Honeywell

Control Language Advanced Process Manager Reference Manual

AP27-410

Honeywell

Implementation
Advanced Process Manager - 2

Control Language Advanced Process Manager Reference Manual

AP27-410
7/93

Copyright, Trademarks, and Notices

Revision 02 - July 23, 1993

About This Publication

This publication provides reference information about the Honeywell Control Language for the Advanced Process Manager (CL/APM).

This publication supports TDC 3000 software releases 400 through 410.

Change bars are used to indicate paragraphs, tables, or illustrations containing changes that have been made by Document Change Notices or an update. Pages revised only to correct minor typographical errors contain no change bars. All changes made by previous Document Change Notices have been incorporated in this update.

Table of Contents

Table of Contents

Table of Contents

INTRODUCTION
Section 1

This section tells you what this manual is about and refers you to other TDC 3000 publications for information related to CL.

1.1 PURPOSE

This publication provides reference information about Honeywell's Control Language for the Advanced Process Manager (CL/APM). CL/APM is used to build custom-control strategies that cannot be accommodated by standard TDC 3000 PV/Control Algorithms. CL/APM contains many similarities to the other TDC 3000 control languages—especially to CL/PM—but it also contains much that is unique to the construction of sequences for the Advanced Process Manager. See Appendix B.3 for a summary of CL/APM differences from CL/PM.

This manual **does not** provide instruction on how to transform a particular control strategy into a CL/APM structure; rather, it outlines the rules and describes all the components that can be used to build a CL/APM structure that will execute your control strategy.

This manual assumes that you are a practicing control engineer with knowledge of TDC 3000 product capabilities—specifically, the Advanced Process Manager (APM) and the data points that reside in it. You should also have an operational knowledge of the Universal Station's TEXT EDITOR and the COMMAND PROCESSOR and DATA ENTITY BUILDER functions available through the Engineer's Main Menu.

Sometimes, in order to make a concept more easily understood, an analogous concept in another programming language (such as Pascal or FORTRAN) is used. If you are not familiar with either of these languages, you can ignore the comments relating to them without eliminating any substance related to CL/APM.

1.2 REFERENCES

Because this manual primarily describes the elements and rules with which CL/APM structures are built, other publications are necessary to gain a complete knowledge of how CL/APM relates to the rest of TDC 3000 (in other words, how to implement a CL/APM structure once it is written). It is recommended that you read the following publications (at least the material under heading 1.2.2) before using this manual.

1.2.1 General CL/APM Information

Information Directory, SW01-400, in the *System Summary* binder—Tells which publications contain information on CL/APM.

Control Language/Advanced Process Manager Data Entry, AP11-400, in the *Implementation/Advanced Process Manager - 2* binder—Tells how a CL/APM structure is configured into the TDC 3000 System.

Configuration Data Collection Guide, SW12-400, in the *Implementation/Startup & Reconfiguration - 2* binder—Information in this publication ensures that you have collected the required data for implementing a CL/APM structure.

1.2.2 Publications with CL-Specific Information

Advanced Process Manager Control Functions and Algorithms, AP09-400, in the *Implementation/Advanced Process Manager - 2* binder—Describes the TDC 3000 On-Line Control Software and strategies that may include CL/APM structures. Includes discussions of data points that contain CL structures and run time information, such as parameters the operator interacts with to monitor/control CL execution.

Process Operations Manual, SW11-401, in the *Operation/Process Operations* binder—Section 6 describes the procedure to load a compiled CL/APM Sequence program to a Process Module Data Point residing in an Advanced Process Manager.

Data Entity Builder Manual, SW11-411, in the *Implementation/Engineering Operations - 1* binder—Describes how to use the Data Entity Builder to configure data points including Process Module Data Points in an APM.

Text Editor Operation, SW11-406, in the *Implementation/Engineering Operations - 1* binder—Describes how to use the Text Editor to enter CL structures (CL source files).

1.3 CL/APM OVERVIEW

CL/APM structures can be used to build a custom-control strategy to augment or replace standard TDC 3000 algorithms. The CL/APM structures are the Sequence Program, Abnormal Condition Handlers, and Subroutines.

NOTE

Structures can be independently compiled and sometimes are referred to as compilation units.

The following are the general steps required to build and implement a CL/APM structure (refer to heading 1.2, REFERENCES, for publications associated with the **BOLD-FACED** functions in the following steps). A complete description of these steps is given in the *Control Language/Advanced Process Manager Data Entry, AP11-400.*

1. Use the **DATA ENTITY BUILDER** (DEB) to configure (build) all APM data points associated with the CL/APM structure(s).

2. Use the Universal Station's **TEXT EDITOR** to create the CL/APM source file(s).

3. Use **CL** in the Command Processor on the US to compile the CL/APM structure. Compiling your source code turns it into an object file that can be executed in an APM.

4. Use the Process Operations Personality (Process Module Detail Display) to load the Sequence program to a Process Module Data Point in an APM.

RULES AND ELEMENTS OF CL/APM
Section 2

This section introduces you to the fundamental building blocks of CL/APM.

2.1 INTRODUCTION TO CL/APM RULES AND ELEMENTS

CL/APM, like any language, has certain characteristics that allow you to do certain things, while not allowing other things. For instance, comparing the characteristics of the English language with analogous characteristics in CL/APM, one arrives (roughly) at the following:

ENGLISH	CL/APM
grammar	syntax, rules
characters	characters
words, phrases	data types, variables, declarations,
clauses	expressions, conditions
sentences	statements (simple command or instruction to manipulate an element)
paragraph	step or phase
story, essay	sequence program

Heading 2.2 describes the basic rules (grammar) and elements (words) of CL/APM. The remainder of this section (headings 2.3, 2.4, and 2.5) describe more complex elements of CL/APM (that is, the "phrases" and "clauses" of CL/APM). After reading this section, you will be able to go to Section 3 and construct statements (sentences) using the building blocks from this section and the syntax governing statement construction.

