SPL User Manual

AMERICAN AUTO-MATRIX®
SMART BUILDING SOLUTIONS®

SPL User Manual

Part Number 1E-04-00-0083

Updated 3/16/2007

WORLD HEADQUARTERS

American Auto-Matrix
One Technology Lane
Export, Pennsylvania 15632-8903 USA
Tel (1) 724-733-2000
Fax (1) 724-327-6124
Email aam@aamatrix.com
www.aamatrix.com

Updated 3/16/2007

- Document uses new date revision scheme for Technical Documentation
- §2.3.2 - Added EQU Statement and expression examples
- §3.4.2 - Fixed BACnet statement expression examples
- §Appendix E - Change header for BACnet Data Types from “Number” to “Identifier Number”

Version 2.1

- Fixed various SPL sample errors found by Tech Services (various pages in manual)
- Updated SPL Error Code Appendixes (Appendix B and Appendix C)
- Re-wrote SPL and BACnet section (Section 3) to provide better understanding of how to write SPL programs for NB-GPC Product Family devices.
- Fixed BACnet SPL Reference Tables (Appendix E) to reflect the correct Object References for NB-GPC family devices.
- Updated DREF Statement (Section 2.4.1)
- Updated Program Control Attributes (Section 2.15)
- Updated Overview Section which discusses Compiler Control Statements

Version 2.0 - Initial Manual Re-Release

1.1 Introduction 1-3
1.2 The Parts of SPL Programs 1-4
1.3 Program Names 1-5
1.4 The .SPL, .PLB and .LST Files 1-6
1.5 The Program Reference Block (PRB) 1-7
1.6 The Initial Value (INI) File 1-8
1.7 Attributes and Registers 1-9
1.8 Compiler Control Statements 1-10
1.9 Comments 1-13
1.10 Labels 1-14
1.11 Expressions 1-15
1.12 Constants 1-17
1.13 Named Constants 1-19
1.14 Registers 1-22
1.15 Program Attributes 1-23
1.16 Named Object Attributes ....
(SAGE) 1-24
1.17 Named PUP Object Attributes.................... (GX,
DX, GPC) 1-25
    1.17.1 Named BACnet Object Properties (GPC) 1-25
1.18 Tables (SAGE) 1-26
    1.18.1 RAM-based Tables 1-26
1.19 Functions 1-28
1.20 Expression Operators 1-35
2.1 Introduction 2-3
2.2 Working with Attributes 2-5
    2.2.1 ATTR Statement 2-5
    2.2.2 TABLE and DATA Statements 2-5
    2.2.3 SAVE Statement (SAGE) 2-6
2.3 Assignment Statements and Equates 2-8
    2.3.1 Standard Value assignment 2-8
    2.3.2 EQU 2-9
2.4 References 2-10
    2.4.1 DREF Statement 2-10
    2.4.2 REF Statement 2-10
2.5 Virtual Attributes 2-11
2.6 Iteration, Branching and Subroutines 2-12
    2.6.1 GOTO statement 2-12
    2.6.2 IF... THEN... {ELSE...} Statement 2-12
    2.6.3 ON... GOTO... statement 2-13
    2.6.4 LOOP Statement 2-13
    2.6.5 GOSUB Statement 2-14
    2.6.6 CALL Statement 2-14
    2.6.7 RETURN Statement 2-15
2.7 Program Delays 2-16
    2.7.1 SWAIT and MWAIT Statements 2-16

2.7.2 WAIT Statement 2-16
2.8 Printing, Logging, and Alarms 2-17
2.8.1 PRINT Statement (SAGE) 2-17
2.8.2 LOG Statement (SAGE) 2-19
2.8.3 ALARM Statement 2-20
2.9 Job Execution (SAGE) 2-22
2.9.1 The REPORT Job 2-22
2.9.2 The SPOOL Job 2-25
2.9.3 The DATA CAPTURE /DATA STUFF Job 2-27
2.10 Spooling Report and Log Files (SAGE) 2-35
2.10.1 SPOOL Statement 2-35
2.11 Trending Control Statements (SAGE) 2-36
2.11.1 STARTTREND & STOPTREND Statements 2-36
2.12 Program Execution Statements (SAGE) 2-37
2.12.1 ACTIVATE Statement 2-37
2.12.2 DEACTIVATE Statement 2-37
2.12.3 The RESTART Statement 2-37
2.12.4 STOP Statement 2-38
2.12.5 UNLOAD Statement 2-38
2.13 Execution Error Control 2-40
2.13.1 ERRORABORT Statement 2-40
2.13.2 ERRORWAIT Statement 2-40
2.13.3 ONERROR Statement 2-41
2.14 Debugging Statements 2-42
2.14.1 SECTION Statement 2-42
2.14.2 NOP Statement 2-42
2.15 Program Control Attributes 2-43
3.1 Introduction 3-3
3.2 Fundamentals of SPL in BACnet 3-4
3.2.1 The PROP Statement 3-4
3.2.2 Prop Statement Examples 3-4
3.3 Working with Object Properties 3-6
3.3.1 Referencing Objects 3-6
3.3.2 Referencing Properties 3-6
3.3.3 Addressing Object Properties 3-6
3.3.4 Addressing User-Defined properties 3-7
3.3.5 Peer-To-Peer Addressing 3-7
3.3.6 Writing Values to Object Properties 3-8
3.3.7 Data Type Sensitivity with BACnet SPL 3-10
3.3.8 EQU Function Limitations in BACnet SPL 3-10
3.4 Advanced BACnet SPL Functions 3-11
3.4.1 The OID Function 3-11
3.4.2 The BACNET Statement 3-11
A.1 Introduction A-3
A.2 ACTIVATE A-5
A.3 ALARM A-5

A.4 ATTR A-5
A.5 BACNET A-6
A.6 CALL A-6
A.7 DATA A-7
A.8 DEACTIVATE A-7
A.9 DREF A-7
A.10 ERRORABORT A-8
A.11 ERRORWAIT A-8
A.12 GOSUB A-8
A.13 GOTO A-9
A.14 IF... THEN... {ELSE...} A-9
A.15 JOB A-9
A.15.1 REPORT Job A-9
A.15.2 SPOOL Job A-10
A.15.3 BROADCAST (BC) Job A-10
A.15.4 DATA CAPTURE/DATA STUFF Job A-10
A.15.5 UPLOAD/DOWNLOAD Job A-11
A.15.6 EXPORT Job A-11
A.16 LOG A-12
A.17 LOOP A-12
A.18 MWAIT A-12
A.19 NOP A-13
A.20 OID A-13
A.21 ON... GOTO A-13
A.22 ONERROR A-14
A.23 PRINT A-14
A.24 PROP A-14
A.25 RESTART A-15
A.26 RETURN A-15
A.27 SAVE A-15
A.28 SECTION A-16
A.29 SPOOL A-16
A.30 STARTTREND A-16
A.31 STOP A-17
A.32 STOPTREND A-17
A.33 SWAIT A-17
A.34 TABLE A-18
A.35 UNLOAD A-18
A.36 WAIT A-18
E.1 BACnet Data Types E-1
E.2 BACnet Objects E-2
E.3 Property Identifiers E-4

Section 1: Overview

In this section

Introduction 1-3
The Parts of SPL Programs 1-4
Program Names 1-5
The .SPL, .PLB and .LST Files 1-6
The Program Reference Block (PRB) 1-7
The Initial Value (INI) File 1-8
Attributes and Registers 1-9
Compiler Control Statements 1-10
Comments 1-13
Labels 1-14
Expressions 1-15
Constants 1-17
Named Constants 1-19
Registers 1-22
Program Attributes 1-23
Named Object Attributes (SAGE) 1-24
Named PUP Object Attributes (GX, DX, GPC) 1-25
Tables (SAGE) 1-26
Functions 1-28
Expression Operators 1-35

1.1 INTRODUCTION

Historically the large number of REX Program Language (RPL) programs which exist for RCU and STAR, and the general familiarity with RPL among end-users, led to the enhancement of the RPL language for SAGE application programming. The resulting extended language was called SAGE Programming Language (SPL). This language has since been extended first for SOLO/DX and SOLO/GX, and now again to accommodate BACnet programming concepts.

Overall, the language supports a large number of features, summarized below:

▾ Unlimited number of program lines up to maximum pcode size
▾ Extension in pcode size to 64K bytes.
▾ Up to 255 program attributes
▾ Indirect references within the program up to 256 references
▾ Floating point math and type coercion
▾ Access to lookup tables for scaling, conversion and general purpose storage in programs
▾ Job execution including Report Generation
▾ Formatted printing
▾ Ability of printing to a log disk file for batch report generation and printing
▾ Standard function blocks such as PID that are usable by multiple programs
▾ Re-entrant subroutine calling ability for multiple programs
▾ Up to sixteen (16) program registers capable of 32 bit values and data type
▾ Six (6) Level expression stack to resolve nested expressions
▾ Asynchronous read/write of named object attributes
▾ Emulation of BACnet Program Objects
▾ Asynchronous read/write of BACnet object properties
▾ Support for BACnet object properties in addition to attributes

1.2 THE PARTS OF SPL PROGRAMS

Application programs that run in the SAGE, SOLO/DX, SOLO/GX and GPC controllers are written in SPL. An SPL program consists of a collection of structures that are used by the SPL compiler and execution system. An SPL program consists of the following items:

- a program name
- a Program Logic Block (PLB) file
- an optional Program Reference Block (PRB) file
- an optional attribute Initial Value (INI) file
- program attributes and registers
- an Engineering Units (EU) override file
- a set of program options

These components are explained in the following sections of this chapter.

1.3 PROGRAM NAMES

SPL programs must have an associated name that is limited only by the operating system. The valid characters for program names are shown below:

- ▼ A-Z (uppercase letters “A” through “Z”)
- ▼ a-z (lowercase letters “a” through “z”)
- ▼ 0-9 (numbers “0” through “9”)
- ▼ _ (under bar)
- ▼   (space)
- ▼ . (period)
- ▼ $ (dollar sign)

Program names are case-insensitive, meaning lowercase letters are treated the same as uppercase letters in program object names (e.g., "abc" is the same as "ABC").

1.4 THE .SPL, .PLB AND .LST FILES

SPL programs must reference a Program Logic Block (PLB). The PLB is a binary data file that contains compiled binary pseudocode in a form which can be executed by the controller. This file is created by the SPL Compiler after you create, edit and compile an ASCII text file (i.e., an SPL source file) which contains program logic statements that are easily edited and understood by programmers.

ASCII SPL source code files have the .spl extension and Binary Program Logic Block files (PLBs) have the .plb extension.

The name given to the SPL file can be as long as allowed by the operating system on your computer. However, the controllers impose a limitation on file length. When a PLB is loaded into a SAGE of a unitary controller, the file name is shortened to 8 characters. If the name was originally longer, only the first 8 characters will be used for the program name.

The source file contains SPL program logic statements which are discussed in detail later in this section. SPL source code can be created and/or edited by using the SPL editor built in to the NB-Pro and SoloPro software packages, any unformatted text editor, or the Program Editor in the SAGE^{MAX}. The number of lines in an SPL source file is unlimited.

The source code that you create in ASCII form is converted to a binary pseudocode file (the PLB) by the SPL compiler. Compiled PLBs cannot exceed 65,535 bytes in size.

NOTE

If any errors are generated during the compiling process, a PLB is not created.

You may choose to have the SPL Compilers optionally create a list file during the compile process. The list file is an ASCII text file that contains the source code statements along with the pseudocode and their respective *relative* locations in memory and any error messages generated by the compiler. List files are useful in debugging the execution of SPL programs. List files have the same name as the SPL source file except that they have the extension .lst and are found on the *C:\SPL* subdirectory of the SAGE^{MAX}.

NOTE

The LST file does not point out execution or logic errors in your program logic. Only syntax errors (errors due to the improper construction or format of program statements) are flagged by the compiler and shown in the list file.

1.5 THE PROGRAM REFERENCE BLOCK (PRB)

SPL programs may contain an optional Program Reference Block (PRB) file. PRBs allow programs to refer to named objects indirectly so that the actual object names do not need to be used in the program logic statements. This allows many different program objects to share the same PLB.

If a PLB uses *references* as part of its logic, the associated program object must contain a PRB. The PRB is an ASCII text file that specifies the object name/reference association.

Each reference in the PRB text file must be on a separate line and must adhere to a specific format so that it may be recognized by the SAGE^{MAX}. References must contain the object name to be referenced, and may optionally contain the object type code (i.e., PT, PG, VR, GL) and/or an attribute. Indices into the PRB are zero-based (e.g., reference 0 is the first reference). The four formats for PRB references are shown below:

- \ objecttype \ objectname ; attribute
- objectname ; attribute
- \ objecttype \ objectname
- objectname

The *objecttype* is a two character mnemonic (PT, VR, GL, PG, etc.). If missing, the correct type will be determined by a search of *all* object types

If no object type is specified in a reference, then the SAGE^{MAX} searches all object types in its database for the first object name that matches. This search is performed in the following order: Points, Programs, Variables, and Globals.

1.6 THE INITIAL VALUE (INI) FILE

A third component of an SPL program is an attribute Initial Value File or INI file. This optional ASCII text file is used to set program attributes to initial values at the start of program execution. If a program attribute is not listed in the INI File, the attribute is initialized to zero. If no INI file is specified for a program, all its program attributes are initialized to zero. SPL provides mechanisms that can be used to save new initial values to this file. Programs with the same attributes that have the same initial values may share the same Initial Value File.

1.7 ATTRIBUTES AND REGISTERS

All programs have 16 registers (**%A-%P**) and a number of program control attributes (or properties for BACnet controllers) depending on the platform being used. To an operator, program registers always begin with a percent sign (**%**) and program control attributes always begin with a dollar sign (**$**). Up to 255 additional two-character attributes can be defined for every program.

On a SAGE, program registers, program control attributes and user-defined program attributes can be accessed from the Monitor Program Submenu by qualified users. The first user-defined program attribute is the *default attribute*. If no user-defined program attributes are specified in the program, the **$$** program control attribute is displayed as the default when you monitor the program object.

1.8 COMPILER CONTROL STATEMENTS

Compiler control statements are non-executable directives to the SPL compiler that it uses to control the generation and format of SPL compiler listings and PLBs. The various SPL compiler control statements are summarized below:

#NOLIST
#TITLE "*titletext*"
#PAGE *length,width*
#NOLABELS
#LABELS
#FIXED
#FLOAT
#SAGE
#SOLODX
#SOLOGX
#GPC
#PLB08K, #PLB16K, #PLB64K
#ONESEC
#ENDONESEC

Compiler control statements always begin with the pound sign character (**#**). They must begin in the left-most column.

#NOLIST

The no listing command is used if suppression of a compiler list file is desired. Unless otherwise directed by this command, the SPL compiler generates a companion list file *sourcefilename.LST* from the designated SPL program logic source file. The compiler *always* generates a PLB from the source file. If the #NOLIST command is used, it **must** be on the first line of the source file.

#TITLE "*titletext*"

The title directive is used to put the specified ***titletext*** at the top of each page of a compiler list file in order to help identify the program logic. The text string ***titletext*** must be enclosed in double quotation marks (**"**) and can be up to 79 characters long.

#PAGE length,width
#PAGE

The page control commands direct the SPL compiler to set the maximum number of characters per line in the listing file to ***width*** and the maximum number of lines per page to ***length***. When the page control command is used without arguments, a new page is started in the compiler listing by writing a form-feed character to the list file.

#NOLABELS
#LABELS

The no labels directive commands the SPL compiler to *not* generate pseudo-code for statement labels in the PLB file. Using this command results in smaller, faster executing PLBs, but eliminates the ability to visually locate labels in the PLB files during troubleshooting. The **#LABELS** command turns on the inclusion of label pcodes in the PLB. In order to conserve RAM and optimize execution, **#NOLABELS** is the default for **#SOLODX, #SOLOGX** and **#GPC**. #**LABELS** remains the default for #**SAGE**. The inclusion of label pcodes can be turned on/off throughout the SPL source file.

#FIXED

The fixed data type directive commands the SPL compiler to generate a fixed point data type when it encounters a number with a decimal point, e.g., 1.234. The data type is determined by the number of digits and whether a sign is included. So, 1.234 results in a data type 0F8H and -1.23 results in a data type 0FBH. Note that numbers that are expressed in exponent form, e.g., 1.2E-2 **always** result in floating point data types.

#FLOAT

The floating point data type directive commands the SPL compiler to generate a floating point data type when it encounters a number with a decimal point, e.g., 1.234.

The default mode is **#FIXED**.

#SAGE
#SOLODX
#SOLOGX
#GPC

The **#SAGE, #SOLODX, #SOLOGX** and **#GPC** commands identify the target platform for the resulting Program Logic Block (PLB). A summary of statements, terms, operators and features supported by the compiler for each target is shown in Table 1-1. The **#SAGE, #SOLODX, #SOLOGX** and **#GPC** commands can appear any place in the SPL source file, but it is recommended that they appear early in the source file, i.e., directly after the **#NOLIST** and/or **#PLBxx** commands if they are included. If no target option is used, **#SAGE** target option is used as the default.

NOTE

When writing a program for a GC1 controller, the **#SOLOGX** command should be used to specify the target platform.

When it executes, the SPL compiler requires a block of RAM in addition to what the executable SPL module uses. The additional block is used to build the PLB. The amount required depends on the maximum anticipated size of the PLB. The **#PLB08K, #PLB16K** and **#PLB64K** commands set the maximum size of the PLB generated by the SPL compiler at 8192-x, 16384-x and 65535-x bytes respectively (where x = 142 bytes). Using **#PLB08K** or **#PLB16K** when it is known that the PLB is less than 8k bytes and 16k bytes allocates smaller amounts of RAM from the free-space list in the SAGE-resident version the SPL compiler and allows the PC-based version to execute in a smaller amount of RAM. If present, the **#PLBxx** statement must be on the second line of the source file if there is a **#NOLIST** command, otherwise it must be on the very first line. There can be no comment lines prior to the **#NOLIST** and/or **#PLBxx** statements. **#PLB64K** is the default if there are no **#PLBxx** commands. Unless the program size is known to be greater than 8K, you should use **#PLB08K** at the beginning of every **#SAGE** based program.

#ONESEC
#ENDONESEC

The **#ONESEC** and **#ENDONESEC** statements are valid only for **#SOLOGX** and **#GPC**. They are used to define the extent of a once-a-second routine for use during program execution. There can be at most a single pair of **#ONESEC/#ENDONESEC** statements per SPL source. If the **#ONESEC** statement is present and the **#ENDONESEC** statement is not, then the once-a-second routine is assumed to extend to the end of the PLB. There is a set of eight general purpose 32-bit registers (A-H) available for use inside the once-a-second routine. In the **#SOLOGX**, these registers named A-H, are a separate set from the main program registers (A-P). The **#ONESEC** command causes the SPL compiler to make an entry in the PLB header identifying the starting offset of the once-a-second routine for use by the SPL program executor. It also causes the complier to add a STOP pcode directly preceding the **#ONESEC** statement. The **#ENDONESEC** statement causes the SPL compiler to automatically insert a RETURN pcode directly preceding it.

Table Table 1-1 shows which compiler control statements are compatible with the different target platforms.

Table 1-1 Compiler Control Statements

Statement	SAGE	DX	GX	GPC
#ENDONESEC			✓	✓
#FIXED	✓	✓	✓	✓
#FLOAT	✓			✓
#LABELS	✓	✓	✓	✓
#NOLABELS	✓	✓	✓	✓
#NOLIST	✓	✓	✓	✓
#ONESEC			✓	✓
#PAGE *length,width*	✓	✓	✓	✓
#PLB08K	✓			
#PLB16K	✓			
#PLB64K	✓			
#SAGE	✓			
#SOLODX		✓		
#SOLOGX			✓	
#GPC				✓
#TITLE "*titletext*"	✓	✓	✓	✓

1.9 COMMENTS

All lines in SPL programs that have a semicolon (;) in the leftmost column are comments. They are for documentation purposes only and are ignored at compile time. The generous use of comment statements within your programs will make them more readable and easier to troubleshoot, especially if *you* are not the person doing the troubleshooting.

As a guideline, the top lines of programs should be reserved for program identification comments. This area may contain information such as:

▾ the name of the program
▾ the date the program was written
▾ the name of the author
▾ what the program does
▾ the meaning/use of program attributes
▾ the meaning/use of program registers
▾ any assumptions made by the author
▾ any input variables used by the program
▾ any output values calculated by the program
▾ an *edit trail* indicating any modifications made to the program logic, when they were made, and by whom
▾ in general, any information that may prove useful to someone looking at the program for the first time

In addition to using comments at the beginning of your program logic, it is also helpful to create a series of comment lines prior to any program segments with logic that may need special explanation. The extra effort you take in adding useful comments to your programs is well worth the benefits you (or someone else) will reap in the future.

1.10 LABELS

SPL programs are composed of one or more *statements* which define the actions and logical operations that the program is to take when it is executed. SPL program statements are grouped into *lines* which contain a single program statement. These lines may be labeled with symbolic names to identify them. Typically this is done so that **GOTO** and other branching statements can refer to them.

Labels cannot use any of the reserved names that identify SPL statements. Labels are case-insensitive, meaning that the label **ABC** is treated the same as the label **ABc** or **abc**. Labels must begin in the leftmost column of the line. Labels may contain up to eight characters or up to eight characters and numbers. Labels can consist of the following:

- **A** through **Z**
- **a** through **z**
- **0** through **9** (not as the first character)
- **$**, **.** and _ characters.

Labels **cannot** begin with the numbers **0-9**. If a line contains any statement following the label, then the statement must be separated from the label by one or more **TAB**s or spaces. Labels may optionally end with a colon character (**:**), which is not counted in the length of the label.

The lines of an SPL program may be labeled with a symbolic name to identify that line, typically so that **GOTO** and other branching statements can refer to it. Labels may be symbolic and numeric, or symbolic and can be up to 8 characters long. Symbolic labels may not use any of the reserved names that identify SPL statements.

The following program fragment demonstrates several labels:

```
                C=\PT\TEMP2;CV
LABEL1
                A=\PT\TEMP1;CV
                B=A+3
                IF B >32 THEN LABEL003
                C=5
LABEL2:         IF ZONE;CV>72 THEN LABEL003
                SWAIT 30
                GOTO LABEL2
```

1.11 EXPRESSIONS

Expressions are symbolic formulas which represent a chain of arithmetic calculations on data from various sources. Expressions are used to convey values for parameters in many of the primitive statements in the SPL language. SPL expressions can contain an arbitrary number of *terms* and *operators* which may represent mixed mode arithmetic (i.e., integer, floating point and fixed point). SPL automatically performs type coercion on mixed mode values.

Expression evaluation is performed from left to right. Up to six levels of nesting (i.e., the use of parentheses) may be used in expressions to define an order or *precedence* for evaluation. The expression within the innermost set of parentheses is evaluated first from left to right. This procedure continues outward until the expression within the outermost set of parentheses is evaluated from left to right.

Expressions may contain constants, variables, registers, named object attributes, references, tables, built-in functions and arithmetic and logical operators.

In a very general sense, expressions are composed of terms and operators. In the simplest case, an expression is simply a term with no operators. An expression is defined as follows:

expression ::= term or
expression ::= term operator expression

An expression may also be nested within parentheses and used as a term anywhere within an expression. Up to six levels of parentheses may be used.

The syntax for a nested expression is shown below.

(expression)

The evaluation of nested expressions occurs first from the innermost set of parentheses and continues outward. Expressions that are at similar levels of nesting are simply evaluated from left to right.
The code below shows some complex SPL programming examples using nested expressions.

```
A=MAX(MEAN(B,C,D), MEAN(E,F,G))
H=SQRT((B**2)+(C**2))
;SV=((;MX-;MN)/255)*;CV+;MN
```

Because expressions may contain terms that are objects on networks, an expression does not necessarily have to be completely resolved before execution is passed to another program. Such network accessing is processed asynchronously, causing only the program using the value to be delayed until the value has been fetched. This provides for a fair method in dealing with network-intensive programs, and not penalizing other programs by waiting for a network device object value.

Terms in expressions may be one of several possible types, indicating one of several possible sources of a value to be used during expression evaluation. In general, each term has a data type as well as a value. Data types identify the way in which the values are represented numerically, and may imply additional hidden operations or coercions to be performed when arithmetic operations are required between dissimilar types. The types of terms that can be used in expressions are as follows:

- constants
- named terms
- registers
- program control attributes
- user-defined program attributes
- named object attributes
- BACnet object properties ***(BACnet controllers only)***
- references
- virtual attributes
- tables
- functions
- nested expressions

Each type of expression term is explained in detail in the following sections.

