SIEMENS

APOGEE Powers Process Control Language (PPCL) User's Manual

125-1896 Rev. 5, 10/00

Rev. 5, October, 2000

Table of Contents

1

Introduction

Introduction to PPCL

Imagine a world without computers. Suddenly, a large number of daily functions that are controlled by computers would have to be controlled by hand. How would you like to solve arithmetic formulas using a pad and pencil instead of a calculator? If you had to make a telephone call, you would have to ask the operator to connect you to the right telephone line. Imagine trying to control a 40-story building's environment without the use of a computer.

Even though computers are very powerful devices, they need instructions so they can process information correctly. The instructions that the computer uses are in the form of a programming language. The programming language used with APOGEE equipment is called Powers Process Control Language (PPCL).

PPCL is a high level language developed specifically to control Heating, Ventilating, and Air Conditioning (HVAC) equipment. PPCL combines the functionality of FORTRAN, yet uses a text-based programming structure like BASIC.

There is much more to writing a PPCL program than putting instructions and commands together. Programs should be logically thought out and their processes understood before writing program code. You can also develop programs that are organized differently, but still perform the same function. Certain programming techniques can work better than others, though no one program is the best solution.

How to Use This Manual

To effectively use this manual, you should be familiar with the equipment controlled by the PPCL programming language. Some of the ways you can become familiar with APOGEE products are as follows:

- Siemens Building Technologies, Inc. Training Classes
- Hands-on experience
- Reading APOGEE user documentation.

If you have knowledge or experience with HVAC equipment, the task of learning the programming language becomes easier.

Manual Organization

The PPCL User's Manual consists of the following six chapters:

Chapter 1 – Introduction describes the manual and the conventions used to convey information.

Chapter 2 – Programming Methodology gives a complete overview of the PPCL environment. This chapter includes discussions about relational, logical, and arithmetic operators as well as information about precedence levels, resident points, special functions, point priority, point status, and modular programming.

Chapter 3 – Control Option Comparisons describes different control applications, how they should be used, and how their PPCL commands should be organized.

Chapter 4 – Syntax lists all control commands. This chapter describes the syntax, functionality, and special exceptions for each command, as well as related commands.

Chapter 5 – Glossary consists common programming terminology, as well as a PPCL Reserved Word List.

Index allows you to quickly locate a specific function or command.

Manual conventions

Table 1-1 provides you with the text conventions used in this manual. This list should help you distinguish between the various syntactical items used in this manual.

Table 1-1. Text Conventions.

Conventions	Example	Identifies
All bold and upper-case letters	**ON**	Any command.
All upper-case letters	FAILED	A status or priority of a point.
All upper-case and italics	*OATEMP*	A point name.
Bold parentheses and commas	**(***pt1***,** *pt2***,** *pt3***)**	Parameters used with a statement.
All lower-case and italics	*if* (*value1*.**EQ**.*value2*)	Other commands or operators used in a program statement.
Computer/printer type	`500 IF(FAN.EQ.ON)`	A programming code example.

Three periods placed vertically or horizontally within program code	`700 ON(SFAN)`
	`710 ON(RFAN)`
	`720 . . .`
		Denotes a continuation of program code. This convention is used to prevent programming examples from becoming cluttered.

The syntax page

Each syntax page consists of the following five sections:

- Compatibility bar
- Syntax (may also be physical or logical syntax)
- Use
- Notes (if applicable)
- See also (if applicable)

Refer to the following page for an example of the **DISALM** command.

DISALM

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **DISALM(***pt1***,...,***pt16***)**

pt1 through pt16	Names of the points that should have alarm reporting disabled.

Use This command disables the alarm printing capabilities for specified points. Up to 16 points can be enabled for alarm reporting by one command. The point status changes to *PDSB* after it has been **DISALM**ed.

Example

```
50  IF (SFAN.EQ.OFF) THEN DISALM(ROOM1) ELSE
    ENALM(ROOM1)
```

Notes This command cannot be used for points that do not reside in the device in which the control program is written. **DISALM** cannot be used to directly disable alarm reporting over the network.

See also **ALARM**, **ENALM**, **HLIMIT**, **LLIMIT**, **NORMAL**

Compatibility bar

At the top of each syntax page is a compatibility bar. This bar identifies if a command can be used with a specific type of firmware. When that command is available for a type of firmware, a symbol is placed under that firmware type's name. This symbol identifies that the command is available in all supported revisions of that specific type of firmware. Figure 1-1 shows the symbol used for all firmware revisions.

Logical

⅄

Figure 1-1. "All Revisions" Logical Firmware Symbol.

When a command is not supported in any revision of a specific type of firmware, no symbol appears under the firmware type indicator.

During the life of a product, functionality is often added. As new functionality is introduced, it becomes necessary to distinguish when specific features were added. When a new command is added to the PPCL programming language, it is recognized by placing the number of that firmware revision under the firmware type's name. An example of how a command added to revision 9.2 logical firmware is shown in Figure 1-2.

Logical

9.2

Figure 1-2. Revisions 9.2 and Higher Logical Firmware Symbol.

The five types of firmware are described as follows:

- *Logical* – Logical firmware is designed to accept engineering units (Deg F°, kPa, psi) for parameters in PPCL commands.
- *Physical* – Physical firmware is designed to accept values in physical, electrical or digital units (for example: 0, 1 represents OFF, ON). Conversely, logical firmware uses standard engineering values instead of physical values.
- *Unitary* – Firmware that uses a subset of PPCL commands found in logical firmware.
- *CM* – Firmware (1.x) used in Controller Modules and Open Processors.
- *APOGEE* – Firmware (2.x) used in APOGEE field panels.

Syntax

There are differences in logical firmware when compared to physical firmware. In order to describe the syntax of both firmware types, the manual uses the following labels accordingly:

- *Syntax* - Describes the usage and parameters for the command which applies to all firmware types.
- *Logical syntax* - Defines the usage and parameters for the commands that are only used in logical firmware.
- *Physical syntax* - Defines the usage and parameters for the commands that are only used in physical firmware.

If you are unsure of the type of firmware running in a field panel, contact your local Siemens Building Technologies, Inc. representative.

Use

The *Use* section describes what function the command performs. There are also some one-line examples of program code that allow you to see how parameters and values are defined.

To help explain conditional and comparison logic, truth tables are provided. Truth tables define inputs which are compared in a logical condition to produce results (outputs). To help you understand the concept of truth tables, imagine you have a barrel of apples and oranges. You reach into the barrel to select two pieces of fruit. To compare the two fruits you selected, refer to table 1-2.

Table 1-2. Comparing Apples to Oranges.

Items	Right hand orange	Right hand apple
Left hand apple	Different	Same
Left hand orange	Same	Different

Table 1-2 represents the possible conclusions you can reach, according to the comparisons between your left and right hand. To use the table, identify the objects you want to compare. For example, you are holding an orange in you left hand and an apple in your right hand. Locate the heading along the left side of the table that identifies the type of fruit you are holding in your left hand. Next, locate the heading across the top of the table that identifies the type of fruit you are holding in your right hand. The box where the vertical and horizontal rows meet is the result of the comparison between the two items. For the example, the apple and orange are different.

Table 1-2 is an example of how truth tables can function. Even though truth tables usually compare more complex items than apples and oranges, the method for determining the result is always the same.

Notes

The *Notes* section contains helpful hints that relate to how the command operates, as well as specific details that are commonly missed and subsequently cause errors in a program. If the command does not contain any notes, there is no *Notes* section.

See also

The *See also* section directs you to other commands, operators or sections that are related to the command. If the command does not have any cross-references, there is no *See also* section.

2

Program Methodology

Introduction

Chapter 2 contains information related to the design, coding, and implementation of a program. In this section, you will learn about the following concepts:

- Using the rules and guidelines of the PPCL language
- Using resident points and storage locations
- Understanding the relationships between priority and point status
- Converting a sequence of operation into program code
- Designing programs using a modular structure

PPCL rules

When someone talks to you, they compose sentences according to the grammatical rules of the spoken language. PPCL also uses grammatical conventions to create commands and instructions. Each PPCL program you write must conform to these rules, otherwise, the computer will not understand what you are trying to communicate.

The general rules for PPCL are as follows:

- Each PPCL program contains one or more PPCL statements.
- Each programming statement must be assigned a unique line number. Valid line numbers are 1 through 32,767.
- The maximum number of program lines a device can contain is limited to the amount of free memory of that device.
- When entering program code through the field panel MMI port (excluding APOGEE), the maximum number of characters per line is 72. If you need to enter more characters, enter an ampersand (&) at the end of the line, and continue entering characters on the next line. The total number of characters on both lines, including the ampersand, cannot exceed 144 characters.
- Program lines are executed in ascending order according to their line numbers, unless directed otherwise. When the last line of the program is reached, the computer automatically returns control to the first line of the program and continues processing.

- For all types of firmware except APOGEE, the last line of the program must be executed on every pass of the program.
- All physical and virtual points used in the program must be defined in the point database.
- Point names that begin with numbers must be prefixed with the *at* character (@).
- Program control must only be transferred from a subroutine using the **RETURN** command.

In addition to the previous rules, the following rules apply only to APOGEE firmware:

- APOGEE PPCL programs use an assigned name. Valid names can use from 1 to 30 characters including: A-Z, a-z, 0-9, spaces ( ), periods (.), commas (,), dashes (-), underlines (_), and apostrophes (').
- If a point name is used in PPCL that is greater than 6 characters or uses characters other than A-Z or 0-9, the point name must use double quotes. For example:

  ```
  560  ON(B2SFN,"BUILDING1.AHU01.SFAN")
  ```

  The first point (*B2SFN*) does not require quotes. The second point (*BUILDING1. AHU01.SFAN*) is a long point name and requires quotes.
- When entering program code through the APOGEE field panel MMI port, the maximum number of characters per line is 66 (including line number). If you need to enter more characters, enter an ampersand (&) at the end of the line, and continue entering characters

on the next line. The maximum number of characters allowed on total of three lines of code (including the ampersands and line numbers) is 198 characters.

- Each field panel running PPCL contains one or more separate programs.

PPCL guidelines

A guideline is a declaration of a policy or procedure. Guidelines are created to help you avoid common programming errors. Guidelines also help you write programs that run faster and are easier to maintain. PPCL guidelines are as follows:

- A program defined in one device should not be used to control points in a different device.
- Control loops should not be closed across a network.
- Time-based commands (for example, **LOOP**, **SAMPLE**, **TOD**, **WAIT**, etc.) should be evaluated through every pass of the program for proper operation.
- When possible, a block of program code should be used in other devices that require the same control. Reusing program code in other devices helps reduce testing time and minimizes the number of errors in program logic. Since PPCL requires a unique point database, you must modify the point names in the reused program code to reflect the database for that device.
- Program lines should be initially numbered using multiples of ten (10, 20, 30) or more. This practice provides space between program lines so that modifications can be made without renumbering the program.
- The first line of the program should always be executed through every pass of the program. If program execution is interrupted (for example, during a power failure), the system

always resumes processing at the first line of the program.

- Routing commands (such as **GOTO**) should transfer program control to a sequentially higher line number. This practice prevents programs from being caught in endless loops.
- When possible, program logic should be documented using comment lines. This guideline enforces one of the principles in structured program design.
- Point names used in programs should be meaningful and describe the function for which they are being used. For example, a point name monitoring outside air temperature could be named *OATEMP*.
- A subroutine should only be used in situations where it is more beneficial to place a block of code in a subroutine instead of using straight-line code.
- When using time-based commands, be aware that devices evaluate as many lines of program code as possible. For example, Version 3.0 controller boards are capable of evaluating an average of 350 lines of program code in one second. Field panels using Version 4.0 controller boards are capable of evaluating an average of 500 lines of program code in one second. Two devices can run a program that contains the same number of lines, yet one device may execute the code several times faster than another device.

  The factor that has the greatest influence on program code evaluation speed, is the number of FLN devices connected to the field panel running PPCL. The line evaluation rate is also affected by the number of program lines defined in a device.

- When using multiple programs in an APOGEE field panel, try to create programs that are roughly the same number of lines per program. APOGEE field panels sequentially execute one line of each enabled PPCL program. Programs that contain fewer line numbers will be executed faster than longer programs. For example, a field panel contains two programs, one with 10 lines and the other with 50 lines. The shorter program will be executed five times before the longer program is executed once.

Relational operators

Relational operators compare two values. The result of a comparison, called a *condition*, determines an action. The action depends upon which condition is true. For example, one morning you wake up. As you are eating breakfast, you happen to look at the newspaper and discover that it is going to be a beautiful day. You might say to yourself, "If it gets warmer than 80°F, I will go to the beach. If not, then I will stay home."

You have just used a relational operator to determine your actions for the day. In the statement, you said that the temperature must be greater than 80°F. If the result of this condition is true, then you get to go to the beach. If the result of this condition is false, then you will stay home. This example describes the function of the relational operator.

A relational operator also has a precedence level associated with it. Precedence levels represent the order at which expressions and functions are evaluated. A complete discussion of this topic is located in the *Order of precedence* section. PPCL supports the following relational operators:

- Equal to (.**EQ**.)
- Greater than or equal to (.**GE**.)
- Greater than (.**GT**.)
- Less than or equal to (.**LE**.)
- Less than (.**LT**.)
- Not equal to (.**NE**.)

Each relational operator is described in more detail on the following pages.

Equal to (.EQ.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***value1.***EQ**.*value2***)** *then...*

value1, *value2*	These values represent a number. Values are defined as analog or digital numbers, local variables, or point names.

Use Compares two values to determine if the first value (*value1*) is equal to the second value (*value2*). The comparison between the two values is true if *value1* is equal to *value2*.

Example

```
500  C  IF THE ROOM TEMP IS EQUAL TO 80,
510  C  THEN SET THE SET POINT TO 70.
520  C
530  IF (RMTEMP.EQ.80.0) THEN RMSET = 70.0
```

Notes Certain points (primarily analog input) may contain precise values. When compared to a whole number, the result would be false because the actual value may not exactly match the compared value. It is a good idea to know the resolution of a point before comparing a value with this relational operator.

Greater than or equal to (.GE.)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax *if* **(***value1*.**GE**.*value2***)** *then...*

value1, *value2*	These values represent a number. Values are defined as analog or digital numbers, local variables, or point names.

Use Compares two values to determine if the first value (*value1*) is greater than or equal to the second value (*value2*). The comparison between the two values is true if *value1* is greater than or equal to *value2*.

Example

```
700  C  IF THE ROOM TEMP IS GREATER THAN
710  C  OR EQUAL TO 80, THEN SET THE SET
720  C  POINT TO 70
730  C
740  IF (RMTEMP.GE.80.0) THEN RMSET = 70.0
```

Greater than (.GT.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***value1*.**GT**.*value2***)** *then...*.

value1, *value2*	These values represent a number. Values are defined as analog or digital numbers, local variables, or point names.

Use Compares two values to determine if the first value (*value1*) is greater than the second value (*value2*). The comparison between the two values is true if *value1* is greater than *value2*.

Example

```
250  C  IF THE ROOM TEMP IS GREATER THAN
260  C  80, THEN SET THE SET POINT TO 70
270  C
280  IF (RMTEMP.GT.80.0) THEN RMSET = 70.0
```

Less than or equal to (.LE.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***value1*.**LE**.*value2***)** *then...*

value1, *value2*	These values represent a number. Values are defined as analog or digital numbers, local variables, or point names.

Use Compares two values to determine if the first value (*value1*) is less than or equal to the second value (*value2*). The comparison between the two values is true if *value1* is less than or equal to *value2*.

Example

```
300  C  IF THE ROOM TEMP IS LESS THAN OR
310  C  EQUAL TO 80, THEN SET THE SET POINT
320  C  TO 70
330  C
340  IF (RMTEMP.LE.80.0) THEN RMSET = 70.0
```

Less than (.LT.)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax *if* **(***value1*.**LT**.*value2***)** *then...*

value1, *value2*	These values represent a number. Values are defined as analog or digital numbers, local variables, or point names.

Use Compares two values to determine if the first value (*value1*) is less than the second value (*value2*). The comparison between the two values is true if *value1* is less than *value2*.

Example

```
900  C  IF THE ROOM TEMP IS LESS THAN 80,
910  C  THEN SET THE SET POINT TO 70.
920  C
930  IF (RMTEMP.LT.80.0) THEN RMSET = 70.0
```

Not equal to (.NE.)

Physical	Logical	Unitary	CM	APOGEE
▲	▲	▲	▲	▲

Syntax *if* **(***value1*.**NE**.*value2***)** *then...*

value1, *value2*	These values represent a number. Values are defined as analog or digital numbers, local variables, or point names.

Use Compares two values to determine if the first value (*value1*) is not equal to the second value (*value2*). The comparison between the two values is true if *value1* is not equal to *value2*.

Example

```
600  C  IF THE ROOM TEMP IS NOT EQUAL TO
610  C  80, THEN SET THE SET POINT TO 70.
620  C
630  IF (RMTEMP.NE.80.0) THEN RMSET = 70.0
```

Logical operators

While relational operators compare two values and produce a condition, logical operators compare two conditions. The result of a comparison between the two conditions is called a condition. If the result of the conditions is true, then a specific action is taken. If the result is false, then an alternative action is performed. In the example given for relational operators, you made the following statement:

> "If it gets warmer than 80°F, I will go to the beach."

You might want to expand your statement to take into account your work schedule. Your statement then might look like the following example:

> "If it gets warmer than 80°F and I do not have to work, I will go to the beach."

In the second example, you have defined two conditions. The first condition tests if the temperature is above 80°F. The second condition tests if you have to work. The determining factor in this example, is that both conditions must be true. The result of the comparison between the two conditions determines the action of the statement.

Even if it is a nice day, you cannot go to the beach because you have to work. This example demonstrates how a logical operator works.

A logical operator also has a precedence level associated with it. Precedence levels represent the order at which expressions and functions are evaluated. For a complete discussion about precedence levels refer to the *Order of precedence* section.

PPCL supports the following logical operators:

- And (.**AND**.)
- Not And (.**NAND**.)
- Or (.**OR**.)
- Exclusive Or (.**XOR**.)

Each logical operator is described in more detail on the following pages.

And (.AND.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***cond1*.**AND**.*cond2***)** *then...*

cond1, *cond2*	Defines a condition which is the result of a comparison between two values.

Use Used to compare two logical conditions. The result of the comparison is true if both conditions are true. Refer to Table 2-1 for a comparison of conditions used with the **.AND.** logical operator:

Table 2-1. Truth Table for .AND. Logical Operator.

Conditional Values	**Condition 1 False**	**Condition 1 True**
Condition 2 False	Result is False	Result is False
Condition 2 True	Result is False	Result is True

Example

```
200  IF (TIME.LT.19:00.AND.TIME.GT.5:00)
     THEN ON(LIGHTS)
```

Notes A single statement can incorporate a combined total of 16 relational and logical operators.

Not And (.NAND.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***cond1*.**NAND**.*cond2***)** *then...*

cond1, *cond2*	Defines a condition which is the result of a comparison between two values.

Use Used to compare two logical conditions. Refer to Tables 2-2 and 2-3 for a comparison of conditions used with the **.NAND.** logical operator:

Revision 8.0 and lower logical firmware:

Table 2-2. Truth Table for .NAND. Logical Operator.

Conditional Values	**Condition 1 False**	**Condition 1 True**
Condition 2 False	Result is True	Result is False
Condition 2 True	Result is False	Result is False

Revision 9.1 and higher logical, physical, CM and APOGEE firmware:

Table 2-3. Truth Table for .NAND. Logical Operator.

Conditional Values	**Condition 1 False**	**Condition 1 True**
Condition 2 False	Result is True	Result is True
Condition 2 True	Result is True	Result is False

Example

```
200  IF (TIME.LE.19:00.NAND.OCC.EQ.OFF) THEN
     OFF(LIGHTS)
```

Notes A single statement can incorporate a combined total of 16 relational and logical operators.

Or (.OR.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***cond1*.**OR**.*cond2***)** *then...*

cond1, *cond2*	Defines a condition which is the result of a comparison between two values.

Use Used to compare two logical conditions. The result of the statement is true if at least one of the conditions is true. Refer to Table 2-4 for a comparison of conditions used with the **.OR.** logical operator:

Table 2-4. Truth Table for .OR. Logical Operator.

Conditional Values	**Condition 1 False**	**Condition 1 True**
Condition 2 False	Result is False	Result is True
Condition 2 True	Result is True	Result is True

Example

```
200  IF (TIME.LT.5:00.OR.TIME.GT.17:00) THEN
     ON(LIGHTS)
```

Notes A single statement can incorporate a combined total of 16 relational and logical operators.

Exclusive Or (.XOR.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *if* **(***cond1*.**XOR**.*cond2***)** *then...*

cond1, *cond2*	Defines a condition which is the result of a comparison between two values.

Use Used to compare two logical conditions. The result of the statement is true if *cond1* is true and *cond2* is false. This statement is also true if *cond1* is false and *cond2* is true. Refer to Table 2-5 for a comparison of conditions used with the **.XOR.** logical operator:

Table 2-5. Truth Table for .XOR. Logical Operator.

Conditional Values	**Condition 1 False**	**Condition 1 True**
Condition 2 False	Result is False	Result is True
Condition 2 True	Result is True	Result is False

Example

```
200  IF (PMP1.EQ.ON.XOR.PMP2.EQ.ON) THEN
     NORMAL(PMPALM)
```

Notes A single statement can incorporate a combined total of 16 relational and logical operators.

Arithmetic operators

Arithmetic operators are mathematically related functions that are performed on two or more operands (numbers). You can specify a maximum of 15 arithmetic operators in one PPCL program line. When used in PPCL, the value of the calculation is determined and assigned to a point name or local variable for future reference.

An arithmetic operator also has a precedence level associated with it. Precedence levels represent the order in which expressions and functions are evaluated. A complete discussion about precedence levels and the order in which equations are evaluated, is located in the *Order of precedence* section.

PPLC supports the following arithmetic operators:

- Addition (**+**)
- Assignment (**=**)
- Division (***/***)
- Multiplication (*****)
- Subtraction (**-**)

Each arithmetic operator is described in more detail on the following pages.

Addition

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1 = value1 + value2*

pt1	A point name or local variable that receives the value of the calculation.
value1, *value2*	Point names, local variables, expressions, or numbers that are used in the calculation.

Use Adds two or more values and stores the sum in a defined location as *pt1*.

Example

```
560  COUNT = COUNT + 1.0
```

Assignment

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1 = value1*

pt1	The point name or local variable that receives the value from *value1*.
value1	A point name, local variable, expression, or number that *pt1* receives.

Use Assigns the value of *value1* to the value of *pt1*.

Example

```
560  COUNT = COUNT + 1.0
```

Notes When assigning values, be careful not to assign values to reserved words. Refer to the *Reserved word list* in the back of the manual.

Example

```
500  C = OATEMP
```

The code in this example would fail due to the “C” being used like a variable, when instead the field panel interprets the line as a comment.

Division

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax pt1 = value1 / value2

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, expression, or number used in the calculation. In PPCL, *value1* represents the numerator.
value2	A point name, local variable, expression or number used in the calculation. In PPCL, *value2* represents the denominator.

Use Divides two values (*value1* by *value2*) and stores the quotient in a defined location (*pt1*).

Example

```
890  AVERAG = TOTAL / 5.0
```

Multiplication

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1 = value1 * value2*

pt1	A point name or local variable that receives the value of the calculation.
value1, *value2*	Point names, local variables, expressions, or numbers used in the calculation.

Use Multiplies two or more values (*value1* and *value2*) and stores the product in a defined location (*pt1*).

Example

```
300   SETPT = TEMP * 0.00438
```

Subtraction

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1* = *value1* - *value2*

pt1	A point name or local variable that receives the value of the calculation.
value1, *value2*	Point names, local variables, expressions, or numbers that are used in the calculation.

Use Subtracts a value (*value2*) from another value (*value1*) and stores the difference in a defined location (*pt1*).

Example

```
300  COUNT = COUNT - 1.0
```

Arithmetic functions

Arithmetic functions perform mathematical calculations on a value (number). When used in PPCL, the value derived from the calculation is usually assigned to point name for future reference.

An arithmetic function also has a precedence level associated with it. Precedence levels represent the order in which expressions and functions are evaluated. A complete discussion about precedence levels and the order in which equations are evaluated, is located in the *Order of precedence* section.

PPCL supports the following arithmetic functions:

- Arc-Tangent (**ATN**)
- Complement (**COM**)
- Cosine (**COS**)
- Natural Antilog (**EXP**)
- Natural Log (**LOG**)
- Root (.**ROOT**.)
- Sine (**SIN**)
- Square Root (**SQRT**)
- Tangent (**TAN**)

Each arithmetic function is described in more detail on the following pages.

Arc-Tangent (ATN)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax *pt1* = **ATN(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the arc-tangent is calculated. All values are expressed in degrees.

Use A trigonometric function that calculates the arc-tangent of a value (*value1*). That value is then stored in a defined location (*pt1*). All values used with this function are expressed in degrees.

Example

```
500  THETA = ATN(VALUE1)
```

Complement (COM)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1* = **COM(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the complement is calculated.

Use A function that calculates a one's complement value for a point (*value1*). The one's complement function converts digital values (0 or 1) to their opposite values. That value is then stored in a defined location (*pt1*).