In general, the description of each element (this section), statement (Section 3), and structure (Section 4) is presented in a 4-part format, as follows:

Definition—a brief "what is it" discussion.

Syntax—elements, statements, and structures are built following the specific form specified by the syntax. Another word for syntax is grammar, as previously mentioned. The form of anything you want to build in CL/APM must *exactly* follow the syntax, so that its structure can be compiled without syntax errors. Heading 2.2.4 in this section discusses the way in which the syntax for an element, statement or structure is presented. Appendix A contains a syntax summary for all elements, statements, and structures covered in this manual.

Description—applies to most, but not all elements, statements, or structures; a description explains less obvious attributes in more detail; for example, complex syntax or any restrictions that may apply.

Examples—contains typical uses of the element, statement, or structure, with incorrect uses included for contrast.

2.2 CL/APM RULES AND ELEMENTS

This subsection explains the rules and basic elements (or building blocks) that are used when building complex elements (Data Types, Variables and Declarations, Expressions and Conditions), or CL/APM Statements and Structures.

2.2.1 Character Set Definition

The CL/APM character set is composed of the 95 printable characters (including blank) of ASCII.

Compatibility with the ISO 646 standard is discussed in heading 2.2.5.2.

Characters can be combined to generate the following basic elements:

- Comments
- Identifiers (including reserved words; see heading 2.2.7.4—Reserved Words Definition)
- Numbers
- Quotes

Basic elements are further combined to produce complex elements such as data types, variables and declarations, and expressions and conditions. The complex elements are then used within statements (Section 3) and the statements are combined to build structures (Section 4).

2.2.2 Spacing

Adjacent elements can be separated by any number of spaces. Spaces are required between elements to prevent confusion; that is, at least one space must separate adjacent identifiers or numbers. Spaces cannot appear within an element other than in quotes and comments.

2.2.3 Lines

Your structure, whether you write it out on paper first or enter it into a file at the Universal Station using the Text Editor, consists of a sequence of lines. (Lines are also called **source** lines; the source code is what the compiler uses to create executable **object** code.) The following applies to the construction of source lines.

No basic element can overlap the end of a source line. Complex elements can continue onto succeeding source lines.

A statement (Section 3) can be continued onto successive lines. Each continuation line must have the ampersand character (**&**) in its first column. The end of the preceding line and the continuation character are treated as spaces.

A statement can start at any column on a line, subject to the restrictions stated in this section. Indentation is optional; refer to the sample structures in Section 4 for an idea of what good indentation techniques can do for the readability of a structure.

Each statement must begin on a new line, unless it is embedded in another statement (such as IF, ELSE, or a WHEN ERROR clause).

Lines may be blank. Blank lines and comment lines cannot appear between continuation lines.

Continuation Line Examples—The following examples show valid use of continuation lines:

```
  SEQUENCE good (APM;
&                     POINT
&                     APM01S01)
  PHASE
&             one
  IF NN(01) > 10.0 THEN (SET NN(01) = 10;
&                         SET NN(02) = 9.5)
  ELSE set FL(01) = on           --NOTE: no continuation for ELSE
  label1:
&             EXIT
  END good
```

The following examples are all invalid:

```
  SEQUENCE bad (APM; POINT APM01
&                       S01)    --identifier can't span lines
  PHASE one
  SEND "str
&         ing one"              -- String can't span lines
  label1:
              exit              -- must use continuation here
  IF NN(01) > 10.0 THEN (SET NN(01) = 10;
                         SET NN(02) = 9.5) -- must use continuation here
  END bad
```

2.2.4 Syntax

Because the syntax definition is part of the discussion of most elements, statements, and structures, you should become familiar with how the syntax is presented.

The syntax for each element, statement, and structure is presented along with its discussion in diagram form. The syntax diagram is entered on the left side with the name of the item shown in reverse-video/bold lettering. Follow the arrows to build the item, looping back optionally, or when necessary (for example, to build an ID, you must loop back through letter or digit for as many times as required to produce the ID String), until you exit the diagram on the right.

Syntax diagrams have three different symbols with text in them: rectangles, rectangles with curved ends, and circles. Items in rectangles refer to a syntax diagram in another part of the manual. Rectangles with curved ends are reserved words in CL/APM and must be written EXACTLY as shown. Items in circles are usually arithmetic operators and delimiters that must be included EXACTLY as shown when building a complex item such as an expression or a CL/APM Statement.

A summary of CL/APM syntax for all elements, statements, and structures in both syntax diagram form and in BNF is found in Appendix A.

2.2.5 CL/APM Restrictions

This section lists the restrictions placed on the language by either the CL compiler or some other element of the TDC 3000 System.

2.2.5.1 Lengths of Identifiers and Messages

The following restrictions are imposed by the TDC 3000 Universal Station:

- These types of identifiers (see heading 2.2.7) can be no longer than eight characters.

  – Parameter identifiers
  – Sequence program identifiers
  – Phase, Step, Subroutine, and Handler identifiers
  – Enumeration-state identifiers

- The length of a data point identifier depends on the tagname size option selected under the the TAG NAME OPTIONS menu, which is accessed from the SYSTEM WIDE VALUES menu.

  - SHORT tagname size = 8-character maximum for data point identifiers
  - LONG tagname size = 16-character maximum for data point identifiers.

- Messages sent to the Operator Station can be no longer than 60 characters.

- Local enumeration states can be up to 64 characters long.

2.2.5.2 ISO 646 Compatibility

The CL/APM character set is compatible with the international standard ISO 646 character set, of which ASCII is a variant. Certain character positions in the ISO 646 character set are permitted to vary for national use (see Table 2-1). Of these, CL/APM uses only the dollar sign.

