ENDIF

-- or --

IF (logical expression) THEN
<statement list>
ELSE
<statement list>
ENDIF
```

3.21.2 LOOP Statements

The LOOP command has many variations. In this section, iterative assignments will be discussed. That is having a variable take on values in an ascending or descending order. This will have any of four forms:

```
LOOP variable = start,end
<statement list>
ENDLOOP

-- or --

LOOP variable = start TO end
<statement list>
ENDLOOP

-- or --

LOOP variable = start TO end BY step
<statement list>
ENDLOOP

-- or --

LOOP variable = start DOWNTO end BY step
<statement list>
ENDLOOP
```

'Variable' must be an unsubscripted variable (i.e., no array elements).

'Start,' 'end,' and 'step' may be integer constants or expressions.

The first two examples assume 1) counting up, and 2) an increment of one. The third example will count from 'start' to 'end' by an increment of 'step.' The last example will count from 'start' down to 'end' by a decrement of 'step.'

Note that the exit condition is tested before the first pass, thus a loop may not be executed at all.

3.21.3 WHILE and UNTIL Statements

The WHILE and UNTIL statements are similar in structure. They both have two forms, one with a zero pass, the other with a one pass.

The following forms are zero pass (i.e., the condition is tested before the loop is entered):

```
LOOP WHILE (logical expression)
<statement list>
ENDLOOP
```

```
LOOP UNTIL (logical expression)
<statement list>
ENDLOOP
```

The following forms are one pass (i.e., the condition is tested at the end of the loop):

```
LOOP
<statement list>
WHILE (logical expression)
```

```
LOOP
<statement list>
UNTIL (logical expression)
```

3.21.4 CONDITIONAL Statements

The CONDITIONAL statement is equivalent to a series of IF/THEN/ELSEIF statements. The structure is a condition followed by a series of commands. For example:

```
CONDITIONAL
(logical expression)
<statement list>
(logical expression)
<statement list>
.
.
.
(OTHERWISE)
<statement list>
ENDCOND
```

Each expression is evaluated and the first to evaluate to true causes the following set of corresponding statements to be executed. The optional condition (OTHERWISE) is also available as a "catch all" condition.

3.21.5 Control Modifiers

There are four commands that provide even more versatility to the above control structures. They are NEXTLOOP, EXITLOOP, NEXTCOND, and EXITCOND.

For LOOP structures:

- The NEXTLOOP command will skip the remainder of that iteration and go on to the next.
- EXITLOOP will immediately exit the LOOP and continue processing statements after the ENDLOOP.

For CONDITIONAL structures:

- NEXTCOND will "jump" to the next condition check. Normally once a condition is matched in a CONDITIONAL structure, the statement list associated with that condition is executed, and then the CONDITIONAL structure is exited.
- EXITCOND will immediately exit the CONDITIONAL statement structure.

If LOOPs are nested, a particular LOOP, not necessarily the current, may be cycled or exited. The following is the generic form:

```
NEXTLOOP integer-constant
EXITLOOP integer-constant
```

where "integer-constant" is a positive value and represents a count of the loops to be cycled or exited. If "integer-constant" is 1, the current loop is cycled or exited.

3.22 INCLUDE Statement

The INCLUDE statement causes a file containing IPLAN program code to be inserted into the current program location. This code may be any valid IPLAN code including INCLUDE statements. Files may be nested up to 10 files deep. The format of the INCLUDE statement is:

```
INCLUDE 'filename'
```

If the file is not found, and the filename was specified without an extension, and attempt to open filename.INC will be made as well.

Use the -SEARCH (Section 4.3.6 SEARCH Option) option to specify directories to look for INCLUDE files.

3.23 Miscellaneous Commands

There are two miscellaneous commands available. They are BEEP and SLEEP. The command BEEP will issue a bell at the terminal (if possible). The SLEEP command will suspend execution for a specified number of seconds. For example:

```
BEEP
SLEEP 35/*sleep for 35 seconds */
BEEP
```

Note that BEEP and SLEEP may not work on some computer systems.

3.24 Program Debugging

The TRACE command causes the execution of a program to be traced for debugging. Tracing may be turned on and off as desired. This allows tracing of only critical sections of code. The form is as follows:

```
TRACE ON
<statement list>
TRACE OFF
```

3.25 Communication With Application Program

3.25.1 Transmitting to the Application: PUSH and PUSHX

It is important to understand what an IPLAN program does. From the perspective of the application program, it responds to a request for input. Actually, it intercepts that request and then takes over its own execution. That is why a PAUSE (see Section 3.12 Program Pauses) or a STOP (see Section 3.11 Program Termination) usually results in a terminal read; the application program resumes what it would have done had IPLAN not interfered. This is what gives IPLAN programs the power to tell the application what to do.

The primary method by which IPLAN programs supply responses to the application program is the PUSH command. The format of the PUSH command is the same as the PRINT command. For example, the statements:

```
I = 1
PUSH 'DO ',I
```

would send the message 'DO 1' to the application.

PUSHX bears the same relationship to PUSH as PRINTX does to PRINT. For example, the statements

```
PUSHX     'This'
PUSHX     ' is'
PUSH      ' an example.'
```

would send the message 'This is an example.' to the application. If the buffer exceeds the buffer wrapping limit (see Section 3.26.2 Controlling New Output Buffer Creation), only the characters up to that limit are used.

Normally, the message 'PUSHed' is displayed at the terminal along with any pending prompts. PUSH is an output statement and has the same data conversion capabilities (see Section 3.15 Input/Output Data Conversions).

3.25.2 Transmitting to the Application: QPUSH and QPUSHX

QPUSH differs from PUSH only in that neither the message or pending prompts are displayed (although pending prompts are cleared). QPUSHX and PUSHX are identical.

3.25.3 Receiving from the Application

To receive information from the application, the CALL command is used. The call commands allows execution of an application dependent routine. For example, if there was a subroutine GETVALUE that would return a special value based on a key, it might be done as follows:

```
CALL GETVALUE (KEY,VALUE)
```

Appendix A contains a list of available application specific subroutines along with descriptions of their use.

Using with Stand-Alone Simulator

When running in the stand-alone simulator (the -RUN and -CRUN compiler options; see Section 4.3 Options), the data retrieval routines cannot access the application. Routines whose argument list

contains an IERR variable will return a 1 for that variable, and all other will be unchanged. Other routines will return zero for integers and reals, false for logicals, and blanks for strings. Any exceptions are documented for that routine in Appendix A.

3.26 SET Statement

The SET command is used to control certain values that may be used by the simulator at a later time, and has the following form:

```
SET keyword value
```

The applicable keywords for the SET command and their meanings are described below.

3.26.1 Transmitting to the Application on Termination

The SET command can be used for issuing a command to the application when the IPLAN program terminates. The following keywords apply for this purpose.

END	*value* must be a string expression, and the effect is to generate a PUSH *value* command if the program terminates because of encountering an END statement. If the PUSHX output buffer is not empty when the program terminates, the SET END command has no effect.
STOP	*value* must be a string expression, and the effect is to generate a PUSH *value* command if the program terminates because of encountering an STOP statement. If the PUSHX output buffer is not empty when the program terminates, the SET STOP command has no effect.
INTERRUPT	*value* must be a string expression, and the effect is to generate a PUSH *value* command if the program terminates because of user interruption (see Chapter 8).

3.26.2 Controlling New Output Buffer Creation

Each of the various output commands, such as PRINT, WRITE, ASK, or REPORT (including the PUSH commands) will write first to output buffers, converting internal data to output format. The maximum size of the output buffers is noted in *Program Limitations*. The point at which the buffer is considered "full", the wrapping limit, will default to that maximum size, but can be set to a smaller size. The SET command is used to control this. The following keywords apply for this purpose.

PRINT	value must be an integer expression, greater than 10, and less than or equal to the output buffer limit noted in *Program Limitations*. PRINT and PRINTX commands will display the contents of its buffer when the next iolist element would have caused the overall buffer length to have exceeded value (see Section 3.13.2 Terminal Output, Temporary Information: PRINT and PRINTX).
WRITE	value must be an integer expression, greater than 10, and less than or equal to the output buffer limit noted in *Program Limitations*. WRITE and WRITEX commands will write the contents of the buffer for that unit when the next iolist element would have caused the overall buffer length to have exceeded value (see Section 3.14.3 File Output).

REPORT value must be an integer expression, greater than 10, and less than or equal to the output buffer limit noted in *Program Limitations*. REPORT and REPORTX commands will display the contents of its buffer when the next iolist element would have caused the overall buffer length to have exceeded value (see Section 3.13.3 Terminal Output, Permanent Information: REPORT and REPORTX).

ASK value must be an integer expression, greater than 10, and less than or equal to the output buffer limit noted in *Program Limitations*. ASK and ASKX commands will begin a new prompt when the next iolist element would have caused the overall length of the current buffer to have exceeded value (see Section 3.13.4 Terminal Output, Prompts: ASK and ASKX).

PUSH value must be an integer expression, greater than 10, and less than or equal to the output buffer limit noted in *Program Limitations*. The buffer associated with the PUSH commands will be truncated to the length value (see Section 3.25.1 Transmitting to the Application: PUSH and PUSHX).

3.27 Program Arguments

If an argument string was specified when the IPLAN program when invoked, then the ARGUMENT command can be used to retrieve the values. The form of the ARGUMENT command is as follows:

```
ARGUMENT ipos,<variable-list>
```

where ipos is an INTEGER, and <variable-list> is a list of zero or more variables of any types.

The argument string is parsed and values assigned to variables in the variable-list in the same way as the INPUT command, except that the argument string can be reread. Whenever the ARGUMENT command is used, IPLAN remembers the position in the buffer of the last value read, and will begin at that point if an ipos value of zero is specified. The nonzero value ipos can be used to specify a different point to begin parsing the argument string. At the first execution, the position is 1. Variables specified for which there are no values follow the same rules as the INPUT command; they are unchanged unless the OPTION ZERO command is used.

The internal function ARGPOS can be used to find the current position of the argument string parser.

For example, if the argument string was:

```
abcdefg hijk 'ABCDEFG HIJK'
```

the following program would produce these results:

IPLAN Program	Results
`PROGRAM SHOWARG`	`abc hij` `14`
`STRING X,Y`	`abc hij` `14`
`ARGUMENT 0,X,Y`	`ABC` `xxx`
`PRINT X:3, '  ', Y:3,ARGPOS:4`	`xxx`

IPLAN Program	Results
`ARGUMENT 1,X`	`def hij 14`
`ARGUMENT 0,Y`	`hij ABC xxx`

```
PRINT X:3, '      ', Y:3,ARGPOS:4
ARGUMENT 0,X,Y
PRINT X:3, '      ', Y:3,ARGPOS:4
ARGUMENT 0,X,Y
PRINT X:3, '      ', Y:3,ARGPOS:4
ARGUMENT 4,X,Y
PRINT X:3, '      ', Y:3,ARGPOS:4
ARGUMENT 8,X,Y
PRINT X:3, '      ', Y:3,ARGPOS:4
END
```

The value of xxx shown in the example will be 1 (one) greater than the length of the buffer used to hold the argument string. This buffer will have the same length as IPLAN input buffers (see *Program Limitations*).

See Section 4.3.7 ARG/NOARG Options for more information on specifying arguments to IPLAN programs.

3.28 Graphics

IPLAN supports a set of graphics subroutines. They allow users to produce plots.

The syntax for a call to a fictional routine, GRAPHCALL, is as follows:

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

A full listing of these subroutines can be found in Appendix B.

3.29 User-Defined Routines

IPLAN has the ability to recognize up to nine predefined subroutine names. They are USREX1, USREX2, USREX3, USREX4, USREX5, USREX6, USREX7, USREX8, and USREX9. The syntax for calling one, say USREX1, is as follows:

```
CALL USREX1 (I,X,A,J)
```

where I and J are IPLAN INTEGER variables, X is an IPLAN REAL variable, and A is an IPLAN STRING variable. Note that the arguments are fixed, and the same for all nine routines.

The intended use of this ability is allow ordinary Fortran programs to be called from within an IPLAN program (languages other than Fortran may be used to create these other programs, but the calling program will be a Fortran program and any cross-language compatibility issues are the responsibility of the user).

The following is a very simple example:

```
PROGRAM CALLUSR
INTEGER I,J
REAL    X
STRING  A
I=1
J=2
X=243.
A='from IPLAN'
CALL USREX1(I,X,A,J)
PRINT I,J,X,A
END

SUBROUTINE USREX1(A,B,C,D)
INTEGER A,D
REAL    B
CHARACTER*(*) C
PRINT *,A,D,B,C
A=21
B=22
D=23
C='from Fortran'
RETURN
END
```

which produced the following output:

```
1 2 243.0000from IPLAN
21 23 22.0from Fortran
```

Please note that this example was run in an executable without a GUI. There are some programming considerations for user-defined routines, particularly in the area of Fortran I/O. Please see Chapter 7 for more details.

SIEMENS

Chapter 4

Developing IPLAN Programs

4.1 Editing

IPLAN programs are text files, and as such, can be built with a standard text editor.

4.2 Compilation

The command IPLAN is used to compile programs. IPLAN utilizes a two-pass compiler. The first pass checks the syntax of each IPLAN statement for correctness. The second pass checks the program semantics. Semantics are activities which are related to the "meaning" of the program; that is, does the structure of the program make sense. By default, IPLAN source files have a file type "IPL", INCLUDE files have a file type "INC", and the executable files have the file type "IRF".

On systems which allow programs to read the command line which started the program, IPLAN can be invoked by entering:

```
IPLAN program options
```

If unable to read the command line or if the program name is not entered on the command line, the user is prompted for the program name:

```
Enter filename and options: TEST -LIST
```

If your file system permits spaces in filenames, then the filename must be quoted, or followed by a comma if options are specified.

Options specified on the command line without a filename are taken as overriding the defaults. Note, however, that -SEARCH is a special case (see comments under -LOOP and -SEARCH below) and that for -ARG, the argument string will not be preserved.

4.3 Options

IPLAN can be invoked with any of several options. Multiple options can be specified; the order is not important. They are:

-LIST	Creates a listing file.
-NOLIST	Turns off -LIST option.
-STAT	Produces program statistics.
-NOSTAT	Turns off -STAT option.

-CRUN	Invokes the simulator after compilation.
-RUN	Invokes the simulator only.
-COMP	Invokes the compiler only.
-LOOP	Requests additional input files without terminating IPLAN.
-INC	Requests additional information in the IRF file to handle run-time errors in INCLUDE files.
-NOINC	Turns off the -INC option.
-SEARCH	Specifies directories to resolve INCLUDE.
-ARG	Specifies a run-time argument string.
-NOARG	Turns off the -ARG option.

4.3.1 LIST/NOLIST Options

An optional listing file can be produced by specifying a -LIST option. This listing file consists of line numbers followed by their corresponding IPLAN statements. If errors occur during compilation, they are noted in the listing file by **** below the offending line. A detailed list is also given at the end of the listing file.

The listing file will have the same name as the ipl file with a.LST suffix.

-NOLIST requests that the listing file not be produced.

-NOLIST is the default.

4.3.2 STAT/NOSTAT Options

An optional statistical printout can be obtained by specifying a -STAT option on either the command line or when prompted. The listing will give present limits of the simulator when this version of IPLAN was released and amount this program took up as well as the number of integers, reals, etc.

-NOSTAT requests that the statistics not be produced.

If the limits are exceeded, the -STAT option will be generated automatically.

-NOSTAT is the default.

4.3.3 COMP, RUN, and CRUN Options

IPLAN programs are normally executed as part of an application program. See Appendix A for instructions for invoking the application specific subroutine calls.

The IPLAN compiler can also execute IPLAN programs. The compiler has no ties to any application program; hence, the application-specific CALL statements will return zero, blank, or false (depending on the types of arguments) rather than expected values (see Section 4.1 Stand-Alone Simulator).

The -COMP option will compile a program. The -RUN option will execute a previously complied program. The -CRUN option will compile the program and, if no compilation errors are found, run the program.

-COMP is the default.

4.3.4 LOOP Option

Entering the following, for example,

```
IPLAN program -LOOP
```

would cause IPLAN to prompt for another filename after compiling. The -LOOP option can be entered from the command line or from the prompt:

```
Enter filename and options:
```

After the first file is compiled and/or run, it is not necessary to continue to enter -LOOP. Additional prompts will be issued as long as new files are entered. All other options except -SEARCH will have their default values unless explicitly overridden for each file (see below concerning omitting the filename). If the source file is not found, the -LOOP option is issued automatically, but only until an existing filename is entered.

-LOOP is not used by default.

4.3.5 INC/NOINC Options

This option will allow errors that occur in source code from INCLUDE files to identify the line number in the INCLUDE files where they occur. Without this option, only the line number of the INCLUDE statement in the main IPL file can be identified. This option will increase the size of the IRF file 1 kB for every 256 lines in an INCLUDE file. This option also affects both compile-time and run-time errors.

-NOINC requests that this information not be included in the IRF file.

-INC is the default.

4.3.6 SEARCH Option

By default, only the local directory is searched for INCLUDE files. With the -SEARCH option, up to five additional directories can be specified. When used with the -LOOP option, subsequent compiles will use the last specified -SEARCH set of directories. Specifying -SEARCH with no directories results in a search of the current directory only.

-SEARCH is not used by default.

4.3.7 ARG/NOARG Options

Specifying -ARG will cause the IPLAN simulator to request an argument string if one is not specified. One can be specified immediately following -ARG (in this case, the string must be placed in quotes if it contains blanks or commas). When the simulator is used within an application program, the method of specifying the argument string will vary, especially if a GUI is used (see Section 8.3 Specifying Arguments). In line mode, it may be possible to specify the arguments following the filename, or the -ARG flag may be required.

-NOARG is the default.

4.4 File Types

IPLAN can only create formatted sequential files. The following Fortran90 specifiers are used when opening files:

```
All files:
ACCESS = 'SEQUENTIAL'
FORM = 'FORMATTED'
```

If	OPEN mode=	'R'	'W'	'RW'	'A'
Then	ACTION=	'READ'	'WRITE'	'READWRITE'	'READWRITE'
	POSITION=	'REWIND'	'REWIND'	'REWIND'	'APPEND'
	STAT=	'OLD'	'UNKNOWN'	'UNKNOWN'	'UNKNOWN'

If 'A' is specified for the OPEN mode, 'R' and/or 'W' are ignored.

The other file characteristic that can be specified is 'F'. This behavior is system dependant and is described in detail in the following sections. It stands for Fortran Carriage Control, a system of using the first column of the output buffer to control line spacing. Most reports written by Siemens PTI products will generate output buffers in this format.

4.4.1 Carriage Control Characters

In the sections below the following characters are discussed. These are single byte ASCII values. The decimal, hexadecimal, and control character values are simply alternate descriptions by which these values may be known:

Symbol	ASCII name	decimal	hexadecimal	control character
<LF>	line feed	10	0A	cntl-J
<FF>	form feed	12	0C	cntl-L
<CR>	carriage return	13	0D	cntl-M

4.4.2 Fortran Carriage Control on UNIX

Fortran Carriage Control is not implemented by compilers used on UNIX systems. Output buffers are terminated by adding a <LF> character to the end of the buffer. The control information that is in column 1 of the output buffer is written to the file unchanged. Siemens PTI products provide an auxiliary program to read such files, remove the first column, and insert <FF> characters where needed. When files are read, they are read expecting records to be terminated by a <LF> character. All the characters up to but not including the next <LF> character are returned in the input buffer.

Because the <LF> character is often interpreted as both line feed and a carriage return on UNIX systems, it is commonly referred to as a new-line character, and coded in C language programs as '\n' (<CR> is '\r' in C, and <FF> is '\f' in C).

4.4.3 Fortran Carriage Control on MS Windows

Two additional specifiers are used when opening files on the MS Windows systems. Their values depend on whether 'F' is specified as part of the OPEN mode:

	mode with 'F'	otherwise
CARRIAGECONTROL =	'FORTRAN'	'LIST'
RECORDTYPE=	'STREAM_CR'	'STREAM_LF'

The CARRIAGECONTROL specifier only affects buffers being written. When 'LIST' is specified, the file is written with no changes to the output buffer, and output buffers are terminated by adding both a <CR> character and a <LF> character to the end of the buffer. When 'FORTRAN' is specified, the first character is removed from the output buffer, and carriage control characters are added depending of the value of that character, as follows:

Column 1	output buffer preceded by	output buffer followed by
+	(nothing)	<CR>
blank	<LF> (nothing if record 1)	<CR>
0	<LF><LF> (<LF> if record 1)	<CR>
1	<FF>	<CR>
$	<LF>(nothing if record 1)	(nothing)
null	(nothing)	(nothing)

Please note that '$' and null are extensions to what is generally recognized as the "standard" set of control characters, but they are supported by the compiler used by IPLAN. The file is terminated by a <LF> character. If any character other than the ones listed above appears in column one, it is treated as if it were a blank. Most reports from Siemens PTI products use blank and one exclusively.

The RECORDTYPE specifier only affects buffers being read. When 'STREAM_LF' is specified, the file is read expecting records to be terminated by the two characters <CR><LF>. All the characters up to but not including the next <CR><LF> pair of characters are returned in the input buffer. When 'STREAM_CR' is specified, the file is read expecting records to be terminated by a <CR> character. All the characters up to but not including the next <CR> character are returned in the input buffer.

An IPLAN program was written to demonstrate these features. The program created two files, Ffile and Lfile, opened with modes of 'WF' and 'W', respectively. The following five strings were written to each file, one string per write:

```
"1JUST"
" THIS"
" LINE"
"1WITH"
" MORE"
```

Each file was then read in with 'RF' and 'R' specified. Then each file was reopened for append (with the same specification of either 'F' or not) and the same five records were written again. Again, both

files were read the two ways described above. Finally, both files were opened again for append, but this time just 'A' was specified for Ffile, and 'AF' was specified for Lfile. Again, the same five records were written to each file, and again each file was opened the two possible ways. The results are shown on the next page. In that display, wherever a <LF> character would appear, <10> is shown, and similarly <12> is shown for <FF> and <13> for <CR> (this is because these characters do not have any displayable symbols normally associated with them).

4.4.4 Fortran Carriage Control on Other Systems

No other operating systems are supported as of this release of IPLAN.

The following is an example of reformatted output from demonstration program described in Section 4.4.3 Fortran Carriage Control on MS Windows.

create Ffile with mode of 'wf', and Lfile with mode of 'w'

read Ffile with 'rf'	read Ffile with 'r'	read Lfile with 'rf'	read Lfile with 'r'
1: "<12>JUST"	1: "<12>JUST"	1: "1JUST"	1: "1JUST"
2: "<10>THIS"	2: "THIS"	2: "<10> THIS"	2: " THIS"
3: "<10>LINE"	3: "LINE<13><12>WITH"	3: "<10> LINE"	3: " LINE"
4: "<12>WITH"	4: "MORE"	4: "<10>1WITH"	4: "1WITH"
5: "<10>MORE"		5: "<10> MORE"	5: " MORE"
6: "<10>"		6: "<10>"	

append to Ffile with mode of 'af', and Lfile with mode of 'a'

read Ffile with 'rf'	read Ffile with 'r'	read Lfile with 'rf'	read Lfile with 'r'
1: "<12>JUST"	1: "<12>JUST"	1: "1JUST"	1: "1JUST"
2: "<10>THIS"	2: "THIS"	2: "<10> THIS"	2: " THIS"
3: "<10>LINE"	3: "LINE<13><12>WITH"	3: "<10> LINE"	3: " LINE"
4: "<12>WITH"	4: "MORE"	4: "<10>1WITH"	4: "1WITH"
5: "<10>MORE"	5: "<12>JUST"	5: "<10> MORE"	5: " MORE"
6: "<10><12>JUST"	6: "THIS"	6: "<10>1JUST"	6: "1JUST"
7: "<10>THIS"	7: "LINE<13><12>WITH"	7: "<10> THIS"	7: " THIS"
8: "<10>LINE"	8: "MORE"	8: "<10> LINE"	8: " LINE"
9: "<12>WITH"		9: "<10>1WITH"	9: "1WITH"
10: "<10>MORE"		10: "<10> MORE"	10: " MORE"
11: "<10>"		11: "<10>"	

append to Ffile with mode of 'a', and Lfile with mode of 'af'

read Ffile with 'rf'	read Ffile with 'r'	read Lfile with 'rf'	read Lfile with 'r'
1: "<12>JUST"	1: "<12>JUST"	1: "1JUST"	1: "1JUST"
2: "<10>THIS"	2: "THIS"	2: "<10> THIS"	2: " THIS"
3: "<10>LINE"	3: "LINE<13><12>WITH"	3: "<10> LINE"	3: " LINE"
4: "<12>WITH"	4: "MORE"	4: "<10>1WITH"	4: "1WITH"
5: "<10>MORE"	5: "<12>JUST"	5: "<10> MORE"	5: " MORE"
6: "<10><12>JUST"	6: "THIS"	6: "<10>1JUST"	6: "1JUST"
7: "<10>THIS"	7: "LINE<13><12>WITH"	7: "<10> THIS"	7: " THIS"

```
8: "<10>LINE"
9: "<12>WITH"
10: "<10>MORE"
11: "<10>1JUST"
12: "<10> THIS"
13: "<10> LINE"
14: "<10>1WITH"
15: "<10> MORE"
16: "<10>"
```

```
8: "MORE"
9: "1JUST"
10: " THIS"
11: " LINE"
12: "1WITH"
13: " MORE"
```

```
8: "<10> LINE"
9: "<10>1WITH"
10: "<10> MORE"
11: "<10><12>JUST"
12: "<10>THIS"
13: "<10>LINE"
14: "<12>WITH"
15: "<10>MORE"
16: "<10>"
```

```
8: " LINE"
9: "1WITH"
10: " MORE"
11: "<12>JUST"
12: "THIS"
13: "LINE<13><12>WITH"
14: "MORE"
```

4.1 Stand-Alone Simulator

Section 4.3.3 COMP, RUN, and CRUN Options documents the compiler options –RUN and –CRUN, which allow the compiler to execute IPLAN programs using the stand-alone simulator, so called because has no application data associated with it. It will recognize and allow calls to all the application data retrieval routines documented in Appendix A, but those routines will return error codes or values that best indicate that no data is available.

PUSH statements (and variants) will not be able to pass commands back to an application. Instead, the command "PUSH 'abcdef'" will echo as:

```
' PROGRAM PUSHED abcdef RESUMING'
```

PAUSE statements (and variants) will not pause as there is no application control to pause for. Instead, the command "PAUSE" will echo as:

```
' PROGRAM PAUSED, RESUMING'
```

The stand-alone simulator can be used to more rapidly test formatting, error reporting, general program structure, and any logic that does not require the use of actual program data, than using the application program and simulator combined.

If the IPLAN program uses input file redirection, the PAUSE READ and PAUSE WHILE commands will not pause program execution in the stand-alone simulator.

SIEMENS

Chapter 5

IPLAN Internal Functions

5.1 Functional Summary

Table 5-1 IPLAN Internal Functions shows IPLAN internal functions and as such are available in all IPLAN implementations. Appendix A contains application specific subroutines.

Table 5-1. IPLAN Internal Functions

Name	Description	Type
Conversion:		
TOINT	Convert a value to integer	Integer
TOLOG	Convert a value to logical	Logical
TOREAL	Convert a value to a real	Real
TOSTR	Convert a value to a string	String
TRUNC	Truncate decimal portion of a real	Real
NINT	Round real to nearest integer	Integer
FLOOR	Next lowest integer value of a real	Integer
CEIL	Next greatest integer value of a real	Integer
CHAR	Returns the character in the i^{th} position in the collating sequence	String∗1
ICHAR	Converts character to integer based on collating sequence	Integer
Trigonometric:		
ACOS	Arc cosine of a real	Real
ASIN	Arc sine of a real	Real
ATAN	Arc tangent of a real	Real
ATAN2	Arc tangent of two reals	Real
COS	Cosine of a real	Real
COSH	Hyperbolic cosine of a real	Real
SIN	Sine of a real	Real
SINH	Hyperbolic sine of a real	Real

Table 5-1. IPLAN Internal Functions (Cont.)

Name	Description	Type
TAN	Tangent of a real	Real
TANH	Hyperbolic tangent of a real	Real
Parsing:		
GTINIT	Initialize internal buffer	Integer
GTCHAR	Get one character from buffer	String
GTWORD	Get one word from buffer	String
GTWRDL	Get one word from buffer end return length	String
GTINT	Get one integer from buffer	Integer
GTREAL	Get one real from buffer	Real
GTREST	Get remainder of buffer	String
GTRESET	Reset the buffer	Integer
String Manipulation:		
LEFT	Left-most portion of a string	String
RIGHT	Right-most portion of a string	String
JUSTLEFT	Left justify a string	String
JUSTRIGHT	Right justify a string	String
REPEAT	Initialize string with repeating string	String
SUB	Extract a portion of a string	String
SUBSTR	Substitute one string for another	String
INDEX	Locate one string within another	Integer
SCAN	Locate first of a set of characters in a string	Integer
VERIFY	Locate first character in a string not from a set	Integer
LEN	Return the length of a string	Integer
BUFINIT	Initialize a buffer	String
BUFINT	Place integer into buffer	String
BUFREL	Place real into buffer	String
BUFSTR	Place string into buffer	String
Note: Operators are provided for case folding. See Section 3.10 Expressions.		
I/O:		
SPOOL	Spool a file	Integer
SCREEN	Pass control character to terminal	Integer
EXISTF	Test for the existence of a file	Logical
PROMPT	Issue a prompt	Integer
PROMPTC	Clear prompts	Integer
PROMPTR	Retrieve a prompt	String
PROMPTI	Return the number of pending prompts	Integer
FNDFIL	Initialize file list	Integer

Table 5-1. IPLAN Internal Functions (Cont.)

Name	Description	Type
FNDNXT	Return a name from the file list	String
GETLU	Get an available Fortran unit number	Integer
FRELU	Free a Fortran unit number	Integer
FILESEL	Display file selector window	String
GETDIR	Return current working directory	String
SETDIR	Set current working directory	Integer
Exponential/Logarithms:		
EXP	Raise e to a real power	Real
LOG	Log base e of a real	Integer
LOG10	Log base 10 of a real	Real
SQRT	Square root of a real number	Real
Date/Time:		
GETDATE	Return the current date	String
GETTIME	Return the current time	String
Miscellaneous Arithmetic:		
ABS	Absolute value of a number (integer/real)	Integer/Real
IABS	Absolute value of an integer	Integer
RABS	Absolute value of a real	Real
CABS	Magnitude of complex ordered pair	Real
DIFF	Positive difference of a number (integer/real)	Integer/Real
IDIFF	Positive difference of two integers	Integer
RDIFF	Positive difference of two reals	Real
MAX	Maximum value of a list (integer/real)	Integer/Real
IMAX	Maximum value of a list of integers	Integer
RMAX	Maximum value of a list of reals	Real
MIN	Minimum value of a list (integer/real)	Integer/Real
IMIN	Minimum value of a list of integers	Integer
RMIN	Minimum value of a list of reals	Real
MOD	Modulo of two numbers (integer/real)	Integer/Real
IMOD	Modulo of two integers	Real
RMOD	Modulo of two reals	Real
SIGN	Sign of a number (integer/real)	Integer/Real
ISIGN	Sign of an integer number	Integer
RSIGN	Sign of a real number	Real
ROUND	Round a real number	Real
Other:		
ARGPOS	Current position of argument parser	Integer

Table 5-1. IPLAN Internal Functions (Cont.)

Name	Description	Type
ASIZE	Get declared size of an array	Integer
LENSTR	Get declared size of a string	Integer
GETTYPE	Return the type of a variable	Integer
IPLANINFO	Return IPLAN of application release number	Real
IPLANINFS	Return IPLAN of application release number	String
RND	Generate/seed a random number	Integer

5.2 Alphabetic Listing and Reference

Note that all arguments are for input unless specifically noted.

ABS(x)

Returns the absolute value.

Returns: INTEGER/REAL
Arguments: x = INTEGER or REAL
Errors: NONE

ACOS(x)

Returns the arc cosine of x in radians.

Returns: REAL
Arguments: x = REAL
Errors: x < -1 or x > 1

ARGPOS

Returns current position of parser in the argument buffer.

Returns: INTEGER
Arguments: NONE
Errors: NONE

ASIN(x)

Returns the arc sine of x in radians.

Returns: REAL
Arguments: x = REAL
Errors: x < -1 or x > 1

ASIZE

Returns declared size of array x. x must be specified as an array element (any element). If y is present, it must be 1 or 2, indicating which dimension of a 2-dimensional array is being inquired of. If omitted for a 2-dimensional array, the total size is returned.

Returns: INTEGER
Arguments: x = any type

y = optional INTEGER
Errors: NONE

ATAN(x)

Returns the arc tangent of x in radians.

Returns: REAL
Arguments: x = REAL
Errors: NONE

ATAN2(x,y)

Returns arc tangent of x/y in radians.

Returns: STRING
Arguments: x = REAL
y = REAL
Errors: NONE

BUFINIT(x)

Returns a blank buffer for building strings. Error code x is also returned.

Returns: STRING
Returned Arguments: x = INTEGER
Errors: x > 0

BUFINT(x)

Returns internal buffer with integer x appended to it. BUFINIT must have been called initially.

Returns: STRING
Arguments: x = INTEGER
Errors: NONE

BUFREL(x,y)

Returns internal buffer with real x appended to it using y as total width including decimal point. BUFINIT must have been called initially.

Returns: STRING
Arguments: x = REAL
y = INTEGER
Errors: NONE

BUFSTR(x)

Returns internal buffer with string x appended to it. BUFINIT must have been called initially.

Returns: STRING
Arguments: x = STRING
Errors: NONE

CABS (x,y)

Returns magnitude of complex ordered pair.

Returns:	REAL
Arguments:	x = REAL
	y = REAL
Errors:	NONE

CEIL(x)

Returns the smallest integer greater than or equal to x.

Returns:	INTEGER
Arguments:	x = REAL
Errors:	NONE

CHAR(x)

Returns the character in the i^{th} position in the ASCII or EBCDIC character set. See Appendix C for the ASCII and EBCDIC character sets.

Returns:	STRING * 1
Arguments:	x = INTEGER
Errors:	NONE

COS(x)

Calculates the cosine of x where x is in radians.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

COSH(x)

Returns the hyperbolic cosine of x where x is in radians.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

DIFF(x,y)

Returns the positive difference, i.e., the absolute value of x-y. Note that both x and y must be of the same type.

Returns:	INTEGER/REAL
Arguments:	x = INTEGER or REAL
	y = INTEGER or REAL
Errors:	NONE

EXISTF(x)

Tests for the existence of the file x.

Returns:	LOGICAL
Arguments:	x = STRING
Errors:	NONE

EXP(x)

Raises 'e' to the power of x.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

FILESEL(mask,width,lines,title)

When running under windows, displays a standard file selector using the file mask specified in the argument "mask", and returns the selected file name; if supported by the application, the title field will be displayed in the title bar. (The title argument may be omitted.) When not running under windows, displays a list of files, using the arguments width and lines to control the display of the output (number of characters wide, number of lines before pausing), and requests that a file name be typed in; if 0 < lines < 500 then the display will pause and request an input indicating whether to quit or continue for every "page". Note that this could impact response files.

Returns:	STRING
Arguments:	mask = STRING
	width = INTEGER
	lines = INTEGER
	title = STRING
Errors:	NONE

FLOOR(x)

Returns the largest integer less than or equal to x.

Returns:	INTEGER
Arguments:	x = REAL
Errors:	NONE

FNDFIL(x)

Returns the number of files matching the file specification, x, and initializes the file list for FNDNXT.

Returns:	INTEGER
Arguments:	x = STRING
Errors:	NONE

FNDNXT(x)

Returns the x^{th} file names initialized by FNDFIL.

Returns:	STRING
Arguments:	x = INTEGER
Errors:	NONE

FRELU(lu)

Returns an error code.

Returns:	Integer
Arguments:	lu = INTEGER, the unit number from previous GETLU call to be freed
Errors:	0 if no error: 1 if unit number is invalid 2 if unit number not in use

GETDATE(x)

Returns the date based on x.

Returns:	STRING
Arguments:	x = INTEGER 0 = the entire date (DDD, MMM ## YYYY) 1 = the day name only (DDD) 2 = the day number only (##) 3 = the month only (MMM) 4 = the year only (YYYY)
Errors:	NONE

GETDIR

Returns the name of the current working directory.

Returns:	STRING
Arguments:	NONE
Errors:	Returns blanks if any error occurs

GETLU

Returns a Fortran unit number that is available for use, and marks that number as unusable until FRELU is used.

Returns:	Integer
Arguments:	NONE
Errors:	If no units available, returns zero

GETTIME(x)

Returns the time based on x.

Returns:	STRING
Arguments:	x = INTEGER
	0 = the entire time (HH:MM)
	1 = the hour only (HH)
	2 = the minute only (MM)
	3 = the current CPU time (if available)
	4 = the current connect time (if available)
	5 = the current I/O time (if available)
Errors:	NONE

GETTYPE(x)

Returns the type of variable x.

Returns:	INTEGER
	0 = INTEGER
	1 = REAL
	2 = STRING
	3 = LOGICAL
Arguments:	x = ANY
Errors:	NONE

GTCHAR(x)

Returns a single character from the internal buffer initialized by GTINIT, and an error code x. (The buffer must have been initialized using GTINIT.) See Section 8.4 Parsing for information about parsing.

Returns:	STRING
Returned Arguments:	x = INTEGER (error code)
Errors:	x = 0; no error
	x = 1; at end of buffer

GTERR(x)

Returns the current value of the GTxxx parser error flag and sets the flag to x. The parser will display errors for data conversions errors (see GTINT, GTREAL, GTWORD) when this flag is on (true). Display of these error messages cannot be suppressed any other way.

Returns:	LOGICAL
Arguments:	x = LOGICAL
Errors:	NONE

GTINIT(y,x)

Initializes parser using buffer y. Returns the number of characters initialized, and an error code x. See Section 8.4 Parsing for information about parsing.

Returns:	INTEGER
Returned Arguments:	x = INTEGER (error code)
Arguments:	y = STRING
Errors:	x = NONE

GTINT(x)

Returns an integer from the internal buffer initialized by GTINIT, and an error code x. (The buffer must have been initialized using GTINIT.) See Section 8.4 Parsing for information about parsing.

Returns:	INTEGER
Returned Arguments:	x = INTEGER
Errors:	x = 0; no error
	x = 1; missing data, 0 returned
	x = 2 or 3; end of buffer, 0 returned
	x = 4; data conversion error; 0 returned

GTREAL(x)

Returns a real from the internal buffer initialized by GTINIT, and an error code x. (The buffer must have been initialized using GTINIT.) See Section 8.4 Parsing for information about parsing.

Returns:	REAL
Returned Arguments:	x = INTEGER
Errors:	x = 0; no error
	x = 1; missing data, 0.0 returned
	x = 2 or 3; end of buffer, 0.0 returned
	x = 4; data conversion error; 0.0 returned

GTRESET(x)

Returns the number of characters reinitialized, and an error code x. Resets the parser to the state following the preceding GTINIT call. See Section 8.4 Parsing for information about parsing.

Returns:	INTEGER
Returned Arguments:	x = INTEGER
Errors:	x = 0; no error
	x = 1; buffer already at initial point

GTREST(x)

Returns the remainder of the internal buffer initialized by GTINIT, and an error code x. (The buffer must have been initialized using GTINIT.) See Section 8.4 Parsing for information about parsing.

Returns:	STRING
Returned Arguments:	x = INTEGER
Errors:	x = 0; no error
	x = 1; at end of buffer

GTWORD(x)

Returns a word from the internal buffer initialized by GTINIT, and an error code x. (The buffer must have been initialized using GTINIT.) See Section 8.4 Parsing for information about parsing.

Returns:	STRING
Returned Arguments:	x = INTEGER
Errors:	x = 0; no error
	x = 1; missing data, blanks returned
	x = 2 or 3; end of buffer, blanks returned
	x = 4; data conversion error, blanks returned

GTWRDL(n,x)

Returns a word from the internal buffer initialized by GTINIT, the number of characters returned, and an error code x. (The buffer must have been initialized using GTINIT.) See Section 8.3 Specifying Arguments for information about parsing.

Returns:	STRING
Returned Arguments:	n = INTEGER
	x = INTEGER
Errors:	x = 0; no error
	x = 1; missing data, blanks returned
	x = 2 or 3; end of buffer, blanks returned
	x = 4; data conversion error, blanks returned

IABS(x)

Returns the absolute value of the number x.

Returns:	INTEGER
Arguments:	x = INTEGER
Errors:	NONE

ICHAR(x)

Returns the corresponding integer in the ASCII or EBCDIC character set. See Appendix C for the ASCII and EBCDIC character sets.

Returns:	INTEGER
Arguments:	x = STRING * 1
Errors:	NONE

IDIFF(x,y)

Returns the positive difference, i.e., the absolute value of x-y.

Returns:	INTEGER
Arguments:	x = INTEGER
	y = INTEGER
Errors:	NONE

IMAX(x_1,...,x_{10})

Returns the maximum of a list of up to 10 integers.

Returns:	INTEGER
Arguments:	x_n = INTEGER
Errors:	NONE

IMIN(x_1,...,x_{10})

Returns the minimum of a list of up to 10 integers.

Returns:	INTEGER
Arguments:	x_n = INTEGER
Errors:	NONE

IMOD(x,y)

Returns the result of x modulo y.

Returns:	INTEGER
Arguments:	x = INTEGER
	y = INTEGER
Errors:	NONE

INDEX(s1,s2)

Returns the starting position of string 's2' in string 's1'. Zero will be returned if the string is not found.

Returns:	INTEGER
Arguments:	s1 = STRING
	s2 = STRING
Errors:	NONE

IPLANINFO(s)

Returns the IPLAN or application version/release number. The integer portion is the version number; the decimal portion is the release number. Returns zero for an invalid argument.

Returns:	REAL
Arguments:	s = string
	'compiler'
	'simulator'
	'application'
Errors:	NONE

IPLANINFS(s)

Returns IPLAN or application release number or application name. Release number is of the form "major.minor.mod"..

Returns:	STRING
Arguments:	s = string 'compiler' 'simulator' 'application' 'appname'
Errors:	NONE

ISIGN(x)

Returns -1,1 based on sign of x.

Returns:	INTEGER
Arguments:	x = INTEGER
Errors:	NONE

JUSTLEFT(s)

Left justifies a string.

Returns:	STRING
Arguments:	s = STRING
Errors:	NONE

JUSTRIGHT(s)

Right justifies a string.

Returns:	STRING
Arguments:	s = STRING
Errors:	NONE

LEFT(x,y)

Returns the left-most 'y' characters of x.

Returns:	STRING
Arguments:	x = STRING y = INTEGER
Errors:	NONE

LEN(s)

Returns the operational length of the string 's'. Trailing spaces will be ignored.

Returns:	INTEGER
Arguments:	x = STRING
Errors:	NONE

LENSTR(s)

Returns declared length of the string 's'.

Returns:	INTEGER
Arguments:	x = STRING
Errors:	NONE

LOG(x)

Returns the natural logarithm of x.

Returns:	REAL
Arguments:	x = REAL
Errors:	$x \leq 0$

LOG10(x)

Returns the logarithm base 10 of x.

Returns:	REAL
Arguments:	x = REAL
Errors:	$x < 0$

MAX(x_1,...,x_{10})

Returns the maximum value of a list of up to ten values. Note that all ten values must be of the same type.

Returns:	INTEGER/REAL
Arguments:	x_n = INTEGER or REAL
Errors:	NONE

MIN(x_1,...,x_{10})

Returns the minimum value of a list of up to ten values. Note that all ten values must be of the same type.

Returns:	INTEGER/REAL
Arguments:	x_n = INTEGER or REAL
Errors:	NONE

MOD(x,y)

Returns x modulo y. Note that x and y must be of the same type.

Returns:	INTEGER/REAL
Arguments:	x = INTEGER or REAL
	y = INTEGER or REAL
Errors:	NONE

NINT(x)

Returns the nearest integer to x.

Returns:	INTEGER
Arguments:	x = REAL
Errors:	NONE

PROMPT(s,i)

Issues string s as a prompt. i is an options field where 1 requests suppression of CR/LF and 10 requests echoing to progress area. The options can be added together. The value of the function is an error flag, zero if no error.

This function is considered obsolete. Use the ASK and ASKx commands.

Returns:	INTEGER
Arguments:	s = STRING
	i = INTEGER
Errors:	NONE

PROMPTC

Clears all prompts. Value of function is the return code, zero if no error.

Returns:	INTEGER
Arguments:	NONE
Errors:	NONE

PROMPTI

Returns number of prompt buffers pending.

Returns:	INTEGER
Arguments:	NONE
Errors:	NONE

PROMPTR(i)

Retrieve prompt #i as value of function.

Returns:	STRING
Arguments:	i = INTEGER
Errors:	NONE. If i is invalid, blanks are returned.

RABS(x)

Calculates the absolute value of x.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

RDIFF(x,y)

Returns the positive difference, i.e., the absolute value of x-y.

Returns:	REAL
Arguments:	x = REAL
	y = REAL
Errors:	NONE

REPEAT(x)

Initializes string with repeated occurrence of string x.

Returns:	STRING
Arguments:	x = STRING
Errors:	NONE

RIGHT(x,y)

Returns the right-most 'y' characters of x.

Returns:	STRING
Arguments:	x = STRING
	y = INTEGER
Errors:	NONE

RMAX(x_1,...,x_{10})

Returns the maximum of a list of up to 10 reals.

Returns:	REAL
Arguments:	x_n = REAL
Errors:	NONE

RMIN(x_1,...,x_{10})

Returns the minimum of a list of up to 10 reals.

Returns:	REAL
Arguments:	x_n = REAL
Errors:	NONE

RMOD(x,y)

Returns x modulo y.

Returns:	REAL
Arguments:	x = REAL
	y = REAL
Errors:	NONE

RND(x)

Return a random number or seed the random number generator.

Returns:	INTEGER
Arguments:	x = INTEGER
	x < 0, then: seed generator with (-x)
	x = 0, then: seed generator with default value
	x > 0, then: return a random number less than or equal to x
Errors:	NONE

ROUND(x,y)

Rounds x to y decimal places. y is optional and assumed to be zero.

Returns:	REAL
Arguments:	x = REAL
	y = optional INTEGER
Errors:	NONE

RSIGN(x)

Returns -1.0 or 1.0 depending on the sign of x.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

SCAN(string,set,direction)

Returns index of first character in string that is in set. String is examined from left to right if direction is false, and from right to left if direction is true.

Returns:	INTEGER
Arguments:	string = STRING
	set = STRING
	direction = LOGICAL
Errors:	Returns zero if no character found.

SCREEN(s,s1)

Passes screen command "beep" and "clear" to the terminal.

Returns:	INTEGER
Arguments:	s = STRING = "BEEP" or "CLEAR"
	s1 = STRING = string that contains the sequence to clear your screen. Leave blank to get a carriage return, line feed.
Errors:	NONE

SETDIR(dirname)

Sets the name of the current working directory, returns an error code.

Returns:	INTEGER
Arguments:	dirname = STRING
Errors:	Returns zero for no error, -1 for any error.

SIGN(x)

Returns -1 or 1 depending on the sign of x.

Returns:	INTEGER/REAL
Arguments:	x = INTEGER or REAL
Errors:	NONE

SIN(x)

Returns the sine of x where x is in radians.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

SINH(x)

Returns the hyperbolic sine of x where x is in radians.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

SPOOL(file, form, queue, options, copies)

Spools file to queue with form, options, and number of copies as specified. Returns an error code.

Returns:	INTEGER
Arguments:	File = STRING = name of file
	Form = STRING = specify printer form, or blank
	Queue = STRING = printer queue name
	Options = STRING = ’F’, ’L’, ’FD’, ’LD’, blank
	'F' - file was written with Fortran carriage control
	'L' - file does not contain carriage control characters
	'D' - delete file after spooling
	Copies = INTEGER
Errors:	Error Code > 0

SQRT(x)

Returns the square root of x.

Returns:	REAL
Arguments:	x = REAL
Errors:	x < 0

SUB(x,y[,z])

Returns a portion of the original string x. The substring will start at character position y. If z is not specified, then the remainder of the string will be returned. If z is specified, then z characters will be returned.

Returns:	STRING
Arguments:	x = STRING
	y = INTEGER
	z = INTEGER
Errors:	NONE

SUBSTR(x,y[,z])

Searches for string y in string x. If found, then the substring is replaced with string z. If string z is not specified, then string y is simply removed from string x.

Returns:	STRING
Arguments:	x = STRING
	y = STRING
	z = STRING
Errors:	NONE

TAN(x)

Returns the tangent of x where x is in radians.

Returns:	REAL
Arguments:	x = REAL
Errors:	x = PI/2

TANH(x)

Returns the hyperbolic tangent of x where x is in radians.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

TOINT(x)

Converts a variable of any type to an integer. Reals are truncated. Logicals are 1 if true, else zero. Strings are parsed as free-format input; the first nonblank character up to any trailing blanks or commas is interpreted as an integer. Invalid values return zero.

Returns:	INTEGER
Arguments:	x = ANY
Errors:	NONE

TOLOG(x)

Converts a variable of any type to a logical. Strings are converted to FALSE unless equal to the word TRUE. Numeric values of zero are returned as FALSE, otherwise as TRUE.

Returns:	LOGICAL
Arguments:	x = ANY
Errors:	NONE

TOREAL(x)

Converts variable of any type to real. Logicals are 1 if true, else zero. Strings are parsed as free-format input; the first nonblank character up to any trailing blanks or commas is interpreted as a real. Invalid values return zero.

Returns:	REAL
Arguments:	x = ANY
Errors:	NONE

TOSTR(x)

Converts variable of any type to a string. Numeric values are converted to displayable characters (e.g., a 3 becomes ’3’). Logicals become the string TRUE or the string FALSE.

Returns:	STRING
Arguments:	x = ANY
Errors:	NONE

TRUNC(x)

Returns the number x with the decimal portion truncated.

Returns:	REAL
Arguments:	x = REAL
Errors:	NONE

VERIFY(string,set,direction)

Returns index of first character in string that is not in set. String is examined from left to right if direction is false, and from right to left if direction is true.

Returns:	INTEGER
Arguments:	string = STRING
	set = STRING
	direction = LOGICAL
Errors:	Returns zero if all characters in string are in set.

SIEMENS

Chapter 6

Program Limitations

The following limitations exist in IPLAN programs.

Source line size:	256 characters
Source file size:	10,000 lines
Symbol names:	10,000 symbols
Length of a single symbol name:	80 characters
Total length of all symbol names:	120,000 characters
Maximum nesting of INCLUDE files:	11
Maximum nesting of control structures (IF, LOOP, etc.):	100
Procedure calling depth:	400
Maximum IPLAN internal addresses (for control structures):	4,000
Maximum declared string length:	300 characters
Maximum length of a string literal:	200 characters
Maximum number of files (for OPEN):	15
Maximum file count (for FNDFIL, FNDNXT):	1,000
Maximum number of elements in an iolist:	100
Input buffer length:	256 characters
Maximum output buffer length:	1024 characters
Buffer length of BUFxxx functions:	300 characters
Prompt length limit	256 characters
Pause Until string length	40 characters

There are no fixed limits to the number of variables or the size of the compiled program. However, there will be actual limits imposed by the computer system based on available virtual storage. We cannot predict what these limits will be.

Symbol names include all variable, label, function, subroutine and procedure names. Note that a real array, for example, dimensioned by 100, uses 100 real variables but only one symbol name.

Table 6-1 Program Element Usage for IPLAN Statements gives the program element usage of each IPLAN statement part. For example, the following statement:

```
IF (I>1) THEN PRINT 'ABC' ELSE PRINT 'DEF' ENDIF
```

requires 24 memory elements to store the IF/ELSE, 2 PRINTs, 2 formats, 1 variable reference, 1 integer reference, 2 string references, and 1 relational operator.

Table 6-1. Program Element Usage for IPLAN Statements

Simulator Operators	
Statement	**Usage**
BEEP	1
CALL	3
CLOSE	1
DELETE	1
INPUT	2
INPUTLN	1
NEWREPORT	1
ON	4
OPEN	1
OPTION	0
PAUSE	1
PAUSE READ	1
PAUSE UNTIL	2
PAUSE WHILE	2
PRINT	2
PRINTX	2
PUSH	2
PUSHX	2
QPUSH	2
QPUSHX	2
READ	2
READLN	1
REWIND	1
SET	2
SLEEP	1
STOP	1
TRACE	1
WRITE	2
WRITEX	2

Control Structures	
Statement	**Usage**
CONDITIONAL	4/condition
END	1
ENDLOOP	2
EXITCOND	2
EXITLOOP	2
GOTO	2
IF	2
IF/ELSE	4
LOOP	4
NEXTCOND	2
NEXTLOOP	2
PERFORM	2
RETURN	1

References and Data	
Statement	**Usage**
Format	3
Expression operator	1
Relational operator	1
Variable reference	3
Logical constant reference	1
Other constant reference	2

SIEMENS

Chapter 7

User-Defined Routines

7.1 Usage

User-defined routines are referenced in IPLAN programs via the CALL statement. See Section 3.29 User-Defined Routines.

7.2 Source Language

It is recommended that the same language and compiler be used that is used for IPLAN. This will vary by computer platform and possibly by the Siemens PTI program product that IPLAN is supplied with. It will, however, be a version of Fortran.

If a source language other than Fortran is selected, it will be the user's responsibility to be aware of and deal with any cross-language compatibility issues. In particular, since the calling program will be Fortran, the form of the program arguments must be understood.

The following sections deal with considerations for developing user-defined routines written in Fortran.

7.3 Program Arguments

The calling arguments for all of the user-defined routines are the same. They are an integer, a real, a character string, and another integer (in that order). The intent is to allow various types of data to be passed back and forth without conversion. The intent of the second integer is to allow simple return codes.

The user-defined routine, then, should begin as follows (for USREX1, for example):

```
SUBROUTINE USREX1(A,B,C,D)
INTEGER A,D
REAL    B
CHARACTER*(*) C
```

The variable names could be anything, of course. In this example, an assumed length is coded for the character variable. If the variable used in the IPLAN program is of a different length, it will be truncated or padded as appropriate when control from the user-defined routine returns.

7.4 Terminal I/O in User-Defined Routines

In general, within a user-defined routine, terminal I/O (PRINT, READ from standard input, or WRITE to standard output) should be avoided, especially within an application that is using a GUI. The issues involved are numerous and quite computer system dependent. On most systems, however, such I/O statements will work in some fashion.

7.5 File I/O in User-Defined Routines

With care, file I/O is unrestricted. The problem is in the selection of a Fortran unit number. It will not be possible to predict what unit numbers will be available. An IPLAN program using a user-defined routine could easily find a unit number unavailable that was available in a previous execution.

The recommended method is to use the IPLAN function GETLU to identify an available unit number and pass that number to the user-defined routine. Remember to use the IPLAN function FRELU to release that number when you have closed the file.

7.6 Compilation and Execution of User-Defined Routines

When IPLAN is installed, default user-defined routines exist that will assign zero or spaces to each of the program arguments. The procedure for compiling your source code and for updating the IPLUSR library (this is where the routines are stored) is computer system dependent and is therefore documented in the installation manual for your Siemens PTI program product. Depending on the computer system, it may also be necessary to reinstall IPLAN itself and the executables that use IPLAN.

SIEMENS

Chapter 8

Executing IPLAN Programs

IPLAN programs must be compiled in order to be run. The binary file produced by the IPLAN compiler is used as input to the IPLAN simulator. The binary file itself cannot be run on any machine, or used as input to a build (or bind or link) process to produce an executable file. The IPLAN compiler comes with the simulator built in, which can be accessed using the -RUN and -CRUN options (see Chapter 4 and Section Using with Stand-Alone Simulator). This is primarily a debugging tool, however, since the true power of IPLAN comes from its ability to communicate with a host application through the PUSH commands (Section 3.25 Communication With Application Program) and the routines in Appendix A. The host program will have the IPLAN simulator built into it and provide a means to load a compiled IPLAN program.

8.1 How IPLAN Interacts With the Host Application

When the simulator in the host application has a loaded, active IPLAN program, it will intercept terminal read requests and instead begin execution of the IPLAN program (an IPLAN program can be active, or paused by executing a PAUSE command, or terminated by executing a STOP command, reaching the END of the main procedure, or being interrupted). IPLAN programs interrogate the host application via the routines in Appendix A, and control the host application via the PUSH commands. The values entered for the PUSH commands appear to the host application as if they were entered from the terminal.

8.2 Interrupting IPLAN Programs

Program interruption is a system-dependent feature, so the method for initiating an interrupt is not discussed here (the host application should document how programs are interrupted). Only the "IP" control code is defined for IPLAN. If an "IP" code is entered, it instructs the simulator to terminate the IPLAN program. The IPLAN simulator will ignore any other codes that it finds.

Program interruption is an asynchronous event, therefore, it is not possible to stop the program precisely at a particular point. The IPLAN simulator is internally structured in such a way as to permit interruptions to be effective only in between operation codes. Some IPLAN statements are simple and only generate a single operation code (e.g., STOP). Most statements are more complex and generate several operation codes (e.g., PRINT I,J would generate five operation codes – one for the PRINT, one each for each variable reference, and one format code for each variable).

It is important to realize that the "IP" code will not have an effect on the application program. If a running program is interrupted and the "IP" code set and after the execution of a PUSH command, but before the resumption of the IPLAN program, the application program will not stop, but the IPLAN program will stop before it can resume.

The SET INTERRUPT command (see Section 3.26.1 Transmitting to the Application on Termination) can be used to automatically PUSH a value to the application program when an IPLAN program stops due to user interruption.

8.3 Specifying Arguments

Unfortunately, it is not possible to completely define how this is done. This is because the IPLAN simulator will be incorporated as part of another application, and how that application's interface allows IPLAN programs to be specified is independent of IPLAN itself.

The IPLAN simulator expects to be given a single string as instructions to load an IPLAN program. That string will contain the name of the compiled IPLAN program, and, optionally, an argument string. The argument string can be in one of two forms. Either the four-character string "-arg", or anything else. Please note that the case of the "-arg" is not important (i.e., these strings are equivalent: "-arg", "-ARG", "-aRg"). If the string is not "-arg", then the remainder of the string following the filename is considered the argument string. If the string is "-arg", then the IPLAN simulator will interpret this as an instruction to issue a prompt for the argument string.

For example, let's say that the application interface is command driven, and that the command to load an IPLAN program is EXEC. The format of the command is "EXEC <filename> <arguments>" and everything following EXEC is passed as is to the IPLAN simulator. Now consider this IPLAN program:

```
Program t
String a,b,c
Argument 0,a,b,c
Print 'a=',a,', b=',b,', c=',c
End
```

If this is entered:	Then this is displayed:	Then enter:
EXEC myfile.irf what's this stuff	a=what's, b=this, c=stuff	
EXEC myfile.irf "what's this stuff"	a=what's this stuff, b=, c=	
EXEC myfile.irf what is this stuff	a=what, b=is, c=this	
EXEC myfile.irf what, 'is this',, stuff	a=what, b=is this, c=	
EXEC myfile.irf –arg	Enter argument string for IPLAN program:	here you go
	a=here, b=you, c=go	

Note that these examples assume that a space is a valid delimiter for a filename. This is not always true. For systems where a space may be part of a filename, the filename will have to be delimited appropriately. For example: EXEC "myfile.irf" an argument string

The rules for entering arguments for the simulator to the compiler are slightly different because other compiler options can be specified on the same line (see Section 4.3.7 ARG/NOARG Options).

8.4 Parsing

8.4.1 Definition

Parsing is the action of taking a sequence of characters and identifying its parts. The criteria used will vary depending on the application. For example, the English reader will parse this sentence into words, using spaces as delimiters, and will also identify punctuation and capitalization. A delimiter is something that separates one part from another.

8.4.2 Application

The concepts here apply to the IPLAN commands READ, INPUT, ARGUMENT, the IPLAN parser functions (all named starting with "GT"), and the IPLAN compiler command line. Each of these uses an independent buffer, even though the same parser is used, and will not interfere with the results of any other.

8.4.3 Position and Delimiters

The parser is initially positioned at the first character in the buffer. Whenever a string of characters is identified and returned the parser will positioned at the first nonblank character following the delimiters that follow the string that was returned. For example, if the string

```
abc def
```

is in the buffer (where a is the first character in the string) the parser will be initially positioned at the "a", and then, after "abc" is returned, positioned at the "d".

A single delimiter is a space or a comma, optionally preceded or followed by any number of spaces. For example, if the string above had been

```
abc     ,       def
```

the results would be the same.

If a parsing error occurs, the parser will position itself at the end of the buffer.

GTCHAR is a special case. It returns the next character wherever the parser happens to be positioned, and leaves the parser positioned at the following character, regardless of what values either character may have. This can produce surprising results occasionally. For example, if the buffer contains

```
a,bcd
```

calls to GTCHAR, GTWORD, GTWORD would return "a", " ", and "bcd", respectively.

The parser also recognizes a logical end-of-line character. If the parser encounters the character "/" (slash) that is not part of a filename or a quoted string it will behave as if it had reached the end of the buffer.

8.4.4 Considerations for Specific Data Types

Characters	Applies to function GTWORD, commands READ, INPUT, and ARGUMENT, with STRING variables, and to command line options that are not filenames. Generally will pick up any character between delimiters. If a character string (string) contains a blank, a comma, or a slash, the string must be quoted. Either apostrophes or quotes may for used as the quote character. When a quoted string is read, it is returned without the quote characters. Consecutive quote characters within a quoted string will be interpreted as a single embedded quote character. Quote characters in a nonquoted string, or embedded in a string quoted using the other quote character, will not receive special treatment.
Integers	Applies to function GTINT, and to commands READ, INPUT, and ARGUMENT, with INTEGER variables. May only contain the digits 0 through 9, optionally preceded by a sign, + or –. Integers may have a range of 2147483648 to –2147483649.
Reals	Applies to function GTREAL, and to commands READ, INPUT, and ARGUMENT, with REAL variables. A real number is a number with either a decimal point, or with an exponent specified. Valid characters for a real number include the digits 0 through 9, and the characters +, –, ., E, and D.
Logicals	Applies to commands READ, INPUT, and ARGUMENT with LOGICAL variables. A character string is read, according to the rules above. A case-insensitive comparison is made to the string TRUE. If it matches, the value is recognized as TRUE; otherwise the value is considered to be FALSE.
Filenames	Applies to command line values that are expected to be filenames or pathnames. Filenames are generally treated like character strings. Exceptions are where characters that would normally have special meaning to the parser can be part of a filename, such as a blank, a comma, a slash, or quotes. For filenames where a comma is a valid character, the parser still recognizes it as a delimiter if it is not quoted. For filenames where a space is a valid character, special attention must be given by the user if there is additional information following the filename; spaces will not be recognized as delimiters following such a filename (use a comma, or put the filename in quotes).

SIEMENS

Appendix A

PSS®E Application Notes

A.1 Invoking an IPLAN Program from Within PSS®E

A.1.1 Line Mode

An IPLAN program may be invoked from within PSS®E by issuing EXEC at the line mode activity selector. The syntax is:

```
ACTIVITY?  EXEC program_name
```

You must have compiled the program beforehand and no file extension is required. If the program (*.irf file) cannot be found in the local directory, the programs subdirectory is searched.

To specify arguments to the IPLAN program, enter:

```
ACTIVITY?  EXEC program_name argument_string
```

or:

```
ACTIVITY?  EXEC program_name -ARG
```

In the latter case, you will be asked to enter the arguments on a separate line.

A.1.2 GUI

From the PSS®E GUI select ***IO Control > Run Program Automation FIle***. Select ***Files of Type: IPLAN File (*.irf)*** and navigate to the file. Use *Arguments* button at lower left of screen to specify arguments.

A.1.3 Python

Use the API routine *runiplanfile*. Arguments are specified on the same line, as for activity EXEC.

A.2 PSS®E Subroutines: Functional Summary

Table A-1. PSS®E Subroutines

Name	Data Retrieved
Miscellaneous:	
SELCTR	Returns activity selector indicator.
OPENUN	Returns the logical unit set by activity OPEN.
SFILES	Returns current saved case and snapshot file names.
BSYSISDEF	Is bus subsystem number defined
MDLFLDCNT	Returns number of folders in the list to be searched for dynamics data defined models.
MDLFOLDER	Returns the name of a folder in the list that is searched for dynamics data defined models.
MDLLIBCNT	Returns number of libraries in the list to be searched for dynamics library models.
Case Data:	
TOTBUS	Returns the total number of buses in the working case including three-winding transformer "star" point buses.
TITLDT	Returns the two-line case title.
SOLVED	Checks if last solution attempt reached tolerance.
SYSMVA	Returns system base MVA.
SYSTOT	Returns the system total load, generation, or losses (P+jQ).
SYSMSM	Returns total system MVA mismatch.
CHKTRE	Checks for Type 4 (or greater) buses with in-service branches and counts number of islands not containing a Type 3 (swing) bus.
ISLAND	Trips in-service branches connected to Type 4 (or greater) buses and disconnected islands not containing a swing bus.
GENCNV	Checks if generators are "converted."
ITERAT	Returns number of iterations used in last solution attempt.
MAXMSM	Returns largest MVA mismatch.
Bus Data:	
INIBUS	Initializes the bus fetching routine 'NXTBUS' for retrieving buses in ascending numerical order. Dummy buses for multisection lines are included. Hidden "star" point buses of three-winding transformers are excluded.
INIBUX	Initializes the bus fetching routine 'NXTBUS' for retrieving buses in ascending numerical order. Dummy buses of multisection lines are excluded. Hidden "star" point buses of three-winding transformers are excluded.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
ININAM	Initializes the bus fetching routine 'NXTBUS' for retrieving buses in ascending alphabetical order. Dummy buses for multisection lines are included. Hidden "star" point buses of three-winding transformers are excluded.
ININAX	Initializes the bus fetching routine 'NXTBUS' for retrieving buses in ascending alphabetical order. Dummy buses of multisection lines are excluded. Hidden "star" point buses of three-winding transformers are excluded.
STAREA	Restricts 'NXTBUS' to a single area.
STZONE	Restricts 'NXTBUS' to a single zone.
STBSKV	Restricts 'NXTBUS' to a single base kV level.
NXTBUS	Returns the next bus in ordered sequence. One of the routines INIBUS, INIBUX, ININAM or ININAX must be called first.
ORDBUS	Returns the bus number corresponding to the specified position in the ordered list (numeric or alphabetic) of buses.
BUSEXS	Tests for the existence of a specified bus.
NOTONA	Returns the 18-character extended bus name for a specified bus number.
NATONO	Returns the bus number for a specified 18-character extended bus name.
BUSDAT	Returns real bus values.
BUSDT1	Returns absolute value of complex bus values.
BUSDT2	Returns complex bus values.
BUSINT	Returns integer bus values.
BUSMSM	Returns bus mismatch.
Load Data:	
INIIND	Initializes load fetching routine 'NXTLOD' for returning loads attached to a bus in ascending order by load ID..
NXTIND	Returns the next load connected to a bus. 'INILOD' must be called first.
LODDT1	Returns absolute value of complex load values.
LODDT2	Returns complex load values.
LODINT	Returns integer load values.
LODIND	Returns load array index.
Fixed Shunt Data:	
INIFXS	Initializes the fixed shunt fetching routine "NXTFXS" for returning fixed shunts at a bus.
NXTFXS	Returns the next fixed shunt. The routine INIFXS must be called first.
FXSDT1	Returns absolute value of complex fixed shunt values.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
FXSDT2	Returns complex fixed shunt values.
FXSINT	Returns integer fixed shunt values.
Generator Data:	
INIMAC	Initializes the machine fetching routine 'NXTMAC' for returning machines connected to a bus in ascending order by machine ID.
NXTMAC	Returns the next machine connected to a bus. 'INIMAC' must be called first
MACDAT	Returns real machine values.
MACDT2	Returns complex machine values.
MACINT	Returns integer machine values.
MACIND	Returns machine array index.
GENDAT	Returns plant total power output as complex value.
GENDT1	Returns plant total power output as MVA.
Branch Data:	
INIBRN	Initializes the branch fetching routines 'NXTBRN' or 'NXTBRN3' for returning branches connected to IBUS. For multisection lines, the branch fetching routine will return adjacent dummy bus as the TO bus.
INIBRX	Initializes the branch fetching routines 'NXTBRN' or 'NXTBRN3' for returning branches connected to IBUS. For multisection lines, the branch fetching routine will return the other endpoint bus as to TO bus.
NXTBRN	Returns the next branch connected to a bus, excluding three-winding transformers. INIBRN or INIBRX must be called first.
NXTBRN3	Returns the next branch connected to a bus, including three-winding transformers. INIBRN or INIBRX must be called first.
BRNDAT	Returns real branch values.
BRNDT2	Returns complex branch values.
BRNINT	Returns integer branch values.
BRNFLO	Returns the branch flow (P+jQ) as calculated at IBUS.
BRNMSC	Returns branch flow values.
Transformer Data:	
XFRNAM	Returns transformer name.
XFRDAT	Returns real transformer values.
XFRINT	Returns integer transformer values.
TR3NAM	Returns three-winding transformer name.
TR3DAT	Returns three-winding transformer real values.
TR3DT2	Returns three-winding transformer complex values.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
TR3INT	Returns three-winding transformer integer values.
WNDDAT	Returns real values for a three-winding transformer winding.
WNDDT2	Returns complex values for a three-winding transformer winding.
WNDINT	Returns integer values for a three-winding transformer winding.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
Area Interchange Data:	
ARDAT	Returns real area values.
AREINT	Returns integer area values.
ARENAM	Returns area name.
ARENUM	Returns area number.
AREUSE	Indicates if area is in use.
AREDAT	Returns complex area values.
ARITOJ	Returns the interchange between two areas.
INITIE	Initializes the tie branch fetching routines 'NXTTIE' or 'NXTTIE3' for returning branches in area IAR. For multisection lines dummy buses are included in the processing of the tie branch fetching routine and for multisection lines connected to a FROM area bus, the adjacent dummy bus is treated as the TO bus.
INITIX	Initializes the tie branch fetching routines 'NXTTIE' or 'NXTTIE3' for returning branches in area IAR. For multisection lines dummy buses are ignored in the processing of the tie branch fetching routine and for multisection lines connected to a FROM area bus, the other endpoint bus is treated as the TO bus.
NXTTIE	Returns the next tie branch from an area, excluding three-winding transformers. 'INITIE' or 'INITIX' must be called first.
NXTTIE3	Returns the next tie branch from an area, including three-winding transformers. 'INITIE' or 'INITIX' must be called first.
Two-Terminal dc Data:	
DC2INT_2	Returns integer two-terminal dc line values.
DC2DAT_2	Returns real two-terminal dc line values for the specified bus.
INI2DC	Initializes the two terminal dc line fetching routine 'NXT2DC' for returning dc lines in ascending order by dc line name.
NXT2DC	Returns the next two terminal dc line. The routine 'INI2DC' must be called first.
VSC dc Data:	
VSCINT	Returns integer VSC dc line values.
VSCCIN	Returns real VSC dc line converter values.
VSCCDT	Returns real VSC dc line values.
INIVSC	Initializes the VSC dc line fetching routine 'NXTVSC' for returning dc lines in ascending order by dc line name.
NXTVSC	Returns the next VSC dc line. The routine 'INIVSC' must be called first.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
Multiterminal dc Data:	
DCNINT_2	Returns integer multiterminal dc line values.
DCNDAT_2	Returns real multiterminal dc line values.
DCNCIN_2	Returns integer multiterminal dc line converter values.
INIMDC	Initializes the multiterminal dc line fetching routine 'NXTMDC' for returning dc lines in ascending order by dc line name.
NXTMDC	Returns the next multiterminal dc line. The routine 'INIMDC' must be called first.
Multi-Section Line Data:	
INIMSL	Initializes the multi-section line member fetching routine 'NXTMSL'.
NXTMSL	Returns next multi-section line member branch. 'INIMSL' must be called first.
Zone Data:	
ZONNAM	Returns the zone name.
ZONNUM	Returns the zone number.
ZONUSE	Indicates if zone is in use.
ZNDAT	Returns complex values for the desired zone.
ZNITOJ	Returns the interchange between two zones.
Interarea Transfer Data:	
TRXDAT	Returns the interarea transfer MW.
Owner Data:	
OWNNAM	Returns owner name.
OWNNUM	Returns owner number.
OWNDAT	Returns data associated with an owner.
OWNUSE	Indicates if an area is in use.
FACTS Device Data:	
FCDINT_2	Returns FACTS device integer values.
FCDDAT_2	Returns FACTS device real values.
INIFAX	Initializes the FACTS device fetching routine 'NXTFAX' for returning FACTS devices in ascending order by FACTS device name.
NXTFAX	Returns the next FACTS device. The routine 'INIFAX' must be called first.
Switched Shunt Data:	
SWSDT1	Returns switched shunt real values.
SWSINT	Returns switched shunt integer values.
SWSBLK	Returns the switched shunt number of steps and step size for block IBLK.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
SWSBLZ	Returns the switched shunt number of steps and zero sequence step size for block IBLK.
Dynamics Data:	
DSIVAL	Returns dynamics integer array values.
DSRVAL	Returns dynamics real array values.
DSCVAL	Returns dynamics character array values.
CHNVAL	Returns output channel value.
OKSTRT	Indicates errors after activities STRT or MSTR.
GET_MSTATE	Returns the current MSTATE value.
Dynamics Model Data:	
MDLIND	Returns plant-related model starting array indices and status.
MDLNAM	Returns plant-related model name.
LMODIND	Returns load model starting array indices and status.
LMODNAM	Returns load model name.
SLMODIND	Returns starting array indices and status for load models specified by subsystem.
SLMODNAM	Returns model name for load models specified by subsystem.
RMODIND	Returns relay model starting array indices and status.
RMODNAM	Returns relay model name.
DC2MIND	Returns two-terminal dc line model starting array indices and status.
DCNMIND	Returns multitermianl dc line model starting array indices and status.
FCDMIND	Returns FACTS device model starting array indices and status.
SWSMIND	Returns switched shunt model starting array indices and status.
VSCMIND	Returns VSC dc line model starting array indices and status.
WINDMIND	Returns wind model model starting array indices and status.
DC2AUXMIND	Returns the starting array indices and status of auxiliary signal model associated with two-terminal dc line for the specified signal index.
DCNAUXMIND	Returns the starting array indices and status of auxiliary signal model associated with multiterminal dc line for the specified signal index.
FCDAUXMIND	Returns the starting array indices and status of auxiliary signal model associated with FACTS device for the specified signal index.
VSCAUXMIND	Returns the starting array indices and status of auxiliary signal model associated with VSC dc line for the specified signal index.
DC2MNAM	Returns two-terminal dc line model name.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
DCNMNAM	Returns multiterminal dc line model name.
FCDMNAM	Returns FACTS device model name.
SWSMNAM	Returns switched shunt model name.
VSCMNAM	Returns VSC dc line model name.
WINDMNAM	Returns wind model name.
DC2AUXMNAM	Returns the auxiliary signal model name associated with a two-terminal dc line for the specified auxiliary signal index.
DCNAUXMNAM	Returns the auxiliary signal model name associated with a multiterminal dc line for the specified auxiliary signal index.
FCDAUXMNAM	Returns the auxiliary signal model name associated with a FACTS device for the specified auxiliary signal index.
VSCAUXMNAM	Returns the auxiliary signal model name associated with a VSC dc line for the specified auxiliary signal index.
Fault Analysis Data:	
SCINIT	Initializes the short circuit IPLAN routines.
SCBUS2	Returns sequence voltages.
SCMAC2	Returns machine short circuit currents.
SCBRN2	Returns branch sequence currents.
SCDONE	Returns the working case after 'SCINIT'.
SC3WND	Returns three-winding transformer short circuit currents.
Obsolete Routines:	
BRNCUR	Returns the branch current flow in amps as calculated at IBUS.
BRNMVA	Returns the branch MVA flow calculated at IBUS.
BRNSTT	Returns the branch status value.
LODCNV	Checks if loads are "converted."
MACSTT	Returns machine status value.
SWSDAT	Returns switched shunt schedule voltage band.
TRNDAT	Returns transformer ratio and phase shift angle.
DC2DAT	Returns real two-terminal dc line values for the specified bus.
DC2INT	Returns integer two-terminal dc line values.
DCNCIN	Returns integer multiterminal dc line converter values.
DCNDAT	Returns real multiterminal dc line values.
DCNINT	Returns integer multiterminal dc line values.
FCDDAT_2	Returns FACTS device real values.

Table A-1. PSS®E Subroutines (Cont.)

Name	Data Retrieved
FCDINT_2	Returns FACTS device integer values.

A.3 IPLAN and Response Files

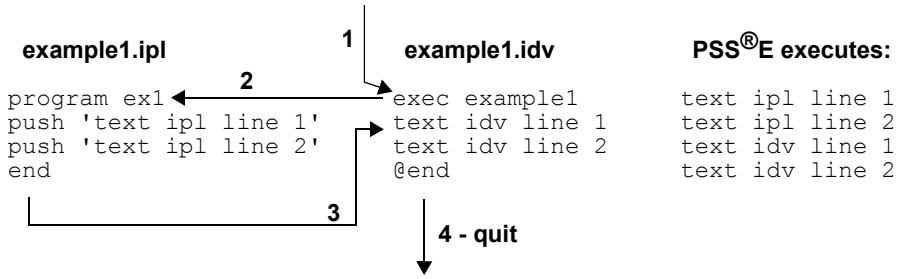
Note that the PAUSE READ command added at Release 12 of IPLAN provides the funtionality that had previously generated most of the interest in the material presented in this section. All the following information is still accurate, however.

PSS®E provides two methods of input automation, IPLAN (activity EXEC) and Response Files (See *PSS®E Program Operation Manual*, Section 16.12, Running a Response File and activities PSEB and PSAS, and immediate commands @INPUT and @CHAIN). What happens when they are used together?

Following are several examples, along with very simple explanations. A more detailed overview of the interaction of IPLAN and Response Files is presented in Section A.3.1 How IPLAN and Response Files Control PSS®E Execution.

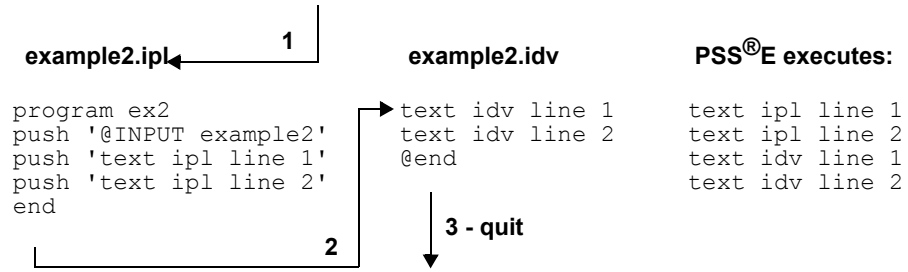
Each example consists of an @INPUT (or @CHAIN) or EXEC command, which is shown on the first line, followed by any additional input required. Then three columns listing the IPLAN program used in the example (in the first column), all Response Files used (in the second column), and the commands that PSS®E executes as a result (in the third column). The third column could be viewed as an equivalent Response File.

1. Command executed: @INPUT example1



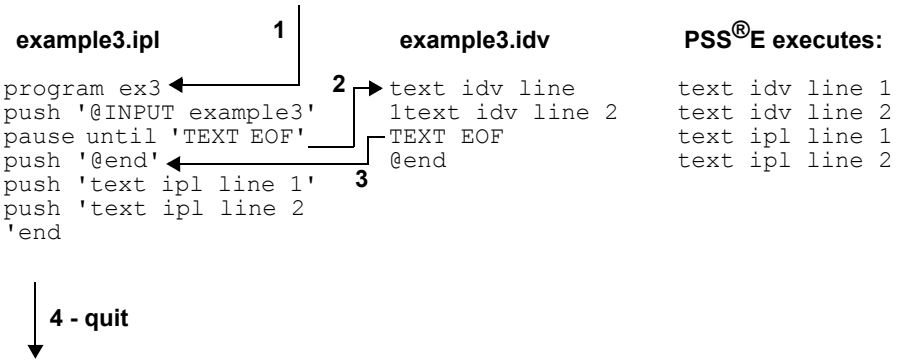
Summary. The Response File invokes IPLAN, IPLAN pushes two TEXT commands, then terminates, and then the Response File continues, supplying two TEXT commands.

2. Command executed: exec example2



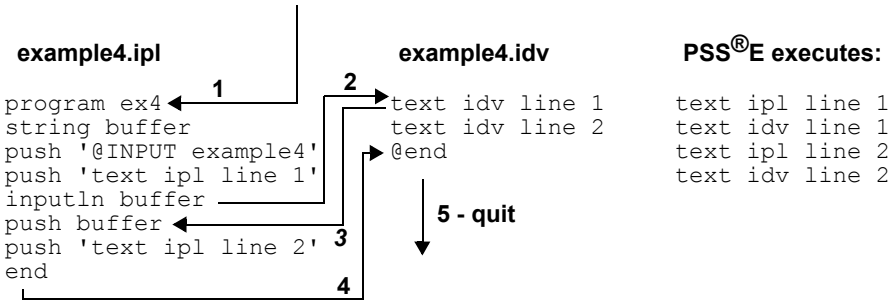
Summary. The IPLAN program opens a Response File, pushes two TEXT commands, then terminates, at which point input returns to the terminal, which has been redirected to a Response File, which supplies two TEXT commands.

3. Command executed: exec example3



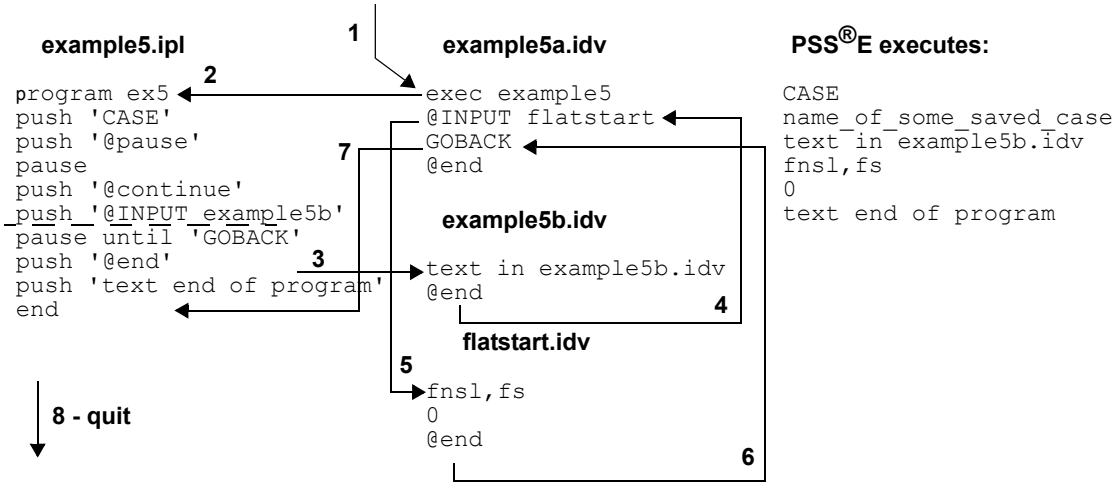
Summary. The IPLAN program opens a Response File, then pauses, returning input to the terminal, which has been redirected to a Response File. The Response File supplies two TEXT commands, then the string 'TEXT EOF' which is recognized as the pause termination string.The IPLAN program resumes, pushes '@end' to close the Response File, pushes two TEXT commands, and then terminates. Note that if the "push '@end'" line had not been present, then, when the IPLAN program terminated, the next line of the Response File would have been read. That line contains '@end', which would have closed the Response File, and the results would have been the same. If you try this exact example, it may appear that 'TEXT EOF' executes; it is only echoed. TEXT EOF was selected for this example to demonstrate a technique that allows a single Response File to be used in this way by IPLAN while strictly containing valid activities.

4. Command executed: exec example4



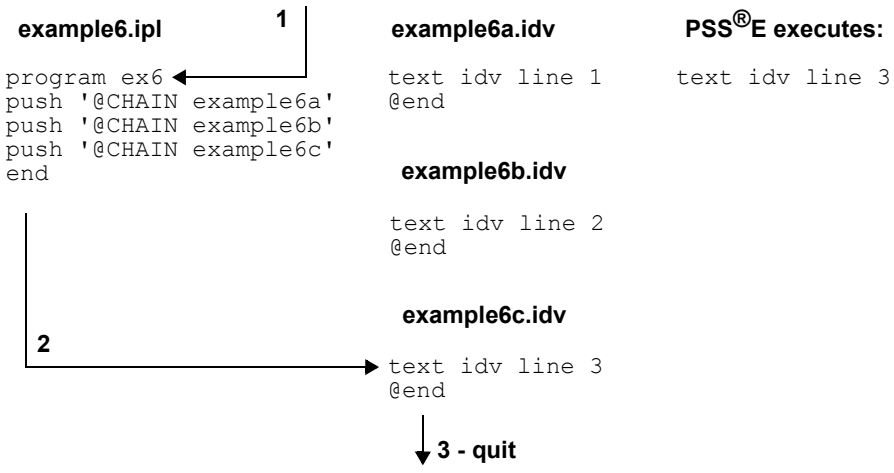
Summary. The IPLAN program opens a Response File, pushes a TEXT command, then issues a terminal input request. Since terminal input has been redirected to a Response File, the line 'text idv line 1' is read from that file. IPLAN then pushes that string, which contains a valid TEXT command, then pushes another TEXT command, and then terminates, returning input to the terminal, which has been redirected to a Response File. The Response File supplies another TEXT command.

5. Command executed: @INPUT example5a. Additional input: name_of_some_saved_case (indicated below by a dashed line)



Summary. The Response File invokes IPLAN, IPLAN pushes the command CASE, then pushes '@pause' which pauses the Response File. Then the IPLAN program pauses for the duration of one input request. The input request is directed to the terminal, where the user is requested to supply a case name (the prompts from PSS®E are displayed). After activity case terminates, the IPLAN program resumes. It reactivates the Response File, opens another Response File (nested, because @INPUT is used), then pauses. Input returns to the terminal, which is redirected to a Response File. The lowest level Response File is read, which supplies a text command, and is then closed due to a @end command. The higher level Response File then causes another Response File to be opened (again, nested). That Response File supplies a FNSL command, and another line of input to that activity, and then is closed (due to @end). The higher level Response File then supplies the string 'GOBACK', which is recognized as the pause termination string. The IPLAN program resumes, pushes '@end' to close the Response File, then pushes a TEXT command, then terminates.

6. Command executed: exec example6



Summary. The IPLAN program opens a Response File. It then opens another Response File, but, since @CHAIN is used, causes the lowest level Response File pre-

viously open to be closed. It then 'chains'* another Response File and terminates. Input returns to the terminal, which has been redirected to a Response File, which supplies one TEXT command. Notice that example61.idv and example62.idv are each opened and closed without ever being read.

A.3.1 How IPLAN and Response Files Control PSS®E Execution

IPLAN programs gain control by intercepting any input request by PSS®E. When an IPLAN PUSH** statement is executed, the value 'PUSH'ed is, to PSS®E, indistinguishable from the same value entered by the user at the terminal, or read from a Response File. It is important to note that no attempt to read from the terminal, or from a Response File, or from a windows dialog, ever takes place.

Response Files, on the other hand, replace the terminal as the source of data input. If running under windows, the act of opening a Response File will normally also disable the windows dialogs (which otherwise would prevent an attempt to read from the terminal). This is all IDEV, or @INPUT, or @CHAIN does; open a Response File, and, if needed, disable the windows dialogs. It does not 'execute' the file. What happens is that at the time of the next request for input, the Response File is read to satisfy that request. This is, unless an IPLAN program is active.

If an IPLAN program is active***, the read request never takes place, whether a Response File is open or not. Of course, IPLAN can issue its own read requests to the terminal via the INPUT and INPUTLN commands. If a Response File is open and active at the time that that read request takes place, it is the Response File that is read to satisfy the request.

IPLAN programs can be suspended via the commands PAUSE, PAUSE UNTIL, and PAUSE READ. During the time that that command is in effect, the read request reverts to being satisfied as if the IPLAN program had not been loaded (except that windows dialogs may be disabled).

A.3.2 Common Questions


In the following paragraphs, remember that @CHAIN and the IDEV activity are equivalent.

Why can't IPLAN execute more than one @CHAIN command?

Example 6 in Section A.3 IPLAN and Response Files above explains this.

When I push an @CHAIN command in an IPLAN program, it doesn't work. Then the Response File runs when the IPLAN program finishes. Why?

Example 2 in Section A.3 IPLAN and Response Files above explains this.

How do I use a Response File from inside my IPLAN program without stopping the IPLAN program?

There are three ways:

1. The simplest and, generally, the best way is to use the PAUSE READ command (see Section 3.12.3 PAUSE READ). When, say, an @CHAIN command is PUSHed, and followed by a PAUSE READ, PSS®E will execute all the commands in the Response File

* The term 'chain', in this context, means to connect in series. While it is possible to have a circular chain ('daisy chain'), a chained function replaces its predecessor and it is not possible to return along a chain.
** PUSH is used to indicate any of the commands PUSH, PUSHX, QPUSH, or QPUSHX.
*** An IPLAN program can be loaded (by activity EXEC) and not active if PAUSE'd.

and then continue with the IPLAN program at the statement following the PAUSE READ command.

2. As of Release 16, another available alternative is the PAUSE WHILE command (see Section 3.12.4 PAUSE WHILE). This is similar to PAUSE READ but distinguishes between response files that were already open and does not PAUSE for those.

3. An alternative that may sometimes be more appropriate (and the best way prior to Release 12.0) is to use the PAUSE UNTIL command (see Section 3.12.2 PAUSE UNTIL). This will also execute all the commands in the Response File, but then return control to the terminal input. To resume the IPLAN program at the statement following the PAUSE UNTIL command, it will be necessary to enter the UNTIL string.

4. Another way is to use the PUSH command (see Section 3.25 Communication With Application Program). In this case, you will need to write a loop, executing INPUTLN and PUSH until the end of the file. This is probably only useful if there is some particular reason you wish to examine the contents of the Response File before executing the commands.

A.4 Warning Concerning PSS®E Activity ECHO

Be careful using IPLAN programs that start Response Files (e.g., push @INPUT) with an ECHO file (PSS®E activity ECHO) enabled. If that echo file is then used as a Response File, it will contain, in addition to the EXEC command to run the IPLAN program, all the commands in the Response File that is started by that IPLAN program. In other words, all the commands in that Response File (the one started by the IPLAN program) will be executed twice!

A.5 Interaction of Prompts from IPLAN and PSS®E

IPLAN can be used for many reasons. For example, to produce a customized report, to run a sequence of activities, perhaps varying that sequence depending on data, or to implement an analysis of data not currently performed by the application (i.e., by a PSS®E activity). It might not seem obvious, then, that from the application's (i.e., PSS®E) perspective, the only thing IPLAN programs do is execute PUSH statements (that is slightly exaggerated; PAUSE and STOP are noticed in a way, and opening a file could be noticed indirectly). That perspective will be helpful in understanding what follows.

PSS®E operates by responding to a series of input values. If you have ever looked at a Response File, you have seen an example of such a series of input values. These might come from a person through a GUI or terminal interface, from a file, or from an IPLAN program. PSS®E doesn't know, and doesn't care. It prepares a set of questions, or prompts, describing the value it needs next and issues an input request. When input is coming from a terminal interface, for example, these prompts would be displayed on the terminal, the user would enter the response, the prompts would be erased (to prepare for the next input request), and the response would be returned to PSS®E.

Whenever IPLAN gains (at the beginning of the program) or resumes (after a PUSH or PAUSE) control, the prompts that PSS®E has created are in buffers, called prompt buffers. You have complete control of these buffers. You can count them (PROMPTI), you can erase them (PROMPTC), you can get copies of them placed in local variables (PROMPTR), you can add a new line to the existing prompts (ASK). Of course, you can also ignore them. This might be the proper thing to do if, say, the next statement were a PUSH.

Because IPLAN programs can vary so much, it would be difficult to describe every situation that one might encounter, and explain the alternatives that might be available concerning how prompts may be dealt with. However, some questions have arisen repeatedly, and those situations are described in the following paragraphs.

A.5.1 Common Questions

Why does "ACTIVITY?" get printed with my first input request?

When you execute the EXEC activity, an IPLAN program gets loaded. That's all; loaded (i.e., read in), not executed. PSS®E having now completed activity EXEC, creates the prompt "ACTIVITY?" (actually, ACTIVITY? will be on the second line, the first will be blank), and issues an input request. Since an IPLAN program is loaded and active (the default) the input request is diverted to the IPLAN program which begins executing. Let's say, for example, that the first two statements in your IPLAN program are "ASK 'Enter a number' ", and "INPUT A". Then three prompt lines will be displayed: " ", "ACTIVITY?", and "Enter a number". You could use the PROMPTC function before the ASK statement to clear PSS®E's prompts.

Why do I get an extra "Input?" question when my IPLAN program ends? (When I hit Enter, PSS®E responds with "INVALID ACTIVITY--PLEASE TRY AGAIN.")

This seems capricious, but it's not. Your program has issued a terminal read since the last time it gained control. Once the read request was satisfied, all the prompts are cleared and the GUI is disabled. If the IPLAN program terminates in this state, the input request made by PSS®E that caused the IPLAN program to run in the first place has still not been satisfied. So the input process continues. Unfortunately, it no longer knows the question. The simplest solution is to terminate your program with a PUSH statement, say "PUSH TEXT 'IPLAN program has terminated'". PSS®E will run activity TEXT, and start a new activity request cycle, but the IPLAN program will no longer be active. The SET command END option is another way of doing the same thing.

I have an old IPLAN program that used to run fine. When I run it now all the prompts appear in the progress window. Why?

You probably were using the PRINTX command to create prompts. One of the growing pains that IPLAN has experienced has been that how the PRINTX command works used to be different. The change occurred at Release 9 (PSS®E-22). The shortest answer, if you have no other reason to change the program, is to add an OPTION PRINTX command to the program. This restores the old operation of the PRINTX command. Long term, this is not a good idea. A better answer is to replace all your PRINTX commands with ASK commands. Unfortunately, this assumes a great deal about how your program works, and these assumptions may not all be true. The best answer, but the most work, is to determine what you are really trying to accomplish with all of your existing PRINT and PRINTX commands, and see which of the existing output commands (PRINT, PRINTX, REPORT, REPORTX, ASK, ASKX) is best suited to accomplish your purpose.

A.6 Interaction of IPLAN and PSS®E activities PSEB and PSAS

PSS®E contains two activities that behave like macro language processors, PSEB and PSAS. They are, in fact, macro language translators. The language that the PSEB and PSAS languages are translated into is the PSS®E command language. In other words, what is produced by activities PSEB and PSAS is not the results of the instructions contained in the input program, but a response file containing the PSS®E commands that will carry out those instructions. When either activity is run, it requests an output file; lets say the filename mtest is given. Both activities terminate by doing the equivalent of issuing the command @input mtest.

Absent the use of IPLAN, PSS®E would immediately execute the instructions in the file mtest.idv (the .idv extension would have been automatically added. But this will not happen if the PSEB command was issued from inside an IPLAN program, as explained in section A.4.

The PAUSE READ command can be used to cause the instructions in the PSEB (or PSAS) generated response file to be executed without stopping the IPLAN program. For example:

Given this IPLAN program (in file psebtest.ipl):