Example

```
500  LLSTAT = COM(PROF01)
```

Cosine (COS)

Physical	Logical	Unitary	CM	APOGEE
	9.2		✓	✓

Syntax *pt1* = **COS(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the cosine is calculated. All values are expressed in degrees.

Use A trigonometric function that calculates the cosine of a value (*value1*). That value is then stored in a defined location (*pt1*). All values used with this function are expressed in degrees.

Example

```
700  C
701  C  THIS FORMULA COMPUTES REAL POWER
702  C  WHEN VOLTS, AMPS, AND PHASE ANGLE
703  C  ARE KNOWN.
704  C
706  PWR = V * I * COS(THETA)
```

Natural Antilog (EXP)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1* = **EXP(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the antilog is calculated.

Use This function calculates the natural antilog of a point (*value1*) and stores that value in a defined location (*pt1*).

Example

```
860  NATANT = EXP(VALUE1)
```

Natural Log (LOG)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax *pt1* = **LOG(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the log is calculated.

Use This function calculates the natural log of a point (*value1*) and stores that value in a defined location (*pt1*).

Example

```
200  ENTH = LOG(VALUE1)
```

Root (.ROOT.)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Syntax *pt1* = **(***value1*.**ROOT**.*value2***)**

pt1	A point name or local variable that receives the value of the calculation.
value1, *value2*	Point names, local variables, or numbers from which the value of the root is calculated.

Use This function calculates the root of the defined points by raising the value of *value1* to the inverse power of *value2*. The value is then stored in a defined location (*pt1*). The mathematical notation for the root is represented as follows:

$$\sqrt[3]{TEMP}$$

The following example demonstrates the same values defined as a line of PPCL program code:

Example

```
310  RVAL = (TEMP.ROOT.3.0)
```

Sine (SIN)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax *pt1* = **SIN(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the sine is calculated. All values are expressed in degrees.

Use A trigonometric function that derives the sine of a value (*value1*). That value is then stored in a defined location (*pt1*). All values used with this function are expressed in degrees.

Example

```
180  VAR = SIN(THETA)
```

Square Root (SQRT)

Physical	Logical	Unitary	CM	APOGEE
✓	✓	✓	✓	✓

Syntax *pt1* = **SQRT(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the square root is calculated.

Use Derives the square root of a value. The value calculated by the square root function is stored in a defined location (*pt1*).

Example

```
180  SRPT1 = SQRT(PT1)
```

Tangent (TAN)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax *pt1* = **TAN(***value1***)**

pt1	A point name or local variable that receives the value of the calculation.
value1	A point name, local variable, or number from which the value of the tangent is calculated. All values are expressed in degrees.

Use A trigonometric function that calculates the tangent of a value (*value1*). That value is then stored in a defined location (*pt1*). All values used with this function are expressed in degrees.

Example

```
930  TANPT1 = TAN(PT1)
```

Special functions

A special function is used to access a specific value that is unique to a point. The value of the point can then be tested or assigned to other points. Since special functions are maintained by the system, they cannot be manually commanded to a different value. Special functions cannot be used over the network.

A special function also has a precedence level associated with it. Precedence levels represent the order at which expressions and functions are evaluated. A complete discussion about precedence levels and the order in which equations are evaluated, is located in the *Order of precedence* section.

PPCL supports the following special functions:

- Alarm priority (**ALMPRI**)
- Totalized value (**TOTAL**)

Each special function is described in more detail on the following pages.

Alarm priority (ALMPRI)

Physical	Logical	Unitary	CM	APOGEE
	12.0		⅄	⅄

Syntax **ALMPRI(*pt1*)**

pt1	A point name for which the current alarm priority is taken.

Use This function accesses the alarm priority level of a point defined in an enhanced alarming control strategy. The value of **ALMPRI** can range from 1 through 6. Besides testing **ALMPRI** for a specific value, you can also assign the value of **ALMPRI** to another point.

Example 1

```
230  C
232  C  THIS LINE TESTS IF THE POINT CALLED
234  C  RMTEMP IS AT ALARM PRIORITY LEVEL 1.
236  C
240  IF (ALMPRI(RMTEMP).EQ.1) THEN ON (OPBELL)
```

Example 2

```
330  C
332  C  THIS LINE ASSIGNS THE ALARM
334  C  PRIORITY LEVEL OF RMTEMP TO THE
336  C  POINT CALLED TMPPRI.
338  C
340  TMPPRI = ALMPRI(RMTEMP)
```

Notes **ALMPRI** must reside in the same field panel as the point on which the function is being performed.

Totalized value (TOTAL)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **TOTAL(***pt1***)**

pt1	A point name for which the totalized value is taken.

Use This function allows you to access the totalized value of a point.

Example 1

```
800  C
801  C  THIS LINE COMPARES THE TOTALIZED
802  C  VALUE OF FAN1 TO DETERMINE IF IT IS
803  C  GREATER THAN 500. IF THE CONDITION
804  C  IS TRUE, FAN1 WILL BE PLACED INTO
805  C  ALARM.
806  C
810  IF(TOTAL(FAN1).GT.500.0)THEN ALARM(FAN1)
```

Example 2

```
800  C
801  C  THIS LINE ASSIGNS THE TOTALIZED
802  C  VALUE OF FAN1 TO THE POINT CALLED
803  C  FANRUN.
804  C
810  FANRUN = TOTAL(FAN1)
```

Notes **TOTAL** must reside in the same field panel as the point on which the function is being performed.

See also **INITTO**

Order of precedence

The order of precedence is the order in which operators (mathematical, relational, logical, and special function) are evaluated. Operators that have a higher precedence are evaluated before operators that have a lower precedence. If all the operators in the formula have equivalent precedence levels, then the operators are evaluated from left to right. Refer to Table 2-6 for a complete list of precedence levels for PPCL operators.

The following example illustrates how precedence levels are applied to operators:

10 - 5 + 2 * 3 = 11

The solution to this problem is as follows:

1. Multiplication has the highest precedence level in this equation. The first step in evaluating the equation is as follows:

   2 * 3 = 6

   After the first level of operators have been evaluated, the equation now looks like the following:

   10 - 5 + 6

2. All other remaining operators are at the same precedence level. Operators are then evaluated in a left to right order. The last step in evaluating the equation is as follows:

   10 - 5 = 5
   5 + 6 = 11

   The solution to this equation is 11.

Order of precedence is also applied to statements containing relational and logical operators. These operators are assigned a precedence level.

In the following example, assume that both comparisons (*condition1* and *condition2*) are true:

> *If* (*value1* .**EQ**. *value2* .**AND**. *value1* .**NE.** *value3*) *then...*

Since the problem compares values using relational operators (**.EQ.** and **.NE.**), the result of the comparison can change the output of the statement. The following steps would solve this problem:

1. The .**EQ**. and .**NE**. relational operators have the highest precedence level in the formula. Since those two operators have the same precedence, they are evaluated from left to right:

   value1 .**EQ**. *value2*   (*condition1* = true)

   value1 .**NE**. *value3*   (*condition2* = true)

   After the first level of operators has been evaluated, the problem now looks like this:

   If (*condition1* .**AND**. *condition2*) *then*...

2. The .**AND**. logical operator is then evaluated. Since .**AND**. requires both conditions to be true, the solution to this problem will also be *true*.

Changing precedence levels with parentheses

The order of precedence can be changed to override how the values are normally viewed. Precedence is changed by placing parentheses around specific values or operations. The information contained within the set of parentheses is evaluated before the information outside of the parentheses. If all the operators in the parentheses have equivalent precedence levels, then the operators are evaluated from left to right.

Table 2-6. Order of Precedence for PPCL Operators.

Precedence Level	**Command**	**Syntax**
1 (Highest)	Parentheses	(expression or value)

2	Alarm priority
	Arc-tangent
	Complement
	Cosine
	Natural antilog
	Natural log
	Sine
	Square root
	Tangent
	Totalized value
		ALMPRI(pt1**)**
		ATN(value1**)**
		COM(value1**)**
		COS(value1**)**
		EXP(value1**)**
		LOG(value1**)**
		SIN(value1**)**
		SQRT(value1**)**
		TAN(value1**)**
		TOTAL(pt1**)**

3	Root	**(**value1.**ROOT**.value 2**)**
4	Multiplication Division	value1 * value2 value1 / value2
5	Addition Subtraction	value1 + value2 value1 - value2

Table 2-6 (continued).
Order of Precedence for PPCL Operators.

Precedence Level	Command	Syntax
6	Equal to Not equal to Greater than Greater than or equal to Less than Less than or equal to	**.EQ.** **.NE.** **.GT.** **.GE.** **.LT.** **.LE.**
7	And Not and	**.AND.** **.NAND.**
8 (Lowest)	Or Exclusive Or	**.OR.** **.XOR.**

When the computer evaluates multiple pairs of parentheses, each pair is evaluated according to how it is positioned in the formula. Multiple parentheses are evaluated as follows:

- For a pair of parentheses defined within another pair of parentheses, the computer always evaluates the innermost set first.
- For parentheses defined as individual pairs, each pair is evaluated from left to right.

Example

The following example will help you understand how parentheses can change the way the computer evaluates formulas. This example uses two levels of precedence (Level 3 - multiplication and Level 4 - addition/subtraction).

10 - 5 + 2 * 3 = 11

While the solution to this equation is 11, the same equation would produce a different solution when parentheses are inserted as follows:

(10 - 5 + 2) * 3 = 21

The following steps will solve this problem:

1. The parentheses is the highest level of precedence and the expression with the parentheses is evaluated first. Since all the operators within the parentheses have the same precedence, the operators are evaluated in a left to right manner. The first step of the problem is evaluated as follows:

   (10 - 5 + 2) = 7

   After the parentheses in the example have been evaluated, the problem appears as follows:

   7 * 3 = 21

2. With the parentheses evaluated, the standard precedence conventions apply. With only one operation left, the multiplication operator is evaluated and produces a value of 21.

Resident points

Resident points provide time-based and system status information. The value of the point can be tested or assigned to other points. Resident points are predefined and maintained by the device. Since each device maintains its own set of resident points, a resident point cannot be directly used across a network.

Resident points can be used in PPCL programs to determine the time, date, month, and day of the month. These points can also monitor alarms, modes of operation (DAY or NIGHT), and communications between nodes. PPCL supports the following resident points:

- Alarm count (**ALMCNT**)
- Alarm count 2 (**ALMCT2**
- Battery status **($BATT**)
- Decimal time (**CRTIME**)
- Day (**DAY**)
- Day of the month (**DAYOFM**)
- Communications link (**LINK**)
- Month (**MONTH**)
- Node number (**NODE0** through **NODE99**)
- Peak Demand Limiting point **($PDL**)
- Seconds counter (**SECNDS**)
- Seconds counters (**SECND1** through **SECND7**)
- Military time (**TIME**)

Each resident point is described in more detail on the following pages.

Alarm count (ALMCNT)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **ALMCNT**

Use The value of **ALMCNT** represents the number of points within a field panel or unitary controller that are currently in the ALARM state. When a point in the device enters the ALARM state, the computer adds one to the value of **ALMCNT**. When a point returns to the NORMAL state, the field panel subtracts one from the value of **ALMCNT**.

Example

```
340  IF (ALMCNT.GT.0) THEN ON(ALARM7)
```

Notes A point must be defined as alarmable in order to be counted when that point enters the ALARM state. In all revisions of field panel firmware, the **ALMCNT** and **ALMCT2** counters work the same with enhanced alarms as they do with regular alarms.

The specific alarm level of a point does not affect **ALMCNT** and **ALMCT2**. These counters do not increment again when an enhanced alarm point changes from one alarm level to another.

If a point is disabled (*PDSB*) or operator disabled (*ODSB*), **ALMCNT** will still increment and decrement accordingly.

See also **ALMCT2, ENALM, DISALM**

Alarm count 2 (ALMCT2)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax **ALMCT2**

Use This resident point is similar to the **ALMCNT** point. **ALMCT2** specifies if the second alarm counter should be incremented when a point enters the ALARM state. When a point in the device enters the ALARM state, the computer adds one (1) to both the value of **ALMCNT** and **ALMCT2**. When a point returns to the NORMAL state, the field panel subtracts one (1) from both values.

Example

```
300  C  IF THE SECOND LEVEL ALARM COUNTER
310  C  BECOMES GREATER THAN 5, COMMAND ON
320  C  ALARM8
330  C
340  IF (ALMCT2.GT.5) THEN ON(ALARM8)
```

Notes A point must be defined as a digital point type, alarmable, and enabled for **ALMCT2** in order to be counted when that point enters the ALARM state. In all revisions of field panel firmware, the **ALMCNT** and **ALMCT2** counters work the same with enhanced alarms as with regular alarms.

The specific alarm level of a point does not affect **ALMCNT** and **ALMCT2**. These counters do not increment again when an enhanced alarm point changes from one alarm level to another.

See also **ALMCNT**, **ENALM**, **DISALM**

Battery condition ($BATT)

Physical	Logical	Unitary	CM	APOGEE
			⅄	⅄

Syntax **$BATT**

Use For field panels that have the ability to monitor the strength of their backup battery, this resident point allows you to access that status. The status of **$BATT** can be tested for a numeric value. The status can also be tested by using the backup battery status indicators.

Testing can be implemented accordingly:

- Test **$BATT** for a numeric value (0, 50 or100). If the value of **$BATT** is equal to 0, then the battery has discharged and needs to be replaced. If the value of **$BATT** is equal to 100, then the battery does not need to be replaced. A value of 50 means the battery is about to discharge and should be replaced to prevent any loss of data.
- To test **$BATT** using status indicators (**OK**, **LOW**, or **DEAD**). If **$BATT** has a status equal to **LOW** or **DEAD**, then the battery has discharged and needs to be replaced. If **$BATT** has a status equal to **OK**, then the battery does not need to be replaced.

Example 1

```
200  IF ($BATT.EQ.0) THEN ALARM(P26BAT)
```

Example 2

```
210  IF ($BATT.EQ.DEAD) THEN ALARM(P26BAT)
```

See also **LOW**, **DEAD**, **OK**

Decimal time (CRTIME)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **CRTIME**

Use This resident point maintains the current time and stores the value in a decimal format. Examples of how **CRTIME** stores values are as follows:

7:15 a.m. = 7.25

7:30 p.m. = 19.50

The values for this point can range from 0.00 to 23.999721.

Example

```
500  C
501  C  THIS CODE DEFINES A TIME PERIOD
502  C  FROM 6:45 A.M. TO 5:30 P.M. FOR
503  C  SFAN TO OPERATE.
504  C
510  IF (CRTIME.GE.6.75.AND.CRTIME.LE.17.50)
     THEN ON(SFAN)ELSE OFF(SFAN)
```

CRTIME can also be used to assign the current value of time to a virtual LAO type point which allows you to read the current time on a graphic, point log, etc. For example:

```
100  VTIME = CRTIME
```

Notes **CRTIME** is updated every second.

Day (DAY)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **DAY**

Use This resident point specifies the current day of the week. The values used for the **DAY** point are as follows:

Number	Day of the Week
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday
7	Sunday

Example

```
300  IF (DAY.EQ.1) THEN TOTRAN = 0
```

Notes These values are not related to the modes used in the **TODMOD** statement.

Day of the month (DAYOFM)

Physical	Logical	Unitary	CM	APOGEE
	✓	✓	✓	✓

Syntax **DAYOFM**

Use This resident point specifies a particular day of any month. The value for this point corresponds to the numerical representation of a specific day in a month. Valid values for the **DAYOFM** point are 1 through 31.

Example

```
160  C  THIS SECTION OF CODE DETERMINES IF
162  C  IT IS THE FIRST DAY OF THE MONTH.
164  C  IF SO, SET TOTMON TO 0.
166  C
180  IF (DAYOFM.EQ.1) THEN TOTMON = 0
```

Notes This point is helpful when you have to perform certain operations on a specific day (for example, generating a report on the first day in the month).

Communications link (LINK)

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **LINK**

Use This resident point indicates the condition of communications. Depending on the status of the communications link, a point contains one of the following values:

0 - The node where the **LINK** point resides is not communicating with the network.

1 - The node where the **LINK** point resides is actively communicating with the network.

Example

```
300  IF (LINK.EQ.0) THEN ON(ALARM)
```

See also **Node Number**

Month (MONTH)

Physical	Logical	Unitary	CM	APOGEE
✓	✓	✓	✓	✓

Syntax **MONTH**

Use This resident point specifies the current month. The values for the **MONTH** point are as follows:

Number	**Month**
1	January
2	February
3	March
4	April
5	May
6	June
7	July
8	August
9	September
10	October
11	November
12	December

Example

```
950  IF (MONTH.GE.4.AND.MONTH.LE.10) THEN
     SEASON = 1 ELSE SEASON = 0
```

Node number (NODE0 through NODE99)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **NODE***n*

n	Represents the number of a **NODE**. Acceptable node numbers for the **NODE** resident point range from 0 through 99.

Use This resident point allows the program to check the status of a node on the network. All devices or CPUs on the network occupy a node corresponding to its address. This point is generally used to test for normal operation of nodes for control strategies that depend on network communication.

Example

```
600  IF (NODE22.EQ.FAILED) THEN ON(ALARM)
```

Notes To check the operation of an Insight for Minicomputer, the node number must be between 0 and 99.

See also **Communications Link**

PDL monitor ($PDL)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **$PDL**

Use This resident point takes on the current value of the demand prediction for each calculated interval made by the **PDLMTR** statement. The point can be assigned to a virtual LAO point, displayed, and trended.

Example

```
350  KWH = $PDL
```

Seconds counters (SECNDS)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **SECNDS**

Use This resident point counts real time seconds and can be used as a timer. The computer adds one (1) to the **SECNDS** variable for every one second of real time that passes. The initial value of the **SECNDS** point is set by a PPCL command. The **SECNDS** point can be set to a maximum value of 9,999.

Example

```
890  IF (SFAN.NE.PRFON) THEN SECNDS = 0
```

Notes For APOGEE field panels, each program has a unique **SECNDS** point. This point can also be viewed in the interface using the program name, system delimiter (:)**SECNDS** format.

Seconds counter (SECND1 through SECND7)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **SECND*n***

n	The number that describes which **SECND*n*** point is referenced. Valid values are 1 through 7.

Use These seven resident points count real time seconds and can be used as timers. The computer adds one (1) to the **SECND*n*** variable for every one second of real time that passes. The value of a **SECND*n*** point can only be set by a PPCL command. The maximum value a **SECND*n*** point can be set to is 9,999.

Example

```
600  IF(SECND1.GT.15) THEN ON(RF) ELSE OFF(RF)
```

Notes For APOGEE field panels, each program has a unique **SECNDS*n*** points. These points can also be viewed in the interface using the program name, system delimiter (:)**SECNDS*n*** format.

Military time (TIME)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **TIME**

Use This resident point maintains the current time and stores the value in military time. The **TIME** value can contain a value from 0:00 to 23:59. Examples of how **TIME** stores values are as follows:

7:15 a.m. = 7:15

7:30 p.m. = 19:30

Example

```
500  C
501  C  THIS CODE DEFINES A TIME PERIOD
502  C  FROM 6:45 A.M. TO 5:30 P.M. FOR
503  C  SFAN TO OPERATE.
502  C
510  IF(TIME.GE.6:45.AND.TIME.LE.17:30)THEN
     ON(SFAN)ELSE OFF(SFAN)
```

Notes The time is updated every second. **TIME** cannot be used to assign a value to a virtual point since its value is not in a standard decimal form. **CRTIME** should be used for this purpose. **TIME** can be used in PPCL for comparison in the **IF**/**THEN**/**ELSE** statement.

Local variables

Local variables are storage locations for data. Local variables function like virtual points except that they require less memory, are predefined, and cannot be directly displayed. Local variables can contain analog or digital values.

Each set of local variables is designed for a specific task. Local variables are divided into two categories: subroutine and global storage variables.

PPCL supports the following local variables:

- **$ARG1** through **$ARG15**
- **$LOC1** through **$LOC15**
- **LOCAL** (definition of user-defined local variables in APOGEE field panels, discussed in *Chapter 4 – Syntax*)

For APOGEE field panels, each program has a unique **$ARG** and **$LOC** variables. **$LOC** points can also be viewed and commanded through the interface using the program name, system delimiter (:), local variable name format.

Each local variable is described in more detail on the following pages.

$ARG1 through $ARG15

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **$ARG*n***

n	The number that describes what **$ARG*n*** point is referenced. Valid values are 1 through 15.

Use A **$ARG*n*** variable represents an actual point name used in a subroutine. Instead of defining actual point names inside of subroutines, values are transferred to **$ARG*n*** variables. The **$ARG*n*** variables represent their corresponding points as functions and calculations are performed in the subroutine. When the value of a **$ARG*n*** variable changes, the value of the corresponding point is also updated.

When a **$ARG*n*** variable is encountered in the subroutine section of the program, the computer checks the calling **GOSUB** command for an available point name.

The following example illustrates this concept:

```
1000  GOSUB 2000 OATEMP, RMTEMP, SETPT
```

The program branches to line number 2000. When the computer encounters *$ARG1* in the program, the computer assigns the available point name's (*OATEMP*) value to *$ARG1*. When the computer encounters *$ARG2*, it assigns the next available point as defined in the **GOSUB** command.

In this example, the values would be assigned accordingly:

The value of *OATEMP = $ARG1*

The value of *RMTEMP = $ARG2*

The value of *SETPT = $ARG3*

Notes **$ARG*n*** points can only be used in subroutines.

Refer to the **GOSUB** command for more information.

$LOC1 through $LOC15

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **$LOC*n***

n	The number that describes what **$LOC*n*** point is referenced. Valid values are 1 through 15.

Use These points are used to store the results of calculations. A **$LOC*n*** variable can store an analog or digital value and can be used throughout the program. The maximum value a **$LOC** point can store is 32767. If you need to store higher values, use a virtual LPACI or LAO point. If you are using APOGEE firmware, you might consider creating a local variable by using a **LOCAL** statement.

Example 1

```
200  MIN ($LOC1,PT1,PT2,PT3)
```

Example 2

```
100  $LOC7 = (50/VALUE1+10.0) * 2.0
```

Notes Local points in APOGEE field panels can store values greater than 32,767 (up to 10,000,000).

At (@) priority indicators

At (@) priority indicators are used to monitor the priority status on points defined in a device. The @priority indicators can be implemented to test if a point is at a specific priority. An @priority indicator can also be used to command a point to a specific priority.

PPCL supports the following @priority indicators:

- Emergency (@**EMER**)
- PPCL (@**NONE**)
- Operator (@**OPER**)
- Peak Demand Limiting (@**PDL**)
- Smoke (@**SMOKE**)

When using an @priority indicator with PPCL statements, the priority level you define in that statement occupies one of the parameters. Statements such as **ON**, **OFF**, **RELEAS,** and **SET** allow you to define a total of 16 parameters in the statement. When you incorporate an @priority indicator within a PPCL statement, that indicator occupies the first parameter location. The total number of points you can define in that statement then decreases to 15. When the @priority is not used, the default priority is **EMER**.

Each @priority indicator is described in more detail on the following pages.

Emergency (@EMER)

Physical	Logical	Unitary	CM	APOGEE
	9.2		▲	▲

Syntax **@EMER**

Use This command identifies the emergency priority level indicator. The **@EMER** command is typically used in one of the following situations:

To test if a point is currently at emergency priority, an **IF**/**THEN**/**ELSE** statement would look like the following:

```
300  IF (SFAN.EQ.@EMER) THEN ON(HORN)
```

To command a point at emergency priority, the program code might look like the following example:

```
500  ON(@EMER,SFAN)
```

To release a point in the device from emergency to PPCL priority, an example of program code might look like the following:

```
700  RELEAS(@EMER,SFAN)
```

Using the @priority indicator in this manner will not release the point if it has a higher priority (SMOKE or OPER).

Notes For logical firmware 9.2 and higher, CM and APOGEE firmware, if the @priority indicator is not specified, the field panel will release points from PDL or EMER priority to NONE.

PPCL (@NONE)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax **@NONE**

Use This command identifies the PPCL priority level indicator. The **@NONE** command can be used to test if a point currently is at PPCL priority. The **IF**/**THEN**/**ELSE** statement would look like the following:

```
300  IF (SFAN.EQ.@NONE) THEN OFF(HORN)
```

Operator (@OPER)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax **@OPER**

Use This command identifies the operator priority level indicator. The **@OPER** command is typically used in one of the following situations:

To test if a point is currently at operator priority, an **IF**/**THEN**/**ELSE** statement would look like the following:

```
300  IF (SFAN.EQ.@OPER) THEN ON(HORN)
```

To command a point to operator priority, an example of program code might look like the following:

```
500  ON(@OPER,SFAN)
```

To release a point from operator to PPCL priority, an example of program code might look like the following:

```
700  RELEAS(@OPER,SFAN)
```

Notes For logical firmware 9.2 and higher, CM and APOGEE firmware, the @priority indicator must be specified to release points in any priority to NONE.

Peak Demand Limiting (@PDL)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax **@PDL**

Use This command identifies the peak demand limiting priority level indicator. The **@PDL** command is typically used in one of the following situations:

To test if a point is currently at PDL priority, an **IF**/**THEN**/**ELSE** statement would look like the following:

```
300  IF (SFAN.EQ.@PDL) THEN ON(HORN)
```

To release a point in the device from PDL to PPCL priority, an example of program code might look like the following:

```
700  RELEAS(@PDL,SFAN)
```

Using the @priority indicator in this manner does not release the point if it has a higher priority (EMER, SMOKE, and OPER).