Characters can be identified by their position (column/row) in the ISO 646 Basic Code Table (similar to an ASCII code chart). Several national variants of ISO 646 use character positions 4/0, 5/11 through 5/14, 6/0 and 7/11 through 7/14 as alphabet extensions such as accented or umlauted characters. Such characters cannot be used as alphabetics in CL/APM. (They can be used in Strings and comments.)

When the characters comma (,), quotation mark ("), apostrophe ('), grave accent (`), or upward arrowhead (^) are preceded or followed by a backspace, ISO 646 prescribes that they be treated as diacritical signs (for example, accents); however, CL/APM does not respect this use. The grave accent and upward-arrowhead characters are not permitted outside of Strings and comments.

Table 2-1 — Variable Characters in ISO 646

Position (col/row)	ASCII	Comments
2/3	#	also pound-sterling sign
2/4	$	also currency sign
4/0	@	varies
5/11	[	varies
5/12	\	varies
5/13	]	varies
5/14	^	varies sometimes
6/0	`	varies sometimes
7/11	{	varies
7/12	|	varies
7/13	}	varies
7/14	~	varies sometimes, also: overline

2.2.6 Comments

A comment begins with a double hyphen (--) and is terminated by the end of the source line. Comments can be seen in examples throughout this manual. A comment is not continued onto a continuation line, but a new comment can appear on each of several continuation lines.

2.2.6.1 Correct Examples of Comments

```
-- A long introductory comment should be written like
-- this, with a separate "--" on each line.
LOCAL numarr:              -- Individual comments
&     NUMBER array (1..3)  -- may be placed on each
&     AT NN(01)            -- continuation line
```

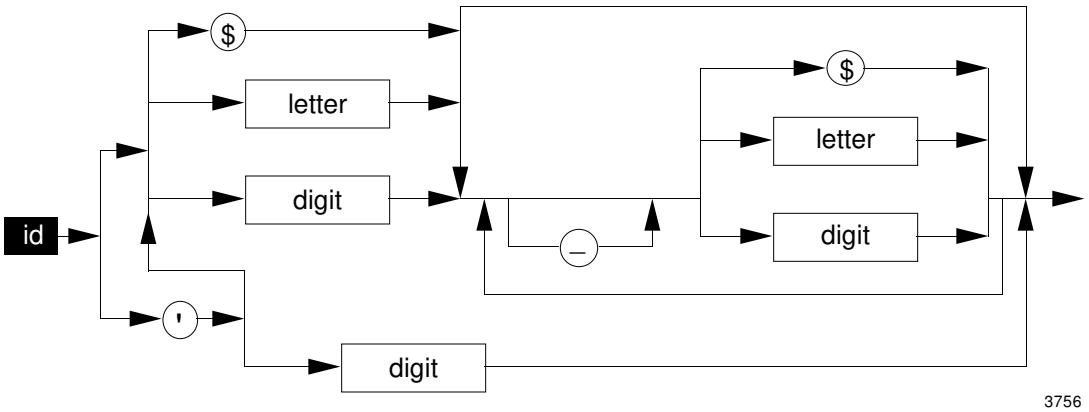
2.2.6.2 Incorrect Examples of Comments

```
-- This is a long comment such as you might use
&  to prefix a subroutine.  It is faulty because
LOCAL num at NN(01)   -- a comment cannot
&                     span source lines
```

2.2.7 Identifiers

Identifiers are used as the names of all kinds of objects in CL/APM: variables and constants, data types, program labels, data point names, etc. The keywords of CL/APM are also identifiers.

2.2.7.1 Identifiers Syntax



2.2.7.2 Identifiers Description

Identifiers are composed of the dollar sign ($), alphabetic characters (A to Z, and a to z), numeric characters (0 to 9), and the break character (underscore or underbar (_)). Special identifiers (see heading 2.2.7.5—Special Identifiers Definition) are prefixed with an apostrophe ('), but the apostrophe is not part of the identifier. An identifier (except for special identifiers) cannot be a single-digit numeric; a single letter or $ is acceptable but not recommended.

NOTE

Identifiers beginning with an exclamation mark (!) represent either digital input hardware points in the APM or a generic representation of the box data point, and have a more restrictive syntax than described here. Refer to headings 2.4.2.3 and 2.5.3 for further discussion of these points.

Break characters are used to divide a long identifier so that it can be more easily read. A break character cannot be the first or last character of an identifier. An identifier cannot contain two adjacent break characters.

Uppercase and lowercase characters can be used in identifiers, but the distinction between cases is not significant; that is, each lowercase character is considered the same as its uppercase counterpart. Within the restriction that no basic element may overlap the end of a source line, an identifier can be of any length (except as noted in heading 2.2.5.1).

An identifier can begin with or contain dollar signs. This permits references to Honeywell-supplied standard identifiers (such as Box data point identifiers—see heading 2.4.2), and other objects whose names begin with or contain dollar signs. Within a CL/APM program, there is no restriction on using identifiers that begin with dollar signs; however, such identifiers cannot be used to define any new object that is visible outside the program. Use of the dollar sign in system-visible object names is reserved to Honeywell.

2.2.7.3 Identifier Examples

```
VALVE     -- a valid identifier
valve     -- the same as VALVE
Valve     -- same as VALVE and valve
hot_pot   -- a valid identifier
hotpot    -- NOT the same as hot_pot
hot__pot  -- NOT VALID (adjacent breaks)
hot_pot_  -- NOT VALID (trailing break)
_hot_pot  -- NOT VALID (leading break)
pump2     -- a valid identifier
2N1401    -- also a valid identifier
14_34_6   -- also valid
14346     -- ok
$abc      -- valid identifier, restricted use
$4995     -- also valid, restricted use
!DI02S02  -- digital input module 2, slot 2
!DI0202   -- NOT VALID (missing "s")
!BOX      -- box data point of the bound data point
```

2.2.7.4 Reserved Words

Table 2-2 lists the CL reserved (identifiers) words that cannot be redefined by any structure.