1.12 CONSTANTS

Constants specify particular unique values implying a data type by their syntax. SPL supports five basic types of syntax for entering constants: integer, fixed, float, time and explicit type.

Integer types can be expressed in decimal, binary or hexadecimal radix. Decimal numbers are a sequence of one or more decimal digits, optionally preceded by a unary minus (**-**) which indicates twos complement negation of the constant.

Hexadecimal (hex) numbers must be preceded with a leading zero if their most significant digit is **A**-**F**. Hex constants end in the letter **H** or **h** to denote hexadecimal format. Binary constants end with the letter **B** or **b** and use the digits **1** and **0**.

The range of values for integer types is 0 to 4*10^9 for unsigned integers and -2*10^9 to 2*10^9 for signed integers.

Examples of integer constants are:

Decimal:	2	27	-6
Hex:	03Eh	0FFH	2Ah
	0x3E	0xFF	0x2A
Binary:	10001110B	11B0	10b

Float constants can be expressed in free form or in scientific notation. Float constants are distinguished from similar integer constants by the presence of a decimal point (**.**) in the sequence of digits, and/or the presence of the exponent indicator (**E** or **e**). The general form of float constants is ±***ddd.dddE***±***ddd*** or ±***ddd.ddd*** where ***ddd*** represents zero or more decimal digits, and the ± represents an optional plus sign or minus sign. At least one decimal digit must be present before the decimal point. The range of float values is from ±1.175494E-38 to 3.402823E+38 with 6-7 significant digits.

Since both fixed and float types can be expressed in free-form notation, the SPL compiler must be told which of the two types to use when it encounters a constant with a decimal point. By default, all free-form constants are fixed types and all scientific form constants are float types. The **#FLOAT** compiler command can be used so that all free-form and scientific form constants are expressed as float types.

Examples of float constants are shown below:

Fixed: 1.6 1234.5678 -.45 -29.60.0 40.123456789

Note that integers are simply fixed types of the form ±*ddd.*

Float constants can be expressed in free-form or scientific notation. Float constants are distinguished from similar integer constants by the presence of a decimal point '**.**' in the sequence of decimal digits, and/or the presence of the exponent indicator '**E**' in upper or lower case. So the general form of float constants is ±*ddd.dddE*±*ddd* where the *ddd* represents zero or more decimal digits, and the **±** represents an optional plus or minus sign. At least one digit must be present before or after the decimal point, but it is not required to have a digit on both sides of the decimal point. Examples of float constants are:

Float:	1.6	1234.5678	-.45
	29.6	0.04	1e6
	102e3	0.45e-5	+39.
	1.056e+10	12.E10	

The range of float values is 8.43*10^{-37} to 3.37*10^{38} with 6-7 significant digits.

Time constants can be expressed in *hours:minutes* or *hours:minutes:seconds* forms. They are distinguished from integer constants by the presence of the colon (**:**) in the string of decimal digits. Examples of time constants are:

Short Time:	10:45	00:00	1:38
	12:5 (means 12:05)		
Long Time:	12:34:56	00:00:00	3:45:20
	12:4:5 (means 12:04:05)		

1.13 NAMED CONSTANTS

There are a number of constants which are given names and are recognized by the SPL compiler. These *named terms* can be used as if they were constants anywhere that a term may be used. Named terms are identified by reserved names in SPL and are explained in the following paragraphs. In the figures that follow, sample SPL program segments are included to illustrate the use of the named terms. The segments vary in their range of complexity.

Table 1-2 Named Constants

Name	Meaning	Value or Range	Type	SAGE	DX	GX	GPC
FALSE	logical false state	0	integer	✓	✓	✓	✓
TRUE	logical true state	1	integer	✓	✓	✓	✓
PI	value of π	3.141593	float	✓			✓
DAY	current day of month	1..31	integer	✓	✓	✓	✓
MONTH	current month of year	1..12	integer	✓	✓	✓	✓
YEAR	current year	e.g. 1990	integer	✓			✓
DATE	current date	-	date	✓	✓	✓	✓
DAYOFWEEK	current day of week	0..6 (0=Monday)	integer	✓	✓	✓	✓
DAYOFYEAR	current julian day	1..366	integer	✓	✓	✓	✓
TIME	current time	00:00:00..23:59:59	time	✓	✓	✓	✓
MON	Monday	0	integer	✓	✓	✓	✓
TUE	Tuesday	1	integer	✓	✓	✓	✓
WED	Wednesday	2	integer	✓	✓	✓	✓
THU	Thursday	3	integer	✓	✓	✓	✓
FRI	Friday	4	integer	✓	✓	✓	✓
SAT	Saturday	5	integer	✓	✓	✓	✓
SUN	Sunday	6	integer	✓	✓	✓	✓
KE	*e*	2.718287	float	✓			✓
CLEAN	have autosave attrs. changed	0..1	integer	✓			

TRUE and **FALSE** are named terms that represent the two logical states *true* and *false*. These named terms are integer types with associated numeric values of 1 and 0 respectively. Program examples of these named terms are shown below:

```
A = FALSE
Check: IF [MX_SEC1;V0] ==1 Then Merge
A = TRUE
IF [MX_SEC1;V1] ==0 Then Merge
A = TRUE
    :
Merge: IF NOT A Then Check
```

PI is a named term that represents the irrational value of pi (π). This constant is the ratio of the circumference of any circle to its radius and is approximated in the SAGE^{MAX} by using the floating point value 3.141593. An example of a program using the named term **PI** is shown below:

```
A=9.0E0         Output:         A = 9.0   (the radius of the duct)
B=PI*(A*A)                      B = 254.469  (the area of the duct)
```

DAY is a named term that represents the current day of the current month as an integer from 1-31.

MONTH is a named term that represents the current month of the year as an integer from 1-12.

YEAR is a named term that represents the current year as an integer, e.g., 2005.

DATE is a named term that represents the current date (e.g., JAN 1, 2005).

The example below shows the **DAY**, **MONTH**, **YEAR** and **DATE** named terms and an SPL programming example using **DAY** and **MONTH:**

```
START:          WAIT MONTH == 7
                WAIT DAY == 4
                CALL HLDY\JULY4TH
                WAIT DAY <> 4
                GOTO START
```

DAYOFWEEK is a named term that represents the current day of the week as an integer from 0-6. The value 0 refers to Monday, 1 refers to Tuesday, etc.

```
                If (DAYOFWEEK==SAT) OR (DAYOFWEEK==SUN) Then WKND Else L2
WKND:           Call WEEK_END
L2:
```

DAYOFYEAR is a named term that represents the current Julian day of the current year as an integer from 1-366. A simple SPL program segment using the named term **DAYOFYEAR** is shown below:

```
A = ( (DAYOFYEAR-1) / 7)+1
```

TIME is a named term that represents the current time in long time format (**HH:MM:SS**) from 00:00:00-23:59:59. A simple SPL program segment using the named term **TIME** is shown below:

```
WAIT TIME == 23:00:00
A = [BLDG1_STATUS;BO]
```

MON, **TUE**, **WED**, **THU**, **FRI**, **SAT** and **SUN** are named terms that represent each of the seven days of the week as an integer from 0-6. The value 0 refers to **MON** (Monday), 1 refers to **TUE** (Tuesday), etc. The example below shows the named terms **MON-SUN** and the named term **DAYOFWEEK** in addition to an SPL program using these named terms.

KE is the natural logarithm base, 2.718287, expressed as a floating point value.

The term **CLEAN** is used to determine if any of the autosave program attributes have been changed. This can be used in the following manner:

```
            IF CLEAN THEN NOTDIRTY
            SAVE
NOTDIRTY:
```

The **SAVE** operation automatically sets the **CLEAN** indicator true.

All of the named constants that are supported by SPL are listed in Table 1-2.

1.14 REGISTERS

For each program, there are 16 arithmetic registers available for storage of temporary values, counters, loop indices and other applications. The registers are named for the first 16 letters of the alphabet (**A** - **P**). Each program register has a 32-bit value and a data type that is determined automatically.

When you access a program's registers from within the program itself, only the register name (A-P) is required.

A program's registers are accessible outside the program itself as named attributes of the program object (e.g., **%A**, **%B**, **%C**, etc.). Likewise, if you access (i.e., *read from* or *write to*) another program's registers, you must use the percent sign as in **PROG1;%A**. This is one of four possible formats for accessing named object attributes.

The example below shows a sample program segment using program registers from within a program and from other programs

```
A = -5
B = [PROGRAM9;%P]*5.0
C = A + B /10.0
[PROGRAM4;%F] = F
D = PROGRAM3;%D
```

1.15 PROGRAM ATTRIBUTES

Each SPL program can have up to 255 unique, case-sensitive, user-defined program attributes. Each attribute has a two-character attribute name, a data type and a 32-bit value. Program attribute names can consist of any characters, but cannot begin with **%** or **$**.

When you select an attribute name, choose one that has some mnemonic significance, that is, one that reflects the use or meaning of the attribute (e.g., **CV** for current value, **MN** for minimum, **TI** for time, etc.). This increases the readability of your program and will aid in the process of troubleshooting in the future.

When working with PUP controllers the data type of program attributes can be any of the standard data types.

Before you reference a user-defined program attribute within an SPL program, you must first *declare* the attribute by using the **ATTR** statement. The **ATTR** statement contains the attribute name and a data type argument.

See *Section 2.2.1*: *ATTR Statement* for more information on using the **ATTR** statement to declare user-defined program attributes.

A program's own attributes should be referenced in the program logic using their short form (**[;AT]** or **;AT**) since this results in the fastest execution of the program. A program can reference its own attributes by their full program attribute name (**[pgname;AT]** or **pgname;AT**) however this can result in much slower execution of the program. The example below shows how to declare user-defined program attributes and illustrates some simple examples using program attributes.

```
ATTR CV,0FDh
ATTR KW,0E0h
ATTR TI,231
ATTR SZ,249

;CV = (ZONE_TEMP;CV - 32.0)*5.0/9.0
A= [KW] + 1.0E3
;TI = TIME
;SZ = 16.358
B = ;CV - ([PROG7;SP]/100)
```

1.16 NAMED OBJECT ATTRIBUTES (SAGE)

Attributes of other objects including points, variables, globals and other programs may be referenced as terms of expressions. The object name and attribute are specified using the same syntax as the Program Reference Block. The example below illustrates the formats available for named object attributes using SPL program segments.

```
A = \PT\HEAT_VALVE;CV
B = BLDG5_ROOM1;ZT + [\VR\OFFSET VALVE]
C = $MODE
[ 7WEST_SETPT;MN] = (C+[SETPOINT B]) / 2
```

For compatibility with RPL programs, the object name and attribute may be optionally enclosed in square brackets, e.g., **[objectname;attribute]**. Object names that begin with **0** to **9** and/or contain one or more spaces **must** be enclosed in square brackets.

Named object attributes may begin with a 2-character object type code. When used, this code is delimited by backslash characters, i.e. \PT\. The named object type codes are:

- PT for points
- VR for variables
- GL for globals
- PG for programs
- MS for MSTP Objects
- BN for BACnet IP or BACnet Ethernet objects

If a named object type code is not specified, the SAGE^{MAX} performs an exhaustive database search for the named object. The search is conducted in a standard order starting with points and followed by programs, variables and globals.

If no attribute is specified in the named object, the default attribute is assumed. For variables (which have a single value), it is not necessary to specify an attribute, since variables do not have attributes. For points and globals, the default attribute is the first attribute that appears when you monitor the point. For programs, the default attribute refers to the user-defined program attribute that is defined first in the program. If no user-defined program attributes are declared or if the program is unloaded, the **$$** control attribute is the default attribute.

1.17 NAMED PUP OBJECT ATTRIBUTES (GX, DX, GPC)

The **#SOLODX, #SOLOGX** and **#GPC** targets support numeric PUP channel and attribute specification using the syntax: **[*pupchannel;attribute*].** The **#SOLODX** and **#GPC** also support this form for specific PUP peer devices identified by their unit number: **[U*unit_pupchannel;attribute*].** Here are some examples:

[F902;AB] attribute AB, pupchannel F902
[F902] pupchannel F902, default attribute
[U100_F902;AB] attribute AB, pupchannel F902 in unit 100
[U100_F902] default attribute, pupchannel F902 in unit 100

1.17.1 NAMED BACNET OBJECT PROPERTIES (GPC)

The **#GPC** also supports access to BACnet object properties enclosed in square brackets. A period is used to delimit the fields of the name:

[.propertyname] a property of this program object
[objecttypeinstance.propertyname] a property of an object in this device
[device.objecttypeinstance.propertyname] a property of an object in another device

The *propertyname* could be a well-known property name like **CV**, or a numeric property identifier. The *objectypeinstance* would use well known mnemonics for standard object types followed by decimal instance numbers, e.g. **AV27.** If the *objectypeinstance* was numeric then it would be the *objectidentifier*. Here are some examples:

[.85]	the **present_value** of this program object
[.2005]	proprietary property 2005 of this program object
[AV27.]	the **present_value** of object AV27
[BV6.change_of_state_count]	**change_of_state_count** for object BV6
[BV6.15]	same as previous
[0x400005.85]	AO5 **present_value**
[123.BO29.present_value]	the **present_value** for object BO29 on device 123

The SPL compiler derives the mnemonics for standard object type names (AV, BO etc.) and well-known property identifiers from sections of its INI file. This allows even proprietary object types and property IDs to use a human-friendlier syntax.

1.18 TABLES (SAGE)

Tables are collections of up to 1,073,741,824 data values which are stored as linear, one-dimensional arrays in disk-resident files. These files have a maximum size of 4,294,967,296 bytes.

SPL recognizes two types of tables: SPL tables and non-SPL tables. All SPL table files are stored in the reserved directory **C:\TABLES**. SPL table files may be stored in this directory or in any of its subdirectories and have the extension **.TBL**. A typical table file might have the full path name **C:\TABLES\XXX\YYY.TBL** which would be represented in an SPL statement by the path fragment **XXX\YYY**.

Individual SPL table terms may be referenced as terms in expressions using the syntax below.

&*tablefragment*(*expression*)

The argument expression in parentheses is evaluated to determine a zero-based index into the table. The resulting index is used to determine which data value to read from the table file. If the result of the index expression is greater than the actual number of values in the table file, an expression evaluation error occurs at run time. Examples of SPL table references are shown below.

```
&TABLE6(A+5)
&MYTABLE(4)
&LOOKUP\CLAIREX(D)
```

To access non-SPL tables (e.g., trend files), the entire pathname for the table must be specified using the following syntax.

&*tablepath*(*expression*)

This is done by placing a backslash character (\) as the first character after the **&** or a colon (**:**) as the second character after the **&**. Examples of non-SPL table references are shown below.

```
&\TREND\TREND1.TRN (J)
&C:\TREND\TREND2.TRN (K+L)
```

The code below shows an SPL programming example of the use of table references:

```
                A=100
                B=0
L1:             B=B+&TABLE3(A-1)
                LOOP A,L1
```

1.18.1 RAM-BASED TABLES

RAM-based tables are supported for all compiler target options. RAM-based table terms are differentiated from file-based table terms by a double ampersand (e.g. ***&&RAMTABL(expression)***) rather than a single ampersand (e.g. &FILETABL). RAM-based table names follow the same rules as symbols and labels and can contain up to 16 characters excluding the two ampersands. RAM-based tables are declared in the following manner:

TABLE name *(size, type)*

where:

name is the table name (without ampersands)

size is number of entries in the table

type id the data type of the table

RAM-based tables can be initialized by DATA statements directly following the TABLE declaration:

```
TABLE name (size, type)
DATA v1,v2,v3,v4....
DATA vn, vn+1....
```

The entire table can be initialized to the same value using the following syntax when declaring it:

```
TABLE name (size, type) = value
```

Table elements not specifically initialized are filled with 0's.

RAM-based tables can be declared anywhere within the body of the SPL source. Each TABLE declaration that is not preceded by a TABLE or DATA statement causes the SPL compiler to insert a STOP statement (or #ENDONESEC for a SOLO/GX or GPC with an open #ONESEC routine) directly preceding the TABLE declaration.

1.19 FUNCTIONS

Functions are procedures which operate on one or more arguments and produce a single value result. SPL provides a number of built-in functions, any of which may be used as terms in any expression. arithmetic functions. The arguments to these functions can be integer or fixed expressions (**ix**), floating expression (**fx**), integer, fixed or floating expressions (**x**) or time expressions (**tx**). All of the functions available through SPL are summarized in Table 1-3. Also included is the datatype of the returned value as well as a listing of what target platforms may use the function.

Table 1-3 SPL Functions

Function	Description	Returned Data Type	SAGE	DX	GX	GPC
RETYPE(x1,x2)	Convert to a specified datatype	x2	✓	✓	✓	✓
FIX(fx,ix)	Convert float to fixed	fixed	✓			✓
FLOAT(ix)	Convert integer to float	float	✓			✓
INT(x)	Convert to integer	integer	✓			✓
ROUND(x)	Round off	fixed or float	✓	✓		✓
ABS(x)	Absolute value	integer or float	✓	✓	✓	✓
SIN(x)	Sine value	float	✓			✓
COS(x)	Cosine value	float	✓			✓
TAN(x)	Tangent value	float	✓			✓
ARCTAN(x)	Arctangent value	float	✓			✓
LOG(x)	Logarithm	float	✓			✓
LN(x)	Natural logarithm	float	✓			✓
EXP(x)	e^x	float	✓			✓
SQRT(x)	Square root	integer or float	✓	✓‡	✓‡	✓
BETWEEN(tx,tx)	Between two times	integer, 0=not between 1=between	✓	✓	✓	✓
MEAN(x1,x2,..x8)	Mean value from list	integer or float	✓	✓		✓
MAX(x1,x2,..x8)	Maximum value from list	integer or float	✓	✓		✓
MIN(x1,x2,..x8)	Minimum value from list	integer or float	✓	✓		✓
TODAY(ix)	Compare today's day of week	integer, 0=no match 1=match with a bitmap pattern	✓	✓	✓	✓
DATAYPE(x)	Datatype of term	integer	✓	✓	✓	✓

‡ For the DX and GX target platforms, SQRT may only be used with Integer arguments and will return an integer result.

The **RETYPE** function is used to convert the data type of an expression to a specified data type. This function uses two expressions. The first expression represents the value to be converted. The second expression represents the new data type of the conversion. The code below shows examples of the **RETYPE** function in simple SPL program segments.

```
[MX_FE00;MX] = RETYPE (A,0FDh)
OAT1;CV = RETYPE (B,251)
```

The **FIX** function is used to convert the data type of an expression from floating point (E0h) to a fixed type. This function uses two expressions. The first expression represents the floating point value to be converted. If the second expression **ix** evaluates to a number from 1-10, then the result is a signed fixed data type with **ix** digits to the right of the decimal point. If the second expression evaluates to zero, then the result has the appropriate number of digits to represent the floating point value in a fixed format. If the second expression **ix** evaluates to a number greater than 10, it represents the desired data type, e.g., **FIX(XYZ,2)** is the same as **FIX(XYZ,0FAh)**.

The code below shows examples of the **FIX** function and examples in simple SPL program segments

```
A=1.06E4         Output:          A=1.06E4       (float)
B=1                               B=1            (integer)
C=FIX (A,B)                       C=10600.0      (fixed)
D=3.4567E-1                       D=3.4567E-1    (float)
E=FIX (D,B-1)                     E=0.34567      (fixed)
F=FIX (D,0FDh)                    F=0.3          (fixed)
```

The **FLOAT** function is used to convert an integer or fixed point value to a floating point value (E0h). This function uses a single expression to represent the integer or fixed point value that is to be converted to floating point. The code below shows an example of the **FLOAT** function using simple SPL statements.

```
A=421            Output:          A=421          (integer)
B=FLOAT(A+100)                    B=5.21E2       (float)
```

The **INT** function is used to convert a floating or fixed point value to an integer that is less than or equal to the value of the specified expression. The resulting data type after using the **INT** function is always either 0FEh or 0FFh. Simple **INT** SPL program examples are shown in the code below:

```
A=7.5            Output:          A=7.5
B=-3.1                            B=-3.1
C=-1.75E0                         C=-1.75E0
D=INT(A)                          D=7
E=INT(B*2)                        E=-7
F=INT(C)                          F=-2
```

The **ROUND** function is used to round off an integer, fixed or floating point value to the closest whole number. The result is either a fixed (for integer or fixed point values) or a floating point number (for floating point values). This function has a single expression **x** which represents the value to be rounded. The code below shows examples of the **ROUND** function using simple SPL statements.

```
A=7.5           Output:         A=7.5
B=-3.1                          B=-3.1
C=-1.75E0                       C=-1.75E0
D=ROUND(A)                      D=8.0
E=ROUND(B*2)                    E=-6.0
F=ROUND(C)                      F=-2.0E0
```

The **ABS** function is used to get the absolute value of an integer, fixed or floating point value. This function has a single expression which represents the input value to the absolute value function. The result is either an integer, fixed or floating point value. Simple **ABS** SPL program examples are shown below:

```
A=7.5           Output:         A=7.5
B=-5.1                          B=-5.1
C=-3.21E0                       C=-3.21E0
D=ABS(A)                        D=7.5
E=ABS(B)                        E=5.1
F=ABS(C)                        F=3.21E0
```

The **SIN** function is used to calculate the sine value of an expression. This function uses a single expression which represents the input value in *radians* (2π radians = 360 °). The result is always in floating point format. The code below shows an example of the **SIN** function:

```
C=5             Output:         A=2.5E0
A=C*SIN(PI/6)
```

The **COS** function is used to calculate the cosine value of an expression. This function uses a single expression which represents the input value in *radians* (2π radians = 360 °). The result is always in floating point format. The code below shows an example of the **COS** function:

```
C=5             Output:         A=2.5E0
A=C*COS(PI/3)
```

The **TAN** function is used to calculate the tangent value of an expression. This function uses a single expression which represents the input value in *radians* (2π radians = 360°). The result is always in floating point format. The code below shows an example of the **TAN** function:

```
Input: angle=45° (pi/4 radians)Output:   B=3.0E0

A=3
B=A*TAN(PI/4)
```

NOTE

As the expression used in the TAN function approaches 90° (1.5707965 = π/2 radians) and 270° (4.7123895 = 3π/2 radians), the limits of the TAN function approach positive infinity (+∞) and negative infinity (-∞) respectively. These values are displayed as +INF and -INF when you use the TAN function.

The named term PI is most likely used in the expression argument for the SIN, COS and TAN functions.

The **ARCTAN** function is used to calculate the arctangent (the inverse function of the tangent) value of an expression. This function uses a single expression which represents the tangent (**TAN**) value. The **ARCTAN** function determines an angular value in radians (2π radians = 360 °) whose tangent is the value specified by the input expression. The result is in floating point format and represents an angular value in radians. The code below shows a simple SPL program using the **ARCTAN** function:

```
Input: A=TANvalue=1.0E0Output:     B=7.853982E-1 (angle in radians)
                                   C=45 (angle size in degrees
A=1.0E0
B=ARCTAN(A)
C=B/(PI/180)
```

The **LOG** function is used to calculate the logarithm (base 10) of an expression. This function uses a single expression in integer, fixed or floating point format. The result of the **LOG** function is in floating point format.

The **LN** function is used to calculate the natural logarithm of an expression. This function uses a single expression in floating point format. The result of the **LN** function is in floating point format.

The code below shows examples of both the **LOG** and **LN** functions.

```
A=1.0E2         Output:        A=1.0E2
B=LOG(A)                       B=2.0E0         (LOG(X))
C=LN(A)                        C=4.6051702E0   (LN(X))
```

The **SQRT** function is used to calculate the positive square root of positive expressions (values greater than or equal to zero). This function uses one input expression that can be an integer, fixed or floating point type. The type of the result is either integer or floating point depending on the type of the input argument. The code below shows an example of the **SQRT** function in an SPL program:

```
A=1000.0        Output:         A=1.0E3
B=SQRT(A)                       B=3.162277E1
```

NOTE

On DX1 and GX1 controllers, the SQRT function may only be used with integer arguments and will return and integer result.

The **BETWEEN** function is used to determine if the current time of day is between two specified times. This function has two input expressions which are separated by a comma. These expressions represent the two times which must be in a time format. The integer result is either **0** (**false**) if the current time is not between the two times specified, or **1** (**true**) if the current time is between the two specified times. An example is shown below:

```
L20:        A=7:00          Output:     A=7:00   (1st time argument)
            B=21:00                     B=21:00 (2nd time argument)
            C=BETWEEN (A,B)             C=1 if current time is between A & B
                                        C=0 if current time is not between A & B
```

NOTE

When you use the BETWEEN function, the value of argument tx1 must be less than the value of argument tx2.