```
program psebtest
integer i
i = promptc
set end 'text iplan program complete'
push 'pseb'
push '1'
push 'psebtest'
push 'recover from savnw'
push 'set loadflow title line 1 to psebtest'
push 'solve using fnsl'
push 'end'
print 'this will be displayed before pseb commands run'
pause read
print 'this will be displayed after pseb commands run'
end
```

⚠ Both PSEB and PSAS contain the command ’EXECUTE’ that loads IPLAN programs. Do not run these activities from IPLAN with input containing that command. It will cause the current IPLAN program to immediately terminate.

Issuing the command "exec psebtest" to PSS®E will produce these results:

```
ACTIVITY?

 Executing activity exec psebtest

exec psebtest

 ACTIVITY? pseb

 ENTER INPUT FILE NAME (0 TO EXIT, 1 FOR TERMINAL): 1

 ENTER OUTPUT FILE NAME (BLANK FOR DEFAULT): psebtest

 PSEB: recover from savnw

 PSS®E PROGRAM APPLICATION GUIDE EXAMPLE
 BASE CASE INCLUDING SEQUENCE DATA

 CASE savnw.SAV WAS SAVED ON MON, MAR 08 1999  15:36

 PSEB: set loadflow title line 1 to psebtest
 PSEB: solve using fnsl
 PSEB: end

 Dialogue input from file C:\Program Files\PTI\PSSE26\Example\psebtest.IDV

 ACTIVITY?  this will be displayed before pseb commands run
TEXT, recover from savnw

 ACTIVITY? CASE,savnw.SAV

 PSS®E PROGRAM APPLICATION GUIDE EXAMPLE
 BASE CASE INCLUDING SEQUENCE DATA

 CASE savnw.SAV WAS SAVED ON MON, MAR 08 1999  15:36

 ACTIVITY? TEXT, set loadflow title line 1 to psebtest

 ACTIVITY? CHNG

 ENTER CHANGE CODE:
  0 = EXIT ACTIVITY                 1 = BUS DATA
  2 = GENERATOR DATA                3 = BRANCH DATA
  4 = TRANSFORMER DATA              5 = AREA INTERCHANGE DATA
  6 = TWO-TERMINAL DC LINE DATA     7 = SOLUTION PARAMETERS
  8 = CASE HEADING                  9 = SWITCHED SHUNT DATA
 10 = IMPEDANCE CORRECTION TABLES  11 = MULTI-TERMINAL DC DATA
 12 = ZONE NAMES                   13 = INTER-AREA TRANSFER DATA
 14 = OWNER NAMES                  15 = MACHINE OWNERSHIP DATA
 16 = BRANCH OWNERSHIP DATA        17 = FACTS CONTROL DEVICE DATA: 8

 HEADING LINE 1:
 PSS®E PROGRAM APPLICATION GUIDE EXAMPLE                    CHANGE IT? 1
 ENTER HEADING LINE
   ************************************************************
 : psebtest

 HEADING LINE 2:
 BASE CASE INCLUDING SEQUENCE DATA                          CHANGE IT? Q

 ENTER CHANGE CODE:
  0 = EXIT ACTIVITY                 1 = BUS DATA
  2 = GENERATOR DATA                3 = BRANCH DATA
  4 = TRANSFORMER DATA              5 = AREA INTERCHANGE DATA
  6 = TWO-TERMINAL DC LINE DATA     7 = SOLUTION PARAMETERS
  8 = CASE HEADING                  9 = SWITCHED SHUNT DATA
 10 = IMPEDANCE CORRECTION TABLES  11 = MULTI-TERMINAL DC DATA
 12 = ZONE NAMES                   13 = INTER-AREA TRANSFER DATA
```

```
 14 = OWNER NAMES                    15 = MACHINE OWNERSHIP DATA
 16 = BRANCH OWNERSHIP DATA          17 = FACTS CONTROL DEVICE DATA: 0