Notes It is not recommend to release points from PDL priority unless absolutely necessary.

For logical firmware 9.2 and higher, CM and APOGEE firmware, if the @priority indicator is not specified, the field panel will release points from PDL to NONE.

Smoke (@SMOKE)

Physical	Logical	Unitary	CM	APOGEE
	9.2		⅄	⅄

Syntax **@SMOKE**

Use This command identifies the smoke control priority level indicator. The **@SMOKE** command is typically used in one of the following situations:

To test if a point is currently at smoke priority, an **IF**/**THEN**/**ELSE** statement would look like the following:

```
300  IF (SFAN.EQ.@SMOKE) THEN ON(HORN)
```

To release a point in the device from smoke to PPCL priority, an example of program code might look like the following:

```
700 RELEAS(@SMOKE,SFAN)
```

Using the @priority indicator in this manner will not release the point if it has a higher priority (OPER).

Notes For logical firmware 9.2 and higher, CM and APOGEE firmware, the @priority indicator must be specified to release points from SMOKE, EMER, PDL to NONE priority.

Point status indicators

Point status indicators are used to monitor the current status of points defined in the device. Point status indicators can be implemented to test if a point is at a specific status, and then act accordingly. Since these points are related specifically to the functions of the device, you cannot use these points over the network.

PPCL supports the following point status indicators:

- Alarm (**ALARM**)
- Alarm acknowledge (**ALMACK**)
- Auto (**AUTO**)
- Battery status - discharged (**DEAD**)
- Battery status - almost discharged (**LOW**)
- Battery status - charged (**OK**)
- Day mode (**DAYMOD**)
- Failed (**FAILED**)
- Fast (**FAST**)
- Manual override (**HAND**)
- Night mode (**NGTMOD**)
- Off (**OFF**)
- On (**ON**)
- Proof on (**PRFON**)
- Slow (**SLOW**)

Each point status indicator is described in more detail on the following pages.

Alarm (ALARM)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax *if* **(***pt1*.*eq*.**ALARM)** *then*...

pt1 A point name whose operational status is compared to the status indicator. This point can be a digital, analog, logical controller, or pulsed accumulator point type.

Use This status indicator determines if the status of point (*pt1*) is in the ALARM state. This comparison will be true if the point is in the ALARM state.

Example

```
200 IF (TEMP02.EQ.ALARM) THEN ON(FAN)
```

Alarm acknowledge (ALMACK)

Physical	Logical	Unitary	CM	APOGEE
				⅄

Syntax *if* **(***pt1*.*eq*.**ALMACK)** *then*...

pt1	A point name that is enabled for alarm acknowledgement

Use Use this indicator to compare the operational status of a point name to the alarm acknowledgement status indicator. This comparison will be true if the point alarm state has been acknowledged by an operator.

Example

```
200  IF (FAN.EQ.ALMACK) THEN ...
```

Auto (AUTO)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax *if* **(***pt1*.*eq*.**AUTO)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be a LOOAP or LOOAL point type.

Use This status indicator compares the operational status of a point to the **AUTO** status indicator. This comparison will be true if the point is in the **AUTO** state.

Example

```
200 IF (HOAFAN.EQ.AUTO) THEN ON(LIGHT3)
```

Battery status - discharged (DEAD)

Physical	Logical	Unitary	CM	APOGEE
			⅄	⅄

Syntax *if* **($BATT**.*eq*.**DEAD)** *then*...

Use This indicator compares the value of the **$BATT** resident point to determine if the backup battery is discharged. The result of the comparison will be true if the battery is discharged. This function can only be used in field panels that have the ability to monitor the strength of their backup battery.

Example

```
600  IF ($BATT.EQ.DEAD) THEN ALARM(P26BAT)
```

An alternate method to test the battery strength is to use the numeric value (0) of **DEAD**.

Example

```
600  IF ($BATT.EQ.0) THEN ALARM(P26BAT)
```

See also **LOW**, **OK**

Battery status - almost discharged (LOW)

Physical	Logical	Unitary	CM	APOGEE
			⅄	⅄

Syntax *if* **($BATT**.*eq*.**LOW)** *then*...

Use This indicator compares the value of the **$BATT** resident point to determine if the backup battery is about to discharge. The result of the comparison will be true if the battery is close to discharging. This function can only be used in field panels that have the ability to monitor the strength of their backup battery.

Example

```
600  IF ($BATT.EQ.LOW) THEN ALARM(P26BAT)
```

An alternate method to test the battery strength is to use the numeric value (50) of **LOW**.

Example

```
600  IF ($BATT.EQ.50) THEN ALARM(P26BAT)
```

See also **DEAD**, **OK**

Battery status - charged (OK)

Physical	Logical	Unitary	CM	APOGEE
			⅄	⅄

Syntax *if* **($BATT**.*eq*.**OK)** *then*...

Use This indicator is used to compare the value of the **$BATT** resident point to determine if the backup battery is operational. The result of the comparison will be true if the battery is charged. This function can only be used in field panels that have the ability to monitor the strength of their backup battery.

Example

```
600 IF ($BATT.EQ.OK) THEN NORMAL(P26BAT)
```

An alternate method to test the battery strength is to use the numeric value (100) of **OK**.

Example

```
600  IF ($BATT.EQ.100) THEN ALARM(P26BAT)
```

See also **DEAD**, **LOW**

Day mode (DAYMOD)

Physical	Logical	Unitary	CM	APOGEE
	9.1		⅄	⅄

Syntax *if* **(***pt1*.*eq*.**DAYMOD)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be defined as an LCTLR point type.

Use This status indicator determines if an equipment controller is in DAY mode. For some equipment controllers, DAY mode is also referred to as OCC (occupied) mode. If an equipment controller is in occupied mode, PPCL recognizes this status as **DAYMOD**.

Example

```
200  IF (CTLR1.EQ.DAYMOD) THEN DAYSP = 75.0
```

Notes This point is only valid for equipment controllers and is only valid when used with the LCTLR or LTCU point types.

Failed (FAILED)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax *if* **(***pt1.eq*.**FAILED)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point can be a digital, analog, logical controller, or pulsed accumulator point type.

Use This status indicator is used to compare the operational status of a point to the **FAILED** status indicator. This comparison will be true if the point is in the FAILED state.

Example

```
200  IF (AHU2.EQ.FAILED) THEN ON(HORN)
```

Fast (FAST)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax *if* **(***pt1*.*eq*.**FAST)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be defined as an LFSSL or LFSSP point type.

Use This indicator is used to compare the operational status of a point name to the **FAST** status indicator. This comparison will be true if the point is in the FAST state.

Example

```
200  IF (FAN.EQ.FAST) THEN DAMPER = 8.0
```

Manual override (HAND)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax *if* **(***pt1*.*eq*.**HAND)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be terminated on a Point Termination Module (PTM) with a manual override switch.

Use This indicator is used to compare the operational status of a point name to the **HAND** status indicator. This comparison will be true if the point PTM is currently being controlled through the use of a PTM manual override switch.

Example

```
200  IF (FAN.EQ.HAND) THEN OADAMP = 8.0
```

Night mode (NGTMOD)

Physical	Logical	Unitary	CM	APOGEE
	9.1		⅄	⅄

Syntax *if* **(***pt1*.*eq*.**NGTMOD)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be defined as an LCTLR point type.

Use This status indicator determines if an equipment controller is in NIGHT mode. For some equipment controllers, NIGHT mode is also referred to as UNOCC (unoccupied) mode. If an equipment controller is in unoccupied mode, PPCL recognizes this status as **NGTMOD**.

Example

```
200  IF (CNTRL1.EQ.NGTMOD) THEN NGTSP = 78.0
```

Notes This point is only valid for equipment controllers and is only valid when used with the LCTLR or LTCU point types.

Off (OFF)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax *if* **(***pt1*.*eq*.**OFF)** *then*...

pt1 A point name whose operational status is compared to the status indicator. This point must be defined as any LDI, LDO, L2SP, L2SL, LOOAL, LOOAP, LFSSL, or LFSSP point type.

Use This indicator is used to compare the operational status of a point name to the **OFF** status indicator. The result of the comparison will be true if the point is in the OFF state.

Example

```
200  IF (FAN.EQ.OFF) THEN RTDAMP = 3.0
```

On (ON)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax *if* **(***pt1*.*eq*.**ON)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be defined as any LDI, LDO, L2SP, L2SL, LOOAL, or LOOAP point type.

Use This indicator is used to compare the operational status of a point name to the **ON** status indicator. The result of the comparison will be true if the point is in an ON state.

Example

```
200  IF (FAN.EQ.ON) THEN RTDAMP = 3.0
```

Proof on (PRFON)

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax *if* **(***pt1*.*eq*.**PRFON)** *then*...

pt1	A point name whose operational status is compared to the status indicator. This point must be defined as any L2SP, L2SL, LOOAL, or LOOAP point type.

Use This indicator is used to compare the operational status of a point to the **PRFON** status indicator. This comparison will be true if the proof for the point is in the ON state.

Example

```
200  IF (FAN.EQ.PRFON) THEN OADAMP = 8.5
```

Slow (SLOW)

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax *if* **(***pt1.eq.***SLOW)** *then...*

pt1	A point name whose operational status is compared to the status indicator. This point must be defined as an LFSSL or LFSSP point type.

Use This indicator is used to compare the operational status of a point name to the **SLOW** status indicator. This comparison will be true if the point is in a SLOW state.

Example

```
200  IF (FAN.EQ.SLOW) THEN DAMPER = 13.0
```

Structured programming

Before writing a program, you must determine the order in which PPCL statements are executed. The method you use to organize your program determines its ease of maintenance. During the life of the program, you may need to make modifications, such as:

- Changing the set point of a control loop.
- Adding code for new equipment.
- Modifying the control strategy.

If you organize the logic of the program, then the program becomes easier to modify and maintain. The following situations can make a program difficult to maintain:

- Statements are not arranged in a logical order.
- Sections of common program code are repeated throughout the program instead of using a subroutine.
- Point names do not describe the function of the point.
- Unnecessary use and misuse of routing (**GOTO**) commands.
- With some firmware (such as APOGEE) the control strategy is duplicated within a single program using new line numbers, rather than creating a separate program in the same field panel.

It also helps to design your program code according to a structured programming style. This style produces an organized program. Errors are easier to trace because the logic flow of the program is from top to bottom. If you are unfamiliar with the program, the comment lines in the program help you understand what functions specific groups of statements perform.

Characteristics of structured programming are as follows:

- Program logic flows from the beginning to the end of the program.
- A subroutine (module) is defined for a group of statements that perform a specific task more than once during each pass of the program. A subroutine reduces the need for duplicate code lines.
- If possible, each section of program code is described and documented using comment lines.

Modular programming

Modular programming is a style of programming which logically organizes code into common functions. This style of programming can be applied to a single program, or to multiple programs running on the network or in one field panel. The objectives of modular programming are as follows:

- **To reduce program code.** If you use a section of code many times during one pass of the program, that code can be defined as a subroutine. When you need to access that routine, you transfer control to that part of the program.

- **To standardize program code.** By defining functions, procedures, and formulas in subroutines, programs become faster to create, easier to write, and require less time to test. The subroutine you use for one device can also be adapted to other devices.

Subroutines also help set programming standards that define where functionality is placed in the program. If you use an established block of code and you do not change the line numbers, then that subroutine will always be placed in the same location in the program.

For example, if you to have to perform a specific operation five different times during one pass of the program, you have two programming options. The first option is to duplicate the code in five different places in the program. This would add five times as much program code. However, when making a change to that segment, you would have to remember to change all occurrences of that code.

The second option is to use a subroutine (also called a *module*). Every time you need to perform the operation, just transfer program control to the subroutine. This process allows for a smaller amount of mainline program code. If you need to change the code, you do not have to search for all the other occurrences as explained in the first option.

In order to get the most use out of a subroutine, you should also determine if the subroutine will be accessed enough times through one pass of the program. If you implement a subroutine that has only one line of code, and it is only called one time in the program, you would create more program code than by just using straight-line program code.

Note the use of a subroutine that only contains one line of program code. A single line of program code placed in a subroutine will never be executed enough times to be of any benefit. Conversely, a subroutine containing four lines of program code becomes beneficial when it is called more than once. Plan your programs carefully so you can optimize your program code.

Table 2-7 helps you determine if a subroutine will require more lines of program code than straight-line code and when it becomes beneficial to use a subroutine.

Table 2-7. Efficient Use of Subroutines.

Number of calls	**1 Line**	**2 Lines**	**3 Lines**	**4 Lines or more**
1 Call	No	No	No	No
2 Calls	No	No	Even	Yes
3 Calls	No	No	Yes	Yes
4 Calls or more	No	Yes	Yes	Yes

To use Table 2-7, determine how many lines of code a subroutine will contain (excluding the **RETURN** statement) and how many times it will be called through one pass of the program. If the result is "NO," then it is more efficient not to use a subroutine. If the result is "EVEN," then there is no benefit for using either method. If you find the result is "YES," then it is more efficient to use a subroutine.

Using GOTO

The **GOTO** command unconditionally transfers control to another location in the program. A **GOTO** command is commonly used to skip over subroutines and blocks of code.

NOTE: Do not use the **GOTO** command to transfer program control to and from subroutines or to jump back in a program without first executing the last line of the program.

Using GOSUB

The **GOSUB** command transfers control to a subroutine. Variable arguments can be passed to the subroutine from the mainline code using **$ARG** local variables (if applicable). When the subroutine processing is complete, the program control is transferred back to the **GOSUB** statement. The values in the **$ARG** points replace their respective point values. The field panel then continues processing at the next sequential line number.

Multiple programs versus subroutines

For APOGEE field panels, the same concepts apply as previously described, except individual PPCL programs are used to replace subroutines. In some cases, time-based statements should not be used in subroutines, and therefore require a program. The **DEFINE** statement can be used to replace major portions of point names. Refer to the following example.

Five Air Handling Units (AHUs) use the same control strategy. Two possible options for PPCL programming are as follows:

Option 1 - Use one PPCL program that contains five duplicate blocks of program code.

Option 2 - Five programs that duplicate one block of program code. Each block contains one additional statement (**DEFINE**) for use with the point names. Facilities that use a standardized naming convention will benefit the most from this structure.

Program testing

To ensure that a program is working properly, it should be tested. Testing program code verifies that the program works properly under normal operating conditions. Testing also allows you to check the durability of the program by providing a wide variety of operating conditions that the program must evaluate.

Structured programming practices allow program testing to be more complete and less time consuming. Tests can be developed so that individual modules as well as the complete program are tested. Programs with a modular design are easier to test because the number of variables and lines of code that must be tested are small.

When a program is tested, the data used should adequately test all conditions that the program may encounter. Test conditions not only include testing the range at which an input responds, but also testing values outside the range of response. If an input is defined to accept values from 50°F to 70°F, use an input value of 40°F to find out how the system responds. If the program turns a point ON, find out what happens when you turn it OFF.

Test how your program will react if a sensor or fan motor would fail.

When you use values that are not expected to be encountered by the program, you discover how the program processes errors. The precautions you take in programming and testing can save you downtime when the system encounters unusual operating conditions.

Program documentation

After you have written and tested your program, the final step in completing a structured program is documentation. When using a structured method of programming, documentation of that program becomes an easier task. Programs should have some type of documentation written about their functions and the equipment they control.

The following sections describe the methods for documenting programs. One or more of these methods can be used to document operation of the program. In addition to documenting the program, these methods can also be used as planning tools when designing your PPCL program.

Internal documentation

Internal documentation refers to information written into the program lines. To prevent that information from being interpreted as program commands, a comment line is used. Comment lines allow you to enter text information describing the functionality of a specific section of code. Comment lines are especially helpful for describing subroutines and areas of program code that are difficult to understand.

The device and software you use to program PPCL determines how you enter comment lines. Comment lines require you to enter a line number followed by the letter C. For example:

```
200  C  THIS COMMAND WILL TURN ON IF
210  C  SFAN'S VALUE FALLS BELOW 68 OR
220  C  RISES ABOVE 78. THE POINT SFAN01
230  C  WILL REMAIN ON/OFF FOR AT LEAST 5
```

For information on how to properly enter comment lines, refer to the user's manual for the device or software you are using.

External documentation

External documentation refers to physical control diagrams, layouts, and information written about the program. This information can be used as a reference to learn about the functions of the control system. This section describes two of the more common methods of external documentation.

***Decision tables* -** Decision tables are similar to truth tables. Like truth tables, a decision table defines objects which are compared to each other and produce a result. The objects for a decision table are the equipment and modes of operation. A result is determined by the comparison between a piece of equipment and a mode of operation. The result of this comparison is considered to be the status of that piece of equipment during a specific mode of operation. Decision tables are useful because they provide a visual representation of how the specific equipment interacts with other equipment during each mode of operation.

Table 2-8 provides an example of a decision table that defines six pieces of equipment and four modes of operation.

Table 2-8. Example of a Decision Table.

Equipment type	Shutdown	Day mode	Smoke	Warm-up
Supply Fan	Off	On	On	On
Return Fan	Off	On	On	On
CC Valve	Closed	Modulate	Modulate	Closed
Mixing Dampers	Closed	Modulate	Modulate	Closed
Supply Fan Vol	Modulate	Modulate	Open	Modulate
Return Fan Vol	Modulate	Modulate	Open	Modulate

Pseudocode - Another method used to document a program is pseudocode. When solving a programming problem, you might ask yourself questions about what you have to accomplish. As you define what must be accomplished, you create a list of the logical steps the program should perform. The following list is an example of how pseudocode works:

1. Check mixed air temperatures
2. Check outside air temperatures
3. Open/close dampers as needed
4. Start/stop fans

Each line of pseudocode is written in a non-syntactical manner. Programmers can quickly organize their ideas before writing the code. After the code is written, pseudocode can be used as a reference to help others learn about the functionality of the program.

Steps to solving a programming problem

When you first start programming in PPCL, you should have some type of problem solving method. The following problem solving method begins with the sequence of operation, and continues through the testing and documenting of the final code solution.

Step 1 - *Read through the problem*. Read the problem to gain an understanding of what the sequence of operation is asking you to do. If you have a large sequence of operation, you may want to break it down into smaller, manageable sections.

Step 2 - *Determine modes of operation*. Determine the types of modes that will be used during the operation of the system. Examples of operational modes are:

- Day mode
- Night mode
- Emergency mode

Defining the modes of operation allows you to design code in a highly organized fashion.

Step 3 - *Identify specific controls*. Identify procedures and controls that you will write program code to solve. You should look for key phrases such as:

"This system must control..."

"Will perform..."

"Cycle..."

"Calculate..."

All of these phrases identify some type of control for which you will write PPCL code. Classify each control according to the modes you have defined.

Step 4 - *Produce pseudocode/decision tables for your modes.* After you have written down all the tasks you must accomplish, you must decide the order in which those tasks should be performed. There are various methods to organize programming tasks. These methods include:

- Pseudocode
- Decision tables

Use the method which is the easiest and most helpful for you. The time you spend organizing your ideas will help you prevent program logic flaws.

Step 5 - *Design the modules of code*. Once you have your ideas organized, start writing program code for each of the modes. Start with one mode and code the solution. Refer to Section 4, "Syntax" for help with PPCL statements. Remember to comment the program where appropriate. If you have had experience writing programs, you might write two or three modes before continuing.

Step 6 - *Enter the program into the system*. Type the program module into the computer. Remember to define any points that you use in the database.

Step 7 - *Execute and test the program code*. Depending on the experience you have writing PPCL programs, you may not write an error-free program every time. Getting programs to work properly takes time and patience.

Before you actually use a PPCL program, run the program and verify that it works. The more testing you do to simulate conditions that the program will encounter, the better your chances of detecting errors. Through testing, you may even discover an easier method for accomplishing a particular control strategy.

3

Control Option Comparisons

Introduction

The APOGEE system has many different types of control applications. To make these applications work properly, you must apply the correct program to the function you want to control. This chapter discusses the concepts behind some of the major applications. This chapter also helps you understand the procedures the system uses in performing everyday operations.

This chapter also discusses the organization of PPCL statements and how they relate to each other. For a complete description of PPCL commands identified in this chapter, refer to *Chapter 4 – Syntax*.

Duty Cycling (DC)

To understand how the Duty Cycling (DC) function works, imagine you have a home furnace to control. The duty cycling function works much like the thermostat that controls the ON/OFF times of the furnace.

The DC function shuts down equipment for short periods of time during the normal operational period of the system. Shutting off loads helps reduce operating costs. Most systems are designed to meet the expected maximum demand. Since the maximum demand is only reached a few times a year and for a short period, the system is usually oversized for the load.

When the system is not at maximum demand, the extra capacity of the system is wasted due to the system running at only partial capacity. If your home furnace only needs to run 20 minutes out of an hour to maintain the set point temperature, then the other 40 minutes of run time are spent overheating the space and wasting energy.

The following example uses DC to control a hall fan in a school building. The fan you are controlling is called *HFAN*. In order to conserve energy, you have decided to run the fan during times when students are in the hall. In this school, classes start on the hour and last for 50 minutes. There is a 10-minute period of time when the students are in that hall before the next class.

Since the hall that *HFAN* serves has good ventilation, you decide to run the fan for the last 15 minutes of the hour and for the first five minutes of the next hour. The rest of the time the fan is shut off.

Applications, which require ON/OFF switching in hourly patterns, work well with a duty cycling program.

You should be careful when points you use for DC are commanded by other applications such as Time-Of-Day (TOD). If a point is controlled by Duty Cycling and TOD during the same time period, one program may interfere with the operations of the other. Be careful when designing your programs so that functions do not conflict.

PPCL commands used for DC do not have to be defined in a specific order. DC has two commands that can be used independently of each other:

- **DC** - Turns points ON and OFF according to the schedule defined for the points. This command is best applied to equipment where the load is constant or encounters minor deviations in space temperature or humidity. Some examples of where this command is best used include a storeroom, maintenance shop, building core, or service area.
- **DCR -** Turns points ON and OFF according to a temperature defined with a dead band. Every five minutes, the **DCR** statement checks the value of the space temperature point. The temperature sensor that monitors the equipment for an area should be located to provide an accurate representation of the space temperature serviced by the equipment.

  The **DCR** command is best used to control areas where the load changes or when deviations in space temperature or humidity are undesirable. Such areas include a classroom, meeting room, lobby area, or areas located adjacent to an exterior wall of the building.

Peak Demand Limiting (PDL)

To better understand how the Peak Demand Limiting (PDL) function works, imagine you have to control a home furnace. The PDL function is similar to the high limit control on the furnace. If the temperature of the furnace gets too hot, the contacts for the burner open and shut down the burner.

In the case of building control, PDL reduces the electrical peak of a system. Electrical demand is reduced by shedding (turning OFF) electrical loads when the actual demand exceeds the set point. When demand falls below the set point (if the restored loads will not cause the set point to be exceeded), the loads are restored (turned ON).

If you have some experience with APOGEE functionality, you might identify that Duty Cycling (DC) loads also reduces the demand for the building. While this statement is true, duty cycling only controls loads according to a time schedule. The PDL function can monitor the total demand of a building, but duty cycling has no way of knowing when the system will exceed a demand set point. When a peak set point is exceeded, building owners might have to pay a higher electrical rate over a long period of time.

Example

To help you understand how PDL works, refer to the following example.

PDL is monitoring an electrical meter that supplies power to three air handling units and some low voltage equipment. Two of the air handlers are currently ON while the third air handler is about to be turned ON. When all three air handlers and the auxiliary equipment are operating at one time, the peak set point of 500 KWH will be exceeded.

After the third air handler is started, PDL predicts that the load will exceed the set point. To prevent the peak set point from being exceeded, PDL shuts OFF (sheds) loads that are not critical. Once the electrical demand of the building is operating at a level where the actual demand is less than the peak set point, the loads that were shed are restored.

PDL application programs

There are several different types of PDL application programs. A description of these programs is as follows:

Insight for Minicomputers Peak Demand Limiting

This type of PDL is run on minicomputer based systems. Refer to *Insight for Minicomputers User's Documentation Set (125-1910)* for a complete description of this PDL control method.

Target Peak Demand Limiting (TPDL)

With this version of PDL, Insight for Minicomputers monitors demand meter readings, makes a demand forecast, and assigns demand shed targets to each field panel for the calculation interval. The demand shed target is based upon the percentage of total loads that can be shed by the field panel.

Each field panel is responsible for shedding and restoring loads defined in it. If communications are lost, the field panel continues to limit demand to its last assigned target.

TPDL requires the following two PPCL commands defined in the load-handling field panel. They must be specified in the following sequence:

- **PDL** - Maintains the target kilowatt consumption level by shedding and restoring loads as needed.
- **PDLDAT** - Defines a power consuming load that will be controlled by a PDL command.

Distributed Peak Demand Limiting

This version of PDL is very similar to the TPDL. In this implementation, a field panel on the network reads the meters, forecasts the demands, and sends out targets to the other field panels on the network. Distributed PDL can only work on Protocol 2 networks.

Distributed PDL can also be used on systems that use Insight for Minicomputers, although the minicomputer is not required. All meter monitoring and demand predictions (called *forecasts*) for a meter area are done by PPCL statements in one specified field panel (also called the *predictor panel*). All meter inputs from the field must be connected directly to that field panel.