Table 2-2 — CL Reserved Words*

ABORT	ENUMERATION	KEEPENB	READ
ACCESS	ERROR	LOCAL	REPEAT
ALARM	EU	LOOP	RESTART
AND	EXIT	MINS	RESUME
ARRAY	EXTERNAL	MOD	SECS
AT	FAIL	NOT	SEND
BLD_VISIBLE	FOR	OR	SEQUENCE
BLOCK	FROM	OTHERS	SET
CALL	GENERIC	OUT	SHUTDOWN
CUSTOM	GOTO	PACKAGE	STEP
CLASS	HANDLER	PARALLEL	SUBROUTINE
DAYS	HELP	PARAMETER	THEN
DEFINE	HOLD	PARAM_LIST	VALUE
ELSE	HOURS	PAUSE	WAIT
EMERGENCY	IF	PHASE	WHEN
ENB	IN	POINT	WRITE
END	INITIATE	RANGE	XOR

There are also a number of predefined identifiers in CL/APM. These identifiers are not reserved and can be redefined in a program.

We recommend that you avoid redefining any predefined identifiers, except when the result would not be confusing.

Predefined identifiers are type names, state names of predefined discrete types, and built-in function and subroutine names.

Predefined Discrete Types—The following type and its states is predefined:

Logical = Off/On

Alphabetical List—The following is a list of all the predefined identifiers in alphabetical order.

Abs	-- Built-in Function
APM	-- Sequence Program Type
Atan	-- Built-in Function
Avg	-- Built-in Function
Badval	-- Built-in Predicate
Cos	-- Built-in Function

* Those reserved words listed in Table 2-2 that are used by only CL/AM (e.g., BLOCK) are treated as reserved for CL/APM compilations as well.

Date_Time	-- Built-in Function
Equal_String	-- Built-in Function
Exp	-- Built-in Function
Finite	-- Built-in Predicate
Int	-- Built-in Function
Len	-- Built-in Function
Ln	-- Built-in Function
Log10	-- Built-in Function
Logical	-- Type name
Max	-- Built-in Function
MC	-- Sequence Program Type
Min	-- Built-in Function
Modify_String	-- Built-in Subroutine
Now	-- Built-in Function
Number	-- Type name and Built-in Function
Number_To_String	-- Built-in Subroutine
Off	-- Logical state name
On	-- Logical state name
PM	-- Sequence Program Type
Round	-- Built-in Function
Set_Bad	-- Built-in Subroutine
Sin	-- Built-in Function
Sqrt	-- Built-in Function
String	-- Type name
Sum	-- Built-in Function
Time	-- Type name
Tan	-- Built-in Function

2.2.7.5 Special Identifiers Definition

If an identifier is directly preceded by an apostrophe ('), that identifier is treated as an identifier, even though it may be spelled the same as a reserved word or is all numeric. There must be no spaces between the apostrophe and the identifier.

Except for conflict with reserved words or numbers, a special identifier must follow all the usual rules. The apostrophe cannot make a bad identifier good. Exception: A single-digit numeric identifier preceded by an apostrophe (for example, '9) passes the compiler, but is of questionable use or value.

2.2.7.6 Special Identifiers Examples

```
LOCAL foo: set/reset at FL(01)  -- invalid, SET is reserved
LOCAL bar: 'set/reset at FL(02) -- OK, set is an identifier
EXTERNAL 7                      -- invalid
EXTERNAL '7                     -- OK
LOCAL ' xyz at NN(01)           -- invalid, space follows '
LOCAL 'xyz_ at NN(02)           -- invalid, break character, "_",
                                -- cannot be the last character
                                -- of an identifier
```

2.2.7.7 Conflicts Between Identifiers

Under some circumstances, the same identifier can be used to name more than one thing without conflict. Under other circumstances, an attempt to reuse an identifier can cause a compile-time error.

The rules under which an identifier can safely name more than one thing are as follows:

1. There are three groups of identifiers that can be named: data types, objects, and program units. Two identifiers from the same group cannot have the same name if they are visible in the same scope. (See rule 2. for a discussion of scopes.) The identifier group can always be distinguished by the compiler, so a data type name and a program-unit name (for example) can never cause a conflict, even if they use the same identifier.

   Data Types are
   - Number
   - Time
   - Logical
   - Enumerations
   - String

   Objects are
   - Local variables
   - Local constants
   - Built in Functions
   - Arguments
   - Data Points
   - Parameters
   - Enumeration states

   Program Units are
   - Phases
   - Steps
   - Labels
   - Sequence Programs
   - Abnormal Condition Handlers
   - Subroutines

   Note that Subroutines are program units, but Functions are objects. This means that a Subroutine can have the same name as a local variable, but a Function cannot.

   Note also that data type names do not conflict with object names. This means, for example, that a parameter can have the same name as its data type. In fact, the names of TDC 3000 System-defined enumeration types are often the same as the data point parameters that possess those types. The MODE parameter of the Regulatory Control Data Point is an example. The MODE parameter is of type MODE enumeration. Pascal programmers should be aware of this to avoid confusion.

2. Identifiers do not cause conflict if they are declared in different scopes.

   Scopes are

Sequence Programs	Abnormal Condition Handlers
Phases	Steps
Subroutines	

   Most things can be declared in only a few of these scopes. For example, a Step can contain only Label declarations and a Subroutine can contain only Steps and/or Label declarations.

   Scopes can sometimes be nested. A Step can be within a Phase, and a Phase within a Sequence Program. Local Subroutines of a Sequence Program are considered to be nested within the Sequence Program; a Subroutine can, in turn, contain Steps.