 ACTIVITY? TEXT, solve using fnsl

 ACTIVITY? FNSL OPT

 TAP CODE IS 0 TO LOCK, 1 FOR STEPPING, 2 FOR DIRECT
 AREA INT CODE IS 0 TO DISABLE, 1 FOR TIE LINES ONLY, 2 FOR TIE LINES AND LOADS

 ENTER:
  [TAP ] , [AREA INT] , [1 FOR PHASE] , [1 TO FLAT] , [1 TO LOCK] , [1 TO LOCK ]
  [CODE]   [  CODE  ]   [ SHIFTERS  ]   [  START  ]   [D.C. TAPS]   [SWCH SHNTS]
 0 0 0 0 0 0

 ENTER ITERATION NUMBER FOR VAR LIMITS
  0 FOR IMMEDIATELY, -1 TO IGNORE COMPLETELY:  99

 ITER   DELTAP   BUS     DELTAQ   BUS     DELTA/V/   BUS   DELTAANG   BUS
   0    0.0002(  154)    0.0718( 3008)     0.07177( 3008)   0.01532(  101)
   1    0.0906(  205)    2.0109( 3008)     0.03074( 3008)   0.00306(  101)
   2    0.0057(  154)    1.8424(  205)     0.04395(  205)   0.01079(  101)
   3    0.0492(  205)    0.0920(  205)     0.00524(  205)   0.00145(  101)
   4    0.0013(  205)    0.0007(  205)     0.00008(  205)   0.00002(  101)
   5    0.0000(  205)    0.0000(  151)

 REACHED TOLERANCE IN   5 ITERATIONS

 LARGEST MISMATCH:    0.00 MW    0.00 MVAR     0.00 MVA-BUS   151 [NUCPANT 500.00]
 SYSTEM TOTAL ABSOLUTE MISMATCH:               0.01 MVA

 SWING BUS SUMMARY:
   BUS X--- NAME ---X      PGEN     PMAX    PMIN      QGEN     QMAX    QMIN
  3001 MINE    230.00     248.9   9999.0 -9999.0     187.2    600.0  -100.0

 ACTIVITY? TEXT ***

 ACTIVITY? TEXT ***

 ACTIVITY? TEXT *** THIS RUN WAS GENERATED FROM RESPONSE FILE psebtest.IDV

 ACTIVITY? TEXT *** MANUALLY DELETE FILE WHEN DESIRED

 ACTIVITY? TEXT ***

 ACTIVITY? TEXT ***

 ACTIVITY? @END
 Response file operation terminated for file psebtest.IDV
 this will be displayed after pseb commands run
text iplan program complete
```

SIEMENS

Appendix B

IPLAN Graphics

B.1 Introduction

The IPLAN graphics package is a set of subroutines that will allow users to create plots or simple drawings. By calling these subroutines, the user defines the numbering of positions and the movement of the plotting "pen" within a plotting field. (The field may be a plotter page or a CRT.)

B.2 Graphic Conventions

All graphics calls operate under the concept of the basic Cartesian Coordinate System. This is a two-dimensional planar coordinate system defined at an origin, from which perpendicular x- and y-axis are located and discrete coordinate positions may be referenced.

In the following graphic subroutines, the following conventions are used.

The x-coordinate minimum value is at the left-hand side of the drawing and values increase in magnitude to the right. The y-coordinate minimum value is at the bottom of the drawing and values increase in magnitude to the top.

On some devices, the orientation may be rotated 90° clockwise. All angles are specified in degrees with 0° in the direction of the positive x-axis and the direction of rotation is specified in the counter-clockwise direction. Figure B-1 depicts the plot orientation pictorially.

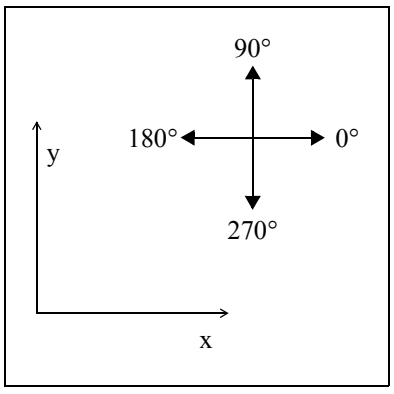


Figure B-1. Plot Orientation

Care should be taken to ensure objects are drawn within the defined limits and that the size of the objects is consistent with the coordinate ranges defined when plotting is initiated.

On some graphic devices such as PostScript laser printers or CRT devices, the flat is always drawn at the size of the device (or window) regardless of the defined plotting limits.

B.3 Using the IPLAN Graphics Package

The following sequence of calls is generally made to generate a plot.

1. Select the plotting device (SELECT).
2. Check for the existence of device (CHECKDEV).
3. Initiate plot page definition, open plot page (STARTPLT).
4. Use plotting calls such as these to generate figures (AXIS, THICK, DRAW, MOVE, TEXT).
5. Terminate plot page (NEXTPAGE).
6. Close plot (CLOSEPLT).

An example program can be found at the end of this Appendix.

B.4 Restrictions

1. Only one graphic display can be defined and used at a time (although multiple graphic displays can be defined and used within a session).
2. No terminal input and/or output can occur while a CRT graphic is in use (i.e., between STARTPLT and CLOSEPLT calls). Output must be via graphic I/O routines.
3. QPUSH and QPUSHX may be used to communicate with the host application provided the hosts output has been redirected to a file.
4. Forms cannot be invoked while CRT graphics is in use.

B.5 Requirements

Use of the IPLAN graphic subroutine library requires one or more of the graphic devices supported on your installation.

B.6 Graphic Subroutines

CALL ARC(X,Y,RAD,ANG1,ANG2,IERR)

Generate an arc or circle.

X

x-coordinate of the focus (real input).

Y

y-coordinate of the focus (real input).

RAD

Arc radius (real input).

ANG1

Beginning angle of arc, in degrees (real input).

ANG2

Ending angle of arc, in degrees (real input).
If ANG1 = ANG2, complete circle is drawn.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL ARCTOL(TOL,IERR)

TOL

Tolerance (integer input).

TOL	***Number of Segments Used***
1	316
2	80
3	36
4	21
5	14

IERR

Error code (integer output):

0 No error.

1 Error.

CALL AREA(FILL,OUTLIN,IERR)

Start/end generation of areas to be toned or filled. The patterns used for filling or toning polygons are device specific. For devices which don't support toning, the polygons are filled with closely spaced parallel lines.

FILL

Defines the shade pattern or color to be used for filling; can be 1-16. (integer input).

0 Performs filling and turns off fill mode.

1 Performs filling of any generated areas and specifies a new pattern.

OUTLIN

Logical flag specifying whether an outline should be drawn around figures (logical input):

True Draw outline.

False Do not draw outline.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL AXIS(X,Y,LABEL,FLAG,LENG,ANGLE,STVAL,DELTA,IERR)

Generates an axis line with labels, scale annotations, and tic marks. Tic marks are drawn and labeled at every unit and values are printed to two decimal places. (See AXISPRM to change these defaults.)

X

x-coordinate for start of axis line (real input).

Y

y-coordinate for start of axis line (real input).

LABEL

Label string which is centered on the generated axis (string input).

FLAG

Logical flag to determine whether annotations are to be placed above or below axis (logical input):

True Annotation placed above.

False Annotation placed below.

LENG

Length of axis in units (real input)

ANGLE

Angle, in degrees, at which the axis is to be drawn. Normally, 0.0° is used for generating the x-axis and 90.0° is used for generating the y-axis (real input).

STVAL

The first value (either MAX or MIN) which will be used for annotating the axis at unit (i.e., inch) intervals (real input).

DELTA

The number of data units per axis unit. This data value will be added to the STVAL starting value for generating scale annotations at each succeeding intervals along the axis line (real input).

IERR

Error codes (integer output):

0 No error.

1 Error.

CALL AXISPRM(DELTA,NDEC,IERR)

Modifies the default parameters used by AXIS.

DELTA

Values placed every DELTA ticks (starting at origin) (real input).

NDEC

Sets number of decimal places used for annotation (integer input)

IERR

Error code (integer output):

0 No error.

1 Error.

CALL BKGRD(ICOLOR,IERR)

Sets background colors on devices that support this feature.

ICOLOR

Background color index (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL CHECKDEV(ICLS,IDEV,IERR)

Checks for the existence of a device. May be called before ‘STARTPLOT.

ICLS

Returns device class or -1 if device is not supported (integer output).

IDEV

Device number to be checked (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL CLIP(MINX,MAXX,MINY,MAXY,IOFF,IERR)

Enables software clipping of vectors drawn outside a rectangular area.

MINX

Minimum limit in x-direction (real input).

MAXX

Maximum limit in x-direction (real input).

MINY

Minimum limit in y-direction (real input).

MAXY

Maximum limit in y-direction (real input).

IOFF

A switch that, if nonzero, causes MINX, MAXX, MINY, and MAXY to be offset/scaled by the values specified in the last call to OFFSET (integer input):

0 0ff.

1 On.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL CLOSEPLT(IERR)

Close plot. See subroutine NEXTPAGE for terminating a plotting page.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL COLOR(IPEN,IERR)

Changes line color on devices that support color. The number of available colors and the assignment of those colors to pen numbers is device specific. If a pen number outside the range of pen numbers for that device is specified, the modulo of IPEN, NPEN is used.

IPEN

Line color (1 to 16), color is device dependent (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL COPYVEC(FLAG,IERR)

Determines whether or not subsequent plotting can be replotted to a new device (not enabled at this time).

FLAG

Logical switch (logical input).

False Subsequent plotting will not appear when the plot is replotted.

True Subsequent plotting will appear when the plot is replotted.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL DASH(IDASH,IERR)

Sets line pattern for subsequent plotting. Appearance of line patterns is device specific.

IDASH

Line pattern specification, integer input (1 to 6).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL DRAW(XCOOR,YCOOR,IERR)

Processes straight line moves with "visible" lines being drawn during movement. The final position of the pen given by the coordinates passed.

XCOOR

x-coordinate (real input).

YCOOR

y-coordinate (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL DRAWOFFST(XCOOR,YCOOR,IERR)

Processes straight line moves with "visible" lines being drawn during movement. The final position (x,y) will be offset and scaled as specified by the last call to OFFSET.

XCOOR

x-coordinate (real input).

YCOOR

y-coordinate (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL DRAWREL(XCOOR,YCOOR,IERR)

Processes straight line moves with "visible" lines being drawn during movement. The final position passed relative to the current position.

XCOOR

x-coordinate (real input).

YCOOR

y-coordinate (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL FONT(IFONT,IERR)

Allows for the selection of different character fonts.

IFONT

Font set to be used (integer input).

ERR

Error code (integer output):

0 No error.

1 Error.

CALL GRID(XL,XR,XD,NNX,MODX,YL,RY,YD,NNY,MODY)

Subroutine GRID allows the user to generate a variety of grid patterns or overlays.

XL

First x-coordinate value to be used in generating the outline of the grid (real input).

XR

Last x-coordinate value to be used in generating the outline of the grid (real input).

XD

Distance between uniformly spaced vertical lines (real input).

NNX

Number of intervals in the x-direction (NNX < 1000) (integer input).

MODX

If greater than zero, indicates that every MODX vertical line is to be skipped (integer input).

YL

First y-coordinate value to be used in generating the outline of the grid (real input).

YR

Last y-coordinate value to be used in generating the outline of the grid (real input).

YD

Distance between uniformly spaced horizontal lines (real input).

NNY

Number of intervals in the y-direction (NNY < 1000) (integer input).

MODY

If greater than zero, indicates that every MODY horizontal line is to be skipped (integer input).

CALL LOGO(XC,YC,SIZE,ANGLE,IERR)

Allows the programmer to draw the Siemens PTI logo on a plot.

XC,YC

Lower left x,y coordinates for positioning the drawing (real input).

SIZE

Size of the logo with respect to the current value of units in each direction (real input).

ANGLE

Angle in degrees; the logo is rotated about (xy,yc). Rotation is counterclockwise, measured from the horizontal (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL MOVE(XCOOR,YCOOR,IERR)

Processes straight line moves with "invisible" lines being drawn. The final position of the pen is determined by XCOOR,YCOOR.

XCOOR,
YCOOR

x,y coordinates (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL MOVEOFFST(XCOOR,YCOOR,IERR)

Processes straight line moves with "invisible" lines being drawn during movement. The final position of the pen will be offset and scaled as specified by OFFSET.

XCOOR,
YCOOR

x,y coordinates (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL MOVEREL(XCOOR,YCOOR,IERR)

Processes straight line moves with "invisible" lines being drawn during movement. The final position of the pen "relative" to the current position.

XCOOR,
YCOOR

x,y coordinates (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL NAMEDEV(NAME,IDEV,IERR)

Returns the name associated with the device number. When the device is not installed in a given installation, the string "interface not installed" is appended to the name.

NAME

Device name (string output).

IDEV

Device number (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL NEWPEN(IPEN,IERR)

Selects a different line width (or color on devices that support color). The number of available colors and the assignment of those colors to pen numbers is device specific. If a pen number outside the range of pen numbers for that device is specified, the modulo of IPEN, NPEN is used.

IPEN

Specifies either the line thickness or pen number or color index to be used for subsequent plotting (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL NEXTPAGE(I,N,IERR)

Terminate a plotting page. See subroutine CLOSEPLT to terminate all plotting.

I

Reserved for future enhancement, must be variable (integer input).

N

Switch that governs the action taken by NEXTPAGE (integer input).

0 Terminate the plotting page.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL NOCLIP(IERR)

Disables the software clipping limits established in a previous call to CLIP.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL NUMBER(XCOOR,YCOOR,SIZE,FPN,ANGLE,NDEC,IERR)

Subroutine NUMBER may be used to convert a floating point number to the appropriate character equivalent which is then drawn on the user's plot

XCOOR,
YCOOR

x,y coordinate for lower left-hand corner of the first character to be plotted (real input).

SIZE

Character height in units (real input).

FPN

Floating point number to be plotted (real input).

ANGLE

Angle, in degrees, that the annotation is to be rotated about (XCOOR, YCOOR) (real input).

NDEC

Specifies the number of digits to be plotted (integer input):

>0 NDEC is the number of digits to be right of the decimal point that are to be plotted after rounding.

0 Indicates that only the integer position and the decimal point are to be plotted after rounding.

-1 Indicates that only the integer position is to be plotted with no decimal point after rounding.

<-1 Absolute (NDEC)-1 digits are to be right truncated from the integer portion and plotted after rounding.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL OFFSET(XOFF,XFAC,YOFF,YFAC,IERR)

Subroutine OFFSET allows the user to set special scale factors and offsets to be used by DRAWOFFST and MOVEOFFST calls and to PLOT calls with IPEN codes 12 or 13. These values are applied to the x,y coordinates prior to other factors.

XOFF

Defines the offset applied to x values when PLOT is called (real input).

XFAC

Defines the scaling factor applied to x values when PLOT is called (real input).

YOFF

Defines the offset applied to y values when PLOT is called (real input).

YFAC

Defines the scaling factor applied to y values when PLOT is called (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL PLOT(XCOOR,YCOOR,IPEN,IERR)

Processes straight line moves with either "visible" or "invisible" lines being drawn during movement. The XCOOR,YCOOR determine the final position of the pen.

XCOOR,
YCOOR

x,y coordinates (real input).

IPEN

Plotting option control (integer input).

2 Draw to (XCOOR,YCOOR).

3 Move to (XCOOR,YCOOR).

12 Draw to offset/scaled (XCOOR,YCOOR).

13 Move to offset/scaled (XCOOR,YCOOR).

22 Draw relative from current point.

23 Move relative from current point, anything else is ignored.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL PRINTTEXT(XCOOR,YCOOR,SIZE,TEXT,LEN,ANGLE,IERR)

Outputs character text using hardware character generators, when possible, rather than plotting characters.

XCOOR,
YCOOR

Starting x,y coordinate for lower left-hand corner of the first character to be plotted (real input).

SIZE

Character height in units (real input); currently unavailable.

TEXT

Character string containing text to be printed (string input).

LEN

Length of the character string TEXT (integer input).

ANGLE

Angle, in degrees, that the text is rotated about (XCOOR,YCOOR) before it is plotted (real input); currently unavailable.

IERR

Error code (integer output):

0 No error.

1 Error.

CALL REVISION(MSGBUF,IERR)

Returns the version number and date of the current underlying graphics package being used.

MSGBUF

Name, release number, and date (string output).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL ROTATE(XCOOR,YCOOR,ANGLE,IERR)

Causes subsequent plotting to be rotated around a given axis.

XCOOR,
YCOOR

x,y coordinates pivot (real input).

ANGLE

Angle, in degrees; the plotting is to be rotated counterclockwise (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL SELECT(IDEV,IERR)

Prompts the user for the desired plotting device from a list of installed devices.

IDEV

Device number chosen (integer output).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL STARTPLT(IDEV,XMIN,XMAX,YMIN,YMAX,XCHNG,IERR)

Start plot.

IDEV

Desired device number to be plotted to (integer input). Note that legal plotting devices may be obtained by calling subroutine SELECT first.

XMIN

Minimum x value to be plotted in units (real input).

XMAX

Maximum x value to be plotted in units (real input).

YMIN

Minimum y value to be plotted in units (real input).

YMAX

Maximum y value to be plotted in units (real input).

XCHNG

If nonzero causes plot to be rotated 90° around axis (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL SYMBOL(XCOOR,YCOOR,SIZE,ISYMB,ANGLE,IERR)

Plots special values at various angles and sizes. See CALL FONT(IFONT,IERR). Symbol numbers greater than 128 use symbols shown in Figure B-2.

XCOOR,
YCOOR

Starting x,y coordinate for the symbol (real input).

SIZE

Character height in units (real input).

ISYMB

Symbol number. If ISYMB is negative, a visible line is drawn during the move from the present position to (XCOOR,YCOOR) (real input).

ANGLE

Angle, in degrees, that the symbol is rotated about (XCOOR,YCOOR) (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

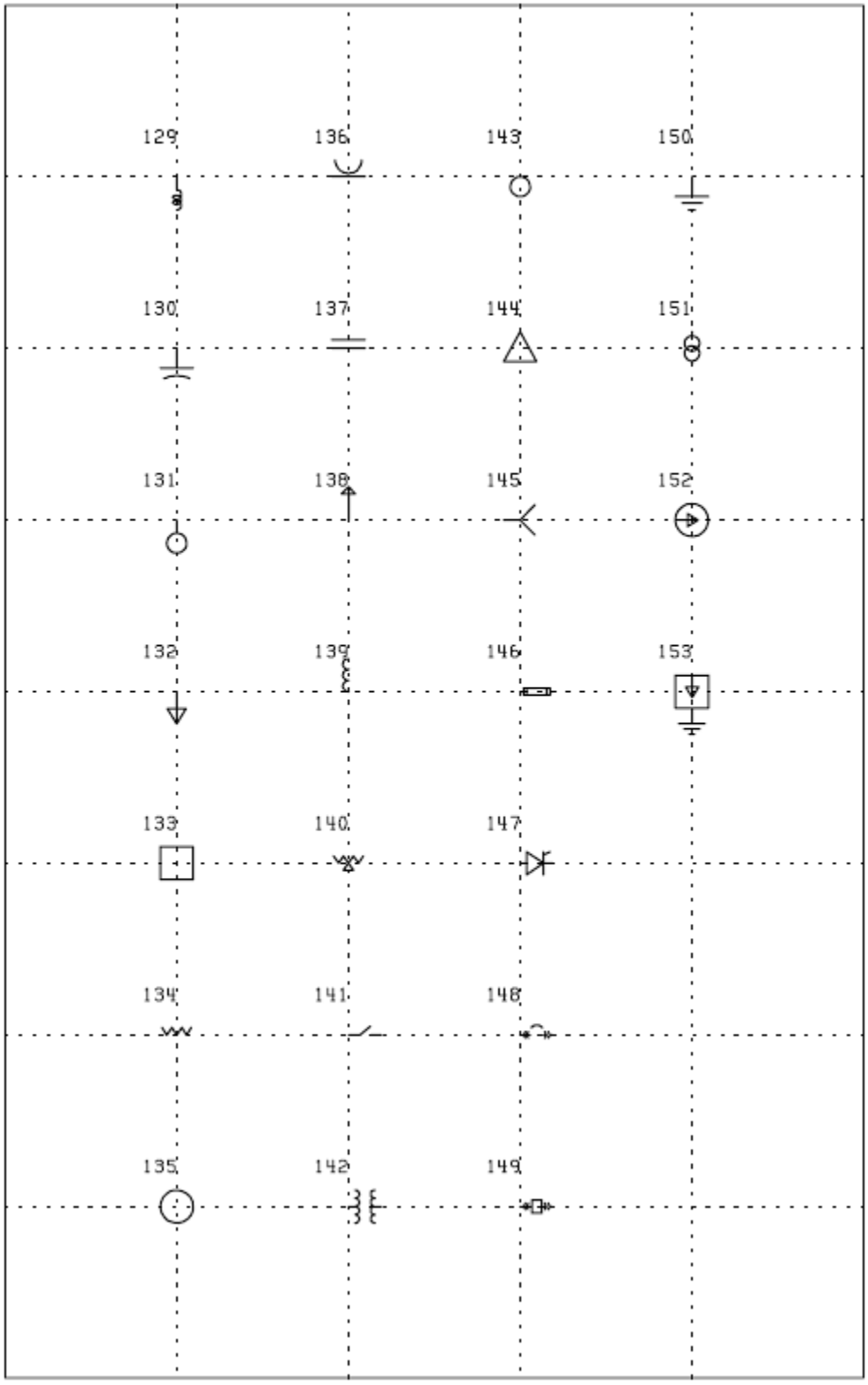


Figure B-2. Symbol Numbers Greater Than 128

CALL TEXT(XCOOR,YCOOR,SIZE,TEXT,LENGTH,ANGLE,IERR)

Plots alphanumeric annotations at various angles and sizes. See CALL FONT(IFONT,IERR).

XCOOR,
YCOOR

Starting x,y coordinate for the text (real input).

SIZE

Character height in units (real input).

TEXT

String containing text to be plotted (string input).

LENGTH

Length of TEXT in characters, including blanks (integer input).

ANGLE

Angle, in degrees, that the text is rotated about (XCOOR,YCOOR) (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL THICK(IPEN,IERR)

Changes line thickness on devices which do not support colors.

IPEN

Line width (1 to 5), 1 being thinnest (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL TYPE(ITYPE,IDEV,IERR)

Returns a code to designate the type of device selected for plotting.

ITYPE

Device type (integer output):

0 IDEV is not a graphic CRT.

1 IDEV is a graphic CRT.

IDEV

Device number (integer input).

IERR

Error code (integer output):

0 No error.

1 Error.

CALL WHERE(XNOW,YNOW,IERR)

Returns current location of the "graphics" pen.

XNOW,
YNOW

Current location of pen in coordinate plane (real input).

IERR

Error code (integer output):

0 No error.

1 Error.

B.7 Sample Graphics Program