Based on the comparison of what is predicted for demand and the demand limit set point, the predictor panel decides how many kilowatts of load must be shed for the meter area. In addition to monitoring and predicting demand, the predictor panel is also responsible for keeping data for the reports.

Each predictor panel can control a total of seven load handling field panels. Each panel receives one demand target from the predictor panel. The load handling field panels maintain their demand target level by shedding and restoring loads defined under its control.

Distributed PDL uses the following five commands. These commands must be defined in the following order:

- **PDLMTR** - Monitors consumption meters to determine power usage, maintains consumption reports, predicts usage, issues warnings, and restarts meters.
- **PDLSET** - Assigns various consumption limiting set points to specific time intervals.
- **PDLDPG** - Distributes the difference between the PDL resident point and the set point.
- **PDL** - Maintains the target kilowatt consumption level by shedding and restoring loads as needed.
- **PDLDAT** - Defines a power consuming load that will be controlled by a PDL command.

When using this PDL control method, the predictor panel must have the **PDLMTR**, **PDLSET**, and **PDLDPG** statements defined in its PPCL program. This configuration assumes that the predictor panel does not have any loads defined in the panel. If the predictor panel is also controlling loads, the **PDL** and **PDLDAT** statements must also be resident in the PPCL program.

For load-handling field panels, the **PDL** and **PDLDAT** statements must be defined in the PPCL program.

Point priority

A point priority identifies the level at which a point is commanded. When statements are examined, the commanding priority of the functions is compared to the priorities of the points. A point can only be commanded by a program line whose priority is equal to or higher than the current priority of the point.

Physical firmware

Physical firmware PPCL has only two priorities: NONE and EMER. All points are considered to be at NONE priority unless their priorities are changed with the **EMON**/**EMOFF** commands. The **EMON** command turns the point ON and can change its priority to EMER. The **EMOFF** command turns the point OFF and can change its priority to NONE.

This section discusses using the NONE and EMER priorities. It does not address Insight for minicomputers priorities such as the PDL priority. Refer to the *Insight for Minicomputers User's Documentation Set (125-1910)* for more information about priorities and how they interact with PPCL programs running in the field panel.

EMON and EMOFF commands

The **EMON** and **EMOFF** commands change the status (ON or OFF) of a digital point. With that change, you can also specify if you want to change the priority of the point.

Note: Before using the **EMON** and **EMOFF** commands, you should have a good understanding of how they function. For complete descriptions of the **EMON** and **EMOFF** commands, refer to *Chapter 4 – Syntax*.

Logical firmware

The logical firmware revision you have determines the functionality of your system. Your firmware may not support the concepts presented in this section.

Revision 5.0 through 9.1 logical firmware

Revision 5.0 through 9.1 logical firmware contains the following four priority levels:

- **NONE** (PPCL priority, lowest priority level)
- **PDL** (Peak Demand Limiting)
- **EMER** (Emergency)
- **OPER** (Operator, highest priority level)

With the priorities organized in this manner, NONE is the lowest priority. Points in PDL priority can be commanded only by operations that have a PDL priority or higher. Points in EMER priority can be commanded only by operations that have an emergency priority or higher. Points in OPER can only be commanded by the operator.

Releasing point priorities

To quickly change the priority of a point to NONE, you can use the **RELEAS** command. The **RELEAS** command returns the priority of any point to NONE. If you use this command and monitor point priorities with flags, remember to set the flags to the appropriate value. If you release points to NONE priority and forget to set the priority flags, the program will not work correctly.

Revision 9.2 and higher logical firmware

Revision 9.2 and higher logical firmware support the following five priority levels:

- **NONE** (PPCL priority, lowest priority level)
- **PDL** (Peak Demand Limiting)
- **EMER** (Emergency)
- **SMOKE** (Smoke control)
- **OPER** (Operator, highest priority level)

The five priorities are listed in order from lowest to highest. A point is only commanded if the operation has a priority equal to or higher than the point's current priority. For example, a point whose priority is set to EMER can be changed by a program line with a EMER, SMOKE, or OPER priority.

The significant difference in revision 9.2 and higher logical firmware is that priorities can be directly monitored. This allows you to command and test point priorities without using priority flags. When referring to priorities, the at (@) symbol must be placed before the priority syntax. Refer to Table 3-1.

Revision 9.2 and higher logical firmware also include the SMOKE priority. The SMOKE priority is used for smoke control applications.

Table 3-1. System Variables for Priority Monitoring.

Priority	Priority syntax
Operator	**@OPER**
Smoke	**@SMOKE**
Emergency	**@EMER**
Peak Demand Limiting	**@PDL**
No priority (PPCL)	**@NONE**

Commanding and testing priorities

Logical firmware offers various methods for controlling point priorities. As previously discussed in this section, earlier revisions of logical firmware can only command a point to a specific status. For example:

```
120  OFF(SFAN,RFAN)
```

Since the introduction of revision 9.2 logical firmware, you can command points to a specific priority, as well as a specific status. For example:

```
120  OFF(@OPER,SFAN,RFAN)
```

When this command is executed, *SFAN* and *RFAN* are shut OFF and their priorities are changed to OPER. You no longer have to use an operator interface command (such as **OIP**) to change the priority of a point.

When you design your program, you might have to test the priority of the point. If you are performing numerous tasks in the program, using point priority levels is one method of preventing those tasks from colliding. Sometimes programs fail because a command, not related to a function being performed, changes a point inside of that function.

The following example demonstrates how to test for a priority condition:

```
200  IF (SFAN.EQ.@EMER) THEN ON(@EMER,RFAN)
```

This line of program code tests *SFAN* to determine if it's in EMER priority. If *SFAN* is at EMER priority, *RFAN* will be command ON. This statement verifies that *SFAN* is at EMER priority before turning ON *RFAN*.

Releasing point priorities

You can use the **RELEAS** command to quickly change the priority of a point or group of points. When using the **RELEAS** command, you can only change a higher priority to a lower priority. You cannot command a point with a low priority to a higher priority using the **RELEAS** command.

When using the **RELEAS** command to change point priorities, you have two options:

Release the point to NONE priority

Release the point at a specific priority to the NONE priority.

The first option allows you to release points to the NONE priority. In the following example, the two points (FAN1 and FAN2) are released to the NONE priority.

```
160  RELEAS(FAN1,FAN2)
```

The second option allows you to release points at a specific priority to the NONE priority. In the following example, any of the included points in the **RELEAS** command that have a priority of EMER or lower, will be released to the NONE priority:

```
300  RELEAS(@EMER,FAN1,FAN2)
```

Unitary firmware

Unitary firmware does not support the use of priority indicators and manual priority modifications.

CM and APOGEE firmware

CM and APOGEE firmware support the same priority indicators as 9.2 and higher logical firmware.

Point status

The operational condition of a point is called the point status. This section discusses the different methods used to determine the operational status of a point.

Determining point status using physical firmware

Physical firmware does not use syntactical words (ON, OFF, AUTO...) to check the status of a point. Instead, status must be determined by comparing a point value to a value.

Value	Meaning
0	OFF
1	ON

Testing for point status

To understand the concept of testing point status, you should be familiar with the point types of the APOGEE system. Table 3-2 shows the number of addresses used in each point type and the functions each address performs. You should know how these point types are addressed when defining, testing, or commanding points in physical firmware. The following 11 point types are recognized by every field panel, equipment controller, and mass storage device.

Table 3-2. The 11 Point Types and Their Organizations.

Point type	Address 1	Address 2	Address 3	Address 4
LAI	AI			
LAO	AO			
LDI	DI			
LDO	DO			
LFSSL	DO (OFF/FAST)	DO (OFF/SLOW)	DI (PROOF)	
LFSSP	DO (OFF)	DO (FAST)	DO (SLOW)	DI (PROOF)
LOOAL	DO (ON/OFF)	DO (AUTO)	DI (PROOF)	
LOOAP	DO (ON)	DO (OFF)	DO (AUTO)	DI (PROOF)
LPACI	DI (COUNT)			
L2SL	DO (ON/OFF)	DI (PROOF)		
L2SP	DO (ON)	DO (OFF)	DI (PROOF)	

Each point type is assigned a specific number of addresses. The point type determines the total number of addresses you assign for that point. Some point types require up to four addresses, while others require only one.

Example

To explain testing a point for a status, an LFSSL point type is used in this example. The LFSSL point is similar to an OFF/SLOW/FAST switch that could be used to control a fan. When you command LFSSL point to FAST, it has the same effect as turning the control switch on a fan to FAST. The actual process that occurs is somewhat more complex than turning a switch.

Single point testing

Single point testing refers to the individual comparison of a point to an ON or OFF value. The following example uses the conventions of an LFSSL point (OFF/FAST/SLOW), except that two independent LDOs and one LDI point are used in place of the LFSSL. Even though the example uses three points, those points work together like a LFSSL point to control the operation of the fan. The following names have been given to the example:

FANDO1	Point 1
FANDO2	Point 2
FANDI1	Point 3

If you want to turn the fan ON, you must decide which speed you want to set the switch (FAST or SLOW). The position you set the switch determines which point name you command. If you decide to set the fan to FAST, you would command *FANDO1*. If you decide to set the fan to SLOW, you would command *FANDO2*.

CAUTION:

When using this method of point control, it is very important that only one digital output point be ON at any time. Commanding multiple digital output points ON will cause serious damage to that piece of HVAC equipment.

If you decide to set the fan to FAST, the program code to command *FANDO1* might look like the following:

```
550  ON(FANDO1)
```

When you command *FANDO1* to FAST, you are really commanding *FANDO1* to ON. Table 3-3 shows the comparisons between the ON/OFF values of the digital points and how they relate to their respective logical positions.

Table 3-3. Digital Point Values of Logical Positions.

Names of digital output points	**Logical positions**		
	Off	**Fast**	**Slow**
Value of *FANDO1*	0 (OFF)	0 (OFF)	1 (ON)
Value of *FANDO2*	0 (OFF)	1 (ON)	0 (OFF)

To determine if the fan is ON, compare the values of *FANDO1* and *FANDO2* to one. If either point is equal to one, the fan is ON. A test for that condition might look like the following:

```
600  IF(FANDO1.EQ.1.0.OR.FANDO2.EQ.1.0) THEN...
```

To determine the true status of the fan, you have to define more specific conditions. The previous example only tested for an ON status. If you want to determine the position of the switch, you must test both points individually. An example of program code that can accomplish that test might look like the following:

```
610  IF (FANDO1.EQ.1.0) THEN ...
620  IF (FANDO2.EQ.1.0) THEN ...
```

If the first test at line 610 is TRUE, then the fan is set to the FAST position. If the second test is TRUE, then the fan is set to the SLOW position. If both tests are FALSE, then both digital output points are OFF. Since both points are OFF, you can conclude that the fan is OFF.

Depending on the application you are controlling, the logical position (SLOW, FAST, AUTO...) can always be represented in terms of being ON or OFF. When you know what status the point is controlling, you can test that point for a value of zero or one.

Single point proofing

The digital input point defined as *FANDI1* is used to verify the operation of the fan. With most digital point types, this point is optional. You can test the proof point using the same concepts as described above. When the digital input point is equal to one, the point is considered ON. When the digital input point is equal to zero, the proof point is OFF.

Bundled point testing

Bundled point testing refers to the comparison of bundled point values to an ON or OFF status. For this example, the LFSSL point will be defined to control a fan (the point name is *FAN*). The point requires three addresses, which are as follows:

Address 1	A digital output (DO) point that controls an OFF/FAST switching.
Address 2	A DO point that controls an OFF/SLOW switching.
Address 3	A digital input (DI) point used for proofing. This point is optional.

If you want to turn the fan ON, you must decide which position you want to set the switch (SLOW or FAST). If you decide to set the switch to SLOW, you would command the address that controls the OFF/SLOW switching. If you decide to set the switch to FAST, you would command the address that controls the OFF/FAST switching.

Commanding a DO address to ON does not command all other DO addresses to OFF. If two addresses that are controlling one piece of equipment are commanded ON at the same time, damage will occur to that HVAC equipment.

For example, if you wanted to set the fan to FAST, you could enter the following program code to command the fan:

```
550  ON(FAN#1)
```

When you command the first address of the fan ON, you really command that address to FAST. Table 3-4 demonstrates the comparisons between the ON/OFF values of the digital points and how they relate to their respective logical positions.

Table 3-4. Digital Point Values of Logical Positions.

Addresses for a LFSSL point	**Logical positions**		
	Off	**Fast**	**Slow**
OFF/FAST address switch	0 (OFF)	0 (OFF)	1 (ON)
OFF/Slow address switch	0 (OFF)	1 (ON)	0 (OFF)

The syntactical conventions used with this command demonstrate how to change the value of a bundled point. When you originally define a LFSSL point, you define three point addresses. The first address you define is a DO point controlling the OFF/FAST switching. The second address you define is a DO point that controls the OFF/SLOW switching. The third address you define is an optional DI point used for proofing. Since one LFSSL point name controls three addresses, you must use the pound sign (#) and a number to distinguish which address the computer should command.

To determine if the fan is ON, compare the values of *FAN* to 1. A test for an ON condition might look like the following:

```
600  IF(FAN#1.EQ.1.0.OR.FAN#2.EQ.1.0) THEN...
```

To determine the true status of the fan, you must define more specific conditions. The previous example only tests for an ON status. If you want to determine the position of the switch, you would have to test both addresses (1 and 2) individually. An example of program code that can accomplish that test might look like the following:

```
610  IF (FAN#1.EQ.1.0) THEN ...
620  IF (FAN#2.EQ.1.0) THEN ...
```

If the first test at line 610 is TRUE, then the fan is in the FAST position. If the second test is TRUE, then the fan is in the SLOW position. If both tests are FALSE, then both digital output points are OFF. Since both points are OFF, you can conclude that the fan is OFF.

Depending on the application you are controlling, the logical position (SLOW, FAST, AUTO...) can always be represented in terms of being ON or OFF. When you know the status that the point is controlling, you can test that point for a value of zero or one.

Bundled point proofing

The digital input point defined as the third address is used to verify the operation of the fan. With most digital point types, this point is optional. You can test the proof point using the same concepts as described in the *Bundled point testing* section. When the digital input point is equal to one, the point is considered ON. When the digital input point is equal to zero, the proof point is OFF.

Pulsed points

When commanding pulsed points in physical firmware, there are some precautions you should program into your code. Since pulsed points can be either ON or OFF, they are sometimes commanded to the wrong position because their status is not checked. If pulsed points are commanded incorrectly, the relays in the point will produce a *chattering* effect. A chattering relay causes the point to wear out quicker and degrade the response of the system.

Commanding pulsed points, using the following code, will help your system run more efficiently. This methodology for commanding a pulsed point only works with physical firmware.

```
100  C
101  C  THIS CODE PROVIDES A TEST FOR
102  C  COMMANDING PULSED POINTS. TYPES CAN
103  C  BE A L2SP, LOOAP, OR LFSSP POINT.
104  C  ANY POINTS STARTING WITH "PUL" ARE
105  C  PULSED POINTS. VLDO IS A POINT
106  C  FLAG.
107  C
119  C  CHECK IF THE PULSE POINTS ARE ON
120  IF (VLDO.EQ.1) THEN GOTO 140
129  C  TURN ON PULSE FLAG
130  ON(PUL1#1,PUL2#1,PUL3#1,VLDO)
140  ...
159  C  CHECK IF PULSE POINTS ARE OFF
160  IF (VLDO.EQ.0) THEN GOTO 190
169  C  TURN OFF PULSE POINTS
170  OFF(PUL1#2,PUL2#2,PUL3#2,VLDO)
179  C  TURN OFF PULSE FLAG
180  OFF(VLDO)
190  ...
```

Failed point values

If a point fails that is used in a PPCL statement, the point's last known value is still used in the PPCL statement. For example:

```
MIN(RMMIN,RM1TMP,RM2TMP,RM3TMP)
```

If *RM1TMP* is a physical point that fails when it has the lowest value of the three *RMTMP* points, the **MIN** statement will set *RMMIN* to the last known value for *RM1TMP*.

Determining point status using logical firmware

Logical firmware uses syntactical words (**ON**, **OFF**, **AUTO**...) to check the status of a point. These commands are used with **IF**/**THEN**/**ELSE** commands to determine the status of a point. The following section explains the methodology for determining the status of a point in logical firmware.

Revision 5.0 through 9.1 logical firmware

Revision 5.0 through 9.1 logical firmware supports the following ten status indicators.

- **AUTO**
- **ALARM**
- **DAYMOD (Rev. 9.1+)**
- **FAILED**
- **FAST**
- **NGTMOD (Rev. 9.1+)**
- **OFF**
- **ON**
- **PRFON**
- **SLOW**

These commands eliminate the need to test points individually for an ON or OFF status. For example, a LOOAL point can have an ON, OFF, and AUTO status. If you had to check the status of each of those three points, that particular line of program code might look confusing. Status indicators allow you to easily check the status of a point without having to compare point addresses to ON/OFF conditions.

Point status indicators can be used with **IF**/**THEN**/**ELSE** commands. When a point is at a specific status, a certain action is taken. These commands also make programs easier to understand because the syntactical words relate to the status of the point.

Testing point status

To explain the methodology of testing point status, a L2SL point is used in this example. The name of this point is *LIGHTS* and its function is to control the lights for a large conference room.

The configuration for an L2SL point uses two addresses. The first point address controls an ON/OFF switch, while the second optional point monitors a proof condition. If you have to verify that the lights are ON in the conference room, you could use the following test:

```
740  IF (LIGHTS.EQ.ON) THEN ...
```

When this line of code is executed, the *LIGHTS* point is checked to verify it is ON. If the point is not ON, then an alternate action will occur.

Revision 9.2 through 11.2 logical firmware

Revision 9.2 and through 11.2 logical firmware uses the same syntactical conventions for status indicators as Revision 5.0 through 9.1 logical firmware. For more information, refer to the *Revision 5.0 through 9.1 logical firmware* section.

CM 1.x and revision 12.0 and higher logical firmware

CM version 1.0 and higher, as well as revision 12.0 and higher logical firmware support the following status indicators:

- **ALARM**
- **AUTO**
- **DAYMOD**
- **DEAD**
- **FAILED**
- **FAST**
- **HAND**
- **NGTMOD**
- **OFF**
- **OK**
- **ON**
- **PRFON**
- **SLOW**

The three status indicators (**OK**, **LOW** and **DEAD**) are used specifically to support the **$BATT** resident point. The **HAND** status indicator is used to monitor points terminated on Point Termination Modules (PTMs) with an optional manual override switch.

APOGEE firmware

In addition to CM 1.x and revision 12.0 and higher logical firmware, APOGEE firmware supports the following additional status indicator:

- **ALMACK**

Commanding and evaluating the HAND status

The HAND status is only used in conjunction with Point Termination Modules (PTMs) in CM and APOGEE firmware. Some PTMs are built with an optional manual override switch that is a mechanical switch that enables you to control equipment at the PTM.

CAUTION:

The PTM manual override switch is not intended to be used as a safety device when performing maintenance. If you use the PTM manual override switch as a safety device, it can result in serious injury to personnel or cause damage to property in the area. Continue to follow safety procedures when performing maintenance.

Because the HAND status identifies a point being controlled by a PTM manual override switch, and not the system, you cannot command points to HAND by using an operator interface. For the same reason, points cannot be commanded to or removed from HAND status by using program statements. The only method you can use to control whether a point is placed into or taken out of HAND status is by using the PTM manual override switch.

When a point is placed in HAND, the ability to control the point is taken away from the system. Before attempting to command points that contain PTM manual override switches, it is recommended that you check the status of the PTM to ensure that the point can be commanded.

The HAND status appears as part of the overall point status and informs you that a PTM manual override switch has taken the ability to control that piece of equipment away from the operator interface and PPCL. Whenever points are in HAND status, you cannot control that equipment until the PTM manual override is switched back to the AUTO position.

When a point is at HAND status, PPCL continues to issue commands to that point. Since the point is being controlled through the use of a PTM, those commands issued by PPCL do not take effect. When the point is switched back to the AUTO position, the last command issued to that point, while in HAND, is executed.

To evaluate the status of a point while in HAND, refer to the following list:

For the HAND status:

- The point is being controlled by the PTM manual override switch.
- The point value cannot be determined.

For the HAND OFF status:

- The point is being controlled by the PTM manual override switch.
- The equipment being controlled is OFF.

For the HAND ON status:

- The point is being controlled by the PTM manual override switch.
- The equipment being controlled is ON.

CAUTION:

Be aware of how your system will react before attempting to take control of the system using a PTM override switch. Manually controlling equipment can cause the system to react in an unstable manner. You may also need to perform other functions once you return control to the system before normal operations can resume.

Unitary firmware

Unitary firmware uses the same syntactical conventions for status indicators as Revision 9.2 and higher logical firmware, but does not support **FAST**, **SLOW** and **AUTO** indicators. Refer to *Revision 9.2 and higher logical firmware* under the *Determining point status using logical firmware*.

Start/Stop Time Optimization (SSTO)

The Start/Stop Time Optimization (SSTO) feature starts and stops equipment at variable times based on calculations that determine the optimal use of the system. Start and stop times of equipment depend upon the outside and inside temperatures of the building.

For example, SSTO is controlling a lobby of a building. When the lobby opens at 8:00 a.m, the temperature must be 75°F. If the temperature of the lobby is 72°F at 7:30 a.m., SSTO determines that the system will need 10 minutes to raise the temperature 3°F to 75°F by 8:00. SSTO starts the warm-up procedure for the lobby at 7:50 a.m.

On another day, the temperature of the lobby is 69°F. SSTO computes that the system needs 20 minutes to warm the lobby to 75°F by 8:00 a.m. With this calculation, SSTO starts the warm-up procedure at 7:40 a.m.

Each time the value of the indoor or outdoor air temperature changes, SSTO recalculates the optimal values. Those values are then applied to the optimization equation so that the system does not waste energy conditioning air that does not need to be conditioned.

You should be careful when points you define for SSTO are commanded by another application such as Duty Cycling (DC). If you have a point being controlled by SSTO and DC during the same time period, one program could interfere with the operations of the other. Be careful when designing your programs so functions do not conflict.

You should use the SSTO program in areas such as extreme (outside) zones, unstable environments, and zones where temperature is affected by elements like wind, sunlight, or auxiliary sources of heating or cooling loads.

There are three PPCL commands for SSTO defined in the field panel. They must be specified in the following order:

TODMOD - This command is shared with the Time-Of-Day group of statements, but is necessary for SSTO to work. **TODMOD** defines the specific mode for each day of the week.

SSTOCO - This command establishes the thermal characteristics of the building based on how the building reacts to changes in temperature.

SSTO - This command calculates optimal start and stop times.

All optimization calculations and equipment control are performed in the field panel where the **SSTO** and **SSTOCO** commands are defined. Each program within a field panel may also perform SSTO calculations for up to 5 zones.

SSTO adjusted start and stop time

Ast (start time) and *Asp* (stop time) times will be added to or subtracted from the calculated times. If these values become large enough, they can force SSTO to start or stop at the earliest (est) or latest (lst) times.

The total adjustment to the calculated start time (*ast*) and the total adjust to the calculated stop time (*as*p) in the SSTO PPCL statement are variables used by each program within the field panel to fine-tune the calculated start time (*cs*t) and the calculated stop times (*csp*). Each day, the

SSTO statement increments or decrements these values by the heating auto-tune coefficient (*hceof4*) and the cooling auto-tune coefficient (*ccoef4*), defined in the SSTOCO statement. This time will be added to or subtracted from the calculation times.

The variables *ast* and *as*p should be set to zero upon initial setup of the SSTO statement. Typically, these values will stay below 5 times the variables *hcoef4* or *ccoef4* in a SSTO statement that has been set up properly. The variables *ast* and *asp* may change to help tune the times. If allowed to reach a high number, it is possible for *ast* to adjust the *cst* past the earliest start time (*est*) or latest start time (*lst*) and *asp* to adjust the *csp* past the latest start time (*lst*) or latest stop time (*lsp*). This means that the equipment will always start or stop at the earliest or latest start or stop time.

If the variables *ast* and/or *asp* are continually incrementing or decrementing calculated times, then the SSTO setup should be checked. The SSTOCO coefficients should be examined. The example values for these coefficients from the PPCL manual should not be used without first calculating what the number should be.

Formulas for SSTO

The formulas on the following pages are used for calculating the optimal start and stop times during the heating and cooling seasons.

For optimum start time in the cooling season

If indoor temperature is less than the desired temperature, then the optimum start time is LB.

Where:

LB	=	latest begin time defined in the **SSTO** command.

If indoor temperature is greater than or equal to the desired temperature, then the optimum start time is:

OB - (d * CC) - (d * f * (CT/10)) + AB

Where:

OB	=	occupancy begin time from **SSTO** command.
d	=	indoor temperature - desired temperature.
CC	=	cooling coefficient (*ccoef1*) from **SSTOCO** command.
f	=	outdoor temperature - desired temperature.
CT	=	cooling transfer coefficient (*ccoef3*) from **SSTOCO** command.
AB		adjust begin time from **SSTO** command.