The **MEAN** function is used to determine the average value of a list of up to 8 expressions. The mean is calculated by summing the values of the expressions and then dividing by the number of expressions. The expressions used by this function can mix integer, fixed and floating point data types in the same statement. The input expressions are separated by commas. The result is the average value of the expressions listed, but may have a data type that is different than the input expressions. This is due to the calculation of the *average* value, in which the data type coercion rules for addition and division apply. Depending on your application, you may have to use the **RETYPE** function to get the data type that you desire. **RETYPE** was discussed earlier in this section.

The code below shows an example of the **MEAN** function using simple SPL program statements:

```
A=10.0          Output:         G=(A+B+C+D+E+F)/6=63.6
B=98.0
C=75.0          E=91.2
```

```
D=62.5          F=44.9

G=MEAN(A,B,C,D,E,F)
```

The **MAX** function is used to determine the maximum value of a list of up to 8 integer, fixed or floating point expressions. The arguments are separated by commas. The resulting data type is the data type of the maximum value. The code below shows an example of the **MAX** function using simple SPL program statements:

```
A=10.0          Output:          G=98.0
B=98.0
C=75.0          E=91.2
D=62.5          F=44.9

G=MAX(A,B,C,D,E,F)
```

The **MIN** function is used to determine the minimum value of a list of up to 8 integer, fixed or floating point expressions. The arguments are separated by commas. The resulting data type is the data type of the minimum value. The code below shows an example of the **MIN** function:

```
A=10.0          Output:          G=10
B=98.0
C=75.0          E=91.2
D=62.5          F=44.9

G=MIN(A,B,C,D,E,F)
```

The **TODAY** function determines whether or not the current day of the week is part of a day-of-the-week bitmap pattern specified by its integer expression. The result of this function is in integer format and is **0** (**false**) if there is no match between the current day and the expression bitmap, and **1** (**true**) if there is a match between the current day and expression bitmap.

To determine whether or not the match exists, the **TODAY** function logically **AND**s a day-of-the-week mask with the argument. The day-of-the-week masks are shown below:

▼ Monday	0000 0001b (1)
▼ Tuesday	0000 0010b (2)
▼ Wednesday	0000 0100b (4)
▼ Thursday	0000 1000b (8)
▼ Friday	0001 0000b (16)
▼ Saturday	0010 0000b (32)
▼ Sunday	0100 0000b (64)
▼ Holiday	1000 0000b (128)

The code below illustrates the **TODAY** function in an SPL program example:

```
                ATTR AD,0E9h
                      :
                ;AD=00011111b
Check:          IF TODAY(;AD) Then Do_It
                MWAIT 1
                GOTO Check

Do_it:          B=100
```

The **DATATYPE** function is used to determine the data type of an expression that you specify as the argument. The result of this function is in integer format. The code below illustrates an SPL example using the **DATATYPE** function:

```
D=DATATYPE(MX_FE01;CV)
      :
C=A*1.035-5
      :
MX_FE01;CV=RETYPE(C,D)
      :
```

1.20 EXPRESSION OPERATORS

An arithmetic expression in SPL may be a simple term, or may be made up of a sequence of terms and operators. SPL provides both unary and binary operators. Unary operators have precedence over binary operators, so any term may be preceded by zero or more unary operators which are evaluated from left to right. Binary operators are always evaluated from left to right with no implied precedence. You must use parentheses to set operator precedence.

Table 1-4 lists all the unary and binary operators available in SPL. Included with each operator is a description of the function of the operator, a sample expression using the operator, and notes relating to the use of the operator.

Table 1-4 Expression Operators

Operator	Description	Example	Notes
-	Unary negation	**-A**	same as **0-A**
NOT	Unary ones complement	**NOT A**	32 bit integer
**	Exponential‡	**A**B**	**A** to the **B** power
*	Multiplication	**A * B**	
/	Division	**A / B**	
MOD	Remainder after division	**A MOD B**	**7 MOD 3** = 1
+	Addition	**A + B**	
-	Subtraction	**A - B**	
==	Equality	**A == B**	0 if not, 1 if equal
<>	Inequality	**A<>B**	1 if not, 0 if equal
>	Greater than	**A > B**	1 if **A>B**, else 0
>=	Greater than or equal	**A>=B**	1 if **A>B** or **A=B**
<	Less than	**A < B**	1 if **A<B**, else 0
<=	Less than or equal	**A<=B**	1 if **A<B** or **A=B**
AND	Bitwise And	**A AND B**	32 bit integer
OR	Bitwise Or	**A OR B**	32 bit integer
XOR	Bitwise Exclusive-Or	**A XOR B**	32 bit integer
SHL	Bitwise Shift Left	**A SHL B**	**1 SHL 3** = 8
SHR	Bitwise Shift Right	**A SHR B**	**256 SHR 7** = 2

‡ The exponential operator is only available on the SAGE and GPC controllers.

NOTE

Since operators are evaluated from left to right with no precedence, you must use parentheses to indicate precedence for operations. For example:

```
    12.34 + 45.7 / 2.0 + 2.1    = 31.12   and
    12.34 + 45.7 / (2.0 + 2.1) = 14.15609756.
```

NOTE

Logical expressions must be enclosed in parentheses for proper evaluation. For example:

```
 IF (DAYOFWEEK==SAT) OR
      (DAYOFWEEK==SAT) THEN ...
```

SECTION 2: SPL PROGRAM STATEMENTS

IN THIS SECTION

Introduction 2-3
Working with Attributes 2-5
- ATTR Statement 2-5
- TABLE and DATA Statements 2-5
- SAVE Statement (SAGE) 2-6

Assignment Statements and Equates 2-8
References 2-10
- DREF Statement 2-10
- REF Statement 2-10

Virtual Attributes 2-11
Iteration, Branching and Subroutines 2-12
- GOTO statement 2-12
- IF... THEN... {ELSE...} Statement 2-12
- ON... GOTO... statement 2-13
- LOOP Statement 2-13
- GOSUB Statement 2-14
- CALL Statement 2-14
- RETURN Statement 2-15

Program Delays 2-16
- SWAIT and MWAIT Statements 2-16
- WAIT Statement 2-16

Printing, Logging, and Alarms 2-17
- PRINT Statement (SAGE) 2-17
- LOG Statement (SAGE) 2-19
- ALARM Statement 2-20

Job Execution (SAGE) 2-22
- The REPORT Job 2-22
- The SPOOL Job 2-25
- The DATA CAPTURE /DATA STUFF Job 2-27
- The UPLOAD / DOWNLOAD FILE Job 2-31
- The EXPORT DATABASE FILE Job 2-32

Spooling Report and Log Files (SAGE) 2-35
- SPOOL Statement 2-35

Trending Control Statements (SAGE) 2-36
- STARTTREND & STOPTREND Statements 2-36

Program Execution Statements (SAGE) 2-37
- ACTIVATE Statement 2-37
- DEACTIVATE Statement 2-37
- The RESTART Statement 2-37
- STOP Statement 2-38
- UNLOAD Statement 2-38

Execution Error Control 2-40
- ERRORABORT Statement 2-40
- ERRORWAIT Statement 2-40
- ONERROR Statement 2-41

Debugging Statements 2-42
- SECTION Statement 2-42
- NOP Statement 2-42

Program Control Attributes 2-43

2.1 INTRODUCTION

This section is intended to familiarize you with all of the SPL programming statements by organizing them into logical groups based on the functions that they perform. Program statements fall into the following categories:

- attribute definitions and references
- assignment statements
- iteration control, program branching and subroutines
- program delays
- printing, logging and alarming
- job execution
- spooling
- trending control
- program execution control
- execution error control
- debugging statements

Attribute definitions and references are used to declare user-defined program attributes and save the values of attributes to the program's attribute initial value (INI) file.

Assignment statements are used to assign the value of an expression to a variable. This type of program statement is characterized by the use of an equal sign (=).

Iteration control, program branching and subroutines are statements perform a statement or group of statements some number of times, change the order in which the logic is executed, or transfer program control to another portion of the program (a subroutine).

Program delays are statements which suspend program execution, either for a set amount of time or until certain conditions are met.

Printing, logging and alarming refers to statements that give you the ability to print information to a port, log information to a file, or generate formatted alarms of definable alarm classes.

Job execution refers to statements that give you the ability to execute any SAGE^{MAX} job from within the SPL program execution environment.

Spooling refers to commands which offer the ability to send specified files to the printer.

Trending control refers to program statements that can control the execution of trends from a program.

Program execution control refers to statements that can start, stop and prepare programs to be executed.

Execution error control refers to program statements that allow you to define a course of action for PEX when network access errors occur.

Debugging statements refer to programming statements that can be used to aid in the diagnosis of program logic errors.

Each SPL programming statement is individually explained, including sample SPL statements in the following pages. All of the SPL programming statements are summarized in Table 2-1**.** To quickly look up a particular statement's syntax and arguments, see *Appendix A: SPL Language Reference*.

Table 2-1 Program Statements

Format	Description
variable = *expression*	assignment statement
ACTIVATE *progname*	start a stopped program - load if required
ALARM *classexpr*, "*formatstring*", *x,x,x...x,x,x*	generates an alarm
ATTR *progattr*,*datatype*	declare program attribute
CALL *PLBname*	go to external subprogram's logic block then load, execute and possibly unload it
CALL *PLBname*,*STICK*	same, but do not unload the program
DATA v1, v2, v2, v4	specifies data for the preceding table
DEACTIVATE *progname*	remove a program from memory (RAM)
DREF *unit, channel;AA*	defines elements for program reference block
ERRORABORT	trap condition - abort on errors
ERRORWAIT	trap condition - wait until no error
symbol EQU *expression*	symbolic equate statement
GOSUB *label*	go to internal subroutine
GOTO *label*	unconditional branch
IF *expr* THEN *label*	conditional branch if expr is true
IF *expr* THEN *label1* ELSE *label2*	conditional branches if expr is true or false
JOB *classexpr*, "*jobstring*", *x,x,x...x,x,x*	request to SAGE^{MAX} job scheduler
LOG *logfilename*, "*formatstring*",*x,x,x...x,x,x*	print (log) information to a file
LOOP *register*,*label*	iteration control
MWAIT *expression*	wait a certain amount of minutes
NOP	no operation used for debugging
ON *expression* GOTO *label0,label1...labeln*	indexed conditional branches
ONERROR *label*	trap condition - branch if error occurs
PRINT *portexpr*,*classexpr*,"*formatstring*",*x,x,x...x,x,x*	print information to a port
PROP *progproperty,BACnetDatatype*	declares a BACnet property for BACnet based devices
RESTART *progname*	start a program from beginning - load if required
RETURN	return from a subroutine
SAVE	copy all program attributes to INI file
SAVE *aa*,*bb*,*cc*,*dd*...	copy selected attributes to INI file
SECTION *number*	section marker used for debugging
SPOOL *portexpr*,*pathname*	send a file to be printed
SPOOL *portexpr*,*pathname*,DELETE	send a file to be printed, then delete it
STARTTREND *trendname*	activate a trend
STOP	halt execution of this program
STOP *progname*	halt execution of specified program
STOPTREND *trendname*	deactivate a trend
SWAIT *expression*	wait a certain amount of seconds
TABLE *name (size, type)*	initializes a RAM-based table
UNLOAD *programname*	remove this program from memory (RAM)
WAIT (*expression*)	wait until an expression is true, then go on

2.2 WORKING WITH ATTRIBUTES

2.2.1 ATTR STATEMENT

ATTR {*channel*};*attr*,*datatype*{=*initialvalue*}{,*autosave*}

where:

channel is the channel number (optional)
attr is the attribute
datatype is the data type
initialvalue is the initial value (optional)
autosave is the autosave flag

Local program attribute names and their data types are declared using the ATTR command. Although local program attributes can be declared anyplace within in the SPL text, they **must** be declared before they are referenced. It is strongly recommended that all local attributes are declared before any other SPL statements. Local program attributes that are referenced, but not declared, result in compile-time errors.

To maintain backward compatibility, the short-form of attribute declaration is supported:

ATTR *attr,datatype*

If an initial value and/or auto save indicator is required but no channel is to be defined, a semicolon (;) must precede the attribute:

User defined attributes are normally stored in the program channel in which they were created. However, the GC and GX controllers allow you to use the optional *channel* argument in the attribute declaration to specify a channel. By default, user defined attributes will be stored in channel 0000h but can be specified for any channel from 0000h-7FFFh. This can allow you to have multiple attributes with the same two-letter name, but different channel numbers. For example 1000;**AA**, 1001;**AA**, and 1002;**AA**, could all be used as valid attributes within a single program.

2.2.2 TABLE AND DATA STATEMENTS

TABLE *name* (*size, datatype*){=*value*}

where:

name is the name of the table
size is the number of entries contained in the table
datatype is the datatype
value is the value to which all cells in the table will be initialized

Tables are collections of up to 1,073,741,824 data values which are stored as linear, one-dimensional arrays. These files have a maximum size of 4,294,967,296 bytes.

SPL recognizes two types of tables: SPL tables and non-SPL tables. All SPL table files are stored in the reserved directory **C:\TABLES**. SPL table files may be stored in this directory or in any of its subdirectories and have the extension **.TBL**. A typical table file might have the full path name **C:\TABLES\XXX\YYY.TBL** which would be represented in an SPL statement by the path fragment **XXX\YYY**.

Individual SPL table terms may be referenced as terms in expressions using the syntax below.

&*tablefragment*(*expression*)

The argument *expression* in parentheses is evaluated to determine a zero-based index into the table. The resulting index is used to determine which data value to read from the table. If the result of the index expression is greater than the actual number of values in the table file, an expression evaluation error occurs at run time. Examples of SPL table references are shown below.

&TABLE6(A+5)
&MYTABLE(4)
&LOOKUP\CLAIREX(D)

To access non-SPL tables (e.g., trend files), the entire pathname for the table must be specified using the following syntax.

&*tablepath*(*expression*)

This is done by placing a backslash character (\) as the first character after the **&** or a dive letter and a colon (**:**) after the **&**. Examples of non-SPL table references are shown below.

&\TREND\TREND1.TRN (J)
&C:\TREND\TREND2.TRN (K+L)

The code below shows an SPL programming example of the use of table references:

```
            A=100
            B=0
L1:         B=B+&TABLE3(A-1)
            LOOP A,L1
```

2.2.3 SAVE STATEMENT (SAGE)

SAVE {*programattr1,programattr2,...programattr16*}

where:

programattr1,programattr2,...programattr16 are the program attributes (up to 16) which are to be saved

The **SAVE** statement is an attribute manipulation command that is used to write the values of custom program attributes to an initial value (INI) file. The **SAVE** command has two formats to accommodate saving all or selected program attribute values. **SAVE** by itself causes the current value of all program attributes to be written to the INI file. **SAVE** followed by up to 16 program attributes saves the values of only those attributes which are specified. Only those attributes specified will appear in the **.INI** file.

Regardless of the format used, the current INI file is renamed with a **.$NI** extension and a new INI file is created with the updated data.

The **SAVE** statement behaves like a **NOP** statement if no INI file is specified in the program definition. If an INI path fragment that does not currently exist is specified, PEX will create the file when the first **SAVE** is executed.

The code below illustrates the use of the **SAVE** statement in an SPL programming example:

```
ATTR HL,0FBh
ATTR LL,0FBh
ATTR CS,0FBh
ATTR HS,0FBh
        :
;CS=A+5.0
;HS=B-7.0
SAVE CS,HS
        :
```

2.3 ASSIGNMENT STATEMENTS AND EQUATES

2.3.1 STANDARD VALUE ASSIGNMENT

variable = expr

SPL allows various forms of value assignment statement. In each case, a variable on the left side of the **=** (equal sign) is assigned the new value dictated by the expression on the right side. The right side expression produces a value and a data type. Because automatic data type *coercion* may occur during evaluation, the data type of the expression may not match the data type of the variable on the left side. In this case the data type and value from the expression *may* have to be coerced into the variable's data type according to certain rules. The table below summarizes the conversions in general:

Left Side	Right Side	Effect
fixed	float	left=FIX(right)
float	fixed	left=FLOAT(right)
fixed*	fixed	left=RETYPE(right)

*Different fixed data type.

Integers and time data types are treated as fixed types. The table above does not reflect that there are 20 distinct types of fixed types, i.e. 10 decimal point positions each for signed and unsigned types.

When the left side variable is a local program attribute, coercion of the expression into the proper data type is done automatically by PEX. When the left side variable is a register, unless the data type of the result is converted according to the table above, the data type of the register is automatically changed to the data type of the result. When the left side variable is any other type of object, the result *must* be converted according to the table or incorrect values may be assigned to the variable. For example, if the left side variable is a point whose data type is F9H (xxxxxxx.xxx) and the expression has a data type of F7H (xxxxxx.xxxx) then a RETYPE (F9H) must be done so that the value assigned to the variable is not 10 times too large (in this case.)

There are several forms of assignment statements that may be used in SPL. These are summarized below:

register = expression	A = B+C
;*programattribute = expression*	;CV = B+C
namedobject = *expression*	OAT = B+C
namedobject = expression	[ZONE TEMP] = B+C
namedobject = expression	[1STFLOOR] = B+C
\*objecttype***\***namedobject = expression*	\VR\OAT = B+C
*namedobject***;***attribute = expression*	[LOOP;SP]= B+C
\*objecttype***\namedobject;***attribute = expression*	\PT\LOOP;SP = B+C
REF(*expression***)** = *expression*	REF(6) = B+C
&*tablename***(***expression***)** = *expression*	&CLAIREX(29) = B+C

UNS(*x1,x2,x3,x4,x5,attribute***)** = *expression*	UNS(1,0,0,0,FB00h,CV) = B+C

At first there would appear to be a conflict in syntax between local attribute references that are used as variables on the lefthand side of assignment statements, e.g., **;AT=***expression*, and comments since both begin with semicolons. The difference in syntax between the two is that comments begin in the leftmost column and local program attribute references used as variables must have at least one leading space or tab. In order to avoid confusion, local program attribute references can be enclosed in brackets, i.e. **[;AT]**=*expression*.

2.3.2 EQU

symbol EQU expression

EQU (Equate) provides a simple method to assign substitute names to commonly used point references in an SPL program, providing the ability to easily read and interpret an SPL program in a more basic form. **EQU** is a symbolic equate in its rawest form.

EQU statements must be defined in a program *before* they are used, because the compiler considers all terms that are not SPL keywords, numeric values or SPL symbols to be object names The symbol part of the **EQU** statement can be up to 16 characters in length, which must all be printable characters (A-Z, 0-9, !, @, etc.) and cannot begin with a digit. The right-hand side of the expression can be up to 32 printable characters and can be a programmatic expression or point data location (e.g. FE01;CV).

The code below illustrates the use of the **EQU** statement in an SPL programming example:

```
#GPC
#LABELS
;
FANSTATUS EQU FE01;CV
FANOUTPUT EQU FB02;CV

;start of program
L0:   SWAIT 1
   IF FANSTATUS==1 THEN TURNON ELSE TURNOFF
;
TURNON
   FANOUTPUT = 1
GOTO L0:
;
TURNOFF
   FANOUTPUT = 0
GOTO L0
```

2.4 REFERENCES

2.4.1 DREF STATEMENT

The **DREF** (**D**efine **REF**erence) statement is used with the **#SOLODX, #SOLOGX** and **#GPC** target options for the purpose of building Program Reference Blocks (PRBs) equivalents. **DREF** statements take the following form:

DREF *unit,channel;attr*

where:

unit is the unit number (optional)
channel is the channel number
attr is the attribute

DREF statements cause entries to be made into the attribute table at the end of the PLB, taking the place of the PRB. The attribute/reference table can contain up to 255 entries, i.e., there can be a total of 255 combined program attributes and references per PLB for the **#SOLODX, #SOLOGX** and **#GPC** target options. If **DREF** statements are used along with the **#SOLODX** option, PRBs *cannot* be used when the DX executes the program linked with the PLB. Note that the SOLO/GX and GPC *never* use PRBs, so all references must be declared using **DREF** statements. Attributes and references are added to the table in the order in which they are defined. This ordering identifies **REF** indices. Attribute and reference declarations can be mixed freely if desired. The **DREF** statement is invalid for the **#SAGE** option.

2.4.2 REF STATEMENT

The **REF** statement is used to read the values of a reference created using the **DREF** statement.

REF (*index*)

where:

index is the index number of the referenced attribute

When the **REF** statement is used, *index* is evaluated at run time to determine an index value from 0-255. The resulting index is used to determine which of 256 possible referenced values to use in place of the **REF**. If the result of the index expression is greater than 255, or greater than the actual number of references present, then an expression evaluation error occurs at run time.

```
  DREF 0FE01H;CV
; when no unit ID is defined, DREF looks locally.
DREF 15,0FE02H;CV
;
L0:    A = REF(0)
       B = REF(1)
       SWAIT 10
       GOTO L0
```

2.5 VIRTUAL ATTRIBUTES

In some applications, it may be necessary or desirable to read attribute values from a point that may not have a point object created for it. SPL provides a mechanism for using arithmetic expressions to specify each of the necessary parameters for identifying the particular point attribute that is desired. This type of access is called *virtual attribute access.* The special function **UNS** may be used as a term in any expression to accomplish reading of a point attribute value. The UNS function makes use of the SAGE's Unified Network Services for reading, hence its name. The syntax for the UNS function is:

UNS*(portexpr, unitexpr, ftypeexpr, cardexpr, chanexpr, attribute)*

In this syntax, *portexpr* is an expression whose value must be in the range 0..7 representing the SAGE port number where the point is located. *unitexpr* is an expression representing the unit number (0..65534) on that port where the point is. *ftypeexpr* is an expression representing the fundamental type code and subchannel code for the point. The fundamental type is 0..15 and the subchannel is 0..15. They are combined by the formula: ***ftypeexpr = fundamental type + (subchannel * 16).*** *cardexpr* is an expression representing the card number (0..255) for the point. *chanexpr* is an expression representing the channel number (0..65535 or FFFFh). *attribute* is not an expression, but instead is the literal attribute name as two characters.

Not all SAGE port types or protocols require all of the UNS parameters to fully qualify a point. In those cases, the other parameters may be specified as zero.

2.6 ITERATION, BRANCHING AND SUBROUTINES

2.6.1 GOTO STATEMENT

GOTO *label*

where:

label is the label of the point to which program execution will be switched

The **GOTO** statement is an unconditional branch statement that causes program logic to jump to some other location that is identified by a label.
The code below illustrates the use of the **GOTO** statement and shows a sample SPL programming example. It may increase the readability of your program logic if you add a blank line after **GOTO** statements.

```
             :
L1:     C = A+B
        GOTO L3

L2:     C = B-A
L3:     D = C*2
             :
```

2.6.2 IF... THEN... {ELSE...} STATEMENT

IF *expr* THEN *label1* {ELSE *label2*}

where:

expr is the logical expression which determines conditional branching behavior
label1 is the label to jump to if *expr* evaluates to *true*
label2 is the label to jump to if *expr* evaluates to *false* (optional)

The **IF... THEN...** statement is a conditional statement that causes the program logic to jump to some other location identified by a label if a certain condition is true. If the condition is false, execution *falls through* to the next sequential statement. If the optional **ELSE** statement is included, then program execution will jump to the label following the **ELSE** statement if the condition evaluates to *false*.

The code below illustrates the use of the **IF... THEN... ELSE...** statement and shows its usage in an SPL programming example.

```
        IF (DAYOFWEEK==SUN) THEN L3
L0:     IF (A>B) THEN L1 ELSE L2
L1:     C=A+B
        GOTO L3

L2:     C=B-A
L3:     D=C*2
           :
```

2.6.3 ON... GOTO... STATEMENT

ON *expr* GOTO *label0*,*label1*,*label2*,*label3*,....

where:

expr is the expression which determines which label is selected

label0,label1,label2,label3,.... are the labels of the sections to which program control can be switched

The **ON/GOTO** statement is a conditional statement that identifies a series of indexed labels to which PEX transfers control based on the value of an expression. The code below illustrates the use of the **ON/GOTO** statement.

```
                    ATTR ER,07
                          :
                    ON INT(B-10) GOTO L0,L1,L2
                    :ER=1
                    PRINT 13,226,"Unsuccessful."
                    GOTO DONE

          L0:       D = (C+1)/2
                    GOTO MERGE

          L1:       D = (C+20)/2
                    GOTO MERGE

          L2:       D = (C+50)/2
          MERGE:    PRINT 13,226,"Success. D=%?%",D
          DONE:     STOP
```

The indices of the **ON/GOTO** statement are zero-based. In addition, if an index evaluates to a number that is greater than the number of indices, program execution continues with the next line of the program.