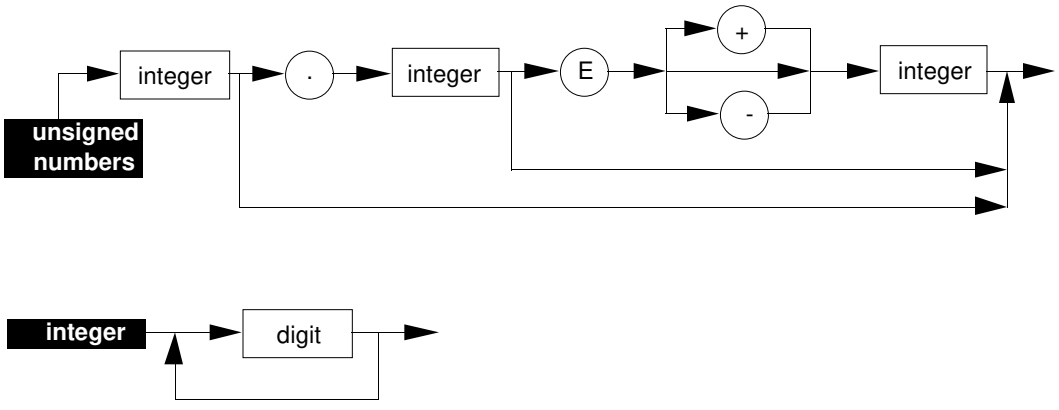
   When two scopes are nested, an identifier in an inner scope hides anything in an outer scope that has the same identifier and the same class (Program Unit, Data Type, or Object). For example, a Subroutine argument called X hides any local variable called X in the main program.

3. The only exception to rule 2 deals with parameters of the bound data point (the Process Module Data Point specified in the Sequence Heading). Bound data point parameters appear in every scope, exactly as local variables declared in that scope; therefore, no object can be declared in any scope that conflicts with the name of any bound data point parameters.

2.2.8 Numbers

A Number (or Numeric Literal) is an ordinary decimal number, with or without decimal point, and with an optional exponent.

2.2.8.1 Numbers Syntax



2.2.8.2 Numbers Description

Numbers that contain a decimal point must have at least one digit both to the left and to the right of the decimal point.

A Number that contains a decimal point can also have an exponent. An exponent consists of the letter E (either upper case or lower case), optionally followed by a plus or minus sign, followed by one or more digits.

A Number cannot contain spaces or break characters. In particular, spaces between a numeric literal and its exponent are not permitted.

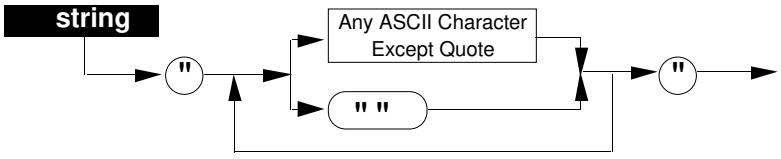
2.2.8.3 Numbers Examples

```
1000        -- valid
1000.       -- NOT VALID; no trailing digit
1000.0      -- valid; same as 1000
0.5         -- valid
.5          -- NOT VALID; no leading digit
10.02E1     -- valid
10.02e1     -- valid; same as 10.02E1
10.02 e1    -- NOT VALID; embedded space
1234.0E-2   -- valid
1234E-2     -- NOT VALID; no decimal point & trailing digit
1234.0E -2  -- NOT VALID; embedded space
1234.0E+2   -- valid
```

2.2.9 Strings

A String (or String Literal) is a sequence of zero or more characters enclosed at each end by quotation marks (").

2.2.9.1 String Syntax



2.2.9.2 String Description

Any printable character (see heading 2.2.5.2) can appear in a String Literal. If a quotation mark appears in a String Literal, it must be written twice.

A String Literal can contain a maximum of 64 characters. Literals longer than 64 characters are truncated to 64 characters by the compiler (with an appropriate user warning).

2.2.9.3 String Examples

```
"This is a String"    -- a String Literal
"&@$?*! system"       -- can contain any printable characters
""                    -- the empty String
"He said ""hello"""   -- he said "hello"
"A"   " "             -- two Strings of length 1
""""                  -- a string of length 1 (containing only a
                      -- quotation mark)
```

2.2.10 Special Symbols

The characters and combinations of characters in Table 2-3 are special symbols:

Table 2-3 — Special Symbols

Symbols	Meaning
+ – * / **	Arithmetic operators
< <= > >= <>	Relational operators
=	Equality, assignment operator
..	Range separator
( )	Parentheses
: ; . ,	Punctuation
"	String separator
– –	Comment separator
&	Line continuation
'	Special identifier prefix
!	First character of special point name form
$	First character of Honeywell reserved names

2.3 CL/APM DATA TYPES

This subsection describes the types of data that CL/APM can manipulate. In CL/APM, all data types are built into the language. There are two kinds of data types: scalar and composite.

- Scalar types have no components. They are the built-in types: Number, Time, and Discrete types (Logical and Enumeration types).
- Composite types are data points (because a data point can have lots of parameters/components), arrays (again, lots of components), and the built-in type String.

2.3.1 Number Data Type

All numeric values in CL/APM are of the single type, Number. This type is conceptually a subset of the real numbers and is internally implemented in single-precision floating point.

There is no separate Integer type in CL/APM; numbers may, of course, have integer values, and CL/APM built-in functions support truncation and rounding of non-integer values. In CL, the MOD operator, usually used to obtain the fractional remainder from an integer division, also can be applied to non-integer values. (In CL, the MOD value is calculated by subtracting the INT value of a divide result from the divide result.)

When a Standard Parameter of type integer is fetched, CL/APM converts it to a real number (single-precision floating point). When a number is stored to an integer parameter, the real number is rounded to an integer before it is stored.

2.3.1.1 Bad Values

In TDC 3000, a number may not always have a well-defined finite value. A numeric value may instead be Bad, Infinite, or Uncertain.