For optimum stop time in the cooling season

If indoor temperature is less than the desired temperature, then the optimum stop time is:

OE + ((10 * CR * d)/f) + AE

Where:

OB	=	occupancy end time from SSTO command.
CR	=	cooling retention coefficient (*ccoef2*) from **SSTOCO** command.
d	=	indoor temperature - desired temperature.
f	=	outdoor temperature - desired temperature.
AE	=	adjust end time from **SSTO** command.

If indoor temperature is greater than or equal to the desired temperature, the optimum stop time is LE:

Where:

LE	=	latest end time from **SSTO** command.

For optimum start time in the heating season

If indoor temperature is less than the desired temperature, then the optimum start time is:

OB + (d x HC) - ((d * f * HT) / 25) + AB

Where:

OB	=	occupancy begin time from **SSTO** command.
d	=	indoor temperature - desired temperature.
HC	=	heating coefficient (*hcoef1*) from **SSTOCO** command.
f	=	outdoor temperature - desired temperature.
HT		heating transfer coefficient (*hcoef3*) from **SSTOCO** command.
AB		adjust begin time from **SSTO** command.

If indoor temperature is greater than or equal to the desired temperature, then the optimum stop time is LB:

Where:

LB	=	latest begin time from **SSTO** command.

For optimum stop time in the heating season

If indoor temperature is less than the desired temperature, then the optimum stop time is LE:

Where:

LE	=	latest end time from **SSTO** command.

If indoor temperature is greater than or equal to the desired temperature, then the optimum stop time is:

OE + ((25 * HR * d) / f) + AE

Where:

OE	=	occupancy end time from **SSTO** command.
HR	=	heating retention coefficient (hcoef2) from **SSTOCO** command.
d	=	indoor temperature - desired temperature.
f	=	outdoor temperature - desired temperature.
AE	=	adjust end time from **SSTO** command.

For cooling mode at occupancy begin

If indoor temperature is greater than the desired temperature plus one degree, then:

AB = AB - CA

Where:

AB	=	adjust begin time from **SSTO** command.
CA	=	cooling auto-step coefficient (*ccoef4*) from **SSTOCO** command.

If indoor temperature is less than the desired temperature minus one degree, then:

AB = AB + CA

Where:

AB	=	adjust begin time from **SSTO** command.
CA	=	cooling auto-step coefficient (*ccoef4*) from **SSTOCO** command.

For cooling mode at occupancy end

If indoor temperature is greater than the desired temperature plus one degree, then:

AE = AE + CA

Where:

AE	=	adjust end time from **SSTO** command.
CA	=	cooling auto-step coefficient (*ccoef4*) from **SSTOCO** command.

If indoor temperature is less than the desired temperature minus one degree, then:

AE = AE - CA

Where:

AE	=	adjust end time from **SSTO** command.
CA	=	cooling auto-step coefficient (*ccoef4*) from **SSTOCO** command.

For heating mode at occupancy begin

If indoor temperature is greater than the desired temperature plus one degree, then:

AB = AB + HA

Where:

AB	=	adjust begin time from **SSTO** command.
HA	=	heating auto-stop coefficient (*hcoef4*) from **SSTOCO** command.

If indoor temperature is less than desired the temperature minus one degree, then:

AB = AB - HA

Where:

AB	=	adjust begin time from **SSTO** command.
HA	=	heating auto-stop coefficient (*hcoef4*) from **SSTOCO** command.

For heating mode at occupancy end

If indoor temperature is greater than the desired temperature plus one degree, then

AE = AE **-** HA:

Where:

AE	=	adjust end time from **SSTO** command.
HA	=	heating auto-stop coefficient (*hcoef4*) from **SSTOCO** command.

If indoor temperature is less than the desired temperature minus one degree, then:

AE = AE + HA

Where:

AE	=	adjust end time from **SSTO** command.
HA	=	heating auto-stop coefficient (*hcoef4*) from **SSTOCO** command.

Time-Of-Day (TOD)

The Time-Of-Day (TOD) function is the software equivalent of the time clock. TOD starts and stops equipment according to a defined schedule. TOD can be used for running equipment on a scheduled basis. Equipment that can use the functionality of TOD are lights, doors (locking and unlocking), or exhaust fans.

CAUTION:

Be careful when points you define for TOD are commanded by another application such as Duty Cycling (DC). If you have a point being controlled by TOD and DC during the same time period, one program could interfere with the operation of the other.

Time-Of-Day commands

These commands are the actual TOD specific functions used in TOD programs. The **TODMOD** statement must come first before any number of **TOD** or **TODSET** statements. Refer to the following:

TODMOD - Defines the types of schedules (normal, weekend, etc.) for each day of the week.

TOD - Commands up to 16 digital points ON or OFF at specified times.

TODSET - Sets the values of up to ten analog points at specified start and stop times during the day.

You can also use the **HOLIDA** command in conjunction with the group of **TOD** commands. The **HOLIDA** command allows you to change the schedules for equipment during holidays. This command must be placed before any **TOD** or **TODSET** commands.

NOTE: You can also create schedules using the Time-of-Day (TOD) calendar that offers you a menu-driven interface. For complete information on defining and maintaining daily operating schedules using the TOD calendar, refer to *Field Panel User's Manual* (125-1895). If there is an APOGEE field panel available, refer to the *APOGEE Field Panel User's Manual* (125-3000) for an alternate method of TOD programming.

TOD examples

The following examples illustrate the results when defining holidays using the PPCL **HOLIDA** statement or the TOD Calendar.

Example 1: Holidays defined in TOD calendar, but not in PPCL

The TOD calendar has the following entries:

03-May-1994	Holiday
08-Sep-1994	Holiday

The program contains the following PPCL TOD commands:

```
100  TODMOD(1,1,1,1,1,1,1)
110  TOD(1,1,18:00,07:00,LITE1)
120  TOD(1,1,18:00,19:00,FAN1)
130  TOD(16,1,17:00,10:30,LITE1)
```

Based upon this schedule, every day except May 3 or September 8, *LITE1* will go ON at 18:00 and turn OFF at 07:00, and *FAN1* will go ON at 18:00 and turn OFF at 19:00. On May 3 and September 8, *LITE1* will go ON at 17:00 and turn OFF at 10:30, while *FAN1* will not be commanded.

Example 2: Holidays defined in PPCL, but not in the TOD calendar

```
50  HOLIDA(5,3,9,8)
```

The following TOD commands are defined in the TOD database:

Monday - Sunday

07:00:00	OFF	*LITE1*
18:00:00	ON	*FAN1*
18:00:00	ON	*LITE1*
19:00:00	OFF	*FAN1*

Holiday

10:30:00	OFF	*LITE1*
17:00:00	ON	*LITE1*

Based upon this database, everyday, except May 3 and September 8, *LITE1* will go ON at 18:00 and turn OFF at 07:00, and *FAN1* will go ON at 18:00 and turn OFF at 19:00. On May 3 and September 8, *LITE1* will go ON at 17:00 and turn OFF at 10:30, while *FAN1* will not be commanded.

Example 3: Holidays defined in both PPCL HOLIDA and the TOD calendar

Problems can occur if different holiday schedules are defined in the **HOLIDA** statement and defined in the TOD calendar, and then both methods are used at the same time. The equipment commanded by the PPCL TOD command for holiday mode 16 will execute not only the holidays defined in the **HOLIDA** statement, but also those defined in the TOD calendar. As the following example illustrates, the holiday schedule will result in the equipment operating over its required timetable.

NOTE: If you choose to define a holiday using the TOD calendar and the PPCL **HOLIDA** statement, make sure that the holiday is defined as the same day in both places.

In Example 3, the **HOLIDA** statement defines these holidays:

```
100  HOLIDA(4,1,6,14)
```

The TOD calendar defines the following holidays:

03-May-1994	Holiday
08-Sep-1994	Holiday

The following TOD commands are defined in the TOD database:

Monday - Sunday

07:00:00	OFF	*LITE1*
18:00:00	ON	*FAN1*
18:00:00	ON	*LITE1*
19:00:00	OFF	*FAN1*

Holiday

10:30:00	OFF	*LITE1*
17:00:00	ON	*LITE1*

Because TOD recognizes holidays defined in PPCL and in the TOD calendar, *LITE1* will operate according to the holiday schedule on four days: 4/1, 5/3, 6/14, and 9/8. This would cause a problem if *LITE1* was originally assigned to operate on a holiday schedule of May 3rd and September 8th.

IF/THEN/ELSE Time-Of-Day

Although Time-Of-Day uses actual commands dedicated to performing TOD functions, time scheduling can be accomplished using the **IF**/**THEN**/**ELSE** commands.

The following example illustrates TOD scheduling using conditional testing:

One of the simplest applications for time scheduling using conditional tests is the control of lights. You decide to control the outdoor parking lights for a building using **IF**/**THEN**/**ELSE** commands. You set up a schedule so that the lights are ON from 7:00 p.m. until 5:00 a.m. The **IF**/**THEN**/**ELSE** statement might look like the following:

```
100  C  THIS LINE OF CODE TURNS THE LIGHTS
102  C  OFF AT 5:00 AM AND ON AT 7:00 PM.
104  C
110  IF(TIME.GT.5:00.AND.TIME.LT.19:00)THEN
     OFF(LITES)ELSE ON(LITES)
```

APOGEE interaction with PPCL TOD

APOGEE field panels contain an application called *equipment scheduling*, which replaces TOD in PPCL. Some portions of TOD programming are shared by equipment scheduling, specifically the system calendar. Equipment scheduling uses *replacement days* and *special days*. The R1 replacement day in equipment scheduling corresponds to the holiday in the **TODMOD** statement.

4

Syntax

ACT

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **ACT(***line1*,...,*line16***)**

line1 through line16	Valid line numbers for commands within the same device as the control command. You must enter line numbers as integers, which can range from 1 to 32,767.

Use This command activates from 1 to 16 lines of PPCL code so they can be examined and executed. **ACT** only enables PPCL lines that are specifically defined in the command. A range of PPCL lines cannot be defined using the **ACT** command.

Example

```
100  IF (TIME.GT.8:00.AND.TIME.LT.17:00) THEN
     ACT(120) ELSE DEACT(120)
```

Notes The **ACT** command only affects PPCL lines defined in the same device as the statement.

See also **DEACT**, **DISABL**, **ENABLE**

ALARM

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax ALARM(pt1,...,pt16)

pt1 *through* *pt16*	Names of the points to be alarmed. They have to be in the same device. They cannot be local variables.

Use This command forces a point into the ALARM state. Up to 16 points can be placed into an ALARM state on a single line.

Example

```
100  IF (ROOM.GT.80.0) THEN ALARM(ROOM25)
```

Notes The status *AC* is displayed when a point is commanded to the ALARM state.

Points must be defined as alarmable and be enabled for alarming in order to be placed into the alarm-by-command state.

See also **DISALM, ENALM, HLIMIT, LLIMIT, NORMAL**

AUTO

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **AUTO(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LOOAL or LOOAP through type points.

Use This command sets an ON/OFF/AUTO point to the AUTO position. It can be used only with LOOAL or LOOAP points. Up to 16 points can be changed with one command.

Example

```
100  AUTO(EFAN1,EFAN2,EFAN3)
```

See also **FAST, OFF, ON, SLOW**

DAY

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **DAY(***pt1*,...,*pt16***)**

pt1 through pt16	Point names of logical controller *through* (LCTLR) points.

Use This command changes a logical controller point to DAY mode. Logical controller point types can be commanded into DAY or NIGHT mode.

Example

```
100  IF (TIME.LT.7:00.OR.TIME.GT.18:00) THEN
     NIGHT(LCTLR2) ELSE DAY(LCTLR2)
```

For some equipment controllers, DAY mode is also referred to as OCC (occupied) mode. If an equipment controller is in the occupied mode, PPCL recognizes this status as DAY.

See also **NIGHT**

DBSWIT

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **DBSWIT(***type***,***input***,***low***,***high***,***pt1***,...,***pt12***)**

type	The type of dead band switch action. Valid values are 0 and 1. **0** = All output points (*pt1*,...,*pt12*) are commanded ON when the input point value rises above the high limit, and are commanded OFF when the input point value falls below the low limit. **1** = All output points (*pt1*,...,*pt12*) are commanded ON when the input point value falls below the low limit, and are commanded OFF when the input point value rises above the high limit.
input	Point name of the analog point. This can also be a local variable.
low	Represents the low temperature at which a switching action occurs. This parameter can contain a decimal or integer value. The parameter can be defined as a point name, local variable, or number.
high	Represents the high temperature at which a switching action occurs. This parameter can contain a decimal or integer value. The parameter can be defined as a point name, local variable, or number.

pt1 through pt12	Names of the points to be turned ON and OFF. Local variables can be used.

Use This command is used to obtain an ON/OFF switching action of an output point. The operational status of the point is regulated by an analog point using a predefined dead band. This is the software equivalent of a thermostat.

Example 1

```
200  DBSWIT(1,RMTEMP,LDBAND,HDBAND,SFAN, RFAN)
```

Example 2

```
200  DBSWIT(1,RMTEMP,55,58,SFAN,RFAN)
```

DC

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Syntax **DC(***pt1*,*pat1*,...,*pt8*,*pat8***)**

pt1 through pt8	Point names of the output points to be duty cycled. Acceptable point types are LDO, LOOAL, LOOAP, L2SL, and L2SP. Local, virtual, and physical points are valid for *pt1*, *pt2*, *pt3*, and *pt4*, although virtual and physical points must be of the type specified.
pat1 through pat8	Hourly pattern that describes how each point is duty cycled. Each pattern consists of four code numbers between 0 and 7. Each digit represents the on/off pattern for 15 minutes of the hour. Table 4-1 lists the code numbers for each 15-minute interval. Valid values include local, physical, and virtual points, as well as integers.

Table 4-1. Duty Cycle Patterns and Numerical Codes.

First 5 minutes	Second 5 minutes	Third 5 minutes	Duty cycling code number for 15-minute period
OFF	OFF	OFF	0
ON	OFF	OFF	1
OFF	ON	OFF	2
ON	ON	OFF	3
OFF	OFF	ON	4
ON	OFF	ON	5
OFF	ON	ON	6
ON	ON	ON	7

Use This command is used to save energy by reducing the run time of oversized equipment. A 1-hour interval of time is broken down into four 15-minute segments. Each segment is then broken into three 5-minute patterns. The **ON** and **OFF** commands defined in the patterns are represented by numeric codes. Table 4-1 lists the numeric codes and their ON/OFF patterns. After the patterns are defined, the code numbers are entered in reverse order. Refer to the following example:

```
1000  C THIS DC COMMAND DUTY CYCLES EFAN1
1010  C ACCORDING TO THE FOLLOWING SCHEDULE:
1020  C
1030  C FIRST 15 MINUTES - OFF, OFF, OFF (0)
1040  C SECOND 15 MINUTES - ON, OFF, ON (5)
1050  C THIRD 15 MINUTES - OFF, ON, ON (3)
1060  C FOURTH 15 MINUTES - OFF, OFF, ON (1)
1070  DC(EFAN1,1350)
```

The code numbers and patterns are evaluated by reading the time in 5-minute intervals from *right* to *left*. The *first* 15-minute pattern is determined by the right-most code digit. The *last* 15-minute pattern is determined by the left-most code number.

Notes The **DC** command has a priority of NONE. Therefore, the PPCL program must be structured with **IF**/**THEN**/**ELSE** commands to prevent conflicts between **DC** and other commands with the same priority.

See also **DCR**

DCR

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Syntax **DCR(***pt1***,***temp1***,***high1***,***low1***,...,***pt4***,***temp4***,***high4***,***low4***)**

pt1 through pt4	Point names of output points to be duty cycled. Acceptable point types for this command are LDO, logical, physical, and virtual. Virtual and physical types must be of the type specified.
temp1 through temp4	Point names of the space temperature points.
high1 through high4	High temperature limits of the space points. Any number entered for this parameter can be a decimal, integer, point name, or local variable.
low1 through low4	Low temperature limits of the space points. Any number entered for this parameter can be a decimal, integer, point name, or local variable.

Use This command duty cycles an output point to keep a corresponding temperature within a dead band defined with low and high values. This function allows you to define up to four points (*pt1* through *pt4*). For each point, you must define a *temp*, *high,* and *low* parameter. An **ON**/**OFF** decision is made every 5 minutes according to the value of the parameters. The output point is **ON** when the temperature point is above the high limit or below the low limit. When the point value is within the range of the high and low limit, it is commanded **OFF**.

Example

```
200 C   THIS COMMAND WILL TURN ON IF
210 C   SFAN'S VALUE FALLS BELOW 68 OR
220 C   RISES ABOVE 78. THE POINT SFAN01
230 C   WILL REMAIN ON/OFF FOR AT LEAST 5
240 C   MINUTES ON EACH CYCLE.
250 DCR(SFAN01,RM109,78.0,68.0)
```

Notes The **DC** command has a priority of NONE. Therefore, the PPCL program must be structured with **IF**/**THEN**/**ELSE** commands to prevent conflicts between **DCR** and other commands with the same priority.

See also **DC**

DEACT

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Syntax **DEACT(***line1***,...,***line16***)**

line1 through line16	Valid line numbers for commands, which reside in the same device as the control command. The numbers must be entered as integers ranging from 1 to 32,767.

Use This command allows you to disable a maximum of 16 lines of PPCL which prevents them from being examined or executed. **DEACT** only disables PPCL lines that are specifically defined in the command. A range of PPCL lines cannot be defined using the **DEACT** command.

Example

```
100  IF (TIME.GT.8:00.AND.TIME.LE.17:00) THEN
     ACT(120,130) ELSE DEACT(120,130)
```

Notes The **DEACT** command only affects the lines of PPCL program for the device where the program resides.

See also **ACT**, **DISABL**, **ENABLE**

DEFINE

Physical	Logical	Unitary	CM	APOGEE
				⅄

Syntax **DEFINE(***abbrev,string***)**

abbrev	Text string used in other PPCL statements, representing the string parameter.
string	Actual text string that will be substituted where the abbreviation is used. The string text usually contains a significant portion of a long point name.

Use This statement creates an abbreviated notation for a long point name. When used in the program, a percentage (%) must be placed before and after the abbreviation.

Example

```
10   DEFINE(AHU,"BUILDING1.AHU01.")
20   ON("%AHU%SFAN")
```

Without the use of **DEFINE**, this same line of code would look like the following:

```
20   ON("BUILDING1.AHU01.SFAN")
```

Notes This statement is executed when added to the field panel and does not require enabling or execution in the normal program flow.

DEFINE allows program logic to be easily duplicated provided your facility uses structured point names.

DISABL

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **DISABL(***line1***,...,***line16***)**

line1 through line16	Valid line number for commands which reside in the same device as the control command. The numbers must be entered as integers ranging from 1 to 32,767.

Use This command allows you to disable a maximum of 16 lines of PPCL which prevents them from being examined or executed. **DISABL** only disables PPCL lines that are specifically defined in the command. You cannot define a range of PPCL lines using the **DISABL** command.

Example

```
100  IF (TIME.GT.8:00.AND.TIME.LE.17:00) THEN
     ENABLE(120,130) ELSE DISABL (120,130)
```

Notes The **DEACT** and **DISABL** commands can be used interchangeably.

The **DISABL** command only affects the lines of PPCL program for the device where the program resides.

See also **ACT, DEACT, ENABLE**

DISALM

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **DISALM(***pt1***,...,***pt16***)**

pt1 through pt16	Names of the points that should have alarm reporting disabled.

Use This command disables the alarm printing capabilities for specified points. Up to 16 points can be enabled for alarm reporting by one command. The point status changes to *PDSB* after it has been **DISALM**ed.

Example

```
50  IF (SFAN.EQ.OFF) THEN DISALM(ROOM1) ELSE
    ENALM(ROOM1)
```

Notes This command cannot be used for points that do not reside in the device in which the control program is written. **DISALM** cannot be used to directly disable alarm reporting over the network.

See also **ALARM**, **ENALM**, **HLIMIT**, **LLIMIT**, **NORMAL**

DISCOV

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	

Syntax **DISCOV(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of the points for which Change-Of-Value (COV) reporting is disabled. Up to 16 points can be controlled by a single **DISCOV** command.

Use This command is used to stop the reporting of updated values (COVs) by devices to the Insight Minicomputer on a Protocol 1 trunk. On a Protocol 2 trunk, the reporting of COVs will be stopped only if the minicomputer is defined as node 100. When this command is executed, all operations that use COVs to function, will stop. Until the points defined in this command are enabled, updates to graphics, archiving, COV printing, alarming, and in some cases evaluation of equations, will stop. Points defined in the statement must reside in the same device as the **DISCOV** command.

Example

```
50  IF (SFAN.EQ.OFF) THEN DISCOV(ROOM1, ROOM2)
    ELSE ENCOV(ROOM1,ROOM2)
```

See also **ENCOV**

DPHONE

Physical	Logical	Unitary	CM	APOGEE
	9.1		⅄	**2.1**

Syntax **DPHONE(***pn#1***,...,***pn#16***)**

pn#1 through pn#16	Telephone ID numbers that are defined in the device.

Use This command disables telephone ID numbers. The telephone ID number represents the numeric code given to the actual telephone number defined in the device. You can disable a maximum of 16 telephone number IDs with the **DPHONE** command.

Example

```
532  DPHONE(1,2,3,5,6)
```

Notes The **DPHONE** command cannot be used over a network.

See also **EPHONE**

EMAUTO

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **EMAUTO(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LOOAL or LOOAP points.

Use This command is used to change ON/OFF/AUTO points to the AUTO (local control) state with emergency priority. You can command up to 16 LOOAP or LOOAL points.

See also **EMFAST, EMOFF, EMON, EMSET, EMSLOW**

EMFAST

Physical	Logical	Unitary	CM	APOGEE
	✓		✓	✓

Syntax **EMFAST(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LFSSL or LFSSP points.

Use This command is used to change FAST/SLOW/STOP points to the FAST state with emergency priority. You can command up to 16 LFSSP or LFFSL points.

See also **EMAUTO, EMOFF, EMON, EMSET, EMSLOW**

EMOFF

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Logical syntax **EMOFF(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LDI, LDO, L2SL, L2SP, LOOAL, or LOOAP.

Physical syntax **EMOFF(***prior***,***pt1***)**

prior	Identifies the priority that *pt1* should be commanded to. Valid options are: **501** - Commands a point OFF and places the point into EMER priority. **32523** - Commands a point OFF and places the point into NONE priority.
pt1	Point name of a LDI, LDO, L2SL, L2SP, LOOAL, or LOOAP point.

Use When this command is executed, the points defined are commanded OFF. Depending on the type of firmware you have and how you define the parameters, the priority of the point can also be changed. You can command a total of 16 points.

See also **EMAUTO, EMFAST, EMON, EMSET, EMSLOW**

EMON

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Logical syntax **EMON(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LDI, LDO, L2SL, L2SP, LOOAL, or LOOAP.

Physical syntax **EMON(***prior***,***pt1***)**

prior	Identifies the priority that *pt1* should be commanded to. Valid options are: **501** - Commands a point ON and places the point into EMER priority. **32523** - Commands a point ON and places the point into NONE priority.
pt1	Point name of a LDI, LDO, L2SL, L2SP, LOOAL, or LOOAP point.

Use When this command is executed, the defined points are commanded ON. Depending on the type of firmware you have and how you define the parameters, the priority of the point can also be changed. You can specify a maximum of 16 points with a single command.

See also **EMAUTO**, **EMFAST**, **EMOFF**, **EMSET**, **EMSLOW**

EMSET

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **EMSET(***value***,***pt1***,...,***pt15***)**

value	All the points defined in the statement will be commanded to this analog value. This parameter can be a integer, decimal, point name, or local variable.
pt1 through pt15	Point names of LAI or LAO points.

Use This command is used to set the value of analog points with emergency priority. You can specify a maximum of 15 points with a single command.

See also **EMAUTO, EMFAST, EMOFF, EMON, EMSLOW**

EMSLOW

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **EMSLOW(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LFSSL or LFSSP points.

Use This command is used to change FAST/SLOW/STOP points to the SLOW state with emergency priority. You can command a maximum of 16 LFSSP or LFFSL points.

See also **EMAUTO, EMFAST, EMOFF, EMON, EMSET**

ENABLE

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **ENABLE(***line1***,...,***line16***)**

line1 through line16	Valid line numbers for commands within the same device as the control command. Line numbers must be entered as integers from 1 to 32,767.

Use This command allows you to examine and execute lines of PPCL code. You can enable a maximum of 16 lines with a single **ENABLE** command. **ENABLE** only enables PPCL lines that are specifically defined in the command. A range of PPCL lines cannot be defined with the **ENABLE** command. You must specify each PPCL line separately.

Example

```
100  IF (TIME.GT.8:00.AND.TIME.LT.17:00) THEN
     ENABLE(120) ELSE DISABL(120)
```

Notes The **ACT** and **ENABLE** commands can be used interchangeably.

The **ENABLE** command only affects the lines of PPCL program for the device where the program resides.

See also **ACT**, **DEACT**, **DISABL**

ENALM

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **ENALM(***pt1***,...,***pt16***)**

pt1 through pt16	Point names whose alarm reporting *through* capabilities are to be enabled.

Use This command enables the alarm printing capabilities for the specified points. Up to 16 points can be enabled for alarm reporting by one command.

Example

```
50  IF (SFAN.EQ.ON) THEN ENALM(ROOM1) ELSE
    DISALM(ROOM1)
```

Notes This command cannot be used for points that do not reside in the same device as the **ENALM** command.