2.6.4 LOOP STATEMENT

LOOP *register*,*label*

where:

register is the number of times the loop is to be executed

label is the program label to which execution will jump

The **LOOP** statement is an iteration control statement that performs a “decrement register and jump if not zero” function using a specified register and a program label. The **LOOP** statement is a combination of an assignment statement (e.g., **A = A-1**) and a conditional statement (e.g., **IF A>0 THEN Continue**).

The code below illustrates the proper use of the **LOOP** statement in a sample SPL programming example.

```
                    A = 100
                    B = 0
```

```
CALC:       B = REF (A-1)+B
            LOOP A, CALC
            REF (100)=B/100
```

2.6.5 GOSUB STATEMENT

GOSUB *label*

where:

label is the text label which specifies the starting point of the subroutine

The **GOSUB** statement is used to call a subroutine *in the current PLB*. A **RETURN** statement is used to terminate the internal subroutine and return execution control to the statement directly following the **GOSUB**. The subroutine name is actually a label for which all the naming conventions apply.

The code below illustrates the syntax of the **GOSUB** statement and shows its use in a sample SPL program segment:

```
            ATTR AR,0FAH
            A=65
READIT:     D=&DuctDiam(A-1)
            GOSUB AREACALC
            &DuctArea(A-1) = ;AR
            LOOP A, READIT
                :
AREACALC:   ;AR = PI*(D*D)/4
            RETURN
```

2.6.6 CALL STATEMENT

CALL *PLBname*{,STICK}

where:

PLBname is the name of the PLB being called
STICK is included to prevent the PLB from being unloaded

The **CALL** statement is used to execute a PLB from within another PLB. The **CALL**ed PLB may be a block of logic that is shared among several PLBs and/or may be so infrequently used that it is not required that it be RAM-resident all the time.

PLB names are file fragments as discussed in *Section 1.4: The .SPL, .PLB and .LST Files*. If a PLB name begins with a digit (**0**-**9**), it must be enclosed in square brackets.

CALLing a PLB from a program causes the **CALL**er's program counter to be saved. The counter is set to execute the first statement of the **CALL**ed PLB. If the PLB is already loaded into memory, then an *in-use* count for that PLB is increased by one. If the PLB is not in memory, then it will be loaded and its *in-use* count is set to 1.

If the **STICK** argument is used in the **CALL**, then the in-use count is set to FFFFh, preventing the PLB from being unloaded later.

Arguments are passed to the PLB by way of the program registers and user-defined attributes. PLBs which are loaded by way of **CALL** statements ignore the **CALL**ed PLB's attribute declarations because the attribute list is only created when the program is initially loaded. In other words, **CALL**ed PLBs inherit the **CALL**ing program's registers and attributes.

NOTE

CALLed PLBs must have any common attributes defined in the **CALL**ing program. Figure 11-73 illustrates the syntax of the **CALL** statement and shows its use in a sample SPL program.

The code below illustrates the use of the **CALL** statement in a sample SPL program:

```
L1:         D=DAYOFYEAR
L2:         SWAIT 5
            CALL PID,STICK
            [MX_FE01;CV]=0
            IF DAYOFYEAR=D THEN L2
            CALL DAILY_REPORT
            GOTO L1
```

The **CALL**ed PLB must execute a **RETURN** at the end of its execution. When the **RETURN** is executed, the saved program counter of the **CALL**er is restored, and execution begins at the statement following the **CALL**. Once the program **RETURN**s, the **CALL**ed PLB's in-use count is decreased by one, unless it is FFFFh. If the result is zero, then the **CALL**ed PLB is released from memory. If a main program executes a **RETURN** statement while not in an internal subroutine, the program is completely unloaded and its PRB and attribute block are released.

2.6.7 RETURN STATEMENT

RETURN

The **RETURN** statement is used in conjunction with the **CALL** and **GOSUB** statements.

2.7 PROGRAM DELAYS

2.7.1 SWAIT AND MWAIT STATEMENTS

SWAIT ***expr***

where:

expr is the number of seconds to delay program execution

MWAIT ***expr***

where:

expr is the number of minutes to delay program execution

The **SWAIT** and **MWAIT** statements are used to cause a timed delay in program execution. These statements each have a single argument which represents a number of seconds or minutes (respectively) that must pass before program execution continues. The time delay can be viewed as it counts down from the **$D** program control attribute. This attribute shows all time delays in seconds. Once the delay reaches zero, the next program statement is executed.

The code below illustrates the proper use of the **MWAIT** and **SWAIT** statements. Sample SPL programming examples are also shown.

```
L0:         IF (SWITCH==1) Then L1
            MWAIT 5
            GOTO L0

L1:         [PROG1;MN]=[PROG2;SP]+5.0
```

2.7.2 WAIT STATEMENT

WAIT ***expr***

where:

expr is the logical expression that will determine when the **WAIT** will finish

The **WAIT** statement is a conditional statement that halts further program execution until the expression specified in the argument is true.

The code below illustrates the syntax of the **WAIT** statement and shows a sample SPL programming example.

```
L0:         WAIT $ALARMS
            CALL Notify

L1:         CALL Clear, STICK
            IF $ALARMS THEN L1 ELSE L0
```

2.8 PRINTING, LOGGING, AND ALARMS

2.8.1 PRINT STATEMENT (SAGE)

PRINT *portexpr*,*classexpr*,"*formatstring*",*x*,*x*,*x*...*x*,*x*,*x*

where:

portexpr is the SAGE^{MAX} port (0-14) to which data is to be sent
classexpr is the alarm class code (0-255) to be applied to the text message
formatstring contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions
x,x,x...x,x,x,x is the list of expressions to print

The **PRINT** statement is used to send text output to a port on the controller. In the syntax of the **PRINT** statement (see program example below), the *portexpr* expression defines which SAGE^{MAX} port (0-314) to print to. The *classexpr* expression represents the alarm class code (0-255) to be applied to the text message. The *formatstring* field of the **PRINT** statement contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions (**x,x,x...x,x,x**) which follow the format string.

The ***formatstring*** field will typically contain a base text string with Format Specifiers that determine the representation of the corresponding expression in the list of expressions (***x,x,x...x,x,x***) which follow the format string. A list of the Format Specifiers is given in Table 2-2.

Table 2-2 Format Specifiers Used by ALARM, LOG, JOB and PRINT Statements

%%	the character % itself
%?%	use the datatype of the expression
%F*x.y*%	for floating point *xxx.yyy* format
%I*x*%	for integer, *x* digits wide field
%U*x*%	for unsigned integer, *x* digits wide field
%H*x*%	for hexadecimal, *x* digits wide field
%B*x*%	for binary, *x* digits wide field
%*xx*%	for control characters using decimal numbers *xx,* i.e. %10% for <lf>, %13% for <cr> or %07% for <bel>
%R%	the 24 character name of the reference that matches the variable
%T*x*%	time from HH:MM:SS, *x*=2 then HH, *x*=5 then HH:MM, *x*=8 then HH:MM:SS
%W%	current day of week (Mon, Tue, Wed, etc.)
%D%	current day of month as two decimal digits
%M*x*%	current month of year, if *x*=2 then two decimal digits, if *x*=3 then name of month (Jan, Feb, etc.)

Table 2-2 Format Specifiers Used by ALARM, LOG, JOB and PRINT Statements

%Y*x*%	current year of century, if *x*=2 then last two digits, (90, 91, etc.) if *x*=4 then full year, (1990, 1991, etc.)

The program below illustrates the syntax and program examples using the **PRINT** statement. Table 1-7 lists the format specifiers that are supported.

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program illustrates a very simple printer program using the PRINT statement.
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          ATTR PT,0FFh
          ATTR FP,0E0h
          ATTR FX,0F9h
          ATTR PC,0E0h

START: ;PT = -1
          SWAIT 1

WAIT: IF  ( (;PT < 0) OR (;PT > 13) )  THEN START

          ;FP = 1.234567E14
          ;SI = -98765
          ;FX = 85.456
          ;PC = 99.2E0

;First LOG the formatted print statements to a file.
          Print ;PT,0," TSTPRINT Program Variables on %W% %D%-%M3%-%Y4%"
          Print ;PT,0," Signed Integer [;SI] = %I7% (7 digits)%13%%10%",;SI
          Print ;PT,0," Signed Integer [;SI] = %?% (default format)%13%%10%",;SI
          Print ;PT,0," Signed Integer [;SI] = %?L3% (left 3 digits)%13%%10%",;SI
          Print ;PT,0," Signed Integer [;SI] = %?R4% (right 4 digits)%13%%10%",;SI
          Print ;PT,0," Floating Point [;FP] = %?% (scientific)%13%%10%",;FP
          Print ;PT,0," Fixed Point [;FX] = %?% (default format)%13%%10%",;FX
          Print ;PT,0," Percent [;PC] = %F3.1% %% (float with percent
           symbol%%)%13%%10%",;PC
          Print ;PT,0," Referenced Object Ref (0) %R% = %?% (with
name)%13%%10%",0,REF(0)
          Print ;PT,0," The first string from \SPL\SPL.TXT (0-based) = %S%",1
          GOTO  START
```

This code will produce the following output:

```
TSTPRINT Program Variables on Thu 06-Feb-1992
Signed Integer [;SI] =  -98765  (7 digits)
Signed Integer [;SI] =  -98765.   (default format)
Signed Integer [;SI] =  -98       (left 3 digits)
Signed Integer [;SI] =  765.      (right 4 digits)
Floating Point [;FP] =   1.234567E+14   (scientific)
Fixed Point      [;FX] =   85.456     (default format)
Percent          [;PC] =  99.2%     (float with percent symbol %)
Referenced Object Ref  (0)  TSTLOG;PC   =    99.2  (with name)
The first string from \SPL\SPL.TXT (0-based) = User-definable text strings
```

2.8.2 LOG STATEMENT (SAGE)

LOG *logfilename*,"*formatstring*",*x,x,x...x,x,x*

where:

logfilename is complete pathname of a text file to which the text output will be appended
formatstring contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions
x,x,x...x,x,x,x is the list of expressions to print

The **LOG** statement is used to send text output to a file. In the syntax of the **LOG** statement, the *logfilename* argument represents a complete pathname of a text file to which the text output will be appended. If *logfilename* does not exist, it will be created. The *formatstring* field of the **LOG** statement contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions (*x,x,x...x,x,x*) which follow the format string. A list of format specifiers is shown in Table 1-7.

The following code illustrates the use of the **LOG** statement.:

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program illustrates a very simple logger/spooler.
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            ATTR PT,0FFh
            ATTR FP,0E0h
            ATTR SI,0FFh
            ATTR FX,0F9h
            ATTR PC,0E0h
START:      ;PT = -1
            SWAIT 1
WAIT:       IF ( (;PT < 0) OR (;PT > 13) ) THEN START
            ;FP = 1.234567E14
            ;SI = -98765
            ;FX = 85.456
            ;PC = 99.2E0

;First LOG the formatted print statements to a file.

            LOG \LOG\TSTLOG.LOG," TSTLOG Program Variables on %W% %D%-%M3%-%Y4%"
            LOG \LOG\TSTLOG.LOG," Signed Integer [;SI] = %I7% (7 digits)%13%%10%",;SI
            LOG \LOG\TSTLOG.LOG," Signed Integer [;SI] = %?% (default format)%13%%10%",;SI
            LOG \LOG\TSTLOG.LOG," Signed Integer [;SI] = %?L3% (left 3 digits)%13%%10%",;SI
            LOG \LOG\TSTLOG.LOG," Signed Integer [;SI] = %?R4% (right 4 digits)%13%%10%",;SI
            LOG \LOG\TSTLOG.LOG," Floating Point [;FP] = %?% (scientific)%13%%10%",;FP
            LOG \LOG\TSTLOG.LOG," Fixed Point [;FX] = %?% (default format)%13%%10%",;FX
            LOG \LOG\TSTLOG.LOG," Percent [;PC] = %F3.1% %% (float with percent
            LOG \LOG\TSTLOG.LOG," Referenced Object Ref (0) %R% = %?% (with
            LOG \LOG\TSTLOG.LOG," The first string from \SPL\SPL.TXT (0-based) = %S%",1

;Now spool the file to port [;PT] immediately.

SPOOL       ;PT,  \LOG\TSTLOG.LOG
            GOTO  START
```

2.8.3 ALARM STATEMENT

ALARM *classexpr*,"*formatstring*",*x,x,x...x,x,x*

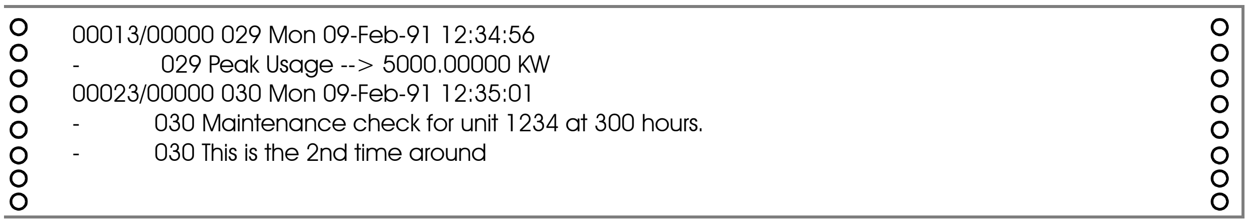
where:

classexpr represents the alarm class code (0-255) to be applied to the text message
formatstring is a base text string with format specifiers that determine the representation of the corresponding expression in the list of expressions
x,x,x...x,x,x,x is the list of expressions to be formatted by *formatstring*

The **ALARM** statement is used to send text to the Alarm Log (ALOG) task for alarm processing. In the syntax of the **ALARM** statement (see **Figure 11-5**), the *classexpr* expression represents the alarm class code (0-255) to be applied to the text message. This class code is used by the ALOG task to determine how to process the alarm message. The *formatstring* field of the **ALARM** statement contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions (x,x,x...x,x,x) which follow the format string. The code below illustrates examples of the **ALARM** statement in use.

```
#SAGE
;
ATTR ;KW, 253
ATTR ;RL, 255
A=1234
B=29
C=19
[;KW] = 5.0
[;RL] = 300
    :
ALARM B,"Peak Usage ---> %F1.5% KW",[;KW]
ALARM C+11, "Maintenance check for unit%I5% at %I3% hours.",A,[:RL]
ALARM 030,"!This is the%I1%nd time around",2
```

Figure 2-1 shows the printed results of the statements shown in the code above. Table 2-2 lists the format specifiers that are supported by the SAGE^{MAX}.



```
00013/00000 029 Mon 09-Feb-91 12:34:56
-           029 Peak Usage --> 5000.00000 KW
00023/00000 030 Mon 09-Feb-91 12:35:01
-           030 Maintenance check for unit 1234 at 300 hours.
-           030 This is the 2nd time around
```

Figure 2-1 Printed Results from the ALARM Statement Example Code

When ALOG processes a message sent to it through the **ALARM** statement, it automatically adds a transaction number, source code, class, day, date and time information in front of the message before routing it, unless the message is preceded by an exclamation point (**!**).

By using an exclamation point (**!**) in the format string, an alarm continuation line is generated. Alarm continuation lines begin with a hyphen (**-**), contain the class and the message text and must follow an alarm line. See the example code shown above.

2.9 JOB EXECUTION (SAGE)

The SPL provides execution of any job file within the SAGE execution environment. SAGE jobs are files with the extension **.JOB** and include **RPT.JOB**, **SPOOL.JOB**, et al. Normally SPOOL jobs are executed by using the native SPL SPOOL command. Note that the *classexpr* in the JOB command line is required, but is currently not used; it should be specified as 0 for now.

2.9.1 THE REPORT JOB

SPL provides enhanced report generation capabilities through the **REPORT** job using the **JOB "RPT"** statement. The **RPT** format of the **JOB** statement is used to issue an "as soon as possible" request to the SAGE^{MAX} scheduler to generate a report.

The **REPORT** job is used to merge a text template file with a file containing data values to produce a final report file. This report file may be printed or saved for later reference.

Reports are created by first creating the *text template file*. This file contains the basic, underlying text of the report. The text file identifies locations to be filled in with data by including special place holders called *format specifiers* (see Table 2-2).

The data file may be either a SAGE^{MAX} table file (**.TBL**) or a trend file. The resulting output file is all text.

JOB 0,"RPT templatepath, valuepath, reportpath",x,x,x...x,x,x

Used to request a report job "as soon as possible." Creates the file specified by the path reportpath from the template file specified in the path templatepath while replacing every **%...%** token with the next expression in the list x,x,x...x,x,x which is contained in the file specified by the path valuepath. Job statement arguments have the following meanings:
template-name of the ASCII text template file
valuepath-data file that contains values in CSV, TBL or TRN format
reportpath-the ASCII text output report file
x,x,x...x,x,x-expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler.

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program is a simple report printer using JOB "RPT ..." and JOB "SPOOL ..." assuming
; that the data has already been collected.  This program prints either at the start of a new day or
; on demand.  This simplifies preset on-demand report generation for an operator by allowing him
; to set a program attribute rather than submit a JOB through the SAGE menuing system.  This
; also allows generation of the report through a host such as EtherView.
;
; Scheduling of the report generation/spooling can also be accomplished through the SAGE
; menuing system.  The report generation at midnight is included here to illustrate the use of the
; BETWEEN statement.
;
;Attributes:        PT request to print report, if <> -1 , PT is the port to spool to
;                   DP default port for auto-printing
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          ATTR PT,0FFh
          ATTR DP,0FFh
START: ;DP = 13
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;If we restarted between midnight and 1:00 AM, then do report now.
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
IF (BETWEEN (0:00, 1:00)) THEN REPORTA
RSTART: ;PT = -1
          D = DAYOFYEAR

TOP: L = 60
WAITMIN: IF (;PT <> -1) THEN  REPORTM
          SWAIT 1
          LOOP L,WAITMIN
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;Check for day rollover every 60 seconds.  Use D register to keep the current Julian
;day.   If D is not the same as the current Julian day, then it is time to auto report.
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
          IF (D == DAYOFYEAR) THEN TOP
; - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;Auto-print.  Set ;PT to default port
; - - - - - - - - - - - - - - - - - - - - - - - - - - - -
REPORTA: ;PT = ;DP
; - - - - - - - - - - - - - - - - - - - - - - - - - - - -
;Submit RPT job request then SPOOL job on port ;PT
; - - - - - - - - - - - - - - - - - - - - - - - - - - - -
REPORTM: JOB 0,"RPT \LOG\TSTRPT.TMP,\TABLES\TSTRPT.TBL,\LOG\TSTRPT.RPT"
          JOB 0,"SPOOL %I2%, \LOG\TSTRPT.RPT",;PT
          GOTO RSTART
```

Figure 2-2 Program TSTRPT.SPL Illustrating the JOB "RPT" Statement

Figure 2-2 illustrates the syntax of the **JOB** statement for a sample report (**RPT**) job. The template file, table file and output file are shown in Figure 2-3, Figure 2-4, and Figure 2-5, respectively.

```
*****************************************************************************************************
O P T I M I Z E D     S T A R T     R E P O R T
Date: %w% %d%-%m3%-%y4% Time:  %t5%

TODAY'S AVERAGE OUTSIDE AIR TEMPERATURE :        %?% deg F
TODAY'S LOW OUTSIDE AIR TEMPERATURE :            %?% deg F
TODAY'S HIGH OUTSIDE AIR TEMPERATURE :           %?% deg F

LOCATIONOPT.    START    OPT. STOP    RUN HRS    MAINT
AHU 1 - Building A%?%  %?%  %I5%  %I5%
AHU 2 - Building A%?%  %?%  %I5%  %I5%
AHU 3 - Building A%?%  %?%  %I5%  %I5%
****************************************************************************************************
```

Figure 2-3 Sample Template File C:\LOG\TSTRPT.TMP

```
Table: C:\TABLES\TSTRTP.TBL contains    19 entries, 5 bytes per entry.

Index   |        DT       |     Value
----------------------------------------
        0:      [FDH]           51.0
        1:      [FDH]           31.0
        2:      [FDH]           59.7
        3:      [E6H]           07:30
        4:      [E6H]           16:45
        5:      [FEH]           2010
        6:      [FEH]           3000
        7:      [E6H]           06:45
        8:      [E6H]           17:00
        9:      [FEH]           1099
       10:      [FEH]           2200
       11:      [E6H]           06:45
       12:      [E6H]           17:30
       13:      [FEH]           2200
       14:      [FEH]           2122
```

Figure 2-4 Sample Table File C:\TABLES\TSTRPR.TBL

```
****************************************************************************************************
                          O P T I M I Z E D    S T A R T    R E P O R T
Date: Wed 12-Feb-1992Time:  11:05

TODAY'S AVERAGE OUTSIDE AIR TEMPERATURE :        51.3 deg F
TODAY'S LOW OUTSIDE AIR TEMPERATURE :            31.0 deg F
TODAY'S HIGH OUTSIDE AIR TEMPERATURE :           59.7 deg F

LOCATION          OPT. START       OPT. STOP          RUN HRS        MAINT
AHU 1 - Building A07:30     16:45              2010          3000
AHU 2 - Building A06:45     17:00              1099          2200
AHU 3 - Building A06:45     17:30              2200          2122

****************************************************************************************************
```

Figure 2-5 Sample Output File C:\LOG\TSTRPT.RPT

2.9.2 THE SPOOL JOB

The **SPOOL** format of the **JOB** statement is used to issue an "as soon as possible" request to the SAGE^{MAX} scheduler to print out (spool) a particular file. Typically the request will be sent to the line printer task (LPT). Other types of tasks also support **SPOOL**ing services. For example, the PHPHDial driver responds to **SPOOL** requests by dialing a particular telephone number and then printing the file once the connection is established.

SPOOL can be executed as a job under SPL. Figure 2-6 illustrates the syntax of the **JOB** statement for a **SPOOL** job in conjunction with an **RPT** job and shows the syntax of the **SPOOL** job statement.

JOB 0,"SPOOL port pathname /D /B /Sbaud /Ntelephonenum",x,x,x...x,x,x

Used to request a spool (print) job "as soon as possible." Arguments have the following meanings:

port*-specifies the SAGE port number (0-31) that provides a spool service*
pathname*-valid DOS pathname of the file to be spooled*
/D*-optional switch used to delete pathname after printing*
/B*-optional switch used to add banner line information to printed listing*
/Sbaud*-optional switch used to specify baud rate for dialout (300, 600, 1200, 2400,4800, 9600, 19200, 38400). If not used, the default is 2400.*
/Ntelephonenum*-optional dialout phone number used to specify destination for dialout port. Valid telephone number characters include 0-9, # and *.*
x,x,x...x,x,x*-expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler.*

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program segment is taken from the complete program in.  It illustrates the
; use of the JOB "SPOOL" statement.  Note the use of the ;PT attribute as a variable used by
; the SPOOL statement.
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
                                                        :
REPORTM:JOB 0,"RPT \LOG\TSTRPT.TMP,\TABLES\TSTRPT.TBL,\LOG\TSTRPT.RPT"
   JOB 0,"SPOOL %I2%, \LOG\TSTRPT.RPT",;PT
                                                        :
```

Figure 2-6The JOB "SPOOL" Statement

The *port* argument refers to any SAGE^{MAX} port number which provides spool services. The *pathname* argument may be any valid DOS pathname.

The */S* and */N* switches are only relevant for dial-out purposes. If the baud rate is not specified for a dial out port, it will default to 38400 baud. The telephone number defaults to *none* which would be ignored by a dial-out port. The telephone number is specified as a sequence of digits, possibly including **#** and *****. You are not permitted to use dashes, parentheses or spaces in the telephone number.

.

CAUTION

Dashes, parentheses and spaces may not be used within the ***Ntelephonenum*** argument.

Note that there may be a significant difference when a file is printed using a **SPOOL** statement versus a **JOB "SPOOL..."** statement. The latter submits a **SPOOL** job to the SAGE^{MAX} job scheduler while the former submits a **SPOOL** command directly to the port task. In the latter case, if there are jobs in the schedule queue, the new **SPOOL** job will not be submitted to the port immediately. If you are generating a report that needs to be **SPOOL**ed in the same program, you should use the **JOB "SPOOL..."** statement to make sure that the report is finished before it is **SPOOL**ed, i.e., the **SPOOL** job is inserted into the job queue after the **REPORT** job.

2.9.2.1 THE BROADCAST JOB

The broadcast job (**BC**) is used to issue an "as soon as possible" broadcast of *message* text to the SAGE^{MAX} scheduler to a specified SAGE^{MAX} *unit* number on a specified SAGE^{MAX} *port*.

Figure 2-7 illustrates the syntax of the **JOB** statement for broadcast (**BC**) jobs.