```
PROGRAM TEST1

/*************************************************************************/
/*      THIS PROGRAM DEMONSTRATES THE GRAPHICS SUBROUTINES OF IPLAN 6.0. */
/*   IT PROVIDES EXAMPLES OF:  SCALING FUNCTION VALUES TO A PLOTTING     */
/*   RANGE;  CREATING BORDERS; GENERATING AND PLOTTING A SERIES OF       */
/*   POINTS;  AND THE USE OF SEVERAL COMMON ACTIVITIES.                  */
/*************************************************************************/


INTEGER  IER            !  ERROR CODE RETURNED BY GRAPHICS CALLS
INTEGER  PLOTR          !  PLOTTING DEVICE NUMBER
INTEGER  B              !  DUMMY INTEGER VALUE
INTEGER  NDEC           !  NUMBER OF DECIMAL PLACES ON AXIS TIC-MARKS
REAL     XBIG,XSML      !  LIMITS OF PLOTTER--RIGHT, LEFT EDGES
REAL     YBIG,YSML      !  LIMITS OF PLOTTER--TOP, BOTTOM EDGES
REAL     XMAX,XMIN      !  LARGEST, SMALLEST X-VALUES TO BE PLOTTED
REAL     YMAX,YMIN      !  LARGEST, SMALLEST Y-VALUES TO BE PLOTTED
REAL     MLEFT,MRIGHT   !  PLOT DEVICE SPACE ALLOWED FOR MARGINS
REAL     TOP, BOT       !  PLOT DEVICE SPACE ALLOWED FOR MARGINS
REAL     XBGN,XEND      !  STARTING, ENDING POSITIONS OF X-AXIS
REAL     YBGN,YEND      !  STARTING, ENDING POSITIONS OF Y-AXIS
REAL     XS,YS          !  LENGTHS OF X, Y-AXES
REAL     XTIC,YTIC      !  CHANGE IN VALUES BETWEEN AXES' TICK-MARKS
REAL     XPT,YPT        !  SCALED PLOTTER POSITION-REPRESENTS (XVAL,YVAL)
REAL     XVAL,YVAL      !  FUNCTION VALUES
STRING   XTTL,YTTL,FX   !  X-AXIS, Y-AXIS, MAIN TITLES

B    = PROMTC
SET END   'PTEXT END OF PROGRAM'
SET STOP  'TEXT END OF PROGRAM'

CALL SELECT(PLOTR,IER)  !  SELECT PLOTTING DEVICE
  IF (IER <> 0) THEN
    PRINT 'ERROR IN SELECT '
    STOP
  ENDIF

XMAX =  10.            !  MAXIMUM, MINIMUM NUMERICAL (FUNCTION)
```

```
XMIN =   0.            !  VALUES TO BE PLOTTED
YMAX = 100.
YMIN =-100.

TOP = 3.0              !  TOP MARGIN, IN SCREEN SPACES
BOT = 1.0              !  BOTTOM MARGIN
MRIGHT= 1.0             !  RIGHT MARGIN
MLEFT =1.0              !  LEFT MARGIN
XBIG = 20.0            !  CALL LAST DEVICE X-LOCATION 20.0, FIRST 0.0
XSML = 0.0             !  (PAGE IS DIVIED UP INTO A FIXED NUMBER OF
YBIG = 20.0            !  LOCATIONS WHICH VARIES WITH THE DEVICE.
YSML = 0.0             !  FRACTIONAL VALUES ARE ALLOWED.)
XBGN = XSML+MLEFT       !  STARTING PLOTS > .7  GIVES GOOD BORDER
YBGN = YSML+BOT
XEND = XBIG-MRIGHT
YEND = YBIG-TOP        !  TOP MARGIN IS BIGGER BECAUSE OF PLOT'S TITLE
NDEC = 2

CALL STARTPLT(PLOTR,XSML,XBIG,YSML,YBIG,10.0,IER) ! DEFINE THE
IF (IER <> 0) THEN                                ! NUMBERING OF
 PRINT 'ERROR IN STRTPLOT'                        ! DEVICE DIVISONS
 STOP                                 ! # OF DIVISIONS~ DEVICE RESOLUTION
ENDIF

XTTL = 'TIME'          !  AXES' TITLES
YTTL = 'MAGNITUDE'

XS   = (XEND-XBGN)     !  AXES LENGTHS IN PLOTTER DEVICE DIVISIONS
YS   = (YEND-YBGN)
XTIC = (XMAX-XMIN)/(XS)          !  NOTE: LOGIC ASSUMES XMAX > XMIN
YTIC = (YMAX-YMIN)/(YS)          !        IN CALCULATION OF TIC-MARK DELTA

CALL AXISPRM(1.0,NDEC,IER)       ! SETS DEFAULT AXIS PARAMS
CALL THICK(2,IER)                ! SELECTS PEN THICKNESS (WIDTH)

CALL AXIS(XBGN,YBGN,XTTL,FALSE,XS,0.0,XMIN,XTIC,IER)  ! DRAW AXES
CALL AXIS(XBGN,YBGN,YTTL,TRUE,YS,90.0,YMIN,YTIC,IER)

CALL MOVE(XEND,YBGN,IER)  !  DRAW A BORDER AROUND PLOTTING AREA...
CALL DRAW(XEND,YEND,IER)
CALL DRAW(XBGN,YEND,IER)

CALL AREA(3,TRUE,IER)              ! TEST SHADING FUNCTION...
CALL MOVE(XEND,YEND,IER)
CALL DRAW(XEND-2.0,YEND,IER)
CALL DRAW(XEND-2.0,YEND-2.0,IER)
CALL DRAW(XEND,YEND-2.0,IER)
CALL DRAW(XEND,YEND,IER)
CALL AREA(0,TRUE,IER)

/*  CREATE A SAMPLE FUNCTION  */

YVAL =  0.0
XVAL =  0.0
YPT  =  YBGN+((YVAL-YMIN)/(YMAX-YMIN))*(YS)
XPT  =  XBGN+((XVAL-XMIN)/(XMAX-XMIN))*(XS)
CALL MOVE(XPT,YPT,IER)          ! MOVE TO ORIGIN

XVAL = 0.0001                   ! AVOID INFINITE VALUES
LOOP WHILE (XVAL<=XMAX)

    YVAL = 100.0*SIN(2.0*XVAL)*EXP(-XVAL/3.14)     ! EXAMPLE FUNCTION
    FX   ='100.0*SIN(2.0*XVAL)*EXP(-XVAL/3.14)'    ! EXAMPLE FUNCTION STRNG

    YPT=YBGN+((YVAL-YMIN)/(YMAX-YMIN))*(YS)    ! SCALE  VALUE
    XPT=XBGN+((XVAL-XMIN)/(XMAX-XMIN))*(XS)    ! TO A SCREEN (DEVICE) LOCATION
    CALL PLOT(XPT,YPT,2,IER)                   ! PLOT AT LOCATION
    XVAL=XVAL+0.015

ENDLOOP
```

```
/*  LAY A TITLE IN THE TOP MARGIN  */
CALL TEXT(XBGN,YEND+(TOP/2.0),0.60,'GRAPHICS EXAMPLE',16,0.0,IER)
CALL PRINTTEXT(XBGN,YEND+(TOP/2.0)-.5,1.0,FX,100,1.0,IER)

CALL THICK(1,IER)
CALL LOGO (0.0,0.0,0.5,0.0,IER)               ! PTI LOGO IN LOWER LEFT

CALL DASH(5,IER)           ! FLOP DOWN A GRID
CALL GRID(XBGN,XEND,1.0,NINT(XS),0,YBGN,YEND,1.0,NINT(YS),0)

CALL NEXTPAGE(B,0,IER)     ! SHUT DOWN PLOTTING-USING NEXTPAGE IS RECOMMENDED
                           ! FOR GRAPHICS DISPLAYS--KEEPS PLOT ON SCREEN
CALL CLOSEPLT(IER)

END
```

B.8 Common Questions

Question: When using the graphics routines in IPLAN, some of the text is displayed incorrectly. Instead of letters it appears to print mathematical symbols and Greek letters. Why?

Answer: Add this statement to your IPLAN program before producing your graphics:

```
CALL FONT(2)
```

This will switch the graphics software font to the second font, which contains lowercase letters; the first font does not. Be sure to set the font back to 1 or you may affect PSS®E graphics activities that expect to use those mathematical symbols and Greek letters.

Details: If you look closely, I think you will find that only lowercase letters are being printed incorrectly. PTI's plotting package supplies several software fonts. Two are available at any one time. If you have a standard installation, the two fonts that are available to you are ftueng and ftleng.

SIEMENS

Appendix C

Character Sets

This appendix represents the ASCII/EBCDIC character sets. To the left side of the table are the base decimal values for each row, and at the top of the table are the amounts to add to the row for each column of characters. To determine the decimal value of an ASCII/EBCDIC character, add the decimal value that corresponds to the row with the decimal value that corresponds to the column. For example, the value of the ASCII character representing the equal sign ("=") is 56+5 or 61.

Table 1 ASCII Character Set shows the ASCII character set.

Table 1: ASCII Character Set

Row/Column	+0	+1	+2	+3	+4	+5	+6	+7
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL
8	BS	HT	LF	VT	FF	CR	SO	SI
16	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB
24	CAN	EM	SUB	ESC	FS	GS	RS	US
32		!	"	#	$	%	&	’
40	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7
56	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G
72	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W
88	X	Y	Z	[	\	]	^	_
96	‘	a	b	c	d	e	f	g
104	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~	DEL

The EBCDIC character set shown in Table 2 EBCDIC Character Set is used on IBM mainframes. All other machines use the ASCII character set.

Using the previous example, the value of the EBCDIC character representing the equal sign (=) is 120 + 6 OR 126.

Table 2: EBCDIC Character Set

Row/Column	+0	+1	+2	+3	+4	+5	+6	+7
64	space							
72			¢	.	<	(	+	
80	&							
88			!	$	*	)	;	¬
96	-	/						
104				,	%	_	>	?
112								
120		‘	:	#	@	’	=	"
128		a	b	c	d	e	f	g
136	h	i						
144		j	k	l	m	n	o	p
152	q	r						
160		~	s	t	u	v	w	x
168	y	z						
172								
184								
192	{	A	B	C	D	E	F	G
200	H	I			hook		fork	
208	}	J	K	L	M	N	O	P
216	Q	R						
224	\		S	T	U	V	W	X
232	Y	Z			chair			
240	0	1	2	3	4	5	6	7
248	8	9	|					EO