A bad value is represented by a special bit pattern that does not represent any number. (In the IEEE floating-point format, a bad value is represented by a NaN.)

A bad value can arise

- when a variable was never initialized or
- as the result of an invalid arithmetic operation; for example,
  - magnitude subtraction of infinities
  - zero multiplied by infinity
  - zero divided by zero
  - infinity divided by infinity
  - infinity MOD any number
  - any number MOD zero
  - square root of a negative number

- because a value read from an external data point was inaccessible (perhaps temporarily)
- or, because an operator or program explicitly chose to store a bad value.

CL supports bad values through the use of

- the predicate Badval(x), which tests the value x to determine whether it is bad
- the subroutine Set_Bad(x), which stores a bad value into the variable x
- the automatic propagation of bad values that take part in any arithmetic operation, such that if any operation is bad, the result is Bad.

On an attempted store of a bad value to a parameter (by either the SET statement, the READ/WRITE statement or the SET_BAD subroutine), if a bad value is not a legal value for the parameter being written to, the sequence program fails with "illegal value error."

Bad value comparisons work as follows in the APM:

- comparison of two values where one is a bad value and the other is a good value always returns false. For example, if x is a good value and y is a bad value, the following statements return false:

```
IF x = y THEN
IF x > y THEN
IF x <= y THEN
etc.
```

- comparison of two values where both are bad always returns false.

2.3.1.2 Infinite Values

The TDC 3000 numeric-representation format (IEEE floating point) allows for properly signed infinities, as well as ordinary finite numbers. Infinities are Normal values in TDC 3000 and can participate in arithmetic and be stored in parameters of data points with no alarms, warnings, or adverse effects on the CL program.

Infinities are propagated through arithmetic, except as described under **Bad Values**.

New infinities arise

- through division of a nonzero number by zero
- as the result of arithmetic overflow
- or, because a program or operator chose to store an infinite value.

There is no special representation of infinity in CL; rather, any Number whose magnitude is too large to be expressed as a standard floating-point number (e.g., 1.0e9999) is represented as a properly signed infinity.

The built-in predicate, Finite(x), is provided to test for infinities; it is defined under heading 2.7.4.5.

2.3.1.3 Uncertain Values

The PV parameter of an Regulatory Control Data Point can have an uncertain value. Unlike a bad value, an uncertain value has an actual numeric value and can be arithmetically manipulated. The status of a PV is maintained in the parameter PVSTS.

PVSTS has three states: NORMAL, UNCERTN, and BAD. It always tracks the value of the PV. Whenever a bad value is stored into a regulatory point's PV, the accompanying PVSTS parameter is automatically set BAD.

CL does not automatically propagate uncertain values.

2.3.2 Time Data Type

The Time data type represents an interval of time in the TDC 3000 format. Time values can be expressed in seconds, minutes, hours, days, or in any combination of these. Time values less than 1000 days are presented as a duration. Time values of 1000 days or greater are presented as an absolute time.

The minimum resolution of time is one tenth of a millisecond. However, the minimum resolution of a CL Time Literal is one second, and the minimum amount of time addressable with a CL statement is one second. The range of a time value is from -2**31 seconds (in the past) to +2**31 seconds (in the future). (2**31 seconds is approximately 68 years.)

Some other characteristics of time data are:

- Time values are computed by time expressions (see subsection 2.7.3) which are like arithmetic expressions.
- Tenths of milliseconds are maintained throughout time expressions.
- Values of type number are converted to values of type time by designating the number as an operand in a Time Literal (see subsection 2.7.3.5).
- Values of type Time are converted to values of type number by using the NUMBER built in function (see subsection 4.5.2.5).

NOTE

Do not confuse the TIME **data type** with either the Process Module, Box, or Array Data Point **parameter** named TIME. The compiler understands which is which by examination of (correct) use context.

2.3.3 Discrete Data Types

Real-world values are either continuous or discrete. Continuous values are often called **analog** and discrete values **digital**, because of the type of electronic circuitry used to bring these values into and out of a computer, but they are all represented in digital form in the computer.

In CL continuous values have the type Number. Discrete values have the type Logical or are Enumeration types.

A discrete type has two or more states. Each state has a name and is distinct from all other states. The order in which the states are named in the type declaration is significant. This means that two discrete types that have the same state names can be different types, because the order in which the states were declared differed. For example, **red/green/blue** is different from **blue/green/red**.

Variables of discrete types can be compared or assigned to only variables or values of the same type.

2.3.3.1 Shared State Names

A state name can be used in more than one discrete type. For instance, there might be a discrete type whose states are **open** and **close** and another whose states are **open** and **shut**. Although the respective **open** states in this example have the same name, they are not the same state. They cannot be compared, and a value of one type cannot be stored into a variable of the other type.

2.3.3.2 Enumeration Data Types

All discrete types, except Logical, are called Enumeration types. The only operations defined on Enumeration types are assignment and comparison for equality and inequality.

Many Enumeration types are predefined in a TDC 3000 System. These appear just as if those types had been defined in CL and compiled into the system database at some earlier time.

You cannot create enumerations with CL/APM (with the tightly limited exception of state lists used in LOCAL statements and SUBROUTINE argument definitions); however, variables of Enumeration Types can be declared in CL programs. Like all variables, these can be assigned only values of their type; thus, a variable of Enumeration type Red/Blue can be assigned only one of the values, Red, and Blue. The value Green cannot be assigned to such a variable.

2.3.3.3 Logical Data Type

Logical is a predefined discrete type that has two states: on and off. Unlike Enumeration Types, the following operations are defined on Logical values: AND, OR, XOR, and NOT.

Logical should not be considered the same as Pascal's Boolean type, or FORTRAN's LOGICAL type, because it is intended to represent only the state of a discrete variable. It does not represent truth or falsity. In CL, truth values are found in only conditional tests and cannot be stored in variables.