JOB 0,"BC port/unit/message",x,x,x...x,x,x

Used to request a broadcast message job "as soon as possible." Broadcast job statement arguments have the following meanings:

port*-SAGE port number of broadcast destination*
unit-*device unit number of broadcast destination or SAGE peername*
message*-message text to be broadcast*
x,x,x...x,x,x*-expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler.*

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program is a simple broadcaster program that can submit a broadcast request on demand.
; This simplifies the broadcast process and allows it to be done from a host such as EtherView.
;
;Attributes:RQ<>-1 means request a broadcast on this port
;           UN unit (-1 is broadcast to all units on port RQ)
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

            ATTR RQ,0FFh
            ATTR UN,0FFh
START:      ;RQ = -1
WAITRQ: IF (;RQ <> -1) THEN BCAST
            SWAIT 1
            GOTO WAITRQ
BCAST: IF (;RQ >13) THEN START
            IF (;UN == -1) THEN BCAST2
            JOB 0,"BC %I2%/%I5%/This is a message for a specific unit";[;RQ],[;UN]
            GOTO START

BCAST2: JOB 0,"BC %I2%/-/This is a critical message for all units";[;RQ],[;UN]
            GOTO START
```

Figure 2-7 The JOB "BC" Statement

2.9.3 THE DATA CAPTURE /DATA STUFF JOB

The data capture/data stuff job (**DCS**) is used to create an "as soon as possible" request to the SAGE scheduler gather real-time values (capture) from a list of *named object attributes*, and/or modify values (stuff) from a list of *named object attribute/value* pairs.

Captured data may be formatted into several forms for later use. The captured data may be stored in Comma Separated Value (CSV) format which is useful for Lotus or Excel. As an alternative, data may be stored in a format suitable for use as a Data Stuff input file for use at a later date. During data stuffing, an *audit file* can be created to record the success or failure of each stuff.

Figure 2-8 illustrates the syntax of the **JOB** statement for data capture/stuff (**DCS**) jobs along with several SPL programming examples.

```
JOB 0,"DCS pathname /S/D",x,x,x...x,x,x

Used to request a broadcast message job "as soon as possible."  Broadcast job statement arguments have the
following meanings:
pathname-an ASCII text file with a valid DOS name and extension .PDF which is a pointsdescription file.
/S-optional switch to specify captured data in Stuff file format(default is Comma Separated Variable or CSV format is
blank)
/D-optional switch which turns on debug tracing
x,x,x...x,x,x-expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits
the job to the SAGE job scheduler.

; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program illustrates a simple data collection.  It is used to submit a data collection job
; request when it is a holiday.  In this program, "holiday" is defined to be whenever the SAGE
; global calendar $MODE variable is 255 and the time is between 00:00 and 00:30.  This can
; also be submitted on demand by an operator or host such as EtherView.
;
;Attributes: RQ<> 0 means request a data collection
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ATTR RQ,0FFh
; If we restarted between midnight and 12:30 AM AND
; $MODE = 255, then do collection immediately.
START: IF (($MODE == 255) AND (BETWEEN (0:00,0:30))) THEN  COLLECT

RSTART: ;RQ = 0
        D = DAYOFYEAR

TOP: L = 60
WAITMIN: IF (;RQ <> 0) THEN COLLECT
        SWAIT 1
        LOOP L,WAITMIN
; Check for holiday ($MODE=255) and time between 0:00 and 0:30 every 60 seconds.  Use D register to
; keep the current Julian day.  If D is the same as the current Julian day, then we've done the collection for
; that day.

        IF (D == DAYOFYEAR) THEN TOP
        IF ( ($MODE == 255) AND (BETWEEN (0:00,0:30) ) ) THEN COLLECT ELSE TOP

COLLECT:JOB 0,"DCS \LOG\HOLIDAY.PDF"
        GOTO RSTART
```

Figure 2-8 The Job "DCS" Statement

The Points Definition File (**.PDF** file) contains lines of text which adhere to one of several possible formats. The lines may be up to 128 characters long before the carriage return and linefeed. Extra characters will be ignored. The seven possible line formats for **.PDF** files are:

- ;*comment line*
- >*pathname*
- >>*pathname*
- *objectname*,*description*
- !*pathname*
- !!*pathname*
- *objectname=value*

Lines which begin with a semicolon (;) character are treated as comment lines and are ignored by **DCS**.

The *>pathname* format causes further data capture list output to be directed to the file *pathname*. If *pathname* does not exist, then it will be created. The *pathname* can contain *wildcards* as described later in this section.

The *>>pathname* format cause further data capture list output to be appended to the file *pathname*. If *pathname* does not exist, then it will be created. The *pathname* can contain *wildcards* as described later in this section.

The *objectname,description* format will read (capture) the named object/attribute and output its value to the list output file. If the */S* switch is present in the job string, then the output is formatted in a form that is suitable for subsequent use as a data stuff **.PDF** file. The *description* is any arbitrary text, presumably used to describe the named object in more detail.

The *!pathname* format causes data stuff audit output to be directed to the file *pathname*. If *pathname* does not exist, then it will be created. The *pathname* can contain wildcards as described in the following sections.

The *!!pathname* causes data stuff audit output to be appended to the file *pathname*. If *pathname* does not exist, then it will be created. The *pathname* can contain wild cards as described later in this section.

The *objectname=value* format is used to write (stuff) the named object attribute with the new *value*. The acceptable formats for the new value are described in Table 2-3. If there is an open data stuff audit file, then a confirming message is output to it showing the object name, the stuffed value and the success/ failure of the stuff operation. If the stuff is successful, then only the object name and value are output to the audit file. If the operation fails, then the line in the audit file also has a trailing error message that explains the error.

Table 2-3 Formats for Values When Data Stuffing

Format	Notes
##########.	signed integer numbers
#########.#	signed fixed point integers
########.##	"
#######.###	"
######.####	"
#####.#####	"
####.######	"
###.#######	"
##.########	"
#.#########	"
.##########	"
##.##R	floating point numbers
##.##E ±##R	floating point numbers

Table 2-3 Formats for Values When Data Stuffing

Format	Notes
########H	hex numbers
########B	bitmap
##:## BCD	short time format
##:##:## BCD	long time format
Yes/No	yes=1, no=0
On/Off	on=1, off=0

Objectname is the name of an object and attribute to be captured or stuffed. This name can have one of the formats shown below:

- \ *objecttype* \ *objectname* ; *attribute*
- *objectname* ; *attribute*
- \ *objecttype* \ *objectname*
- *objectname*

The *objecttype* is a two character mnemonic (PT, VR, GL, PG, etc.). If missing, the correct type will be determined by a search of *all* object types as shown in the following order: Points, Programs, Variables, and Globals.

The data capture list and data stuff audit output files contain information in ASCII text format, which can be edited with normal text editing programs.

For data capture list files, if the /S option in the job string is **not** selected, information is formatted in Comma Separated Variable (CSV) format, which can be read by many popular programs. The listing contains a header line which identifies the time and date when the information was fetched and the pathname of the **.PDF** file. In general, each line of the output file contains three fields: point name, point description and value. The point name and description fields are quoted text, while the value field is the number that was fetched.

For example:

“SOMEPOINT;AT”,“A description”,75.3

Alternately, if the /S option in the job string **is** selected, information is formatted for data stuffing. For example:

SOMEPOINT;AT=75.3

The **.PDF** file can contain as many lines as desired. There is no limit to the number of times that the list output may be redirected. The pathname for list/audit output is verified before a new list output file is opened or appended. This means that **DCS** checks for the existence of each sub-directory in the pathname. If a sub-directory does not exist, then it is created automatically by **DCS**.

CAUTION

You must be careful to use the >, >>, ! and !! formats correctly, since they may overwrite existing files!

The **>**, **>>**, **!** and **!!** formats allow the percent (**%**) character to appear in the pathname. If it is used, the **%** character must be followed by a single letter code which indicates one of the time and date values shown in Table 2-4. You can specify any combination of these time and date codes in conjunction with normal pathname characters to form unique time/date-based pathnames for **DCS** files. This allows the generation of time-ordered files by **DCS**. Since **DCS** automatically creates sub-directories, you may freely use these **%** codes in sub-directory names, if desired.

Table 2-4 Wildcards Used in >, >>, ! and !! Pathnames

% Code	Time Unit	Digits Generated
N	month	01-12
D	day of month	01-31
C	year of century	00-99
Y	year	e.g., 1991
H	hour	00-23
M	minute	00-59
W	week of year	01-52
K	day of week	1-7, 1=SUN
J	day of year	001-366

For example, the pathname

>\LOG\%W%K%H%M.PRN

might be used as the name of a data capture list file which was run every day, perhaps several times a day. The resulting files would show the week of the year (**%W**), day of the week (**%K**), and time (**%H%M**) as the file name, e.g., **14021335.PRN**.

2.9.3.1 THE UPLOAD / DOWNLOAD FILE JOB

The upload/download file (**UDL**) job is used to transfer files between a SAGE^{MAX} and a network device such as another SAGE^{MAX}, a STAR peer or an XANP or PHP device.

Downloading files from a SAGE^{MAX} to a network device may occur between any existing SAGE^{MAX} **drive:\file.ext** and any valid network device file. The download service is used to open a (source) file on the SAGE^{MAX}, create a (destination) file on the network device, and write each record from the source file to the destination file until all source file records have been read and downloaded.

Uploading files from a network device to a SAGE^{MAX} may occur between any existing network device file and any valid (DOS) SAGE^{MAX} file. The upload service is used to open a network device (source) file, create a SAGE^{MAX} (destination) file, and write each record from the source file to the destination file until all source file records have been uploaded and written.

The syntax of the upload/download file (**UDL**) job is illustrated in Figure 2-9 with sample SPL programming examples.

JOB 0,“UDL t port peername delimiter remotepath localpath”,x,x,x...x,x,x

Used to request a file upload/download job “as soon as possible.” UDL job statement arguments have the following meanings:

***t**-type of file service, where **U** is upload and **D** is download.*
***port**-the SAGE port number to which the network device is connected.*
***peername**-represents the SAGE Ethernet peername or peer unit number for the network device.*
***delimiter**-**:**, **/** or \ character*
***remotepath**-either a standard DOS pathname if the network device is a SAGE peer or **drive:file.ext** for other peers.*
***localpath**-the local DOS pathname of the file in **drive:\path\file.ext** format.*
***x,x,x...x,x,x**-expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler.*

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program illustrates a simple uploader and downloader.  This program allows you to submit,
; on demand, a file upload request followed by a file download request.  The upload/download is
; from/to a peer on the Ethernet (e.g., EtherView or other SAGEs).  This simplifies the file
; upload/download process and allows it to be done from a host such as EtherView.
;
;Attributes:   RQ<>0 requests an upload/download
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
ATTR RQ,0FFh

START:      IF (;RQ <> 0) THEN UL_DL
            SWAIT  1
            GOTO  START

UL_DL:      JOB 0,“UDL U 14 SAGE Peer #12/C:\LOG\SOMEFILE.LOG C:\LOG\LOG1.LOG”
            JOB 0,“UDL D 14 EtherView Host Peer #1/C:\DATA\SOMEFILE.LOG C:\LOG\LOG1.LOG”
            ;RQ = 0
            GOTO  START
```

Figure 2-9 The JOB "UDL" Statement

The *t* refers to the function type and is either a **U** for upload or a **D** for download. *Port* refers to the SAGE^{MAX} port number to which the network is connected. *Peername* is the SAGE^{MAX} Ethernet peer name or peer unit number for the network device. *Delimiter* is either a front slash (**/**), a colon (**:**) or a backslash (\) character. *Remotepath* is either a DOS pathname if the network device is a SAGE^{MAX} peer, or **drive:file.ext** for other devices. *Localpath* is the local DOS pathname of the file in the format **drive:\path\file.ext**.

2.9.3.2 THE EXPORT DATABASE FILE JOB

The export database file (**EXPORT**) job is used to export name bindings files (**.NBF**) by reading and translating SAGE^{MAX}-resident binary object files (**.BOB**). Name bindings files are editable ASCII text files that contain text representations of the objects which exist in a binary form within SAGE^{MAX} **.BOB** files.

The **EXPORT** job can create the new **.NBF** file on any valid DOS path. **EXPORT** reads **.BOB** files from the **\CFG** directory. Figure 2-10 illustrates the syntax of the **EXPORT** job statement and shows sample SPL program examples.

JOB 0,"EXPORT nbfpath /switch /switch ... /switch",x,x,x...x,x,x

Used to request an export name bindings files job "as soon as possible." EXPORT job statement arguments have th following meanings:

nbfpath*-represents the full .NBF pathname which may include an extension, but will be ignored and .NBF will be added.*
/switch*-series of optional switches that represent the type of .BOB files to search for in the \CFG directory. If no switches are listed, then all .BOB files are searched.*
x,x,x...x,x,x*-expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler.*

```
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; This program allows you to submit, on demand, a database export request.  This may be the
; entire database or the points, variables, programs or globals individually.  This simplifies the
; export process and allows it to be done from a host such as EtherView.
;
;           Attributes:     RQ  =0 do nothing
;                               =1entire database
;                               =2 points only
;                               =3 variables only
;                               =4 programs only
;                               =5 globals only
; - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
            ATTR RQ,0FEh

START: ;RQ = 0
WAITRQ: ON [;RQ] GOTO NONE, ALL, POINTS, VARS, PROGS, GLOBS
NONE: SWAIT 1
            GOTO  WAITRQ

; Export entire database
ALL: JOB 0,"EXPORT  \BACKUP\DATABASE.NBF"
            GOTO  START

; Export points only
POINTS: JOB 0,"EXPORT  \BACKUP\DATABASE.NBF  -PT"
            GOTO  START

; Export variables only
VARS:JOB 0,"EXPORT  \BACKUP\DATABASE.NBF  -VR"
            GOTO  START

; Export programs only
PROGS:JOB 0,"EXPORT  \BACKUP\DATABASE.NBF  -PG"
            GOTO  START

; Export globals only
GLOBS:JOB 0,"EXPORT  \BACKUP\DATABASE.NBF  -GL"
            GOTO  START
```

Figure 2-10 The JOB "EXPORT" Statement

The *nbfpath* is a full pathname which may include an extension, however the extension is ignored and **.NBF** is added. If no switches are specified, then the SAGE^{MAX} will search in the **\CFG** directory for all of the **.BOB** files understood by **EXPORT**. If switches are specified, SAGE^{MAX} will search for only those **.BOB** files specified. **EXPORT** recognizes the following switches:

▼ PT	points
▼ VR	variables
▼ PG	programs
▼ GL	globals

2.10 SPOOLING REPORT AND LOG FILES (SAGE)

2.10.1 SPOOL STATEMENT

SPOOL *portexpr*,*pathname*,{DELETE}

where:

portexpr specifies the port number where the file is to be **SPOOL**ed
pathname is specifies the path and filename of the file to be **SPOOL**ed
DELETE specifies whether to delete the file after spooling

The mechanism for spooling report and log files created by the **RPT** and **LOG** commands, is via the **SPOOL** command. The **SPOOL** command causes the specified port to begin printing the file as soon as it can. The **SPOOL** statement is used to send log files that were created by **LOG** statements to a specified port.

The *portexpr* argument specifies the port number where the file is to be **SPOOL**ed.

The *pathname* argument specifies the path and filename of the file to be **SPOOL**ed.

If the **DELETE** argument follows the filename, then the file will be deleted upon completion of the spool. If the **DELETE** argument is used with this command, then the file will be deleted after it is **SPOOL**ed.

The code below illustrates the use of the **SPOOL** statement in an SPL programming example:

```
LOG C:\ValFile.TXT,"Value is %I2% and",A
LOG C:\ValFile.TXT,"Setpoint is %F2.1%",B
LOG C:\ValFile.TXT,"at %T5% on %W%."
LOG C:\ValFile.TXT,"%13% %10%"
SPOOL 13,C:\ValFile.TXT
```

2.11 TRENDING CONTROL STATEMENTS (SAGE)

2.11.1 STARTTREND & STOPTREND STATEMENTS

STARTTREND *trendname*
STOPTREND *trendname*

where:

trendname is the name of a trend file fragment that can be up to 17 characters long

The **STARTTREND** and **STOPTREND** statements are used to start or stop data collection for the specified trend. The trend must be created prior to the execution of the **STARTTREND** or **STOPTREND** statements.

The *trendname* argument represents a trend file fragment that can be up to 17 characters long. All trends are file images that are contained in the directory **C:\TREND** and have the file extension **.TRN**. The complete pathname is **C:\TREND\*trendname*.TRN**.

The code below illustrates the use of the **STARTTREND** and **STOPTREND** statements and shows a sample SPL programming example:

```
L0:         WAIT (DAY==1)
            MWAIT 30
            STOPTREND GET_KW
            JOB 0,"RPT KWTEMP,C:\TREND\GET_KW.TRN,KW"
            JOB 0,"SPOOL 13,C:\RPT\KW.RPT"
            STARTTREND GET_KW
            WAIT (DAY<>1)
            GOTO L0
```

2.12 PROGRAM EXECUTION STATEMENTS (SAGE)

2.12.1 ACTIVATE STATEMENT

ACTIVATE *programname*

where:

programname is the name of the program to be activated

The **ACTIVATE** statement is used to load an inactive program into memory (if necessary), and begin executing its logic. This statement will not restart a program that is already running, but will restart a program that has been aborted. If a program has been stopped, **ACTIVATE** will start the program at that point. The program to be activated is a database object, i.e., a program name as described in *Section 1.3*: *Program Names*.

The code below illustrates the use of the **ACTIVATE** statement in an SPL program:

```
            IF (TIME>=5:00) THEN Wmup
            ACTIVATE Coast
            GOTO End

Wmup:       ACTIVATE Warmup1
End:        STOP
```

2.12.2 DEACTIVATE STATEMENT

DEACTIVATE *programname*

where:

programname is the name of the program to be deactivated

The **DEACTIVATE** statement allows you to remove a program from memory (RAM). This may free memory space for other programs. The program to be **DEACTIVATE**d is a database object, i.e., a program name as described in *Section 1.3*: *Program Names*.

The code below illustrates the use of the **DEACTIVATE** statement in a simple SPL program segment:

```
            ATTR SD,0FEh

L1:         If ;SD=0 Then L1
            DEACTIVATE OSS1
            DEACTIVATE OSS2
            DEACTIVATE OSS3
            STOP
```

2.12.3 THE RESTART STATEMENT

RESTART *programname*

where:

programname is the name of the program to be restarted

The **RESTART** statement is a program control command that is used to restart program execution from the beginning, as if it had just been loaded. **RESTART** can also activate a deactivated program and start it from the beginning. If a program is unloaded, **RESTART** causes it to be loaded then restarted.

The **RESTART** command is also used to start a program that has been suspended by a **STOP** statement. The program that is **RESTART**ed is a database object, i.e., a program name as described in *Section 1.3*: *Program Names*.

The code below illustrates the syntax of the **RESTART** statement and shows a simple SPL programming example:

```
L0:        WAIT (DAY==1)
           CALL ENDMONTH
           JOB 0,"RPT kwtemp,kwvals,kw"
           RESTART KW_CALC
           WAIT (DAY<>1)
           GOTO L0
```

2.12.4 STOP STATEMENT

STOP *programname*

where:

programname is the name of the program to be stopped

The **STOP** statement is used to halt program execution, maintaining it in a suspended state. If a program name is not included with the **STOP** statement, the current program is halted. Otherwise, the specified program is halted. To resume program execution from a halted state, you must use the **RESTART** or **ACTIVATE** statements.

The code below illustrates the syntax of the **STOP** statement and shows a sample SPL programming example:

```
           IF (TIME>=5:00) THEN Wmup
           ACTIVATE Coast
           GOTO End

Wmup:      STOP SETBACK
           ACTIVATE Warmup1
End:       STOP
```

2.12.5 UNLOAD STATEMENT

UNLOAD

The **UNLOAD** statement is used to remove this program from memory (RAM). This frees memory space for other programs.

If the program's PLB is sticky, it is not unloaded, but the program is put in the unload state.

2.13 EXECUTION ERROR CONTROL

2.13.1 ERRORABORT STATEMENT

ERRORABORT

The **ERRORABORT** statement is an error control statement that causes the program executor to abort the program when any trappable or non-trappable error is detected. (See also *Section 2.13.2*:*ERRORWAIT Statement* and *Section 2.13.3*:*ONERROR Statement*).

There can be multiple **ERRORABORT** and **ERRORWAIT** statements within a program. This allows the aborting of errors to be turned on and off. Unless an **ERRORWAIT** statement is included in a program, the **ERRORABORT** statement is in effect.

All errors that are not trappable (e.g., no such object name, invalid operation, etc.) will always cause the program to be aborted.

The code below illustrates the syntax for the **ERRORABORT** statement and shows its use in a simple SPL program segment:

```
ERRORABORT
;CV = [\PT\Zone Temp;ZT]
Print 13,0, "We got it on the first try."
      :
```

2.13.2 ERRORWAIT STATEMENT

ERRORWAIT

The **ERRORWAIT** statement is an error control statement that allows the programmer to specify what PEX should do when it encounters a trappable error. See *Appendix B: SPL Error Codes (SAGE, DX, GX)* for a complete listing of which errors are trappable. If the **ERRORWAIT** statement is included in a program and PEX detects a trappable error, then the statement that caused the trappable error is re-executed forever until the error condition no longer exists. (See also *Section 2.13.1*: *ERRORABORT Statement* and *Section 2.13.3*: *ONERROR Statement* and *Appendix B: SPL Error Codes (SAGE, DX, GX)*).

There can be multiple **ERRORWAIT** and **ERRORABORT** statements within a program. This allows the aborting of errors and error waiting to be staggered throughout the program. Unless an **ERRORWAIT** statement is included in a program, the **ERRORABORT** statement is in effect.

The code below illustrates the syntax of the **ERRORWAIT** statement and shows it being used in an SPL program segment:

```
ERRORWAIT
Print 13,0, "Wait till we get a good value."
;CV = [\PT\Zone Temp;ZT]
Print 13,0, "We got a good value."
      :
```

2.13.3 ONERROR STATEMENT

ONERROR *label*

where:

label is the label of the code to be executed when a trappable error occurs

The **ONERROR** statement identifies a label to which PEX transfers control whenever it detects a *trappable* error (see **Appendix B**). The **ONERROR** statement is in effect only for the statement that precedes it. (See also *Section 2.13.1*:*ERRORABORT Statement* and *Section 2.13.2*:*ERRORWAIT Statement*). When an error is detected, the error code is placed in the program's **$E** control attribute by PEX. **ONERROR** statements take precedence over **ERRORWAIT** statements. The **$E** program control attribute should be reset to zero before leaving the error code handler.

The code below illustrates the syntax of the **ONERROR** statement and shows an SPL programming example.

```
Getit:      ;$E = 0
            A = ZONE_TEMP;CV
            ONERROR Err
L1:         B=A+10
                :

Err:        IF (;$E<>5) THEN END
            A=72.0
            GOTO L1

End:        STOP
```

NOTE

The **ONERROR** statement can only be used with trappable errors such as a timeout, a CRC or checksum error, NAK responses, data rejection, temporarily blocked states, dialer busy states and failed to connect errors. Any other program execution errors cause the program to abort.

2.14 DEBUGGING STATEMENTS

2.14.1 SECTION STATEMENT

SECTION *number*

where:

number is the number designation given to the section

The **SECTION** statement is a debugging statement that stores the *number* argument in the **$S** program control attribute of the program. This command can be placed strategically at multiple locations in the program to be debugged. By using unique *numbers* in the statements, you can track the progress of the program through various logical sections by monitoring the **$S** program control attribute.

The code below illustrates the syntax of the **SECTION** statement and shows an SPL programming example:

```
            SECTION 1
            A = REF (0)
L1:         SECTION 2
            B = B + REF (A-1)
            LOOP L1
            SECTION 3
                  :
```

2.14.2 NOP STATEMENT

NOP

The **NOP** statement is used for low-level debugging and is normally not used in SPL programs. The function of the **NOP** (or No OPeration) statement is to use up time and occupy program space.

2.15 PROGRAM CONTROL ATTRIBUTES

All program control attributes begin with the '**$**' and are referred to as the 'dollar attributes'. The program control attributes are listed below, along with a brief description. Depending on the device you are using, the existence of specific program control attributes differs. Please reference device user documentation for additional information..