This command reverses the **DISALM** command. Points must still be set up for alarming and have the ability to report alarms.

This command does not override *ODSB*.

See also **ALARM**, **DISALM**, **HLIMIT**, **LLIMIT**, **NORMAL**

ENCOV

Physical	Logical	Unitary	CM	APOGEE
✓	✓		✓	

Syntax **ENCOV(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of the points for which Change-Of-Value (COV) reporting is enabled. A maximum of 16 points can be controlled by a single **ENCOV** command.

Use This command is used to enable the reporting of updated values (COVs) by devices to the Insight Minicomputer on a Protocol 1 trunk. On a Protocol 2 trunk, the reporting of COVs can be enabled only if the minicomputer is defined as node 100. When this command is executed, all operations that use COVs to function, will start reporting. Unless the points defined in this command are disabled, updates to graphics, archiving, COV printing, alarming, and in some cases evaluation of equations are reported. Points defined in the statement must reside in the same device as the **ENCOV** command.

Example

```
50  IF (SFAN.EQ.ON) THEN ENCOV(ROOM1, ROOM2)
ELSE DISCOV(ROOM1,ROOM2)
```

See also **DISCOV**

EPHONE

Physical	Logical	Unitary	CM	APOGEE
	9.1		⅄	**2.1**

Syntax **EPHONE(***pn#1***,...,***pn#16***)**

pn#1 through pn#16	Telephone ID numbers that are defined in the device.

Use This command enables a telephone ID number. The telephone ID number represents the numeric code given to the actual telephone number defined in the device. You can enable a maximum of 16 telephone number IDs with the **EPHONE** command.

Example

```
532  EPHONE(1,2,3,5,6)
```

Notes The **EPHONE** command cannot be used over a network.

See also **DPHONE**

FAST

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **FAST(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LFSSL or LFSSP type points.

Feature added to revision 9.2 logical firmware, CM and APOGEE firmware:

FAST(*@ prior***,** *pt1***,...,***pt15***)**

@prior	Defines a specific point priority.
pt1 through pt15	Names of LFSSL or LFSSP type points.

Use This command changes the operational status of a FAST/SLOW/STOP point to FAST. A single **FAST** command can be used to change the operational status of up to 16 LFSSL or LFSSP points. If you change the priority of the points, you can only define a total of 15 points.

Example

```
10  IF (RMTEMP.GT.78.0)THEN FAST(@NONE,
    FAN1,FAN2)
```

See also **AUTO, OFF, ON, SLOW**

GOSUB

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Logical syntax

GOSUB *line# pt1,...,pt15*

or - (parentheses are optional)

GOSUB *line# (pt1,...,pt15)*

line#	Line number. This number indicates which program command line is to be executed next (i.e., where the subroutine starts). This line must be a value from 1 through 32,767.
pt1 through pt15	Point name or local variables whose values are to be passed to the subroutine. They can be omitted if the subroutine does not use variable arguments.

Physical syntax

GOSUB *line#*

line#	Line number. This number indicates which program command line is to be executed next (i.e., where the subroutine starts). This line must be a value from 1 through 32,767.

Use

When this command is executed, the next line to be evaluated will be the line specified in the command. When the computer encounters a **RETURN** command in the subroutine, the next line executed will be the line following the **GOSUB** statement.

Subroutines are particularly useful for programs in which the same calculation is carried out several times using different values.

The variable parameters in a subroutine are designated by the resident point names **$ARG1** through **$ARG15**. These variables can be used by PPCL code in the subroutine in the same way that point names and constants are used. To explain the concept of subroutine program control, a hypothetical section of program code is provided.

Program flow sequence

The following example demonstrates how a number of PPCL commands are used in subroutines. The main subroutine commands, **GOSUB** and **RETURN** are required for all subroutines. The **$ARG** command demonstrates how values are passed between subroutines. The **GOTO** command, which is used to bypass subroutines, is also shown in this example.

Refer to Figure 4-1 as you follow the program flow sequence. This program uses the variables *PT1* and *PT2* to demonstrate the passing of values between subroutines. The program flow through one complete pass is as follows:

1. At line 130, the program branches to line 1010.
2. At line 1010, the program assigns a value of 10 to *PT1* and the value of 20 to *PT2*. Line 1030 branches back to the line after the **GOSUB** command (at line 130). After the program returns to the main line code, the

values of the two points become:

PT1 = 10
PT2 = 20

3. The program continues to sequentially evaluate program lines until it encounters the **GOSUB** command at line 300. The program branches to line 2010. This command also defines two values that are passed (*PT1* and *PT2*).

4. At line 2010, a **$ARG*n*** point is assigned a value. When the program encounters a **$ARG*n*** point, it checks the calling **GOSUB** command for point names. The **GOSUB** command used to call this subroutine contains two point names. When the program encounters the first **$ARG*n*** variable at line 2010, it takes the first point name defined in the **GOSUB** command (*PT1*) and stores that value in *$ARG1* (10). Line 2010 also adds one (1) to the value of *$ARG1*.

   When the program encounters the second **$ARG*n*** variable at line 2020, it takes the second point name defined in the **GOSUB** command (*PT2*) and stores that value in *$ARG2* (20). Line 2020 also adds one (1) to the value of *$ARG2*.

5. Line 2030 returns control of the program to the line after the **GOSUB** command (at line 300). At line 400, the program executes the **GOTO** command to line 3000 that returns control back to mainline code. After one pass of the program, the values of the points after being updated by the **$ARG*n*** variables, will be as follows:

   PT1 = 11
   PT2 = 21

Example

```
100  C MAINLINE PROGRAM FOR LOGICAL FIRMWARE
110  C LEARNING HOW TO USE GOSUB, RETURN, AND
112  C $ARG COMMANDS.***
120
130  GOSUB 1010
     ...
     ...
     ...
300  GOSUB 2010 PT1, PT2
     ...
     ...
     ...
400  GOTO 3000



1000  C --SUBROUTINE 1
1001  C THIS SUBROUTINE ASSIGNS NUMBERS
1002  C TO PT1 AND PT2. IT THEN RETURNS
1004  C CONTROL TO THE MAIN PROGRAM.
1008  C
1010  PT1 = 10
1020  PT2 = 20
1030  RETURN


2000  C --SUBROUTINE 2
2001  C THIS SUBROUTINE PASSES THE VALUE OF
2002  C PT1 AND PT2 TO THEIR RESPECTIVE $ARG
2003  C POINTS. IT THEN ADDS ONE TO BOTH $ARG
2004  C POINTS AND RETURNS THE NEW VALUES TO
2005  C PT1 AND PT2.
2006  C
2010  $ARG1 = $ARG1 + 1
2020  $ARG2 = $ARG2 + 1
2030  RETURN


3000  C After the subroutine is through
3002  C executing, the program returns to
3004  C mainline code.
```

Figure 4-1. Example Program with Two Subroutines.

Nested subroutines

A multiple level subroutine is a block of program code that is called from within another subroutine. Each call you make within a subroutine constitutes a level. You are limited to a total of eight levels.

If you need to use multi-level subroutines, the rules that govern **$ARG*n*** variables and point name declarations for **GOSUB** commands change slightly. When using **$ARG*n*** variables in a multi-level subroutines, you cannot share the **$ARG*n*** variables between subroutine levels. Once a value is assigned to a **$ARG*n*** variable, you must retain that value by not assigning another value to that variable.

The following example demonstrates the use of **$ARG*n*** variables and point declarations used in multiple level subroutines. This example contains a main line section of program code and three subroutines. It demonstrates a method to preserve the values of **$ARG*n*** variables as you transfer program control among subroutines.

Refer to Figure 4-2 as you follow the steps of the example program code:

1. During the evaluation of mainline code, the program encounters a subroutine call at line 1500. The **GOSUB** command transfers control to line 2030. This command also passes the value of *PT1* to the subroutine.
2. While executing the first subroutine, the program encounters a call to a second subroutine (located at line 2500). Since this call is defined within another subroutine, and a variable has been passed (*PT1* to *$ARG1*), you must retain the value of *$ARG1* by

defining *$ARG1* in the **GOSUB** command used to call the second subroutine.

If you do not define *$ARG1* in the call to the second subroutine, any value you introduce in the second subroutine replaces the value in *$ARG1*. By redefining *$ARG1*, you protect that value by passing it to the second subroutine.

As you introduce a new value in the second subroutine (the example defines a point called *PT2*), then that point is defined after the **$ARG*n*** variable. Line 2500 demonstrates a **GOSUB** command which saves a previously used **$ARG*n*** value and passes a new point called *PT2*.

As you work with the variables *$ARG1* and *PT2* in the second subroutine, *$ARG1* continues to represent the value of *PT1*, while *$ARG2* represents the value of *PT2*.

3. At line 3200, the second subroutine calls a third subroutine. In order to preserve the values of *$ARG1* and *$ARG2*, you must define them in the **GOSUB** command (as shown in line 3200) that calls the third subroutine. Note how *$ARG1* and *$ARG2* are placed in a sequential order to maintain their previous point values.

```
1000  C MAINLINE PROGRAM FOR LOGICAL FIRMWARE
1010  C
1020  C   THIS PROGRAM TEACHES HOW GOSUBS AND
1030  C   $ARG POINTS WORK IN MULTIPLE
1040  C   LEVEL SUBROUTINES.
1050  C
1100  ...
1200  ...
1300  ...
1350  C THIS GOSUB CALLS THE FIRST SUBROUTINE
1360  C AND PASSES A VALUE DEFINED IN PT1 TO
1370  C TO $ARG1.
1380  C
1500  GOSUB 2030 PT1

2000  C
2010  C SUBROUTINE #1
2020  C
2030  ...
2040  ...
2050  ...
2060  C IN ORDER TO PRESERVE THE VALUE LOCATED
2070  C IN $ARG1, YOU MUST DEFINE $ARG1 IN THE
2080  C SECOND SUBROUTINE'S CALLING GOSUB. THE
2090  C PROGRAM ALSO PASSES A SECOND POINT
2100  C VALUE NAMED PT2.
2110  C
2500  GOSUB 3030 $ARG1,PT2
2510  RETURN

3000  C
3010  C SUBROUTINE #2
3020  C
3030  ...
3040  ...
3050  C IN ORDER TO PRESERVE THE VALUES
3060  C LOCATED IN $ARG1 AND $ARG2, YOU MUST
3070  C DEFINE BOTH POINTS IN THE THIRD
3080  C SUBROUTINE'S CALLING GOSUB. THE
3090  C PROGRAM ALSO PASSES A THIRD POINT
3100  C VALUE CALLED PT3.
3200  GOSUB 4020 $ARG1,$ARG2,PT3
3210  RETURN

4000  C SUBROUTINE #3
4010  C
4020  ...
4030  ...
4040  RETURN
```

Figure 4-2. Multiple Level Subroutines.

As the program branches from the second subroutine to the third subroutine, *$ARG1* and *$ARG2* continue to represent *PT1* and *PT2* respectively. Both **$ARG*n*** points retain their values because they are passed as parameters in the **GOSUB** command. Line 3200 also defines a new point called *PT3*. Note how *PT3* is placed after the two **$ARG*n*** variables. Any new points you declare in the subroutine must be placed after the variables whose value you want to retain.

Notes The last line of a subroutine must be a **RETURN** command.

Do not use time-based commands such as **LOOP**, **SAMPLE**, **TOD**, and **WAIT** inside a subroutine.

Do not transfer control out of a subroutine without using a **RETURN** command. A **GOTO** command can be used inside of a subroutine only if the command does not transfer program control out of that subroutine.

A **GOSUB** command cannot be used in an **IF**/**THEN**/**ELSE** command.

A **GOSUB** command can only reference point names or local variables.

See also **RETURN**, **GOTO**

GOTO

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **GOTO** *line#*

line#	The next line number to which execution is directed. This line must be a number from 1 to 32,767.

Use This command is used to control program execution by branching to a different section of the program.

Example

```
10  IF (FANRUN.GE.1000) THEN GOTO 50
```

Notes A **GOTO** command should only transfer program control to a sequentially higher line number.

If the line number indicated in the **GOTO** does not exist, execution is transferred to the next line after the line number specified in the **GOTO** command.

A **GOTO** command should not transfer program control to a comment line.

Do not use the **GOTO** command to transfer control to the top of a program. If the last program line is missed because of this, time-based commands (**LOOP**, **WAIT**, etc.) will not function properly.

See also **GOSUB**

HLIMIT

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **HLIMIT(***value***,***pt1***,...,***pt15***)**

value	New value to which the high limit is set. This value can be a decimal, a point name, or a local variable. Integers are not allowed.
pt1 through pt15	Logical names of analog points whose high limits are changed to the new value. These points must be in the same device as the command.

Use This command is used to set a new high alarm limit for alarmable analog points. Up to 15 points can be set to the same high limit with one **HLIMIT** command.

Example

```
100  IF (OATEMP.GT.70.0) THEN HLIMIT(84.0,
     ROOM16)ELSE HLIMIT(78.0,ROOM16)
```

Notes A point must be defined as alarmable if it is to be used in the **HLIMIT** command.

This command forces an upload of PPCL code to the mass storage device in all pre-Revision 11.1 logical firmware.

See also **ALARM, DISALM, ENALM, LLIMIT, NORMAL**

HOLIDA

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **HOLIDA(***month1***,***day1***,...,***month8***,***day8***)**

month1 through month8	Month of the designated holiday. Up to eight holidays can be specified with a single command (January = <1>, June = <6>, December = <12>, etc.).
day1 through day8	Day of the month for the designated holiday. Up to eight holidays can be specified with a single command. The first day of the month is entered as <1>.

Use This command is used to define the dates of holidays up to a year in advance. By entering the pairs of numbers for the dates (month, day) up to eight holidays can be defined by one **HOLIDA** command.

Example

```
630  HOLIDA(12,24,12,25,12,26,12,27)
```

Notes Multiple holiday commands can be used to define more than eight holidays.

The **HOLIDA** and **TODMOD** commands must precede any **TOD** or **TODSET** commands in order for the program to operate correctly.

A **HOLIDA** or **TODMOD** command in a device will only affect **TOD**, **TODSET** and **SSTO** commands in that device.

When a holiday date occurs in a **HOLIDA** command, the mode number for that day in the **TODMOD** command is set to 16.

If holidays are defined using the **HOLIDA** command, as well as the **TOD** calendar in the field panel firmware, make sure that the holidays are defined as the same day in both places. For example:

> If holidays are defined using the PPCL **HOLIDA** statement, and different holidays are defined in the **TOD** calendar, the equipment commanded by the PPCL **TOD** commands for mode 16 (holiday schedule) will execute not only on the holiday defined with the **HOLIDA** statement, but also on the holiday defined with the **TOD** calendar. In this case, the equipment would run for more days than necessary according to the holiday schedule.

See also **TOD**, **TODMOD**, **TODSET**

IF/THEN/ELSE

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **IF**(*exp*) **THEN** *x*

also

IF(*exp*) **THEN** *x* **ELSE** *y*

exp	One or more logical or relational tests that are the basis for the **THEN**/**ELSE** decision making. The logical test can compare variables, constants, status indicators, priority indicators, and/or numbers, state text, and system points. Several logical tests can be linked together with relational or logical operators. You can test a maximum of 13 operands. Operands can be point names, status indicators (i.e., **ON**, **ALARM**, etc.), @ priority indicators, and/or numbers.
x	Represents a condition, assignment, or course of action to take if the expression (*exp*) is true.
y	Represents a condition, assignment, or course of action to take if the expression (*exp*) is false.

Use This conditional logic command is used to provide customized decision logic. When the **IF** expression is true, the **THEN** command is executed. If the condition is false, the **ELSE** command (if defined) is executed. If the condition is false and no **ELSE** command is present, execution continues with the next line of code.

Example 1

```
110  IF (OATEMP.GT.70.0) THEN OADPR = 80.0
```

Example 2

```
310  IF (TIME.GT.8:00.AND.TIME.LT.16:00) THEN
     ON(@NONE,SFAN) ELSE ON(@OPER, SFAN)
```

Notes **GOSUB** commands should not be used for the *x* or *y* parameters.

Time-based commands such as **WAIT**, and **TODMOD** should not be directly used for the *x* or *y* parameters.

INITTO

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **INITTO(***value***,***pt1***,...,***pt15***)**

value	The new value which replaces the current totalized values for all specified points. A number entered for this parameter must be either a decimal, a point name, or a local variable. Integers are not allowed.
pt1 through pt15	Names of points that are defined for totalization. Points must reside in the same device as the control program.

Use This command is used to change the totalized value of a point to a new value (generally 0). A single command can define up to 15 point names that will receive the new value.

Example

```
10  IF (DAYOFM.EQ.1.0) THEN INITTO(0.0,
    PMP1,PMP2)
```

Notes The points defined in this command must reside in the same device as the command.

The **INITTO** command cannot reset the value of LPACI point types. Only points that are defined to be totalized can be initialized by this command.

When used in APOGEE field panels (which can totalize individual states of a point), **INITTO** will reset all totalized states of a digital point to zero.

LLIMIT

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **LLIMIT(***value***,***pt1***,**...**,***pt15***)**

value	New value to which the low limit is set. This value can be a decimal, a point name, or a local variable. Integers are not allowed.
pt1 through pt15	Logical names of analog points whose high limits are changed to the new value. These points must be in the same device as the command.

Use This command is used to set a new low alarm limit for alarmable analog points. A maximum of 15 points can be set to the same low limit with one **LLIMIT** command.

Example

```
100  IF (OATEMP.GT.68.0) THEN LLIMIT(76.0,
     ROOM16) ELSE LLIMIT(68.0,ROOM16)
```

Notes A point must be defined as alarmable if it will be used in the **LLIMIT** command.

This command forces an upload of PPCL code to the mass storage device in all pre-revision 11.1 logical firmware.

See also **ALARM**, **DISALM**, **ENALM**, **HLIMIT**, **NORMAL**

LOCAL

Physical	Logical	Unitary	CM	APOGEE
				⅄

Syntax **LOCAL (***pt1*... *pt16***)**

pt1 through pt16	Names of virtual points created for the program.

Use The program can reference these points as *$pt1* through *$pt16*. Other programs can reference these points by referencing the program name, followed by the system delimiter (:), followed by the local point name.

Example 1

The following is an example of commanding a local point contained within a program:

```
100  LOCAL(FANPT)
200  ON($FANPT)
```

Example 2

The following is an example of a program using the value of a local point in a different program (where *PROG1* is the program name):

```
300  IF(“PROG1:FANPT”.EQ.ON)THEN ...
```

LOOP

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Logical syntax **LOOP(***type***,***pv***,***cv***,***sp***,***pg***,***ig***,***dg***,***st***,***bias***,***lo***,***hi***,***0***)**

type	Designates the type of control action. Your choices are as follows: **0 =** direct acting. **128 =** reverse acting.
pv	Point name of the process variable which is being controlled or regulated. It is usually an LAI point, but can be an LAO point, which represents a temperature, flow rate, air velocity, etc.
cv	Point name of the loop output. It is usually an LAO point that represents an actuator signal such as pressure, current, or voltage. This value can be entered as a local variable.
sp	Set point for the loop against which the process variable is compared. This parameter can be entered as a point name or a decimal number. The value of the set point should represent the same engineering units as the process variable. This value can be entered as a local variable.
pg	Proportional gain determines the part of a PID control action that is directly proportional to the error between the set point (*sp*) and the process variable

(*pv*). This parameter can be entered as a point name, local variable, integer, or decimal. Proportional gain is calculated using the following formula:

$$pg = \frac{\text{Full range of controlled device}}{\text{Throttling range to change the output device from full open to full close.}} \times 1000$$

ig	Integral gain - The contribution of integral control action to the total control action of the loop. Integral action minimizes offset (the difference between the process variable and the set point).

When using integral gain, be sure to reduce the proportional gain so that the total gain of the loop is not high enough to cause instability and cycling. Begin with a low integral gain and increase it slowly if required. A recommended starting point is 2% of the proportional gain. Adding integral gain can increase the time required to tune the loop. This value can be entered as a point name, integer, decimal, or local variable. Integral gain should be calculated by using the following formulas.

If using an integral gain:

$ig = pg \times .02$

If not using an integral gain:

$ig = 0$

dg	Derivative gain - The time rate of response of the control system. Derivative gain is usually applied to fast responding systems. The derivative gain is entered as an integer, decimal, point name, or local variable. When derivative gain is not used, enter a zero. Adding derivative gain can increase the time required to tune the loop.
st	Sample time. This is how frequently the process variable (*pv*) is sampled in seconds. The minimum sampling time that can be specified is 1 second. The sample time is entered as an integer, decimal, point name, or local variable.
bias	Bias (*bias*) is the value (in engineering units) of the control output (*cv*), in a proportional only loop, when the measured variable (*pv*) equals the set point (*sp*). The bias is entered as a decimal number, integer, point name, or logical point. The bias value should always be between the high and low value. Bias is calculated by adding 50% of the output control span to the low limit. For example, you have a valve with a 3 to 8 psi spring range. To calculate the bias, find one-half of the spring range (5 psi / 2 = 2.5) and add that value to the low limit (3 psi). The bias for this device is 5.5 psi.
lo	Low limit of the loop output. This parameter should be set to match the low end range of the controlled device.

	The low limit is entered as a decimal, integer, point name, or local variable.
hi	High limit of the loop output. This parameter should be set to match the high end range of the controlled device. The high limit is entered as a decimal, integer, point name, or local variable.
0	Not used. Enter zero.

Example

```
2000  C  CONTROL LOOP STATISTICS (LOGICAL
2002  C  FIRMWARE)
2004  C  DIRECT CONTROL LOOP
2006  C  INPUT = RM100  OUTPUT = HVALVE
2008  C  SETPOINT = HSETPT
2010  C  PROPORTIONAL GAIN = PGAIN-NO I OR G
2012  C  GAINS
2014  C  SAMPLE TIME = 1 SECOND
2016  C  BIAS = 5.5
2018  C  LOW = 3.0
2020  C  HIGH = 9.0
2022  LOOP(0,RM100,HVALVE,HSETPT,PGAIN,0,0,1,
      5.5,3.0,9.0,0)
```

Physical syntax **LOOP(***type***,***pv***,***cv***,***sp***,***pg***,***ig***,***dg***,***st***,***bias***,***lo***,***hi***,***0***)**

type	Designates the type of control action. Your choices are as follows:
	0 = direct acting.
	128 = reverse acting.
pv	Point name of the process variable which is being controlled or regulated. It is usually an LAI point, but can be a LAO point, which represents a temperature, flow rate, air velocity, etc.
cv	Point name of the loop output. It is usually an LAO point that represents

an actuator signal such as pressure, current, or voltage. This value can be entered a local variable.

sp	Set point for the loop against which the process variable is compared. This parameter can be entered as a point name or a decimal number. The value of the set point should represent the same engineering units as the process variable. This value can be entered as a local variable.
pg	Proportional gain determines the part of a PID control action that is directly proportional to the error between the set point (*sp*) and the process variable (*pv*). This parameter must be entered as an integer. Proportional gain is calculated using the following formula:

$$pg = \frac{\text{Full range of controlled device}}{\text{Throttling range to change the output device from full open to full close.}} \times 1000$$

ig	Integral gain determines the contribution of integral control action to the total control action of the loop. Integral action minimizes offset (the difference between the process variable and the set point).

When using integral gain, be sure to reduce the proportional gain so that the total gain of the loop is not high enough to cause instability and cycling.

Begin with a low integral gain and increase it slowly if required. A

	recommended starting point is 2% of the proportional gain. Adding integral gain can increase the time required to tune the loop.
	This value must be entered as an integer. Integral gain is calculated by using the following formulas:
	ig = *pg* * .02
	If using an integral gain.
	ig = 0
	If not using an integral gain.
dg	Derivative gain defines the time rate of response of the control system. Derivative gain is usually applied to fast responding systems. The derivative gain is entered as an integer, point name, or local variable. When derivative gain is not used, enter a zero.
st	Sample time is how frequent the process variable (*pv*) is sampled in seconds. The minimum sampling time that can be specified is 1 second. The sample time is entered as an integer, point name, or local variable.

bias	This is the integer representation for the percent value of control output (*cv*), when the measured variable (*pv*) equals the set point (*sp*), and expressed in terms of low and high range limits. Bias is entered as a decimal number, point name, or logical point. To calculate the bias, use the following formula:

$$bias = \frac{\text{value desired} - lo}{hi - lo} * 10000$$

lo	Low limit of the loop output. This parameter should be set to match the low end range of the controlled device. The low limit is entered as an decimal number, point name, or local variable. Use the formulas and the values shown in Table 4-2 and Table 4-3 to calculate the low limit.
hi	High limit of the loop output. This parameter should be set to match the high end of the range of the controlled device. The high limit is entered as a decimal number, point name, or local variable. Use the formulas and values shown in Table 4-2 and Table 4-3 to calculate the low limit.
0	Not used. Enter zero.

Table 4-2. Loop Limit Calculations.