If you are familiar with Pascal, the comparison of program fragments in Figure 2-1 should show this difference.

Language:	Pascal	CL
	VAR flag: Boolean; x: real;	LOCAL flag: LOGICAL at FL(01) LOCAL x: NUMBER at NN(04)
Method	...	...
A	flag := x < 5;	SET flag = (WHEN x < 5: On; & WHEN x >= 5: Off)
B	IF x < 5 THEN flag := true ELSE flag := false	IF x < 5 THEN SET flag = On ELSE SET flag = Off

Figure 2-1 — Pascal Boolean vs. CL Logical

In Pascal method A, the result of the comparison **x < 5** is considered to be a value and is stored in the variable **flag**. Engineers with limited programming expertise find this is hard to read and understand. CL method A is just as efficient as Pascal method A and is easier to read.

Pascal method B and CL method B are the same. They are both easy to read, but each is a little less efficient than method A (assuming everything else is equal).

2.3.4 Arrays Data Type

In the APM, local arrays can be of type Flag, Numeric, String, or Time, and can have only one dimension. The index type must be Number. CL/APM can access array parameters such as the timer set point of the APM box data point or logic data point inputs.

All array parameters can be accessed with a variable or calculated subscript in any CL statement except CALLs to user-written subroutines and parameters/variables in READ or WRITE statements. If the result of the index expression is not an integer, it is rounded to the nearest integer. Array parameters cannot contain an off-node reference in the subscript. Note that with the parameters NN and FL, a calculated subscript has a severe performance impact—about five times longer than mapping these parameters to LOCAL variables and using a calculated subscript.

Whole array parameters are valid arguments to user-written subroutines. This includes the parameters NN, FL, TIME, STR8, STR16, STR32, and STR64 as well as any other array parameters, such as the L parameter on a Logic point.

2.3.4.1 Arrays Examples

```
EXTERNAL log1               -- a logic point
EXTERNAL APM02S02           -- a process module point in this node
EXTERNAL !BOX               -- this APM's box data point
EXTERNAL reg1               -- a regulatory point in another APM
EXTERNAL $NM10N02           -- another APM's box data point
LOCAL i AT NN(9)
          . . .
SET log1.L(NN(1)/2) = 25.2                    -- a valid subscript
SET log1.L(i) = 35.4                          -- a valid subscript
SEND : APM02S02.NN(APM02S02.NN(3)+7)          -- a valid subscript
SET APM02S02.NN(1), !BOX.NN(17*NN(8)) = 45.6  -- valid subscripts
CALL sub1 (APM02S02.NN)     -- entire array to a subroutine
CALL sub2 (APM02S02.NN(3))  -- single element of array to a subroutine
READ !BOX.NN(13) FROM $NM10N02.NN(reg1.PV)
                            -- ERROR, off-node variable in subscript
CALL sub3 (APM02S02.NN(i))  -- ERROR, variable subscript sent to a
                            -- user-written subroutine
CALL argsub1 (!BOX.NN, !BOX.FL)   -- valid arguments (whole arrays)
CALL argsub2 (NN, FL)             -- valid arguments (whole arrays)
CALL argsub3 (!BOX.TMSP)          -- valid argument
CALL argsub4 (log1.L)             -- valid argument
```

2.3.5 String Data Type

This CL/APM type consists of variable length strings of up to 64 printable ASCII characters. CL/APM support for the String data type is limited to the following.

- String parameters of APM data points can be read and written.
- Strings (and String Literals) can be assigned.
- Two strings can be compared for equality.
- String lengths can be tested with the built-in function Len.
- Strings can be modified and concatenated with the built-in subroutine Modify_String.
- Numbers can be converted to strings with the built-in subroutine Number_To_String.

Two strings are equal if the contents of the shorter string match contents of the longer string and the remainder of the longer string contains only blanks. For example, `"fred"` is equal to `"fred    "`, but is **not** equal to `"    fred"`.

Two types of string comparisons are provided, the equal (=) and not equal (<>) operators, and the built in function Equal_String.

- The equal and not equal operators are upper/lower case sensitive. For example, `"fred" <> "Fred" <> "FRED"`.
- Equal_String (see subsection 4.5.2.2) is upper/lower case insensitive. For example, `"fred" = "Fred" = "FRED"`.

When a string value is stored to a string parameter, it will be truncated if necessary to make it fit within the space allocated to the parameter when the point was built. Truncation is performed without any warning. Therefore, you cannot safely assume that a string read from a data point parameter is equal to the string that was stored into that parameter.

When string values are assigned from a constant or variable to another string variable, only "countable" characters are transferred. Nonspaces are always countable. Spaces are countable only when a non-space character follows before the end of the string.

When a string variable is assigned, it loses all traces of its former value, even when the new value has fewer countable characters than the old value. For example, if the following statements were executed

```
SET STR8(1) = "ABCDEFGH"
SET STR8(1) = "A"
```

then the final value of STR8(1) would be an A followed by seven blanks. This treatment applies for assignments between string variables of different maximum lengths. Thus, the statements

```
SET STR8(1) = "ABCDEFGH"
SET STR16(2) = "IJKLMNOPQRSTUVWX"
SET STR16(2) = STR8(1)
```

result in STR16(2) holding the value ABCDEFGH followed by eight blanks. When a larger string is assigned into a smaller one, then the value is truncated to fit. Note that the Sequence name, Phase name, Step name, Subroutine name, and Handler name strings are each limited to no more than eight characters. Examples are:

```
PHASE one        -- valid example
STEP A12345678   -- invalid example
```

2.3.6 Data Points Data Type

Data points are named composite structures that have named components called **parameters**. Data points are defined by the Data Entity Builder, rather than by CL. A data point is like a Pascal RECORD. Parameters of data points are identified by dot notation; that is, the point name is followed by a dot, which is then followed by the parameter name.