Table 2-5 Program Control Attributes

Control Attribute	Meaning
$$	Program status 0=Stop 1=Run 2=Unloaded 3=Abort 4=Time Delay 5=Restart 6=Load Request 7=Unload Request 8=Abort Request 9=Network Access 10=Reload Request program execution is stopped program is executing program logic is not loaded into RAM program has aborted due to a run-time error program is waiting for an WMAIT or SWAIT statement to timeout initializes the program and starts executing it from the beginning request to load the program into RAM and begin execution request to stop the program and unload it from RAM PEX has encountered a run-time error and is aborting the program program is waiting for the completion of a network access PEX has received a request to unload, reload and start a program
$D	Number of seconds remaining in time delay
$E	Error code (0=no error) (Refer to Appendix B for a complete list of error codes)
$S	Current section number
$C	Program location counter in hexadecimal of next statement to be executed
$W	Wait/Abort on trappable errors such as timeouts and CRC errors (see Appendix B)
$P	Stack pointer used for the evaluation of nested expressions
$B	Subroutine stack pointer contains 00h or return address of subroutine
$N	Number of program attributes
$Z	Paragraph pointer to Program Context Block (PCB)
$L	Paragraph pointer to Program Logic Block (PLB)
$R	Paragraph pointer to Program Reference Block (PRB)
$A	Paragraph pointer to Program Attribute Block (PAB)
$I	Paragraph pointer to pending Intertask Message (ITM)
$F	Program code fetch state, 0=normal, <>0=fetching expressions
$X	Expression state, 0=preexpr, 1=wait for aterm, 2=wait for bterm
$M	Term state, 0=normal, 1=aterm, 2=bterm
$#	Number of expressions in multiple expression term

Table 2-5 Program Control Attributes

$H	Hard disk access counter is incremented for every hard disk read or write
$1	Single step execution, 0=normal, 1=single step
$T	Pointer to once-per-second routine
$G	Debug break-point

The **$$** attribute indicates the program's current operating status. The **$$** attribute can have one of the following values:

Value	State	Description
0	Stop	program execution is stopped
1	Run	program is executing
2	Unloaded	program logic is not loaded
3	Abort	program has aborted due to a run-time error
4	Time Delay	the program is waiting for an MWAIT or SWAIT to timeout
5	Restart	re-initializes the PCB and starts execution at beginning of program
6	Load Request	requests that the program logic be loaded and the program started
7	Unload Request	requests that the program be stopped and the logic be unloaded
8	Abort Request	PEX has encountered a run-time error and is aborting the program
9	Network Access	the program is waiting for the completion network object read/write

Stop indicates that the program has stopped and is no longer executing its program code. *Run* indicates that the program is in the process of executing its program code. *Unloaded* indicates that the program logic has not yet been loaded into memory. *Abort* indicates that the program has stopped due to a run-time error. *Wait for time* indicates that the program has encountered an **MWAIT** or **SWAIT** statement in its logic and is in a wait state. *Restart* indicates that the program has been initialized and is going to start executing from its beginning. *Load request* indicates a signal for the program to be loaded into memory (i.e., RAM) and to begin execution. Conversely, *unload request* indicates a signal for the program to stop execution and unload (remove) itself from RAM. *Abort request* indicates the state prior to abort when the program executor (PEX) encounters a run-time error and signals that it should be aborted. *Network access* indicates that the program had executed either a network read or network write request and is currently in the process of waiting for the network transaction to be completed. *Reload request* is used to unload a program, then reload and start it.

Not all transitions from one **$$** state to another are valid. The following matrix summarizes the valid state transitions for the **$$** attribute.

The **$E** control attribute specifies the error code number of the previously executed program statement. Normally this attribute equals zero (no error).

This special control attribute can be read as an attribute from the program and can be used to determine a course of action should an error occur (i.e., **$E** <> 0). The example below illustrates the use of the **$E** control attribute in a sample SPL program segment.

```
GETIT:          ;$E = 0
                A = [ZONE_TEMP;CV]
                ONERROR ERR

ERR:            IF ;$E==5 THEN GETIT
                STOP
```

The **$S** control attribute is another special control attribute which reflects the current section number of the program. The **SECTION** statement is used to set the **$S** attribute to any integer value (refer to*Section 2.14.1*:*SECTION Statement*). This control attribute can be used in diagnosing errors in your program logic. By using **SECTION** statements at strategic locations in the logic (e.g., before and after loops, conditional statements, calls to subroutines, etc.), you can check the progress of the program execution. By examining the **$S** control attribute through OPI monitor/modify, you can determine if a particular segment of code is getting executed. The example below illustrates the use of **SECTION** statements so that the **$S** control attribute can be examined.

```
SECTION 1
CALL INITIALIZE
SECTION 2
CALL CALC_LOOP
SECTION 3
CALL PROCESS_LOOP
SECTION 4
CALL SUBMIT_JOB
SECTION 5
STOP
```

For logic errors that are especially difficult to diagnose, you may choose to use the single step mode of execution (control attribute **$1**) in conjunction with the **$E** and **$S** control attributes. When set to single step mode (**$1=1**), the program is executed one line at a time. The program must be set to the **RUN** state (**$$=1**) after each line of the program is executed. In some complex programs, this may be a helpful way to determine if your program logic is doing what you really want it to do or if you are encountering a program error. Using single step mode may also make it easier to follow the logic of large programs at a statement-by-statement level.

The **$$** control attribute reflects the current state of the program. This attribute can assume one of eleven values representing one of eleven possible states for the program. These states are:

- 0 - stop
- 1 - run
- 2 - unloaded
- 3 - abort
- 4 - wait for time
- 5 - restart
- 6 - load request
- 7 - unload request
- 8 - abort request

- 9 - network access
- 10 - reload request

Stop indicates that the program has stopped and is no longer executing its program code. *Run* indicates that the program is in the process of executing its program code. *Unloaded* indicates that the program logic has not yet been loaded into memory. *Abort* indicates that the program has stopped due to a run-time error. *Wait for time* indicates that the program has encountered an **MWAIT** or **SWAIT** statement in its logic and is in a wait state. *Restart* indicates that the program has been initialized and is going to start executing from its beginning. *Load request* indicates a signal for the program to be loaded into memory (i.e., RAM) and to begin execution. Conversely, *unload request* indicates a signal for the program to stop execution and unload (remove) itself from RAM. *Abort request* indicates the state prior to abort when the program executor (PEX) encounters a run-time error and signals that it should be aborted. *Network access* indicates that the program had executed either a network read or network write request and is currently in the process of waiting for the network transaction to be completed. *Reload request* is used to unload a program, then reload and start it.

The **$D** control attribute reflects the number of seconds remaining in a time delay imposed by either an **SWAIT** or **MWAIT** statement. When **$D**<>0, **$$**=4 (wait for time).

The **$C** control attribute indicates the program location counter. As the program executes, **$C** changes, reflecting the relative memory location of the next program statement to be executed. **$C** is used primarily for low-level troubleshooting and diagnosis of program errors.

The **$N** control attribute represents the number of user-defined program attributes that have been declared. The **$N** control attribute will always equal the number of **ATTR** declarations within the program.

The **$H** control attribute displays the current count of the number of times the SAGE^{MAX} hard disk has been accessed by the program. It can be used to monitor the frequency of hard disk accesses by watching how rapidly **$H** increases.

The **$1** control attribute is used to set the single step mode of execution of a program. When set to 0, program execution continues normally. If this attribute is set to 1 (single step mode), program execution stops after each program line is executed. Once program execution is stopped, you can examine the other control attributes, registers and user-defined program attributes. This may be very helpful in troubleshooting and diagnosing hard-to-find errors in your program logic. The next line of program code can be executed by setting **$$**=1 (the RUN state).

The **$W**, **$P**, **$B**, **$Z**, **$L**, **$R**, **$A**, **$I**, **$F**, **$X**, **$M** and **$#** attributes are used by PEX to control the execution of the program. Although their meanings are summarized in Table 2-5, the programmer and/or operator normally does not need to reference them.

SECTION 3: USING SPL WITH BACNET

IN THIS SECTION

Introduction 3-3
Fundamentals of SPL in BACnet 3-4
The PROP Statement......................................3-4
Prop Statement Examples...............................3-4
Floating Point Datatype Creation 3-4
Unsigned Integer Datatype Creation..........3-4
Signed Integer Datatype Creation..............3-4
Text Property Datatype Creation 3-5
Bitstring Property Datatype Creation..........3-5
Time Property Datatype Creation...............3-5
Date Property Datatype Creation 3-5
Enumerated Property Datatyype Creation .3-5
NULL Property Datatype Creation 3-5
Working with Object Properties............................3-6
Referencing Objects..3-6
Referencing Properties 3-6
Addressing Object Properties 3-6
Addressing User-Defined properties...............3-7
Peer-To-Peer Addressing................................3-7
Writing Values to Object Properties 3-8
Writing with Command Prioritization 3-9
Data Type Sensitivity with BACnet SPL........3-10
Advanced BACnet SPL Functions 3-11
The OID Function..3-11
The BACNET Statement 3-11

3.1 INTRODUCTION

SPL has features designed specifically for creating program that work with BACnet devices. From within your program, you may define custom properties. These properties can be used within the program and are also visible to other controllers on the BACnet network just like any other property in the controller. Statements exist which allow you to reference properties that exist on the host controller as well as on other controllers on the network. Functions exist that allow you can generate object identifiers during program execution. These features combine seamlessly to allow you to work with any BACnet properties from within your SPL program.

3.2 FUNDAMENTALS OF SPL IN BACNET

The following section illustrates standard fundamentals for writing SPL programs for NB-GPC Family products. While the information in the previous sections provide explicit information behind the underlying functionality of each statement and its usage, this section will provide a more simplistic approach to learning how to write SPL for NB-GPC Family products.

While the following information provides only a few statements, the majority of existing functions in SPL can obviously be used with BACnet.

3.2.1 THE PROP STATEMENT

The **PROP** statement is equivalent to using *ATTR* in PUP-based devices, where **PROP** allows users to create local program properties initialized to one of the twelve (12) primitive BACnet data types supported throughout the standard. User-defined properties must be declared before any other SPL statements with the exception of compiler control statements such as **#GPC**. The syntax to define a property is:

PROP *propertyname, datatype=xxx*.

where

- ***propertyname*** is a numeric or two-letter reference for the property.
- ***datatype*** is a keyword for one of the twelve primitive BACnet datatypes such as REAL, UNSIGNED, NULL, BOOLEAN, etc. Note that BACnet does not support PUP datatypes (e.g. 0FEh, 254, etc).
- ***=XXX*** is an initial value assignment (this can be placed in optionally).

In addition to being available as arguments for assignment and expression statements, all declared properties are visible to the BACnet network in the form of non-standard properties of the Program Object associated with the program. If you wish to access these properties using a BACnet device manufactured outside of American Auto-Matrix, please make certain that the device supports the ability to address non-standard objects and properties.

3.2.2 PROP STATEMENT EXAMPLES

The following provides examples of how to use the **PROP** statement, with information on how to initialize values for each specific datatype assignment.

3.2.2.1 FLOATING POINT DATATYPE CREATION

For floating point datatypes, use **REAL**. **REAL** is a 32-bi IEEE floating point value, typically used for present-value in Analog Input, Analog Output, and Analog Value objects, as well as setpoints, and any other type of point that is of a floating nature (contains a decimal).

```
PROP 10001, REAL = 65.0
```

3.2.2.2 UNSIGNED INTEGER DATATYPE CREATION

- For unsigned Integer datatype, use **UNSIGNED**.

```
PROP 10002, UNSIGNED = 11
```

3.2.2.3 SIGNED INTEGER DATATYPE CREATION

- For signed Integers, use **INTEGER**.

```
PROP 10003, SIGNED = 6
```

3.2.2.4 TEXT PROPERTY DATATYPE CREATION

- For text properties (character strings), use the term **CHARSTRING**.

```
PROP 10004,CHARSTRING = 0,64,"THIS IS MY TEXT PROPERTY"
```

In the value declaration, the value of zero (0) defines the character string set used for the text property. This value must always be zero (0). The value of 64 limits the size of the text property value to 64 characters. All text properties must have a value defined in order for the program to compile. Text properties are primarily to be used for read-only applications, and cannot be assigned different values from within your SPL program.

3.2.2.5 BITSTRING PROPERTY DATATYPE CREATION

For bitstring properties, use the term **BITSTRING**.

```
PROP 10005,BITSTRING = 5,0b10101
```

In the value declaration, the value of five (5) defines the number of bits for the initialized value. If you attempt to define more bits than the size setting, your program will not compile. All bitstring properties must have a value defined in order for the program to compile. The GPC will support up to a maximum of 32 bits in a bitstring value for any property.

When a custom bitstring is viewed by a client or other front-end, all 32-bits will be returned.

3.2.2.6 TIME PROPERTY DATATYPE CREATION

For time properties, use the term **TIME**.

```
PROP 10006,TIME = 15:30:00.00
PROP 10007,TIME = 16:00
```

Time properties can be declared values in *Hour:Minute* format, or *Hour:Minute:Second.Millisecond* format. Most applications used in American Auto-Matrix Native Series products use *Hour:Minute* format.

3.2.2.7 DATE PROPERTY DATATYPE CREATION

For date properties, use the term **DATE**.

```
PROP 10008, DATE = 0d20051225
```

Date property values are initialized uniquely in BACnet, when compared to PUP applications. The general format is *0dyyyymmdd*, where *yyyy* is the year, *mm* in the month, and *dd* is the day-of-the-month. The example provided above represents December 25, 2005.

3.2.2.8 ENUMERATED PROPERTY DATATYYPE CREATION

For enumeration properties, use the term **ENUM**.

```
PROP 10009, ENUM = 2
```

Enumerated property values are typically used for multiple choice assignments in standard BACnet properties such as the Units property, or present-value of Multi-State object type.

3.2.2.9 NULL PROPERTY DATATYPE CREATION

For **NULL** properties, use the term **NULL**.

```
PROP 10010, NULL
```

A **NULL** Datatype is typically used in SPL to assist with relinquishing control of an Analog Output or Binary Output that was written to at a certain priority. No initial value can be given to a **NULL** property because the datatype reflects no assigned value.

3.3 WORKING WITH OBJECT PROPERTIES

Accessing objects and properties in BACnet using SPL can be done in a variety of methods. The following section reviews the various methods of how to access objects and properties.

3.3.1 REFERENCING OBJECTS

SPL can access objects specifically by pre-defined object references. **Appendix E2** provides a table of supported BACnet Objects, and their predefined SPL object references.

3.3.2 REFERENCING PROPERTIES

SPL can address standard properties by using pre-defined property references. **Appendix E3** provides a table of the standard BACnet properties, and their pre-defined SPL property references. For non-standard properties in GPC family devices, you may use either the two-letter reference for the property (e.g. SP, AE, etc), or the numeric BACnet property identifier.

3.3.3 ADDRESSING OBJECT PROPERTIES

When addressing object properties in SPL, the following format must be used:

[*objectID.property*]

where

- objectID references the pre-defined object reference and its Object Instance number
- property references the pre-defined property reference or numeric BACnet property identifier.

A period (**.**) must separate the objectID and property references.

The following examples are illustrated:

```
[AI1.PRESENT_VALUE]
;references Analog Input with Instance of 1
[BI2.PRESENT_VALUE]
;references Binary Input with Instance of 2
[DE800818.SYSTEM_STATUS]
;references Device object with device instance of 800818
```

In many applications, you will typically deal with an object's PRESENT_VALUE property, as this is the most commonly accessed property in BACnet devices.

However, when you are working with proprietary properties (sometimes referred to as non-standard), SPL requires you to reference the BACnet identifier number for the proprietary property of the object you are referencing. In AAM controllers, property identifiers for proprietary properties can be found in the controller's respective user manual. When you are addressing proprietary properties for a GPC controller (whether local or remote over an MS/TP network connection), you may simply use the two-letter alias that is assigned to it. However, if you are working with an ASC-family device or a third-party BACnet controller that contains proprietary properties, you will need to use the BACnet identifier number.

If you choose to work with the numeric BACnet property identifier or are addressing proprietary properties, the following can be used:

```
[AI1.47410]
[BI2.85]
[DE800818.16520]
```

3.3.4 ADDRESSING USER-DEFINED PROPERTIES

To reference user-defined properties created at the top of your program, the following format must be used:

[*.property*]

where

- .property is the property identifier declared.

By referencing no object, SPL will look inside its own program for the property reference.

```
#GPC
;
PROP 10001,REAL
PROP 10002,REAL
;
L0:          [.10001] =13.00
             [.10002] = 16.25
```

3.3.5 PEER-TO-PEER ADDRESSING

SPL allows users to perform peer-to-peer transactions on the MSTP sub-network that the controller resides on. To address an object property from a remove device, the following format must be used:

[*####.objectID.property*]

where

- **####** is the Device Instance of the Device you wish to access.
- ***ObjectID*** is the Object Type and Instance.
- ***property*** is the property of the object.

The following example illustrates this function:

```
#GPC
;
L0:          A = [12345.AI1.PRESENT_VALUE]
```

When accessing object properties from remote devices, users should place SWAIT statements of about 3 seconds between each peer-to-peer network transaction that is made. This allows for the device to receive the token from the network. If you declare ERRORWAIT, SPL will trap an MSTP communication timeout if encountered.

Please note that NB-GPC family devices can only access other MSTP devices located on the local sub-network it is connected to.

If you wish to access non-standard properties in ASC family devices remotely, you must always use the numeric BACnet property identifier. Numeric BACnet property identifiers for each property can be found in the back of the corresponding device you are using, or through various utilities in NB-Pro.

3.3.6 WRITING VALUES TO OBJECT PROPERTIES

Writing values to object properties is dependent on the datatype of the property you are working with. By general nature, BACnet SPL can write to numeric based data types by simply placing an equal sign after the object property and declaring your value. Datatypes that are acceptable to the right-side of the equal sign are as follows:

- REAL
- UNSIGNED
- INTEGER
- TIME
- DATE
- BOOLEAN
- ENUM
- DOUBLE

The following example provides this action:

```
[AI1.PRESENT_VALUE]=75.2
[AV2314.PRESENT_VALUE]=64.0
```

Similar to PUP applications, you can utilize traditional variable assignment routines. Keep in mind that some datatypes need to be equated to a user-defined property configured for the same datatype if one wishes to modify its value through SPL. The primary datatype that must follow this format is BITSTRING.

The following example illustrates how to write to BITSTRING datatypes:

```
#GPC
;
PROP 10001,BITSTRING=8,0b10101010
;
L0:          [GPCSCHED1.AD] = [.10001]
```

3.3.6.1 WRITING WITH COMMAND PRIORITIZATION

In BACnet, it is possible for many different devices to try to modify the same device's object property values. If multiple devices tried to write to the same object property, errors could occur and values could be set incorrectly. To avoid this, BACnet uses priority arrays to determine the order in which property changes will be performed.

A priority array assigns the unique levels of priority to the different types of devices that could write to a device. There are 16 prioritization levels with 1 being highest, and 16 being lowest. A complete list of BACnet Priority Array Levels and their uses is given below:

Table 3-1: Priority Array Levels

Priority Level	Application	Priority Level	Application
1	Manual-Life Safety	9	Available
2	Automatic-Life Safety	10	Available
3	Available	11	Available
4	Available	12	Available
5	Critical Equipment Control	13	Available
6	Minimum On/Off	14	Available
7	Available	15	Available
8	Manual Operator	16	Available

Valid Objects that need to be commanded with Priority Array are as follows:

- Analog Output
- Analog Value (if commandable)
- Binary Output
- Binary Value (if commandable)

To write to one of the above objects using Priority Array, you must place an at sign (@) followed by the level of priority you wish to write with inside the object property reference. The following example illustrates:

```
#GPC
;
L0:         [AO1.PRESENT_VALUE@2]=100.0
            [BO1.PRESENT_VALUE@2]= 1
```

To relinquish control, you must equate the object property at the same priority to a user-defined property with a NULL datatype. The following example illustrates:

```
#GPC
PROP        10013,NULL
L0:         [AO1.PRESENT_VALUE@2]=[.10013]
            [BO1.PRESENT_VALUE@2]=[.10013]
```

3.3.7 DATA TYPE SENSITIVITY WITH BACNET SPL

In comparison to PUP applications, datatypes in BACnet are mostly 32-bit, which results in sensitivity when writing data through SPL. In SPL, the following items are the most common error when writing BACnet SPL.

- When writing to floating point values, you must include a decimal place. If you do not include a decimal place, your SPL program could abort.
- When performing math functions in SPL, you must use the same datatypes. For example, if you try to add an unsigned value of 15 to a time of 15:00 to get 15:15, this will not work. You must add two times together in order to come to a realistic result. Operating outside of this rule will result in aborted SPL programs
- Follow the rules listed with each datatype listed within this manual. For example, you can only write to bitstring properties by equating a property to a user-defined property.

3.3.8 EQU FUNCTION LIMITATIONS IN BACNET SPL

When using the EQU function with BACnet-based SPL programs, you may use the function to reference commonly accessed objects within your program. Unlike PUP-based SPL, you cannot use EQU functions to write to commandable objects such as Analog Outputs and Binary Outputs in GPC. While the EQU statement can accommodate addressing commandable object types, this functionality is limited to being used for read commands, rather than write commands.

3.4 ADVANCED BACNET SPL FUNCTIONS

3.4.1 THE OID FUNCTION

OID(*objecttype,instexpr*)

where:

objecttype is a numeric object identifier number or SPL object reference
instexpr is an expression for instance

The **OID** function is used to compute object identifier numbers from within an SPL program. In many instances the desired object type will be known, but the object instance will be determined during the program's execution. This would occur, for example, if you knew you wanted to read from an analog input, but you wanted the particular input chosen when the program is run.

The **OID** function will return the object identifier number for the object specified by the object type and instance number entered into the *objecttype* and *instexpr* arguments. The *objecttype* argument can either be the numeric object identifier number for that type of object or the SPL keyword used to refer to that type. The *instexpr* argument can be any expression which results in a positive integer value. A complete list of both the standard BACnet object types as well as the AAM proprietary object types, their object identifier numbers for each, and the SPL Object References for each are given in **Appendix E**.

As an example, to compute the object identifier number for the third instance of the analog output object you could use the numeric value for the object identifier number and write

```
OID(1,3)
```

or you can use the SPL keyword AO to specify the analog output

```
OID(AO,3)
```

Similarly, you could use a program register to decide which object to use. If you wanted to look up the object identifier number for the analog output whose instance number was stored in the B register for the program you could write

```
OID(AO,B)
```

The **OID** function can be combined with the **BACNET** statement to programmatically read properties from, or write properties to, any controller on the MSTP sub-network.

3.4.2 THE BACNET STATEMENT

The **BACNET** statement is used to reference a property on the BACnet network. Your programs can read values from, or write values to, the referenced property using the **BACNET** statement. The syntax for the **BACNET** statement is as follows:

BACNET(*devexpr,objexpr,propexpr,instexpr*)

where

devexpr is an expression whose value specifies the device object instance of the device containing the property to be read or written.

objexpr is an expression specifying the object identifier number of the object whose property is to be read.
propexpr is an expression for the identifier number for the chosen property.
instexpr specifies an array index for use in cases when array properties are being read.

When working with the **BACNET** statement the special value -1 (0xFFFFFFFF) can be used for *devexpr* and *instexpr*. When used in *devexpr*, the value -1 means "this device" and is used to refer to properties present in the device in which the SPL program is running. When used in *instexpr*, a value of -1 indicates that no array index is provided. A value of -1 should be entered for *instexpr* whenever you are referencing a single value. The *instexpr* term is only used for bit string, character string, and octet string properties.

3.4.2.1 READING VALUES WITH THE BACNET STATEMENT

The **BACNET** statement can be used to read values from any device connected to the MSTP sub-network. To read a value, you must specify the device, object identifier number, property identifier number, and index of the property to be read. Each of these values may be functions or expressions, allowing you to determine the property to be read at the time of executions.

For example, if you wanted to read the value of the property who's identifier number was 85 from an object located on the same controller who's identifier number was 127.

```
BACNET(-1,127,85,-1)
```

Here the *devexpr* is -1 because the object is on the same device, *objexpr* is the object identifier number 127, 85 is the identifier number of the property we wish to read, and the value of -1 is included because the property is not an array and, therefore, has no index value.

The combination of the **OID** and **BACNET** statements is particularly useful. The **OID** function can be used as the *objexpr* argument to the **BACNET** statement, allowing you to specify any property without having to know identifier numbers ahead of time. If you wished to read the **present_value** of Analog Output 1 then you would write:

```
D=OID(AO,1)
BACNET(-1,D,85,-1)
```

Here we have used the **OID** function as an argument in the **BACNET** statement. You can also use expressions in the **OID** function. If, in the example above, instead of Analog Output 1, you were interested in reading the value of the Analog Output who's instance number was stored in A, you would write:

```
D=OID(AO,A)
BACNET(-1,D,85,-1)
```

Expressions can also be used in the *devexpr* and *instexpr* arguments. If you had, for example, 10 *NB*-VAV controllers with device numbers 10 through 19, you could average the measured flow (Flow Control:**present_value**) using the following code.

```
        A=0
        B=10
        C=OID(AI,6)
L1:     A=A+BACNET(9+B,C,85,-1)
        LOOP B,L1
        A=A/10
```

In this program, the value of B is used to increment the device from which the flow is being read and A is the average flow.