Output	Range of Values	Calculation
3-15 psi PO (21-103 kPa)	8 - 248	$\frac{\text{Desired PSI - 2.6}}{0.05}$ $\frac{\text{Desired kPa - 17.9}}{0.3445}$
1-15 psi AO-P (7-103 kPa)	14 -211	$\frac{\text{Desired psi}}{0.071}$ $\frac{\text{Desired kPa}}{0.4896}$
4-20 mA AO-E 0-10 Vdc	0 - 1023	$\frac{\text{Desired mA - 4}}{0.015625}$ $\frac{\text{Desired Vdc}}{0.09766}$
Input to another loop (Used for set point calculation)	4,100 - 20,900	(Desired % Output * 16,800) + 4,100

Table 4-3. Loop Limit Values.

PO 3-15 PSI (21-103 kPa)		AO-P 1-15 PSI (7-103 kPa)		AO-E 4-20 mA		AO-E 0-10 Vdc		Input to another LOOP	
PSI (kPa)	**Value**	**PSI (kPa)**	**Value**	**ma**	**Value**	**Vdc**	**Value**	**% Output**	**Value**
-	-	1 (7)	14	4	0	0	0	0	0
-	-	2 (14)	28	5	63	1	101	10	5780
3 (21)	8	3 (21)	42	6	127	2	204	20	7460
4 (28)	28	4 (28)	56	7	191	3	306	30	9140
5 (34)	48	5 (34)	70	8	255	4	409	40	10820
6 (41)	68	6 (41)	85	9	319	5	511	50	12500
7 (48)	88	7 (48)	99	10	383	6	613	60	14180
8 (55)	108	8(55)	113	11	447	7	716	70	15860
9 (62)	128	9 (62)	127	12	511	8	818	80	17540
10 (70)	148	10 (70)	141	13	575	9	921	90	19220
11 (76)	168	11 (76)	155	14	639	10	1023	100	20900
12 (83)	188	12 (83)	169	15	703	-	-	-	-
13 (90)	208	13 (90)	183	16	767	-	-	-	-
14 (96)	228	14 (96)	197	17	831	-	-	-	-
15 (103)	248	15 (103)	211	18	895	-	-	-	-
-	-	-	-	19	959	-	-	-	-
-	-	-	-	20	1023	-	-	-	-

Use

This command performs closed loop control by using any combination of proportional, integral, and derivative control actions in either direct or reverse acting modes. It monitors an input point (process variable), compares it with a desired value (the set point), and adjusts an output (control variable) to bring the input closer to the set point. The **LOOP** command is the software counterpart to a pneumatic receiver controller. Anti-windup is automatically prevented for the integral action once the hi or low limit is reached.

The following example illustrates this concept:

Example

```
2000  C CONTROL LOOP STATISTICS (PHYSICAL
2002  C FIRMWARE)
2004  C DIRECT CONTROL LOOP
2006  C INPUT = RM100  OUTPUT = HVALVE
2008  C SETPOINT = HSETPT
2010  C PROPORTIONAL GAIN = 4000 I GAIN = 10
2012  C D GAIN = 0
2014  C SAMPLE TIME = 1 SECOND
2016  C BIAS = 42
2018  C LOW = 20
2020  C HIGH = 120
2022  LOOP(0,RM100,HVALVE,HSETPT,4000,
      10,0,1,42.0,20.0,120.0,0)
```

MAX

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **MAX(***result***,***pt1***,...,***pt15***)**

result	Point name where the largest value is stored. This can be a virtual point name or a local variable.
pt1 through pt15	Values which are compared. The values can be any combination of point names, decimals, integers, and local variables.

Use This command selects the largest value from 2 to 15 point names or numbers, (numbers must be in decimal format), and stores that value in the result point.

Example

```
10  MAX(HOTZON,ZONE1,ZONE2,ZONE3, 9.0)
```

See also **MIN**

MIN

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **MIN(***result***,***pt1***,...,***pt15***)**

result	Point name where the lowest value is stored. This can be a virtual point name or local variable.
pt1 through pt15	Values which are compared. The values can be any combination of point names, decimals, integers, and local variables.

Use This command selects the lowest value from 2 to 15 point names or numbers (numbers must be in decimal format), and stores that value in the *result* point.

Example

```
10  MIN(COOLZN,ZONE1,ZONE2,ZONE3,9.0)
```

See also **MAX**

NIGHT

Physical	Logical	Unitary	CM	APOGEE
	9.1		▲	▲

Syntax **NIGHT(***pt1***,...,***pt16***)**

pt1 through pt16	Names of logical controller points.

Use This command changes a logical controller point to NIGHT mode. Logical controller point types can be commanded into DAY or NIGHT mode. Refer to the following example:

Example

```
100  IF (TIME.LT.7:00.OR.TIME.GT.18:00) THEN
     NIGHT(LCTLR2) ELSE DAY(LCTLR2)
```

For some equipment controllers, NIGHT mode is also referred to as UNOCC (unoccupied) mode. If an equipment controller is in the unoccupied mode, PPCL recognizes this status as NIGHT.

See also **DAY**

NORMAL

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **NORMAL(***pt1***,...,***pt16***)**

pt1 through pt16	Names of the points to be taken out of the alarm-by-command state.

Use This command removes a maximum of 16 points out of the alarm-by-command condition and returns them to their normal operating mode.

Example

```
100  IF (RM90T.GT.80.0) THEN ALARM(ROOM90)
     ELSE NORMAL(ROOM90)
```

See also **ALARM**, **DISALM**, **ENALM**, **HLIMIT**, **LLIMIT**

OFF

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **OFF(***pt1***,...,***pt16***)**

pt1 through pt16	Point names that will have their priority set to NONE.

Featured added to revision 9.2 logical firmware, CM and APOGEE firmware:

OFF(*@prior***,***pt1***,...,***pt15***)**

@prior	Defines a specific point priority.
pt1 through pt15	Point names of the points to be commanded. Acceptable point types are LDI, LDO, L2SL, L2SP, LOOAL, LOOAP, LFSSL, and LFSSP.

Use This command can be used to switch a point to the OFF state. You can change the operational status for a maximum of 16 points using a single **OFF** command. If you change the priority of the points, you can only specify a maximum of 15 points.

Example

```
20  IF (OATEMP.GE.63.0) THEN OFF(@NONE,
    PUMP1,PUMP2,PUMP3)
```

Notes FAST/SLOW/STOP points (LFSSL, LFSSP) use the **OFF** command for STOP.

See also **AUTO, FAST, ON, SLOW**

OIP

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **OIP(***trigger***,** *"seq"***)**

trigger	Point name of an LDO, LDI point, or local variable used to trigger the operator sequence.
seq	This variable represents the sequence of keystrokes you would enter if you were using a terminal. The sequence must be enclosed in double quotes **" "**. For each level in the sequence that you advance, you must enter a slash ***/***. The sequence must not exceed 60 characters (including slashes) in length.

Use This command is used to mimic operator sequences that are normally entered from a terminal. This command allows most operator functions to be executed from within a PPCL program.

The **OIP** command uses a *trigger* point to determine the conditions for execution. When the trigger point is turned on, then the operator sequence is executed once. To execute the operator sequence again, the trigger point must be turned OFF, and then turned ON again.

Example

```
100  C
102  C  OIP COMMAND INFORMATION:
104  C  OPERATOR SEQUENCE
106  C
108  C  - POINT
110  C  - DISPLAY
112  C  - PRINTER
114  C  - YES
116  C  - VALUE
118  C  - ANY
120  C  - NAME
122  C  - ALL NAMES
124  C
130  C  TRIGGER POINT - RPT7AM
150  OIP(RPT7AM,"P/D/P/Y/V/A/N/*")
```

Notes **OIP** commands used to generate messages, displays, and reports should be staggered in time (that is, do not use the same trigger point for all **OIP** commands). This allows one command to complete before another begins.

The **OIP** command will appear as FAILED if the operator sequence was entered incorrectly and the control program attempts to execute it.

On return from power failure, after an enable command has been executed, or during the first execution of the control program after a database load, the **OIP** command will not execute until the trigger point toggles.

After the **OIP** command has been executed, the trigger point must be reset (commanded back to its original state) before the **OIP** command can be executed again.

When using an **OIP** command with an LDO type subpoint, you must command the point ON/OFF with the number 1 or 0. Using any text other than 1 or 0 will cause the statement to fail.

If the trigger point name begins with a number, then the point name should be preceded by an @ sign. If a variable in the keystroke sequence begins with a number, then the point name should not be preceded by an @ sign. For example:

The following statement is incorrect:

```
100  OIP(TRIG,"P/T/D/H///FAN/@1FAN//60/")
```

The following statement is correct:

```
100  OIP(TRIG,"P/T/D/H///FAN/1FAN//60/")
```

The **OIP** command must be executed with every pass of the program in order to see the trigger point change value.

The **OIP** command cannot be used to perform loop tuning.

In an **OIP** statement, a slash **/** can be used to represent a carriage return. An example of a multi-point trend display would look like:

```
OIP (TRIG, "P/T/D/P///
Name1/Name2/Name3//10/")
```

Changes were made in revision 12.4/12.4.1 logical firmware to allow you to enter which trend instance you want to display. Use the following syntax for the Multi-Point Trend Report:

```
OIP (TRIG, "P/T/D/P///
Name1//Name2//Name3//10/")
```

ON

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **ON(***pt1***,...,***pt16***)**

pt1 through pt16	Point names that will have their priority set to NONE.

Featured added to revision 9.2 logical firmware, CM and APOGEE firmware:

ON(*@prior***,***pt1***,...,***pt15***)**

@prior	Defines a specific point priority.
pt1 through pt15	Point names of the points to be commanded. Acceptable point types are LDO, L2SL, L2SP, LOOAL, and LOOAP.

Use This command can be used to switch a point to the ON state. You can change the operational status for a maximum of 16 points using a single **ON** command. If you change the priority of the points, you can only specify a maximum of 15 points.

Example

```
100  IF (OATEMP.LT.60.0)THEN ON(@NONE,
     PUMP1,PUMP2)
```

See also **AUTO, FAST, OFF, SLOW**

ONPWRT

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Syntax **ONPWRT(***line#***)**

line#	Line number at which execution begins in the control program after returning from power failure. This number must be an integer in the range of 1 to 32,767. If the line number is invalid, this command is ignored.

Use This command is similar to a **GOTO** command and allows you to select the first program line that is executed when power has returned. The **ONPWRT** command is only executed once and is then ignored as long as power stays ON.

Example

```
10  ONPWRT(1800)
```

Notes The **ONPWRT** command should be the first command in a PPCL program since program execution returns to the first line of a PPCL program after a power failure.

If the database for a field panel is lost due to a power failure, the **ONPWRT** command is not executed when power is restored.

PARAMETER

Physical	Logical	Unitary	CM	APOGEE
⅄				

Syntax **PARAMETER** *A* **=** *B*

A	A valid label name. This name cannot be defined in any database and cannot start with a number.
B	The value that is assigned to the label name.

Use The **PARAMETER** command defines a label that is used as a storage location for a value. When the PPCL program compiles, the computer assigns every occurrence of the label (*A*) with the value (*B*). The compiled version of the program then contains the actual value represented by the label.

Example

```
     PARAMETER DELAY = 15
100  WAIT(DELAY,FAN1,FAN2)
```

Notes **PARAMETER** commands are usually placed at the top of the program so they are easy to find.

This command does not require a line number.

PDL

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Physical syntax **PDL(***totkw***,***target***,***g1s***,***g1e***,...,***g4s***,***g4e***)**

Logical syntax **PDL(***area***,***totkw***,***target***,***g1s***,***g1e***,***sh1***,...,***g4s***,***g4e***,***sh4***)**

area	Meter area number. Enter this number as an integer.
totkw	The amount of power that is consumed by the loads which are under control of the **PDL** command. A power consuming load is considered to be under control of a **PDL** command if it is defined in the **PDLDAT** command that is associated with the **PDL** command and the following criteria are met: • The load is currently in NONE or PDL priority. • The PDL command is traced and is enabled. The *totkw* parameter should be the same virtual LAO point that is specified as the *kwtot* parameter in the owning **PDLDPG** command. The value of this parameter is calculated by the **PDL** command.

target	The power consumption limit which is to be maintained by the **PDL** command. This parameter must be the same virtual LAO point that was specified as the *target* parameter for the corresponding **PDLDPG** command. The value of this parameter is calculated by the **PDLDPG** command.
g1s	Start line for priority group 1 (first group to shed) load definition in **PDLDAT** commands.
g1e	End line for group 1 load definition.
sh1	**0** - fixed shedding in group 1, or **1** - round robin shedding in group 1.
g4s	Start line for priority group 4 (last group to shed) load definition in **PDLDAT** command.
g4e	End line for group 4 load definition.
sh4	**0** - fixed shedding in group 4 or **1** - round robin shedding in group 4.

Use A single **PDL** command is used in conjunction with a group of **PDLDAT** commands in a *load handling* field panel to define load parameters and maintain a defined kilowatt level (or target) for that group of loads.

The **PDL** command is responsible for the direct control of a group of **PDLDAT** commands. This control includes:

- Maintaining a target kilowatt (kW) value by shedding and restoring loads. This value is passed to the **PDL** command by the value of an LAO point.

- Monitoring the total available kilowatts under its control. The **PDL** command determines which of the loads defined in the **PDLDAT** command associated with it are actually available for peak demand limiting control. The total kilowatts available to the **PDL** command are placed in an LAO point.
- Maintaining the timing as defined in the PDLDAT statement for *minon*, *minoff*, and *maxoff*.
- Defining how each load under **PDL** control fits into one of four priority groups. This is accomplished by checking the line numbers of **PDLDAT** commands.
- Defining and controlling the sequence of shedding and restoring for each priority group. (The two types of shedding and restoring are fixed and round robin). In logical firmware revision 11.2 and higher, shedding begins at 90% of the set point, and not 100% as in earlier revisions.

Example

```
100  PDL(1,TOTKW1,TGT1,100,199,0,200,299,1,
     300,399,0,400,499,0)
```

Notes With logical and CM firmware, you can only have one meter defined in a field panel. APOGEE firmware can have one meter defined in each program.

When points are controlled by the Peak Demand Limiting function, they are placed into PDL priority. Group 1 is shed first, while Group 4 is shed last.

Even if all groups are not used, the remaining definitions should contain the integer 0.

See also **PDLDAT**, **PDLDPG**, **PDLMTR**, **PDLSET**

PDLDAT

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄		⅄	⅄

Syntax **PDLDAT(***ptname***,***minon***,***minoff***,***maxoff***,***kwval***)**

ptname	Point name of the load. In a network system, a load defined by a **PDLDAT** statement must reside in the same field panel as the **PDL** statement.
minon	Minimum time (in minutes) that PDL must keep the load on after restoring it. This parameter must be less than 546 and can be defined as an integer, decimal, point name, or a local variable.
minoff	Minimum time (in minutes) that the load must remain OFF before PDL can restore it. This parameter must be less than 546 and can be defined as an integer, decimal, point name, or a local variable.
maxoff	Maximum time (in minutes) that PDL can keep the load off after shedding it. This parameter can be defined as an integer, decimal, point name or a local variable. The maximum allowable value of *maxoff* is (*minoff* + 546).
kwval	Kilowatt rating for the load. This parameter can be a decimal, integer, a point name, or a local variable.

Use A group of **PDLDAT** commands are used in conjunction with a single **PDL** command in a *load handling* field panel to define load parameters and maintain a certain kilowatt level (or target) for that group of loads. This command defines the minimum on-time, minimum off-time, maximum off-time, and kilowatt value for a specific load.

Example

```
100  C
102  C  PDLDAT COMMAND INFORMATION:
104  C
106  C - CONTROLLED POINT  - FAN17
108  C - MINIMUM ON TIME   - 10 MINUTES
110  C - MINIMUM OFF TIME  - 5 MINUTES
112  C - MAXIMUM OFF TIME  - 180 MINUTES
114  C - KILOWATT RATING   - 10KW
116  C
118  PDLDAT(FAN17,10,5,180,10)
```

Notes A **PDLDAT** should only be reference by a single **PDL** statement. Multiple references will produce unpredictable results.

See also **PDL, PDLDPG, PDLMTR, PDLSET**

PDLDPG

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **PDLDPG(***area***,***kwtot1***,***target1***,...,***kwtot7***,***target7***)**

area	The meter area designation. This parameter must have the same numeric value as the area parameter that is specified for the corresponding **PDLMTR** statement.
kwtot1 *through* *kwtot7*	The amount of power that is consumed by the loads which are under control of the **PDL** command. A power consuming load is considered to be under control of a **PDL** command if it is defined in the **PDLDAT** command associated with the **PDL** command and the following criteria are met:

- The load is currently in the ON state and was placed in this state at a priority level that is below or equal to PDL priority.
- The load is currently in the OFF state and was placed in this state at PDL priority (that is, if it was shed by the PDL algorithm).
- The **PDLDAT** command is being traced and is enabled.

The *totkw* parameter must be the same virtual LAO point that is specified as the *kwtot* parameter in the owning **PDLDPG** command. The value of this

	parameter is calculated by the **PDL** command.
target1 through target7	The power consumption target value. One *target* parameter must be specified for each **PDL** command in this digital point group. One *kwtot* parameter must be specified for each **PDL** statement that is a number of this meter area. Each of these *kwtot* parameters must be the same virtual LAO point that is specified as the *kwtot* parameter for the corresponding **PDL** command. The value of the parameter is assigned by the **PDL** statement in which it is defined. The sum of the value of all *kwtot* parameters is used by the **PDLDPG** command to calculate the power consumption target value for each **PDL** command. Note that for each of these virtual LAO points, the slope should be defined as one, the intercept should be defined as zero, and the COV limit should be defined as one.

Use Associates a power consumption target parameter with each **PDL** statement. This command also calculates the amount of power that is used by all the loads that are under the control of each **PDL** command in its meter area.

Example

```
100  PDLDPG(1,TOTKW1,TGT1,TOTKW2,TGT2)
```

See also **PDL, PDLDAT, PDLMTR, PDLSET**

PDLMTR

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **PDLMTR(***area***,***hist***,***calc***,***window***,***plot***,***warning***,***mt1***,** *def1***,...,***mt5***,***def5***)**

area	Meter area, entered as an integer (1 to 32,767). This value is usually entered as an integer, but can be a point name or a local variable.
hist	Historical forecast weighting factor as a percent. This parameter should be less than 50% to anticipate demand. The recommended value is 30% and can be a decimal or integer value.
calc	Calculation interval in minutes (one minute is the minimum). This value is usually entered as an integer or decimal, but can be a point name or a local variable.
window	Sliding window interval in minutes (up to 30 samples can be stored per window). This value is usually entered as an integer or decimal, but can be a point name or a local variable.
plot	Maximum value for Time versus Demand plot section of PDL Activity Report. This value should be greater than the highest set point used in PDLSET. This value can be entered as a decimal, integer, point name, or a local variable.

warning	Warning messages enabled = 1. Warning messages disabled = 0.
mt1 *through* *mt5*	Logical point names for demand/*through* consumption meters (can be LPACI or analog points).
def1 *through* *def5*	For historical (latest, best) reading enter -1 (also default value) or enter a constant or a fixed default value. Default values are expressed in kilowatts (demand) for both analog and pulse inputs. The default value is used to replace the meter values (*mt1* through *mt5*) when communication is lost. Acceptable values for this parameter include integers, decimals, point names, or local variables.

Use

The primary role of this command is monitoring meters, making predictions on demand, and keeping report data up-to-date. Only one **PDLMTR** command can be defined per meter area. For APOGEE field panels, one **PDLMTR** command can be defined per program.

PDLMTR commands are used for defining:

- Meters by point name
- Calculation intervals (how often are demand predictions made)
- Predictions on the width of the sliding window
- Historical weighting factors for the sliding window predictions
- Default values to be used if meters cannot be read

- Whether or not warning messages are issued as predicted demand nears or exceeds the set point
- Full-scale demand for Time versus Demand plot section of reports

The **PDLMTR** command is responsible for the following actions:

- Making sliding window demand predictions
- Updating report information pertaining to demand and consumption
- Deciding when warning messages should be issued
- Initializing a meter area

Example

```
200  C
202  C PDLMTR COMMAND INFORMATION:
204  C
206  C  - HISTORICAL WEIGHTING FACTOR - 30%
208  C  - CALCULATION INTERVAL - 1 MINUTE
210  C  - PREDICTION WINDOW - 15 MINUTES
212  C  - THE TIME VERSUS DEMAND SECTION OF
214  C    THE ACTIVITY REPORT PLOT HAS A
216  C    FULL SCALE OF 500 KILOWATTS.
218  C  - WARNING MESSAGES ARE ENABLED TO
220  C    BE SENT TO ALARM DEVICES.
222  C  - THE DEFAULT VALUES OF THE TWO
224  C    METER POINTS (METER1 AND
226  C    METER2) ARE 100 KILOWATTS AND
228  C    50 KILOWATTS RESPECTIVELY.
230  C
250  PDLMTR(1,30,1,15,500,1,METER1,100,
     METER2,50)
```

Notes

With logical firmware, you can have one meter area defined in a field panel.

If a LPACI point is used in a **PDLMTR** statement, then PDL should be restarted for that meter area after the LPACI is reset. If the LPACI is being used with the Minicomputer's Peak Demand Limiting (PDL), then PDL must be stopped, the LPACI reset, and then PDL can be restarted.

When analog points are used for *mt1* through *mt5*, the engineering units are KW. When LPACI points are used for *mt1* through *mt5*, the engineering units are KWH.

See also **PDL, PDLDAT, PDLDPG, PDLSET**

PDLSET

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **PDLSET(***area***,***exceed***,***set1***,***time1***,...,***set7***,***time7***)**

area	Meter area. This is entered as an integer.
exceed	DO point which is commanded OFF at the end of each set point interval. It is then turned ON if the actual floating window peak exceeded the set point at any time during the preceding interval.
set1 through set7	Demand set point (in kilowatts) which is not to be exceeded. This parameter can be a decimal, integer, a point name or a local variable.
time1 through time7	The time at which the corresponding set point ends. This parameter is usually entered as a clock time in military format (24-hour clock). It can also be entered as an integer, a decimal, or a local variable. Times must be in ascending order.

Use The **PDLSET** command is responsible for designating the peak demand limiting levels and ensuring that these levels are not exceeded for a meter area during the time of day during which those set points are in effect. The demand predictions made by the **PDLMTR** command are compared with the appropriate set points defined by this command which determine the number of kilowatts (if any) which must be shed or restored for the meter area.

Example

```
100  C
101  C  PDLSET COMMAND INFORMATION:
102  C
103  C  - THIS COMMAND CONTROLS METER AREA 1.
104  C  - THE VIRTUAL LDO POINT WHICH WILL
105  C    TOGGLE ON AND OFF AT THE END OF
106  C    THE SET POINT INTERVALS WHEN THE
107  C    SET POINT WAS EXCEEDED IS
108  C    CALLED PEAKEX.
109  C  - PDLST1 SET POINT ENDS AT 11:00 A.M.
110  C  - PDLST2 SET POINT ENDS AT 4:30 P.M.
111  C
120  PDLSET(1,PEAKEX,PDLST1,11:00,PDLST2,16:30)
```

Notes The **PDLSET** command requires at least two set point/time definitions for each day. Reports will not generate unless there are at least two set points

See also **PDL**, **PDLDAT**, **PDLDPG**, **PDLMTR**

RELEAS

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **RELEAS(***pt1***,...,***pt16***)**

pt1 through pt16	Point names that will have their priority set to NONE.

Change added to revision 9.2 logical firmware, CM and APOGEE firmware:

RELEAS(*@prior***,** *pt1***,**.....*pt15***)**

@prior	Defines a specific point priority that the point is being released from.
pt1 through pt15	Point names of the points that are to be released.

Use This command changes the priority of up to 16 points to a NONE priority. The feature added to logical firmware allows you to release up to 15 points to NONE priority.

A LAO or LDO point that is used in a **RELEAS** statement can generate multiple entries in the Trend Data Report if a command statement such as **SET**, **=**, **ON**, or **OFF** is used on the same point at the same time.

Notes Always use a priority at least as high as the one that the point will be in to ensure proper release. If the point has been commanded from the keyboard, it will require an @OPER entry to release the priority of the point to NONE.

RETURN

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Syntax **RETURN**

Use This command marks the end of a subroutine. It must be the last command of any **GOSUB** subroutine. When a **RETURN** command is encountered, the program will return to the line following the **GOSUB** command

Example

```
100 GOSUB 310
110 ...
120 ...
130 ...
290 ...GO TO 350

300 C   THIS SUBROUTINE PERFORMS...
310 ...
320 ...
330 ...
340 RETURN
350 CONTINUE WITH REST OF PROGRAM
```

See also **GOSUB**

SAMPLE

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **SAMPLE(***sec***)** *line*

sec	Interval in seconds (1 to 32,767) between evaluations of state. This value is entered as an integer.
line	Any PPCL statement that does not include its own timing function (**WAIT**, **PDL**, **TOD**, **TIMAVG**, **LOOP**, **SSTO,** or another **SAMPLE** command).

Use This command allows you to define how often a command is evaluated. It can be helpful in situations such as preventing short cycling in ON/OFF decisions, and reducing COVs from noisy flow transmitters. Valid commands can include assignment, calculations, and program control (**GOTO**, **GOSUB**).

Example

```
200  SAMPLE(600) ON(HALFAN)
```

Notes The **SAMPLE** command executes immediately on a return from power failure, after an **ENABLE** command, or during the first execution of PPCL following a database load.