Similarly, you could use expressions in arguments to the **BACNET** statement to choose the property being read. If you had, for example, twelve SSB-FI1 devices connected to your NB-GPC1 measuring space temperatures, you could read the current values and calculate an average space temperature using a program similar to the one above or you could simply use the values in the Universal Input Summary Objects. In this case, the current values would be in the **(VD) Current Measured Input 13** through **(VO) Current Measured Input 24** properties. The code to do this would look like this:

```
                A=0
                B=12
L1:             A=A+BACNET(-1,UISUMMARY0,VC+B,-1)
                LOOP B,L1
                A=A/12
```

Here we are using B to increment the property identifier number. The value VC+B is used because we are interested in the values of properties **VD** (identifier number 54852) through **VO** (identifier number 54863). VC has a value of 54851 and we know that B will vary from 12 to 1 as the program executes the **LOOP** statement. The loop will therefore count from 54863, the identifier number for **VO**, down to 54852, the identifier number for **VD**.

3.4.2.2 WRITING VALUES WITH THE BACNET STATEMENT

The syntax used for writing values using the **BACNET** statement is very similar to that used for reading with one additional parameter. When writing values, you must include a priority array level for the write. This value is appended after the *instexpr* argument in the **BACNET** statement. The complete syntax would look like

BACNET(*devexpr,objexpr,propexpr,instexpr,priority*)

where

devexpr is an expression whose value specifies the device object instance of the device containing the property to be written.
objexpr is an expression specifying the object identifier number of the object whose property is to be written.
propexpr is an expression for the identifier number for the chosen property.
instexpr specifies an array index for use in cases when array properties are being written.
priority is an expression for the priority array level for the write command

When writing values using the **BACNET** statement, you may use expressions for any of the arguments in the same way that you could when reading values. For example, if you wanted to turn on all of the digital outputs on an *NB*-GPC1 you would write the following:

```
                B=12
L1:             C=OID(BO,B)
                BACNET(-1,C,85,-1,7)=1
                LOOP B,L1
```

▾ In this example, the **present_value** property for each of the twelve Digital Output objects is set to one. This value is written with a priority of 7.

APPENDIX A: SPL LANGUAGE REFERENCE

This section indicates which program statements are valid for each of the possible target platforms for an SPL Program. It also contains an alphabetical listing of all program statements, the arguments they require and examples of their use.

IN THIS SECTION

Introduction A-3
ACTIVATE A-5
ALARM A-5
ATTR A-5
BACNET A-6
CALL A-6
DATA A-7
DEACTIVATE A-7
DREF A-7
ERRORABORT A-8
ERRORWAIT A-8
GOSUB A-8
GOTO A-9
IF... THEN... {ELSE...} A-9
JOB A-9
  REPORT Job A-9
  SPOOL Job A-10
  BROADCAST (BC) Job A-10
  DATA CAPTURE/DATA STUFF Job A-10
  UPLOAD/DOWNLOAD Job A-11
  EXPORT Job A-11
LOG A-12
LOOP A-12
MWAIT A-12
NOP A-13
OID A-13
ON... GOTO... .... A-13
ONERROR A-14
PRINT A-14
PROP A-14
RESTART A-15
RETURN A-15
SAVE A-15
SECTION A-16
SPOOL A-16
STARTTREND A-16
STOP A-17
STOPTREND A-17
SWAIT A-17
TABLE A-18
UNLOAD A-18
WAIT A-18

A.1 INTRODUCTION

SPL has many statements that can be used to create control programs. However, not all of these statements are applicable to all controller platforms. Table A-1 shows which options are valid for the various target platforms for your SPL program.

Table A-1 Target Platform Dependence of Program Statements

Statement	SAGE	DX	GX	GPC
ACTIVATE *programname*	✓	✓		
ALARM *classexpr,"formatstring",x,x,x...x,x,x*	✓	✓	✓	✓[1]
ATTR *programattr,datatype*	✓	✓	✓	✓[1]
ATTR *channel;attr,datatype=initialvalue,autosave*	✓	✓	✓	✓
BACNET(devexpr,objexpr,propexpr,instexpr,priority)				✓[2]
CALL *PLBname*	✓			
CALL *PLBname*,STICK	✓			
DATA *v1,v2,v3,v4....*	✓	✓	✓	✓
DEACTIVATE *programname*	✓	✓		
DREF *unit,channel;attr*		✓	✓	✓[1]
ERRORABORT	✓	✓	✓	✓
ERRORWAIT	✓	✓	✓	✓
GOSUB *label*	✓	✓	✓	✓
GOTO *label*	✓	✓	✓	✓
IF *expr* THEN *label*	✓	✓	✓	✓
IF *expr* THEN *label* ELSE *label*	✓	✓	✓	✓
JOB *classexpr,"jobstring",x,x,x...x,x,x*	✓			
LOG *logfilename,"formatstring",x,x,x...x,x,x*	✓			
LOOP *register,label*	✓	✓	✓	✓
MWAIT *expr*	✓	✓	✓	✓
NOP	✓	✓	✓	✓
OID *objecttype,instexpr*				✓[2]
ON *expr* GOTO *label0,label1,label2,label3*	✓	✓	✓	✓
ONERROR *label*	✓	✓	✓	✓
PRINT *portexpr,classexpr,"formatstring",x,x,x...x,x,x*	✓			
PROP *propid,datatype=initialvalue,RO*				✓[2]
REF (index)		✓	✓	✓
RESTART *programname*	✓	✓		

Table A-1 Target Platform Dependence of Program Statements

Statement	SAGE	DX	GX	GPC
RETURN	✓	✓	✓	✓
SAVE ***aa,bb,cc,dd...pp***	✓			
SECTION ***number***	✓	✓	✓	✓
SPOOL ***portexpr,pathname***	✓			
SPOOL ***portexpr,pathname,*****DELETE**	✓			
STARTTREND ***trendname***	✓			
STOP	✓	✓	✓	✓
STOP ***programname***	✓	✓		
STOPTREND ***trendname***	✓			
SWAIT ***expr***	✓	✓	✓	✓
TABLE ***name (size, type)*****=*****value***	✓	✓	✓	✓[1]
UNLOAD	✓	✓		
WAIT ***expr***	✓	✓	✓	✓

[1] SBC-GPC only
[2] *NB*-GPC only

A.2 ACTIVATE

SAGE	DX	GX	GPC
✓	✓		

ACTIVATE *programname*

where:

programname is the name of the program to be activated

The **ACTIVATE** statement is used to load an inactive program into memory (if necessary), and begin executing its logic. This statement will not restart a program that is already running, but will restart a program that has been aborted. If a program has been stopped, **ACTIVATE** will start the program at that point.

A.3 ALARM

SAGE	DX	GX	GPC
✓	✓	✓	✓*

*** SBC-GPC only**

ALARM *classexpr*,"*formatstring*",*x*,*x*,*x*...*x*,*x*,*x*

where:

classexpr is represents the alarm class code (0-255) to be applied to the text message
formatstring is a base text string with format specifiers that determine the representation of the corresponding expression in the list of expressions
x,x,x...x,x,x,x is the list of expression to be formatted by *formatstring*

The ALARM statement is used to create an alarm of class *classexpr* using the format specified by *formatstring*.

A.4 ATTR

SAGE	DX	GX	GPC
✓	✓	✓	✓*

*** SBC-GPC only**

ATTR {*channel*};*attr*,*datatype*{=*initialvalue*}{,*autosave*}

where:

channel is the channel number (optional)
attr is the attribute

datatype is the data type
initialvalue is the initial value (optional)
autosave is the autosave flag

Local program attribute names and their data types are declared using the **ATTR** command. Although local program attributes can be declared anyplace within in the SPL text, they **must** be declared before they are referenced. It is strongly recommended that all local attributes are declared before any other SPL statements. Local program attributes that are referenced, but not declared, result in compile-time errors.

When an attribute is declared, it may be set as an autosave attribute by including an autosave flag value of 1. If the autosave flag not present or set to zero, the attribute will not autosave.

A.5 BACNET

SAGE	DX	GX	GPC
			✓*

*** NB-GPC only**

BACNET(*devexpr,objexpr,propexpr,instexpr{,priority}*)

where:

devexpr is an expression whose value specifies the device object instance of the device containing the property to be read or written.
objexpr is an expression specifying the object identifier number of the object whose property is to be read or written.
propexpr is an expression for the identifier number for the chosen property.
instexpr specifies an array index for use in cases when array properties are being read.
priority is an expression for the priority array level for the write command (used for writing).

The **BACNET** statement is used to reference a property on the BACnet network. Using the **BACNET** statement, you can read from or write to a BACnet property from any controller on the network. The device, object, property and instance numbers may all be determined programmatically.

A.6 CALL

SAGE	DX	GX	GPC
✓			

CALL *PLBname*{,STICK}

where:
PLBname is the name of the PLB being called
STICK is included to prevent the PLB from being unloaded

The **CALL** statement is used to execute a PLB from within another PLB. The **CALL**ed PLB may be a block of logic that is shared among several PLBs and/or may be so infrequently used that it is not required that it be RAM-resident all the time.

If the optional **STICK** argument is used in the **CALL**, then the in-use count is set to FFFFh, preventing the PLB from being unloaded later.

A.7 DATA

SAGE	DX	GX	GPC
✓	✓	✓	✓

DATA *v1*,*v2*,*v3*,*v4*...

where:

v1,v2,v3,v4... are the table values

The **DATA** statement is used to initialize the individual elements of a ram-based table. The **DATA** statement(s) must immediately follow the table declaration. To initialize the entire table to a single value see Section A.34: TABLE.

A.8 DEACTIVATE

SAGE	DX	GX	GPC
✓	✓		

DEACTIVATE *programname*

where:

programname is the name of the program to be deactivated

The **DEACTIVATE** statement allows you to remove a program from memory (RAM). This may free memory space for other programs.

A.9 DREF

SAGE	DX	GX	GPC
	✓	✓	✓*

*** SBC-GPC only**

DREF {*unit*,}*channel*;*attr*

where:

unit is the unit number (optional)
channel is the channel number (format 0FFFFH where FFFF is the channel)
AA is the attribute

The **DREF** (**D**efine **REF**erence) statement is used to cause entries to be made into the attribute table at the end of the PLB, taking the place of the PRB. The attribute/reference table can contain up to 255 entries, i.e., there can be a total of 255 combined program attributes and references per PLB.

A.10 ERRORABORT

SAGE	DX	GX	GPC
✓	✓	✓	✓

ERRORABORT

The **ERRORABORT** statement is an error control statement that causes the program executor to abort the program when any trappable or non-trappable error is detected.

A.11 ERRORWAIT

SAGE	DX	GX	GPC
✓	✓	✓	✓

ERRORWAIT

The **ERRORWAIT** statement tells the program executor that when it detects a trappable error, then the statement that caused the trappable error is to be re-executed forever until the error condition no longer exists.

A.12 GOSUB

SAGE	DX	GX	GPC
✓	✓	✓	✓

GOSUB *label*

where:
label is the text label which specifies the starting point of the subroutine

The **GOSUB** statement is used to call a subroutine *in the current PLB*. A **RETURN** statement is used to terminate the internal subroutine and return execution control to the statement directly following the **GOSUB** statement.

A.13 GOTO

SAGE	DX	GX	GPC
✓	✓	✓	✓

GOTO *label*

where:

label is the label of the point to which program execution will be switched

The **GOTO** statement is an unconditional branch statement that causes program logic to jump a location that is identified by a label.

A.14 IF... THEN... {ELSE...}

SAGE	DX	GX	GPC
✓	✓	✓	✓

IF *expr* THEN *label1* {ELSE *label2*}

where:

expr is the logical expression which determines conditional branching behavior
label1 is the label to jump to if *expr* evaluates to *true*
label2 is the label to jump to if *expr* evaluates to *false* (optional)

The **IF...THEN...** statement is a conditional statement which causes program execution to jump to a different part of a program based on the evaluation of a a logical expression to a value of *true*. Using the optional **ELSE...** statement, an additional option corresponding to *expr* evaluating to *false* may be defined.

A.15 JOB

SAGE	DX	GX	GPC
✓			

A.15.1 REPORT JOB

JOB 0,"RPT *templatename*, *valuepathname*, *reportpathname*",*x*,*x*,*x*...*x*,*x*,*x*

where

templatename- name of the ASCII text template file
valuepathname- data file that contains values in CSV, TBL or TRN format

reportpathname- the ASCII text output report file
x,x,x...x,x,x,x are expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler

The **REPORT** job is used to merge a text template file with a file containing data values to produce a final report file. This report file may be printed or saved for later reference.

A.15.2 SPOOL JOB

JOB 0,"SPOOL *port pathname{/D}{/B}{/Sbaud}{/Ntelephonenumber}*",*x,x,x...x,x,x*

where

port- specifies the SAGE port number (0-31) that provides a spool service
pathname- valid DOS pathname of the file to be spooled
/D- optional switch used to delete pathname after printing
/B- optional switch used to add banner line information to printed listing
/Sbaud- optional switch used to specify baud rate for dialout (300, 600, 1200, 2400,4800, 9600, 19200, 38400). If not used, the default is 2400.
/Ntelephonenumber-optional dialout phone number used to specify destination for dialout port. Valid telephone number characters include 0-9, # and *.
x,x,x...x,x,x,x are expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler

The **SPOOL** format of the **JOB** statement is used to issue an “as soon as possible” request to the SAGE^{MAX} scheduler to print out (spool) a particular file.

A.15.3 BROADCAST (BC) JOB

JOB 0,"BC *port/unit/messagetext*",*x,x,x...x,x,x*

where

port-SAGE port number of broadcast destination
unit-device unit number of broadcast destination or SAGE peername
messagetext-message text to be broadcast
x,x,x...x,x,x,x are expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler

The **BROADCAST** job (BC) is used to issue an “as soon as possible” broadcast of message text to the SAGE^{MAX} scheduler to a specified SAGE^{MAX} unit number on a specified SAGE^{MAX} port.

A.15.4 DATA CAPTURE/DATA STUFF JOB

JOB 0,"DCS *pathname {/S}{/D}*",*x,x,x...x,x,x*

where

pathname-an ASCII text file with a valid DOS name and extension .PDF which is a points description file.

/S-optional switch to specify captured data in Stuff file format (default is Comma Separated Variable or CSV format if blank)

/D-optional switch which turns on debug tracing

x,x,x...x,x,x,x are expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler

The data capture/data stuff job (DCS) is used to create an "as soon as possible" request to the SAGE scheduler gather real-time values (capture) from a list of named object attributes, and/or modify values (stuff) from a list of named object attribute/value pairs.

A.15.5 UPLOAD/DOWNLOAD JOB

JOB 0,"UDL *t port peername delimiter remotepath localpath*",*x,x,x...x,x,x*

where

t-type of file service, where U is upload and D is download.

port-the SAGE port number to which the network device is connected.

peername-represents the SAGE Ethernet peername or peer unit number for the network device.

delimiter-:, / or \ character

remotepath-either a standard DOS pathname if the network device is a SAGE peer or drive:file.ext for other peers.

localpath-the local DOS pathname of the file in drive:\path\file.ext format.

x,x,x...x,x,x,x are expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler

The upload/download file (**UDL**) job is used to transfer files between a SAGE^{MAX} and a network device such as another SAGE^{MAX}, a STAR peer or an XANP or PHP device.

A.15.6 EXPORT JOB

JOB 0,"EXPORT *nbfpath {/switch ... /switch}*",*x,x,x...x,x,x*

where

nbfpath-represents the full .NBF pathname which may include an extension, but will be ignored and .NBF will be added.

/switch-series of optional switches that represent the type of .BOB files to search for in the \CFG directory. If no switches are listed, then all .BOB files are searched.

x,x,x...x,x,x,x are expressions whose values are inserted into the jobstring text in place of format specifiers when PEX submits the job to the SAGE job scheduler

The EXPORT job is used to export name bindings files (.NBF) by reading and translating SAGE^{MAX}-resident binary object files (.BOB).

A.16 LOG

SAGE	DX	GX	GPC
✓			

LOG *logfilename*,"*formatstring*",*x*,*x*,*x*...*x*,*x*,*x*

where:

logfilename is complete pathname of a text file to which the text output will be appended
formatstring contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions
x,x,x...x,x,x,x is the list of expressions to print

The **LOG** statement is used to send text output to a file.

A.17 LOOP

SAGE	DX	GX	GPC
✓	✓	✓	✓

LOOP *register*,*label*

where:

register is the number of times the loop is to be executed
label is the program label to which execution will jump

The **LOOP** statement allows you to execute the same block of code multiple times. When a **LOOP** statement is encountered, the value of *register* is decremented (*register=register-1)*. If the value of *register* is greater than zero, then execution will jump to the program label specified by *label*.

A.18 MWAIT

SAGE	DX	GX	GPC
✓	✓	✓	✓

MWAIT *expr*

where:

expr is the number of minutes to delay program execution

The **MWAIT** statement is used to delay program execution for *expr* minutes.

A.19 NOP

SAGE	DX	GX	GPC
✓	✓	✓	✓

NOP

The **NOP** statement is used for low-level debugging and is normally not be used in SPL programs. The function of the **NOP** (or No OPeration) statement is to use up time and occupy program space.

A.20 OID

SAGE	DX	GX	GPC
			✓*

*** NB-GPC only**

OID(*objecttype*,*instexpr*)

where:
objecttype is a numeric object identifier number or SPL object reference
instexpr is an expression for instance

The **OID** function is used to compute object identifier numbers from within an SPL program.

A.21 ON... GOTO...

SAGE	DX	GX	GPC
✓	✓	✓	✓

ON *expr* GOTO *label0*,*label1*,*label2*,*label3*...

where:
expr is the expression which determines which label is selected
label0,label1,label2,label3... are the labels of the sections to which program control can be switched

The **ON... GOTO...** statement is a conditional statement that identifies a series of indexed labels to which program execution can be switched based on the value of the expression *expr*. The indices of the **ON... GOTO...** statement are zero-based. In addition, if an index evaluates to a number that is greater than the number of indices, program execution continues with the next line of the program.

A.22 ONERROR

SAGE	DX	GX	GPC
✓	✓	✓	✓

ONERROR *label*

where:

label is the label of the code to be executed when a trappable error occurs

The **ONERROR** statement identifies the point in the SPL program to jump to if a trappable error is detected in the line immediately preceding the **ONERROR** statement.
The **ONERROR** statement is in effect only for the statement that precedes it. When an error is detected, the error code is placed in the program's **$E** control attribute by PEX. **ONERROR** statements take precedence over **ERRORWAIT** statements.

A.23 PRINT

SAGE	DX	GX	GPC
✓			

PRINT *portexpr*,*classexpr*,"*formatstring*",*x*,*x*,*x*...*x*,*x*,*x*

where:

portexpr is the SAGE^{MAX} port (0-31) to which data is to be sent
classexpr is the alarm class code (0-255) to be applied to the text message
formatstring contains a base text string with *format specifiers* that determine the representation of the corresponding expression in the list of expressions
x,x,x...x,x,x,x is the list of expressions to print

The **PRINT** statement is used to send text output to a port on the controller.

A.24 PROP

SAGE	DX	GX	GPC
			✓*

*** NB-GPC only**

PROP *propid*,*datatype*=*initialvalue*{,RO}

where:

propid is either a well-known property name, e.g. present_value, or a numeric property identifier

datatype is keyword, e.g. NULL, BOOLEAN, UNSIGNED, REAL etc. representing one of the primitive application datatypes
initialvalue is value for the property to have when first loaded.
RO is used to declare a property read only (optional).

A.25 RESTART

SAGE	DX	GX	GPC
✓	✓		

RESTART *programname*

where:
programname is the name of the program to be restarted

The **RESTART** statement is a program control command that is used to restart program execution from the beginning, as if it had just been loaded. **RESTART** can also activate a deactivated program and start it from the beginning. If a program is unloaded, **RESTART** causes it to be loaded then restarted.

A.26 RETURN

SAGE	DX	GX	GPC
✓	✓	✓	✓

RETURN

The **RETURN** statement is used at the end of a subroutine to return program execution to the point immediately following the call to the subroutine.

A.27 SAVE

SAGE	DX	GX	GPC
✓			

SAVE {*programattr1*,*programattr2*,...*programattr16*}

where:
programattr1,*programattr2*,...*programattr16* are the program attributes (up to 16) which are to be saved

The **SAVE** statement is used to write the values of local program attributes to an initial value (INI) file. The *aa,bb,cc,dd...pp* arguments are used to specify which properties (up to 16) will be written to the INI file. If the list of properties is not included, the **SAVE** statement will cause the current value of all program attributes to be written to the INI file.

A.28 SECTION

SAGE	DX	GX	GPC
✓	✓	✓	✓

SECTION *number*

where:

number is the number designation given to the section

The **SECTION** statement is used to assign a number to a given portion of an SPL program. This is a debugging statement that stores the *number* argument in the **$S** program control attribute of the program. This command can be placed strategically at multiple locations in the program to be debugged.

A.29 SPOOL

SAGE	DX	GX	GPC
✓			

SPOOL *portexpr,pathname*,{DELETE}

where:

portexpr specifies the port number where the file is to be **SPOOL**ed
pathename is specifies the path and filename of the file to be **SPOOL**ed
DELETE specifies whether to delete the file after spooling (optional)

The mechanism for spooling report and log files created by the report and log **JOB** commands is via the **SPOOL** command. The **SPOOL** command causes the specified port to begin printing the file as soon as it can. The **SPOOL** statement is used to send log files that were created by **LOG** statements to a specified port. If the **DELETE** argument follows the filename, then the file will be deleted upon completion of the spool.

A.30 STARTTREND

SAGE	DX	GX	GPC
✓			

STARTTREND *trendname*

where:

trendname is the name of a trend file fragment that can be up to 17 characters long

The **STARTTREND** statement is used to start data collection for the specified trend. The trend must be created prior to the execution of the **STARTTREND** statement.

A.31 STOP

SAGE	DX	GX	GPC
✓	✓	✓	✓

STOP {*programname*}

where:

programname is the name of the program you wish to stop (optional)

The **STOP** statement is used to halt program execution, maintaining it in a suspended state. If a program name is not included with the **STOP** statement, the current program is halted. Otherwise, the specified program is halted. To resume program execution from a halted state, you must use the **RESTART** or **ACTIVATE** statements.

A.32 STOPTREND

SAGE	DX	GX	GPC
✓			

STOPTREND *trendname*

where:

trendname is the name of a trend file fragment that can be up to 17 characters long

The **STOPTREND** statement is used to stop data collection for the specified trend. The trend must be created prior to the execution of the **STOPTREND** statement.

A.33 SWAIT

SAGE	DX	GX	GPC
✓	✓	✓	✓

SWAIT *expr*

where:

expr is the number of seconds to delay program execution

The **SWAIT** statement is used to delay program execution for *expr* seconds.

A.34 TABLE

SAGE	DX	GX	GPC
✓	✓	✓	✓*

*** SBC-GPC only**

TABLE *name* (*size*, *datatype*){=*value*}

where:

name is the name of the table
size is the number of entries contained in the table
type is the datatype
value is the value to which all cells in the table will be initialized

The **TABLE** statement is used to generate a linear, one-dimensional array of data values. When the table is created using the TABLE statement, you may optionally initialize all entries in the table using the optional *value* term.

A.35 UNLOAD

SAGE	DX	GX	GPC
✓	✓		

UNLOAD

The **UNLOAD** statement is used to remove this program from memory (RAM). This frees memory space for other programs. If the program's PLB is sticky, it is not unloaded, but the program is put in the unload state.

A.36 WAIT

SAGE	DX	GX	GPC
✓	✓	✓	✓

WAIT *expr*

where:

expr is the logical expression that will determine when the **WAIT** will finish

The **WAIT** statement is a conditional statement that halts further program execution until the expression specified in the argument is *true*.

Appendix B: SPL Error Codes (SAGE, DX, GX)

This section includes a list of the SPL error codes that can be generated for a program with #SAGE, #SOLODX, or #SOLOGX target platforms. Each error listed includes a description of the error a well as whether or not the error is trappable by the compiler's error handling routines. Some errors are listed with a description, while other errors are self descriptive and do not require a description.

Code	Trappable	Meaning	Description
1	✗	**Invalid Request**	An unsupported math function was requested by the program.
2	✗	**Argument Error**	An argument is bad
3	✗	**Invalid Response**	Bad return for an inquiry
4	✓	**CRC Error**	CRC Error from network connection
5	✓	**Timeout**	Too much time (i.e., more than 20 ms) was taken to execute the "once-per-second" routine.
6	✗	**Unknown Datatype**	Datatype is unknown to program executor
7	✓	**NAK Response**	The program tried to write a value to a read-only attribute.
8	✗	**Invalid Channel #**	A read/write request was made to an unknown or non-existent channel.
9	✗	**No Such Attribute**	A read/write request was made to an unknown or non-existent attribute.
10	✗	**Record Deleted**	attempt to read a deleted record
11	✗	**No Such Name**	name was not found in search
12	✗	**Temp Flush Failed**	wild serach file flush failed
13	✗	**Temp Create Failed**	wild search file could not be created
14	✗	**Lseek Cache Failed**	wild search files seek failed
15	✗	**Read Cache Failed**	wild search file read failed
16	✗	**Object File Seek Failed**	database file rewind failed
17	✗	**Object File Open Failed**	database file open failed
18	✗	**Object File Read Failed**	database file read failed

Code	Trappable	Meaning	Description
19	✗	**Record Locked**	could not delete record
20	✗	**No Such Record**	bad record number
21	✗	**No Such Object Type**	bad database type
22	✗	**No Free Locks**	could not lock record
23	✗	**Object File Write Failed**	could not write record
24	✓	**Temporarily Blocked**	communication port cannot accept communications
25	✗	**Invalid Object Name**	object name in source code does not exist in database
26	✗	**Invalid Task**	invalid task parameter specified
27	✗	**Invalid Timer**	invalid timer handle specified
28	✗	**No More Timers**	no more timers available
29	✗	**Object Type Not Initialized**	object type is being initialized
30	✗	**Unknown Peer Name**	Peer Name is not known in database or device
31	✗	**Invalid Driver**	Invalid driver type (driver may not be loaded)
32	✗	**No Such Class**	class does not exist in database or device
33	✗	**DOS Temp Create Failed**	temporary file could not be created
34	✗	**Invalid Port**	invalid port number specified
35	✗	**Invalid Unit**	invalid unit ID specified
36	✗	**Invalid Session**	invalid virtual terminal session
37	✗	**Invalid Service**	protocol service unknown
38	✗	**Already Acknowledged**	transaction already acknowledged
39	✗	**No Such Card**	card does not exist on AI2000 based device
40	✗	**Bad Ftype For Card**	ftype invalid for card that exists on AI2100 device
41	✗	**Bad Datum Size**	data size was rejected
42	✗	**Conversion Error**	data conversion error‘
43	✓	**Data Rejected**	data was rejected

Code	Trappable	Meaning	Description
44	✗	**Bad User or Codeword**	username or password was valid
45	✗	**No Free Sessions**	No free VT or file sessions
46	✗	**Privilege Violation**	attempted to perform a privileged operation
47	✗	**VT Already Open**	vritual terminal session already open
48	✗	**Port Busy**	port is busy carrying out another transaction
49	✗	**Bad Drive**	invalid drive number
50	✗	**End of File**	File end was reached
51	✗	**No Such File**	file name does not exist
52	✗	**Bad File Handle**	invalid file handle number
53	✗	**Bad File Name**	invalid file name format
54	✗	**Bad Record Number**	invalid record number
55	✗	**File Handle Not Open**	a specified file handle is not open
56	✗	**File In Use**	file is currently being written to or used
57	✗	**File Handle in Use**	File number is alraedy assigned
58	✗	**Transient File Too Big**	Job or OBL file was greater than 64k limit
59	✓	**Alarm Queue Failure**	The alarm queue is full (i.e., it has more than 3 alarms and has not been polled).
60	✗	**No Match Found**	Match not found to search
61	✗	**No Such Name**	Name does not exist in database
62	✗	**Conversion Error**	An invalid data type conversion was attempted. For example, trying to assign a negative value to an attribute that has been defined as a BCD time data type would cause a conversion error.
63	✗	**Underflow Error**	The result of a subtraction function resulted in a value that could not be represented in 32 bits.
64	✗	**Overflow Error**	The result of an addition function resulted in a value that could not be represented in 32 bits.
65	✗	**Not a Number**	The square root (SQRT function) of a negative number was attempted.
66	✗	**Invalid Format**	The result of a report generation using a non-ASCII data file.
67	✗	**Invalid Data Type**	The result of RETYPEing an attribute using an invalid or nonexistent data type code.

Code	Trappable	Meaning	Description
68	×	**No Compare**	error is self-explanatory
69	×	**Bad Sample Interval**	error is self-explanatory
70	×	**Invalid Trend Signature**	error is self-explanatory
71	×	**Sample Limit Exceeded**	error is self-explanatory
72	×	**Trend Already Enabled**	error is self-explanatory
73	×	**Trend Not Fould**	error is self-explanatory
74	×	**Invalid Trend Format**	error is self-explanatory
75	×	**No Available ITM Packets**	no Sage resources available to carry out task
76	×	**Sample Failure Alarm**	error is self-explanatory
77	×	**Not Collected**	error is self-explanatory
78	×	**Name Unknown to Peer**	error is self-explanatory
79	×	**Check Byte Error**	error is self-explanatory
80	×	**Invalid Program Register**	a request was made to a program register that does not exist
81	×	**Program Unloaded**	program is unloaded
82	×	**Stack Error**	too many nested expressions in Sage/DX/GX program
83	×	**Invalid Pcode**	incorrect target platform definition (e.g. #SOLOGX) was used
84	×	**Invalid Term**	term is not supported on platform
85	×	**Invalid Operator**	error is self-explanatory
86	×	**Invalid State**	error is self-explanatory
87	×	**Term Error**	error is self-explanatory
88	×	**Expression Error**	error is self-explanatory
89	×	**Unsuccessful Unload**	error is self-explanatory
90	×	**Invalid State Change**	error is self-explanatory
91	×	**Program Not Found**	error is self-explanatory
92	×	**Index Too Large**	error is self-explanatory

Code	Trappable	Meaning	Description
93-95	✗	**reserved**	error is self-explanatory
96	✗	**Password Invalid**	error is self-explanatory
97	✗	**Not an AVL File**	error is self-explanatory
98	✗	**Invalid Message Number**	error is self-explanatory
99	✗	**Invalid Language**	error is self-explanatory
100	✓	**Dial Busy**	error is self-explanatory
101		**No Phone Number Defined**	error is self-explanatory
102	✓	**Failed to Contact**	error is self-explanatory
103	✗	**No Response from Site**	error is self-explanatory
104-199	✗	**reserved in Sage/DX/GX**	error is self-explanatory
128	✗	**Invalid Program Register (DX/GX Only)**	The result of accessing program register %P, for example, from the once-per-second routine (which does not have a %P register).
130	✗	**Stack Error (DX/GX Only**	The result of having an excessive number of nested expressions.
131	✗	**Invalid Program Code (DX/GX Only**	The result of trying to run a program intended for a different controller on a GPC platform, or if the GPC executor encountered unrecognized program logic.
132	✗	**Invalid Term (DX/GX Only**	This occurs when a mathematical calculation or function yields a result (typically a data type) that is not the expected result (e.g., an unexpected data type). For example, if the result of a mathematical operation was something other than an unsigned integer (when an unsigned integer result was expected), an invalid term error would be generated.
133	✗	**Invalid Operator (DX/GX Only**	A math operator that is unsupported was encountered during the execution of the program code.
134	✗	**Invalid State (DX/GX Only**	This occurs if the program code (PCB) or RAM gets corrupted.
146	✗	**Invalid Index (DX/GX Only**	A program statement attempted to access a table element whose index exceeded the size of the table.

APPENDIX C: SPL ERROR CODES (GPC)

This section includes a list of the SPL error codes that can be generated for a program with #GPC target platform. Because the GPCs use an updated program executor, the error codes generated are different than in previous controllers. Each error listed includes a description of the error a well as whether or not the error is trappable by the compiler's error handling routines. Many other errors match up with errors found in the Sage/DX/GX error codes as well.

Code	Trappable	Meaning	Description
1	✕	**Stack Overflow**	A value larger than the largest value supported by the program executor was encountered.
2	✕	**Stack Underflow**	A value smaller than the smallest value supported by the program executor was encountered.
3	✕	**Invalid Format String**	The PLB file is corrupted. The line of SPL is invalid.
4	✕	**Invalid Coercion**	An attempt was made to coerce a variable to an incompatible datatype.
5	✕	**Expression Stack Overflow**	Internal SPL Error.
6	✕	**Expression Stack Underflow**	Internal SPL Error.
7	✕	**Invalid Expression State**	Internal SPL Error.
8	✕	**Invalid PCode**	The function or statement is invalid. Either the function is unsupported or the statement was used incorrectly.
9	✕	**Invalid Term**	A reference or attribute has been misused.
10	✕	**Not Implemented**	The function or statement is not implemented in the controller.
11	✕	**On Goto**	The GOTO statement specified a target which is out of range.
12	✕	**Bad Reference**	A reference in the source code is out of range.
13	✕	**Invalid Datatype**	The referenced datatype does not match the datatype being written.
14	✕	**Format Mismatch**	The PLB file is corrupted. The SPL expression does not match the required parameter format.
15	✕	**Invalid Operator**	A unsupported operator was encountered during the execution of the program code.
16	✕	**Table Read Only**	An attempt has been made to write to a table which is read only.
17	✕	**Nesting Overflow**	The maximum number of nested expressions has been exceeded.
18	✕	**Queue Full**	The network transmission has been denied due to the transmission queue being full.

APPENDIX D: PUP DATA TYPES

This Appendix lists the hexadecimal and decimal PUP numeric data type codes. The hexadecimal codes are followed by h and the decimal codes are provided in parentheses.

Code	Digit Format	Meaning
FFh (255)	±XXXXXXXXXX.	signed 10 digit
FEh (254)	XXXXXXXXXX.	unsigned 10 digit
FDh (253)	±XXXXXXXXX.X	signed 9.1 digit
FCh (252)	XXXXXXXXX.X	unsigned 9.1 digit
FBh (251)	±XXXXXXXX.XX	signed 8.2 digit
FAh (250)	XXXXXXXX.XX	unsigned 8.2 digit
F9h (249)	±XXXXXXX.XXX	signed 7.3 digit
F8h (248)	XXXXXXX.XXX	unsigned 7.3 digit
F7h (247)	±XXXXXX.XXXX	signed 6.4 digit
F6h (246)	XXXXXX.XXXX	unsigned 6.4 digit
F5h (245)	±XXXXX.XXXXX	signed 5.5 digit
F4h (244)	XXXXX.XXXXX	unsigned 5.5 digit
F3h (243)	±XXXX.XXXXXX	signed 4.6 digit
F2h (242)	XXXX.XXXXXX	unsigned 4.6 digit
F1h (241)	±XXX.XXXXXXX	signed 3.7 digit
F0h (240)	XXX.XXXXXXX	unsigned 3.7 digit
EFh (239)	±XX.XXXXXXXX	signed 2.8 digit
EEh (238)	XX.XXXXXXXX	unsigned 2.8 digit
EDh (237)	±X.XXXXXXXXX	signed 1.9 digit
ECh (236)	X.XXXXXXXXX	unsigned 1.9 digit
EBh (235)	±.XXXXXXXXXX	signed .10 digit
EAh (234)	.XXXXXXXXXX	unsigned .10 digit
E9h (233)	channel map	one bit per channel

Code	Digit Format	Meaning
E8h (232)	bitmap of text	one bit per text field
E7h (231)	BCD (H/S/M)	hours is LSB
E6h (230)	BCD (H/M)	hours is LSB
E5h (229)	packed BCD	8 BCD digits as 4 bytes
E4h (228)	BCD date (Y/M/D)	MSW is year LSW/MSB is month LSW/LSB is day
E3h (227)	Binary date	MSW is year LSW/MSB is month LSW/LSB is day
E2h (226)	reserved	
E1h (225)	reserved	
E0h (224)	IEEE 754 32-bit floating point	
DFh-00h (223-0)	reserved	

Appendix E: BACnet SPL Reference

This section lists the BACnet datatypes and object identifier numbers. Also included are the SPL keywords that can be used when referencing the datatypes and object identifiers in your SPL programs.

E.1 BACnet Data Types

A BACnet property must have one of the following thirteen possible datatypes. When declaring a property in your SPL program, you may use either the identifier number associated with the data type or the SPL data type name.

BACnet Data Type	Data Type Identifier	SPL Data Type Reference
Null	0	NULL
Boolean	1	BOOLEAN
Unsigned Integer	2	UNSIGNED
Signed Integer	3	SIGNED
Real	4	REAL
Double	5	DOUBLE
Octet String	6	OCTETSTRING
Character String	7	CHARSTRING
Bit String	8	BITSTRING
Enumerated	9	ENUM
Date	10	DATE
Time	11	TIME
BACnet Object Identifier	12	OBJID

E.2 BACNET OBJECTS

BACnet Object Type	Object Identifier Number	SPL Object Reference
Analog Input	0	AI
Analog Output	1	AO
Analog Value	2	AV
Binary Input	3	BI
Binary Output	4	BO
Binary Value	5	BV
Calendar	6	CAL
Command	7	CMD
Device	8	DE
Event Enrollment	9	EE
File	10	FI
Group	11	GR
Loop	12	LP
Multi-State-Input	13	MSI
Multi-State-Output	14	MSO
Notification Class	15	NC
Program	16	PG
Schedule	17	SC
Average	18	AVG
Multi-State-Value	19	MSV
Trend Log	20	TR
Life Safety Point	21	LSP
Life Safety Point	21	LSP
Life Safety Zone	22	LSZ
Life Safety Zone	22	LSZ
Proprietary (Occupancy Detector on ASC)	131	ASCOCCUPANCY
Proprietary (Proof of Flow on ASC)	131	ASCPROOFOFFLOW
Proprietary (Occupancy Detector on ASC)	131	ASCPULSE
Proprietary (Occupancy Detector on ASC)	133	ASCECONOMIZER
Proprietary (PID on ASC)	133	ASCPID
Proprietary (Input Select on GPC)	240	INPUTSELECT
Proprietary (Broadcast on GPC)	240	GPCBROADCAST
Proprietary (Logic on GPC)	243	GPCLOGIC
Proprietary (Math on GPC)	243	GPCMATH
Proprietary (Min/Max/Avg on GPC)	243	GPCMINMAXAVG
Proprietary (Scale on GPC)	247	GPCSCALING
Proprietary (Piecewise Curve on GPC)	247	GPCPCURVE
Proprietary (Schedule on GPC)	249	GPCSCHEDULE
Proprietary (Floating Point Control on GPC)	250	GPCMOTORCTRL
Proprietary (Thermostatic Control on GPC)	250	GPCTHERMCTRL
Proprietary (PID Control on GPC)	250	GPCPID
Proprietary (Digital Output Summary on GPC)	251	GPCDOSUMMARY
Proprietary (Occupancy Detector on GPC)	252	GPCOCCUPANCY
Proprietary (Analog Output Summary on GPC)	253	GPCAOSUMMARY

BACnet Object Type	Object Identifier Number	SPL Object Reference
Proprietary (Universal Input Summary on GPC)	254	GPCUISUMMARY
Proprietary (Digital Input Summary on GPC)	254	GPCDISUMMARY
Proprietary (StatBus on GPC)	255	GPCSTATBUS

E.3 PROPERTY IDENTIFIERS

Property	Identifier #	SPL Property Reference
acked-transitions	0	ACKED_TRANSITIONS
ack-required	1	ACK_REQUIRED
action	2	ACTION
action-text	3	ACTION_TEXT
active-cov-subscriptions	152	ACTIVE_COV_ SUBSCRIPTIONS
active-text	4	ACTIVE_TEXT
active-vt-sessions	5	ACTIVE_VT_SESSIONS
alarm-value	6	ALARM_VALUE
alarm-values	7	ALARM_VALUES
all	8	ALL
all-writes-successful	9	ALL_WRITES_ SUCCESSFUL
apdu-segment-timeout	10	APDU_SEGMENT_ TIMEOUT
apdu-timeout	11	APDU_TIMEOUT
application-software-version	12	APPLICATION_ SOFTWARE_VERSION
archive	13	ARCHIVE
attempted-samples	124	ATTEMPTED_ SAMPLES
average-value	125	AVERAGE_VALUE
backup-failure-timeout	153	BACKUP_FAILURE_ TIMEOUT
bias	14	BIAS
buffer-size	126	BUFFER_SIZE
change-of-state-count	15	CHANGE_OF_STATE_ COUNT
change-of-state-time	16	CHANGE_OF_STATE_ TIME
client-cov-increment	127	CLIENT_COV_ INCREMENT
configuration-files	154	CONFIGURATION_ FILES
controlled-variable-reference	19	CONTROLLED_ VARIABLE_ REFERENCE
controlled-variable-units	20	CONTROLLED_ VARIABLE_UNITS
controlled-variable-value	21	CONTROLLED_ VARIABLE_VALUE
cov-increment	22	COV_INCREMENT
cov-resubscription-interval	128	COV_ RESUBSCRIPTION_ INTERVAL
current-notify-time	129	CURRENT_NOTIFY_ TIME
database-revision	155	DATABASE_REVISION
datelist	23	DATE_LIST
daylight-savings-status	24	DAYLIGHT_SAVINGS_ STATUS
deadband	25	DEADBAND
derivative-constant	26	DERIVATIVE_ CONSTANT
derivative-constant-units	27	DERIVATIVE_C ONSTANT_UNITS
description	28	DESCRIPTION
description-of-halt	29	DESCRIPTION_OF_ HALT
device-address-binding	30	DEVICE_ADDRESS_ BINDING
device-type	31	DEVICE_TYPE
direct-reading	156	DIRECT_READING
effective-period	32	EFFECTIVE_PERIOD
elapsed-active-time	33	ELAPSED_ACTIVE_ TIME
error-limit	34	ERROR_LIMIT
event-enable	35	EVENT_ENABLE
event-parameters	83	EVENT_PARAMETERS
event-state	36	EVENT_STATE
event-time-stamps	130	EVENT_TIME_STAMPS
event-type	37	EVENT_TYPE

Property	Identifier #	SPL Property Reference
exception-schedule	38	EXCEPTION_ SCHEDULE
fault-values	39	FAULT_VALUES
feedback-value	40	FEEDBACK_VALUE
file-access-method	41	FILE_ACCESS_ METHOD
file-size	42	FILE_SIZE
file-type	43	FILE_TYPE
firmware-version	44	FIRMWARE_REVISION
high-limit	45	HIGH_LIMIT
inactive-text	46	IN_PROCESS
in-process	47	INACTIVE_TEXT
instance-of	48	INSTANCE_OF
integral-constant	49	INTEGRAL_CONSTANT
integral-constant-units	50	INTEGRAL_ CONSTANT_UNITS
issue- confirmednotifications	51	ISSUE_CONFIRMED_ NOTIFICATIONS
last-restore-time	157	LAST_RESTORE_TIME
life-safety-alarm-values	166	LIFE_SAFETY_ALARM_ VALUES
limit-enable	52	LIMIT_ENABLE
list-of-group-members	53	LIST_OF_GROUP_ MEMBERS
list-of-object-property- references	54	LIST_OF_OBJECT_ PROPERTY_ REFERENCES
list-of-session-keys	55	LIST_OF_SESSION_ KEYS
local-date	56	LOCAL_DATE
local-time	57	LOCAL_TIME
location	58	LOCATION
log-buffer	131	LOG_BUFFER
log-device-object- property	132	LOG_DEVICE_ OBJECT_PROPERTY
log-enable	133	LOG_ENABLE

Property	Identifier #	SPL Property Reference
log-interval	134	LOG_INTERVAL
low-limit	59	LOW_LIMIT
maintenance-required	158	MAINTENANCE_ REQUIRED
manipulated-variable- reference	60	MANIPULATED_ VARIABLE_ REFERENCE
max-apdu-length- accepted	62	MAX_APDU_LENGTH_ ACCEPTED
maximum-output	61	MAXIMUM_OUTPUT
maximum-value	135	MAXIMUM_VALUE
maximum-value- timestamp	149	MAXIMUM_VALUE_ TIMESTAMP
max-info-frames	63	MAX_INFO_FRAMES
max-master	64	MAX_MASTER
max-pres-value	65	MAX_PRES_VALUE
max-segments-accepted	167	MAX_SEGMENT_ ACCEPTED
member-of	159	MEMBER_OF
minimum-off-time	66	MINIMUM_OFF_TIME
minimum-on-time	67	MINIMUM_ON_TIME
minimum-output	68	MINIMUM_OUTPUT
minimum-value	136	MINIMUM_VALUE
minimum-value- timestamp	150	MINIMUM_VALUE_ TIMESTAMP
min-pres-value	69	MIN_PRES_VALUE
mode	160	MODE
model-name	70	MODEL_NAME
modification-date	71	MODIFICATION_DATE
notification-class	17	NOTIFICATION_CLASS
notification-threshold	137	NOTIFICATION_ THRESHOLD
notify-type	72	NOTIFY_TYPE
number-of-APDU-retries	73	NUMBER_OF_APDU_ RETRIES

Property	Identifier #	SPL Property Reference
number-of-states	74	NUMBER_OF_STATES
object-identifier	75	OBJECT_IDENTIFIER
object-list	76	OBJECT_LIST
object-name	77	OBJECT_NAME
object-property-reference	78	OBJECT_PROPERTY_ REFERENCE
object-type	79	OBJECT_TYPE
operation-expected	161	OPERATION_ EXPECTED
optional	80	OPTIONAL
out-of-service	81	OUT_OF_SERVICE
output-units	82	OUTPUT_UNITS
polarity	84	POLARITY
present-value	85	PRESENT_VALUE
previous-notify-time	138	PREVIOUS_NOTIFY_ TIME
priority	86	PRIORITY
priority-array	87	PRIORITY_ARRAY
priority-for-writing	88	PRIORITY_FOR_ WRITING
process-identifier	89	PROCESS_IDENTIFIER
profile-name	168	PROFILE_NAME
program-change	90	PROGRAM_CHANGE
program-location	91	PROGRAM_LOCATION
program-state	92	PROGRAM_STATE
proportional-constant	93	PROPORTIONAL_ CONSTANT
proportional-constant-units	94	PROPORTIONAL_ CONSTANT_UNITS
protocol-conformance-class	95	PROTOCOL_ CONFORMANCE_ CLASS
protocol-object-types-supported	96	PROTOCOL_OBJECT_ TYPES_SUPPORTED
protocol-revision	139	PROTOCOL_REVISION
protocol-services-supported	97	PROTOCOL_ SERVICES_ SUPPORTED
protocol-version	98	PROTOCOL_VERSION
read-only	99	READ_ONLY
reason-for-halt	100	REASON_FOR_HALT
recipient	101	RECIPIENT
recipient-list	102	RECIPIENT_LIST
record-count	141	RECORD_COUNT
records-since-notification	140	RECORDS_SINCE_ NOTIFICATION
reliability	103	RELIABILITY
relinquish-default	104	RELINQUISH_DEFAULT
required	105	REQUIRED
resolution	106	RESOLUTION
segmentation-supported	107	SEGMENTATION_ SUPPORTED
setpoint	108	SETPOINT
setpoint-reference	109	SETPOINT_ REFERENCE
setting	162	SETTING
silenced	163	SILENCED
start-time	142	START_TIME
state-text	110	STATE_TEXT
status-flags	111	STATUS_FLAGS
stop-time	143	STOP_TIME
stop-when-full	144	STOP_WHEN_FULL
system-status	112	SYSTEM_STATUS
time-delay	113	TIME_DELAY
time-of-active-time-reset	114	TIME_OF_ACTIVE_T IME_RESET
time-of-state-count-reset	115	TIME_OF_STATE_ COUNT_RESET

Property	Identifier #	SPL Property Reference
time-synchronization-recipients	116	TIME_ SYNCHRONIZATION_ RECIPIENTS
total-record-count	145	TOTAL_RECORD_ COUNT
tracking-value	164	TRACKING_VALUE
units	117	UNITS
update-interval	118	UPDATE_INTERVAL
utc-offset	119	UTC_OFFSET
valid-samples	146	VALID_SAMPLES
variance-value	151	VARIANCE_VALUE
vendor-identifier	120	VENDOR_IDENTIFIER
vendor-name	121	VENDOR_NAME
vt-classes-supported	122	VT_CLASSES_ SUPPORTED
weekly-schedule	123	WEEKLY_SCHEDULE
window-interval	147	WINDOW_INTERVAL
window-samples	148	WINDOW_SAMPLES
zone-members	165	ZONE_MEMBERS