SET

Physical	Logical	Unitary	CM	APOGEE
	⅄	⅄	⅄	⅄

Syntax **SET(***value***,***pt1***,...,***pt15***)**

value	Value to which points should be commanded. This number can be a decimal, logical point, or local variable. Integers are not allowed.
pt1 through pt15	Point names that will have their priority set to NONE.

Feature added to revision 9.2 logical firmware, CM and APOGEE firmware:

SET(*@prior***,** *value***,** *pt1***,...,***pt14***)**

@prior	Defines a specific point priority.
value	Value to which points should be commanded. This number can be a decimal, logical point, or local variable. Integers are allowed in APOGEE firmware.
pt1 through pt14	Point names of LAO or LDO, LFSSL, LFSSP, LOOAL, LOOAP, L2SL, L2SP, and LPACI point types.

Use This command is used to change output points to a new value. Up to 15 points can be assigned a new value (14 points when including a priority) with a single set command.

Example 1

```
450  SET(75.0,RMSET1,RMSET2,RMSET3)
```

Example 2

```
550  SET(@EMER,75.0,RMSET1,RMSET2,RMSET3)
```

SLOW

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **SLOW(***pt1***,...,***pt16***)**

pt1 through pt16	Point names of LFSSL or LFSSP type are commanded to slow.

Feature added to revision 9.2 logical firmware, CM and APOGEE firmware:

SLOW(*@prior***,***pt1***,...,***pt15***)**

@prior	Defines a specific point priority.
pt1 through pt15	Point names of LFSSL or LFSSP type points.

Use This command is used to change FAST/SLOW/STOP points to the low speed (SLOW). A single **SLOW** command can be used to change the operational status of up to 16 LFSSL or LFSSP points. If you change the priority of the points, you can only define a total of 15 points.

Example

```
20  IF (RMTEMP.LT.75.0)THEN SLOW(@NONE,
    FAN1,FAN2)
```

See also **AUTO, FAST, OFF, ON**

SSTO

Physical	Logical	Unitary	CM	APOGEE
	▲		▲	▲

Syntax **SSTO(***zone***,***mode***,***cst***,***csp***,***est***,***lst***,***ost***,***esp***,***lsp***,** *osp***,***ast***,***asp***)**

zone	**SSTO** zone number (valid values are 1 to 5).
mode	Mode number. Valid entries are any combination of 1, 2, 4, 8, or 16 as defined in the following table. Mode numbers can be added together to create customized schedules for the TOD program. Refer to the TODMOD command for more information.

MODE	**Schedule**
1	Normal Schedule
2	Extended Schedule
4	Shortened Schedule
8	Weekend Schedule
16	Holiday Schedule

cst	Point name of the virtual LAO point which stores the calculated start time. If **SSTO** is disabled (*season* = 0 in the **SSTOCO** command), then *cst* is assigned to the latest start time.
csp	Point name of the virtual LAO that stores the calculated stop time. If **SSTO** is disabled (*season* = 0 in the **SSTOCO** command), then *csp* is assigned the latest stop time.

est	Earliest start time. This can be entered in military time, decimal time, or as a logical point name or local variable.
lst	Latest start time. This can be entered in military time, decimal time, or as a logical point name or local variable.
ost	Occupancy start time. This can be entered in military time, decimal time, or as a logical point name or local variable.
esp	Earliest stop time. This can be entered in military time, decimal time, or as a logical point name or local variable.
lsp	Latest stop time. This can be entered in military time, decimal time, or as a logical point name or local variable.
osp	Occupancy stop time. This can be entered in military time, decimal time, or as a logical point name or local variable.
ast	Total adjustment to the calculated start time. This value is changed from day to day. A decimal value or virtual LAO point type can be defined to store the adjustment value.
asp	Total adjustment to the calculated stop time. This value is changed from day to day. A decimal value or virtual LAO point type can be defined to store the adjustment value.

Use The **SSTO** command calculates the optimal start and stop times for each zone based on information derived from the **SSTOCO** command (outside air temperature, zone temperature, desired zone temperature, etc.) as well as parameters for earliest, latest, and occupancy start times provided in the **SSTO** command.

This command has the ability to *tune* itself by incrementing or decrementing the start time after calculating an adjust time (*ast*) when errors between actual temperatures and desired temperatures occur.

Example

```
60  SSTO(1,1,ONTIM,OFTIM,6:30,7:45,8:00,15:30,
    16:45,17:00,0.0,0.0)
```

Notes The **SSTO** command only calculates the optimal start and stop times. **TOD** and **TODSET** commands are needed to command the point.

When *ast* or *asp* are entered as zero, the current adjustment value is displayed each time the command is displayed. If a virtual LAO point name is entered, the operator can specify (command) an initial value for *ast* or *asp*.

See also **SSTOCO**, **TOD**, **TODSET**

SSTOCO

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **SSTOCO(***zone***,***season***,***intemp***,***outemp***,***ctemp***,** *ccoef1***,***ccoef2***,***ccoef3***,***ccoef4***,***htemp***,***hcoef1***,** *hcoef2***,***hcoef3***,***hcoef4***)**

For variables used in this command, you can use numbers, point names, or local variables. The values for the coefficients must be entered as follows:

- in minutes for revisions 5.0 logical firmware only.
- in fractions of an hour for all other revisions of firmware

zone	**SSTO** zone number. Valid values are 1 to 5.
season	Current season. Point name whose value represents the coefficients to be used (heat = 2, cool = 1, and disable **SSTO** = 0).
intemp	Point name or average of indoor zone temperature sensor.
outemp	Point name or average of outdoor air temperature sensor.
ctemp	Desired zone temperature for cooling season. This can be entered as a floating point (decimal) number, a integer, a point name, or a local variable.

ccoef1	Cooling coefficient - The time, in hours, required to lower the zone temperature one degree while ignoring external load factors.
ccoef2	Cooling retention coefficient - The time, in hours, required to raise the zone temperature one degree with the cooling equipment off, outside air dampers open, and the outside temperature 10 degrees higher than the desired zone temperature for the cooling season.
ccoef3	Cooling transfer coefficient - The time, in hours, required to lower the zone temperature 1 degree with outside air dampers closed and the outside temperature 10 degrees higher than the desired zone temperature for the cooling season.
ccoef4	Cooling auto-tune coefficient - The time, in hours, added to or subtracted from the adjust time in the **SSTO** command as part of the self-tuning feature.
htemp	Desired zone temperature for heating season. This can be entered as a floating point (decimal) number, an integer, a point name, or a local variable.
hcoef1	Heating coefficient - The time, in hours, that is required to raise the zone temperature one degree while ignoring external load factors.

hcoef2	Heating retention coefficient - The time, in hours, that is required to lower the zone temperature 1 degree with heating equipment off, outside air dampers open, and the outside temperature 25 degrees lower than the desired zone temperature for the heating season.
hcoef3	Heating transfer coefficient - The time, in hours, that is required to raise the zone temperature 1 degree with outside air dampers closed and the outside temperature 25 degrees lower than the desired zone temperature for the heating season.
hcoef4	Heating auto-tune coefficient - The time, in hours, that is added to or subtracted from the adjust time in the **SSTO** command as part of the self-tuning feature.

Use This command defines the thermal characteristics of a zone based on the season, indoor temperature, outdoor temperature, and a variety of heating and/or cooling coefficients that are necessary to calculate optimal start and stop times.

Example

```
100  SSTOCO(1,1,ROOM10,OATEMP,75.0, 0.01,0.3,
     0.05,0.083,72.0,0.1,0.1,0.2,0.083)
```

See also **SSTO, TOD, TODMOD**

STATE

Physical	Logical	Unitary	CM	APOGEE
				⅄

Syntax **STATE(***@pri*,*statetext*,*pt1*...*pt14***)**

@pri	Specifies an optional parameter that designates a specific point priority.
statetext	State text that can be found in the associated state text table
pt1 *through* *pt14*	Points that can be commanded to a particular state. If using the *@pri* parameter, only 14 points can be defined.

Use This statement commands points using a state text value.

Example

```
500  STATE(NIGHT,TEC1,TEC2,TEC3,TEC4)
```

TABLE

Physical	Logical	Unitary	CM	APOGEE
	✓	✓	✓	✓

Syntax **TABLE(***input***,***output***,***x1***,***y1***,...,***x7***,***y7***)**

input	Point name of the input (*x*) variable. A local variable or virtual LAI point type can be used.
output	Point name of the output (*y*) variable. A local variable or virtual LAO point type can be used.
x1,y1 *through* *x7,y7*	Pairs of coordinates that define the *x-y* relationship. *y1* is the value of the output when the input equals *x1*; *y2* is the value of the output when the input is *x2*, etc. The coordinates can be entered as integers or decimals.

Use This command allows you to define a general function of two variables by specifying pairs of coordinates (*x*,*y*).

The **TABLE** command makes a straight line interpolation for the *output* (*y*) when the *input* is between a pair of *x* values. The *x* points must be entered in ascending order (that is *x3* must be larger than *x2*, etc.). For inputs lower than *x1*, the *output* will always equal *y1*. For inputs larger than the last *x* used, the *output* will equal the last *y* value entered.

Example

```
500  C
502  C  TABLE COMMAND INFORMATION:
504  C
506  C  OATEMP (X)       HWSP (Y)
508  C  ----------      ----------
510  C     0 DEG F       180 DEG F
512  C   60 DEG F       100 DEG F
514  C
520  TABLE(OATEMP,HWSP,0,180,60,100)
```

The hot water set point will remain at 180°F when the outside temperature is below 0°F. The hot water set point will remain at 100°F when the outside temperature is above 60°F.

Notes Table statements can be cascaded by overlapping *x-y* pairs using virtual points.

TIMAVG

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **TIMAVG(***result***,***st***,***samples***,***input***)**

result	A point name used to store the result of the average value. Local variables can also be used.
st	Number of seconds between each sample. This can be entered as an integer, a point name, or a local variable.
samples	Number of samples to be used to calculate the average. This must be an integer between 1 and 10.
input	Point name of the LAI or LAO point whose average value is to be calculated. A local variable can also be used.

Use This command is used to find an average value over time. You decide how often the values should be taken (sample time) and the number of values to be taken. The average is always over the most recent sample count and all previous values are discarded.

Example

```
10  C   SIX SAMPLES WILL BE TAKEN TO
20  C   CALCULATE THE RMAVG.
30  C   THE INTERVAL BETWEEN EACH SAMPLE
40  C   WILL BE 10 MINUTES. RMTEMP WILL
50  C   BE AVERAGED EVERY 60 MINUTES.
60  TIMAVG(RMAVG,600,6,RMTEMP)
```

The average value will be updated every sample time.

Example

```
100  TIMAVG (RMAVG, 600, 6, RMTEMP),
```

In the following statement, RMAVG would change every 600 seconds, not every 100 seconds, (assuming RMTEMP is changing frequently.)

Notes On a return from power failure, after an **ENABLE** command, or during the first execution of PPCL following a database load, the **TIMAVG** command begins executing with one sample. The value of the *result* will equal the current value of *input*.

TOD

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **TOD(***mode***,***recomd***,***time1***,***time2***,***pt1***,...,***pt12***)**

mode Mode number. Valid entries are any combination of 1, 2, 4, 8, or 16 as defined in the following table. Mode numbers can be added together to create customized schedules for the TOD program. The mode number 16 should only be used with the **HOLIDA** command.

MODE	**Schedule**
1	Normal Schedule
2	Extended Schedule
4	Shortened Schedule
8	Weekend Schedule
16	Holiday Schedule

recomd Determines if the points defined will be commanded after a return from a power failure (fail-safe position). *Recomd* is sometimes necessary for field panel warmstarts.

Valid entries are as follows:

0 = Does not recommand the points, *pt1* through *pt12,* on return from power failure.

1 = Recommand the points, *pt1* through *pt12,* on return from power failure.

time1	Time at which an ON command is executed. *Time1* can be entered in military format, decimal format, as a point name, or a local variable.
time2	Time at which an OFF command is executed. *Time2* can be entered in military format, decimal format, as a point name, or a local variable.
pt1 through pt12	Point names of digital output points that are commanded to a value.

Use

This command changes digital output point ON and OFF based on the day of the week and the time of day. If the mode number in the **TOD** command matches the mode number in the **TODMOD** command, then **TOD** commands the particular output points ON and OFF at the specified times.

Example

```
10  TOD(1,1,17:00,07:00,OLITE1,OLITE2)
```

The command time can also be a relative time point that is assigned a value of time. When you calculate a value of time to be used in a TOD command and assign it to the relative time point, make sure that the calculated time is either greater than or equal to the current time. The TOD command will not execute properly if the relative time point is commanded to a time that precedes, or is equal to, the current time.

See also **HOLIDAY**, **SSTO**, **TODMOD**, **TODSET**

TODMOD

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **TODMOD(***momode***,***tumode***,***wemode***,***thmode***,** *frmode***,***samode***,***sumode***)**

For all modes values, refer to the table following the mode descriptions:

momode	Mode number for Monday. Valid entries are 1, 2, 4 or 8.
tumode	Mode number for Tuesday. Valid entries are 1, 2, 4 or 8.
wemode	Mode number for Wednesday. Valid entries are 1, 2, 4 or 8.
thmode	Mode number for Thursday. Valid entries are 1, 2, 4 or 8.
frmode	Mode number for Friday. Valid entries are 1, 2, 4 or 8.
samode	Mode number for Saturday. Valid entries are 1, 2, 4 or 8.
sumode	Mode number for Sunday. Valid entries are 1, 2, 4 or 8.

MODE	**Schedule**
1	Normal Schedule
2	Extended Schedule
4	Shortened Schedule
8	Weekend Schedule
16*	Holiday Schedule

Days that have the same schedule are assigned the same MODE.

*Mode number 16 is not entered into the **TODMOD** command.

Use This command defines specific modes (normal, weekend, etc.) for each day of the week. The mode assigned to a specific day of the week is then used in conjunction with the **TOD** and **TODSET** commands to command points on that day.

Example

```
110  TODMOD(1,1,1,1,2,4,8)
```

In this example, Monday through Thursday are assigned to a *normal* schedule, Friday is assigned to an *extended* schedule, Saturday is assigned to the *shortened* schedule, and Sunday is assigned to the *weekend* schedule.

Notes The **HOLIDA** and **TODMOD** commands must precede any **TOD** or **TODSET** commands in order for the program to operate correctly.

A **HOLIDA** or **TODMOD** command in a field panel will only affect **TOD** and **TODSET** commands in that field panel .

When a holiday date occurs in a **HOLIDA** command, the mode number for that day in the **TODMOD** command is set to 16.

See also **HOLIDA**, **TOD**, **TODSET**

TODSET

Physical	Logical	Unitary	CM	APOGEE
	⅄		⅄	⅄

Syntax **TODSET(***mode***,***recomd***,***time1***,***val1***,***time2***,***val2***,***pt1***,. .., ***pt10***)**

mode	Mode number. Valid entries are 1, 2, 4, 8, or 16 as defined in the following table:

MODE	**Schedule**
1	Normal Schedule
2	Extended Schedule
4	Shortened Schedule
8	Weekend Schedule
16*	Holiday Schedule

Mode numbers can be added together to create customized schedules for the TOD program. The mode number 16 should only be used with the **HOLIDA** command.

recomd Determines if the points defined will be commanded after a return from a power failure (fail-safe position). *Recomd* is sometimes necessary for field panel warmstarts.

Valid entries are as follows:

0 = Does not recommand the points, *pt1* through *pt10,* on return from power failure.

	1 = Recommand the points, *pt1* through *pt10,* on return from power failure.
time1	Time at which the output points (*pt1* through *pt10*) are commanded to *val1*. *Time1* can be entered in military format, as a decimal, a logical point name, or a local variable.
val1	Analog value which the output points (*pt1* through *pt10*) assumes at *time1*. It can be entered in military format, as a decimal, a logical point name, or a local variable.
time2	Time at which the output points (*pt1* through *pt10*) are commanded to *val2*. *Time2* can be entered in military format, as a decimal, a point name, or a local variable.
val2	Analog value which the output points (*pt1* through *pt10*) assumes at *time2*. It can be entered in military format, as a decimal, a logical point name, or a local variable.
pt1 through pt10	Names of analog output points that are commanded ON and OFF.

Use This command is the counterpart of the **TOD** command for analog points. It commands analog output points based on the day of the week and the time of day. If the mode number in the **TODSET** command matches the mode number in the **TODMOD** command, then **TODSET** will command the particular output points to *val1* and *val2* at the specified times.

Example

```
550  TODSET(1,1,9:00,72.0,17:00,55.0, SPTEMP)
```

See also **HOLIDAY**, **TOD**, **TODMOD**

WAIT

Physical	Logical	Unitary	CM	APOGEE
⅄	⅄	⅄	⅄	⅄

Logical syntax **WAIT(***time***,***pt1***,***pt2***,***mode***)**

time	Time delay can be entered in the range of 1 to 32,767 seconds. This can be entered as an integer, a point name, or a local variable.
pt1	Point name of a digital trigger point. Valid point types are LDI, LDO, L2SL, L2SP, LOOAL, and LOOAP. You can also use local variables.
pt2	Point name of digital point to be commanded. Valid point types are LDI, LDO, L2SL, L2SP, LOOAL, and LOOAP. You can also use local variables.
mode	Defines the value to which *pt2* should be commanded based on the value of *pt1*. The following table lists valid modes.

Mode	**Meaning**
11	When *pt1* turns ON, wait *time* seconds then turn *pt2* ON.
10	When *pt1* turns ON, wait *time* seconds then turn *pt2* OFF.
01	When *pt1* turns OFF, wait *time* seconds then turn *pt2* ON.
00	When *pt1* turns OFF, wait *time* seconds then turn *pt2* OFF.

Physical syntax **WAIT(***time***,***pt1***,***pt2***)**

time	Time delay can be entered in the range of 1 to 32,767 seconds. This must be entered as an integer.
pt1	Point name of a digital trigger point. Valid point types are LDI, LDO, L2SL, L2SP, LOOAL, and LOOAP.
pt2	Point name of digital point to be commanded. Valid point types are LDO, L2SL, L2SP, LOOAL, and LOOAP.

Use For physical firmware, the **WAIT** command changes the value of a digital point based on the value of another digital point (the trigger point) after a desired time delay in seconds. When the trigger point is turned on, the program waits the amount of time defined, then turns on the digital point (*pt2*).

For all other firmware, the result point can be turned ON or OFF, based on the trigger point switching ON or OFF. Selection of trigger/result action is based on a mode you enter.

Example of a logical WAIT

```
70  C   EXAMPLE OF A WAIT COMMAND IN LOGICAL
71  C   FIRMWARE. WHEN CNPUMP IS TURNED ON,
72  C   THE FIELD PANEL WAITS 60 SECONDS
74  C   BEFORE TURNING CHPUMP ON.
75  WAIT(60,CNPUMP,CHPUMP,11)
```

Example of a physical WAIT

```
80  C   EXAMPLE OF A WAIT COMMAND IN PHYSICAL
81  C   FIRMWARE. WHEN CNPUMP IS TURNED ON,
82  C   THE FIELD PANEL WAITS 60 SECONDS
84  C   BEFORE TURING CHPUMP ON.
85  WAIT(60,CNPUMP,CHPUMP)
```

Notes

On a return from power failure or when the **WAIT** command is enabled, the trigger point must be toggled before the command executes regardless of the current state of either *pt1* or *pt2*.

The command to *pt2* is only issued once during the normal operation of the program until triggered again.

5

Glossary

This chapter contains programming terminology you will encounter in this manual. Refer to this chapter when you need information about a specific term.

This chapter also contains the PPCL Reserved Word List. The words, letters, and groups of letters placed on this list are used by the PPCL compiler. If you use any of the entries on the word list in a way other than the way they are designed (for example, defining a reserved word as a point name), your PPCL program will not operate properly.

Glossary of terms

APOGEE firmware

Firmware used in APOGEE field panels.

Argument

Type of variable whose value is not a direct function of another variable. Arguments can represent the location of a number in a mathematical operation, or the number with which a function works to produce its result.

Arithmetic function

Function that performs mathematical calculations on a value (number). When used in PPCL, the value derived from the calculation is usually assigned to a point name for future reference.

Arithmetic operator

Mathematically related functions that are performed on two or more operands (numbers). When used in PPCL, the value of the calculation is determined and assigned to a point name or local variable for future reference.

At (@) priority indicators

Indicators that are used to test if a point is at a specific priority, or to command a point to a specific priority. A maximum of 16 parameters can be used in one PPCL statement. When using a @ Priority indicator with PPCL statements, the priority level you define in that statement occupies one of the parameters.

CM firmware

Firmware used in Controller Modules and early revision Open Processors.

Command

Instruction evaluated by the computer.

Comment line

Information that is written into the program but is not interpreted as a program command. The compiler skips over the comment line during compilation. Comment lines allow you to enter text information describing the functionality of a specific section of code. Comment lines are especially helpful for describing subroutines and areas of program code that are difficult to understand.

Condition

Result of a comparison between two values.

Debugging

Process by which the logic of the program is tested for errors. A program is said to have "bugs" when it fails to function properly.

Device

In PPCL, a device represents any field panel or equipment controller that can execute PPCL statements.

Execute

To carry out the instruction of an expression or program.

Expression

Statement that describes a set of variables, constants, or values combined with arithmetic, logical, or relational operators.

Firmware

Portion of software used in a controller that is stored in non-volatile memory. See Logical, Physical, Unitary, CM or AOPGEE firmware.

Integer

Whole number (non-decimal number).

Line

Statement of program code assigned to a unique line number.

Local variable

Points in a program of a field panel that can be used in place of user-defined points to store temporary values ($LOC1 through $LOC15).

Logical firmware

Logical firmware is designed to accept engineering values (Deg °F, kPa, psi) for parameters in PPCL commands.

Logical operators

Operator that compares two conditions. The result of a comparison between the two conditions is called a condition. If the result of the condition is true, then a specific action is taken. If the result is false, then an alternative action is performed.

Modular programming

Style of programming that logical organizes code into common functions, such as operational modes.

Order of precedence

Order in which operators (mathematical, relational, logical, and special function) are evaluated in a command statement. Operators that have a higher precedence are evaluated before operators that have a lower precedence. If all the operators in the statement have equivalent precedence levels, then the operators are evaluated from left to right.

Physical firmware

Physical firmware is designed to accept values for parameters in PPCL commands derived from formulas and calculations.

Program

Collection of instructions combined in a logical order to accomplish a specific task.

Pseudocode

Non-syntactical description of program logic.

Relational operators

Relational operators compare two values. The result of the comparison is called a condition, and determines the type of action that should be taken.

Resident point

Predefined logical point that permanently resides in the PPCL program in the field panel.

Routing

Method for altering the flow of a program to transfer control to a line of the program other than the next sequential line number.

Special function

PPCL functions such as Alarm Priority (**ALMPRI**) that are used to access a specific value that is unique to a point. The value of the point can then be tested or assigned to other points. Since special functions are maintained by the system, they cannot be manually commanded to a different value. Special functions cannot be used over the network.

Statement

Collection of instructions to the computer in order to perform an operation. A statement can also be considered a line of PPCL code.

Subroutine

Portion of program code referenced repeatedly through one pass of the program.

Unitary firmware

Firmware that uses a subset of PPCL commands found in logical firmware.

PPCL reserved word list

The following predefined points permanently reside in the field panel's point database. These names are reserved for specific functions and should not be used as point names

$ARG1 through **$ARG15**	**$BATT**
$LOC1 through **$LOC15**	**$PDL**
.AND.	**.EQ.**
.GE.	**.GT.**
.LE.	**.LT.**
.NAND.	**.NE.**
.OR.	**.ROOT.**
.XOR.	**@EMER**
@NONE	**@OPER**
@PDL	**@SMOKE**
ACT	**ALARM**
ALMACK	**ALMCNT**
ALMCT2	**ALMPRI**
AND	**ARG1** through **ARG15**
ATN	**AUTO**
C (comment)	**COM**
COS	**CRTIME**
DAY	**DAYMOD**

DAYOFM	**DBSWIT**
DC	**DCR**
DEACT	**DEAD**
DEFINE	**DISABL**
DISALM	**DISCOV**
DPHONE	**ELSE**
EMAUTO	**EMER**
EMFAST	**EMOFF**
EMON	**EMSET**
EMSLOW	**ENABLE**
ENALM	**ENCOV**
EPHONE	**EQ**
EQUAL	**EXP**
FAILED	**FAST**
GE	**GOSUB**
GOTO	**GT**
HAND	**HLIMIT**
HOLIDA	**IF**
INITTO	**LE**
LINK	**LLIMIT**
LOC1 through **LOC15**	**LOCAL**
LOG	**LOW**
LOOP	**LT**
MAX	**MIN**
MONTH	**NAND**

NE	**NGTMOD**
NIGHT	**NODE1** through **NODE99**
NONE	**NOR**
NORMAL	**OFF**
OIP	**OK**
ON	**ONPWRT**
OPER	**OR**
PDL	**PDLDAT**
PDLDPG	**PDLMTR**
PDLSET	**PRFON**
RELEAS	**RETURN**
ROOT	**SAMPLE**
SECND1 through **SECND7**	**SECNDS**
SET	**SIN**
SLOW	**SMOKE**
SQRT	**SSTO**
SSTOCO	**STATE**
TABLE	**TAN**
THEN	**TIMAVG**
TIME	**TOD**
TODMOD	**TODSET**
TOTAL	**WAIT**
XOR	

6

Index