2.3.6.1 Data Points Example

```
A100.PV -- parameter PV of data point A100
DV_CTRL1.SVTV -- parameter SVTV of device control point DV_CTRL1
```

2.4 APM DATA POINTS

In addition to the process-connected data points used to monitor and manipulate the process, CL/APM uses two data point types that are related to APM operation. These point types are the Process Module Data Point and the Box Data Point.

2.4.1 Process Module Data Point

A Process Module data point is a data point in the APM that is used as the platform for sequence execution. Each sequence program must name a process module data point in the program header. At download time, the sequence program is bound (loaded) to that Process Module. The Process Module data point is also called the **Bound Data Point** of a sequence program.

Each Process Module Data Point (PMDP) contains flag data points, numeric data points, time variables, and string variables that are local to that PMDP (not to be confused with local variables that are declared in a sequence program). CL/APM programs can reference the local flags, numerics, time variables, and string variables of other PMDPs in the same APM when they are declared in EXTERNAL declarations. These local data points and variables in other nodes on the same UCN can also be referenced, but only by use of READ, WRITE, or INITIATE statements (see headings 3.2.4 and 3.2.15).

Parameters of external points are accessed by dot notation, as previously described. Parameters of the sequence program's bound data point are directly referred to without dot notation. It is an error to reference parameters of the bound data point by dot notation.

The name of the bound data point cannot be used in EXTERNAL declarations.

2.4.2 Box Data Point

A Box Data Point is a data point associated with a process-connected box, such as the PM or APM. It represents box parameters that are visible to TDC 3000 components, including CL/APM programs running in that APM or to CL programs running in PMs or APMs on the same UCN. These parameters include internal variables of the box such as flag and numeric data points, and time and string variables.

A program's view of box data-point parameters varies, depending on whether the program executes inside the box (APM or PM sequence program) or outside the box (BLOCK in an AM). For example, APM box numerics over 4095 are not visible to LCN devices as parameters of the box point, while APM or PM sequence programs have access to all box numerics as box parameters.

A general rule of thumb is that CL/APM can access all PM and APM parameters on the UCN, but cannot access any NIM-resident box parameters. For example, the parameter NODFSTAT (node functional state) is visible to LCN devices but not to CL/PM or CL/APM. See the *Advanced Process Manager Parameter Reference Dictionary* for physical location of the various parameters.

2.4.2.1 Box Data Point Identifier

APM box data-point identifiers follow a naming convention that establishes a set of names of the format

$NMxxNyy

where **xx** is the UCN number and **yy** is the APM number. Note that use of the box data-point identifier must follow its EXTERNAL declaration.

2.4.2.2 Box Data Point Identifier Example

```
$NM10N03.NN(1)      -- box numeric #1
$NM10N03.FL(12)     -- box flag #12
$NM10N03.TMSP(3)    -- box timer 3 setpoint
```

2.4.2.3 !BOX Box Data Point Identifier

The box data point of the bound data point also can be addressed by the special name form !BOX. This name can be used instead of the reserved name form of the box data point of the bound data point ($NMxxNyy), but if used, !BOX must appear in an EXTERNAL statement.

2.4.2.4 !BOX Box Data Point Identifier Example

```
EXTERNAL !BOX
EXTERNAL REACT102       -- another process module in the same APM
    ...
SET REACT102.NN(12) = !BOX.NN(33)
SET !BOX.FL(03) = off
```

2.5 ACCESSING APM PARAMETERS

A CL/APM sequence program can access parameters in the APM of the bound data point. It can also access parameters of points resident in other nodes on the same UCN with READ and WRITE statements, and initiate sequences in the same APM or in another APM or PM on the same UCN with the INITIATE statement. Only PM, APM, and LM parameters can be accessed. Attempting to access NIM-resident parameters causes a compile time error. See heading 1.7, CL Access, in the *Advanced Process Manager Parameter Reference Dictionary* for a list of PM parameters with access restrictions.

Parameters can be referenced (accessed) in four ways:

- as a local variable
- as a parameter of the bound data point or an EXTERNAL data point
- as a parameter of a digital input point named with the hardware addressing name form
- as a parameter of a box data point, either named explicitly or using the !BOX name form

In each of the above cases, the parameter must always be named explicitly; default parameters are not supported.

2.5.1 Local Variable Parameter Access

Number, Logical, String, and Time variables can be declared as LOCAL variables by using the AT clause to equate them with Numeric, Flag, STR8, STR16, STR32, STR64, or Time variables of the bound data point, or of another process module in the same APM, or of the box data point, or of Array points. Refer to subsection 2.6.2.3 for examples.

2.5.2 Bound Data Point/External Data Point Parameter Access

All APM-resident point parameters can be referenced by declaring the point either as the bound data point in the sequence header, or by using the EXTERNAL statement. All points referenced in the program must appear in one of those two places. Points in other nodes on the same UCN can be referenced only by using the READ/WRITE and INITIATE statements.

Note that if a Box Numeric or Flag is defined as an external data point, it is given an artificial .PV parameter that contains its value. See points E100 and F100 in the example that follows.

Parameters of the bound data point must be referenced using only the parameter name. It is a compile time error to reference the bound data point parameters naming the bound data point.

Points referenced in the program must first have been built using the Data Entity Builder. Referencing a point not yet built results in a compile time error.

Example:

```
SEQUENCE filler (APM; POINT REACT101)
   EXTERNAL A100, D100        -- points in this APM
   EXTERNAL B100, C100        -- points in another APM
   EXTERNAL E100              -- numeric point in this APM
```

```
EXTERNAL F100              -- numeric point in another APM
EXTERNAL Valve1, Valve2    -- digital composites in this APM
EXTERNAL DV_CTRL1          -- device control point in this APM
EXTERNAL REACT102          -- process module in this APM
EXTERNAL !BOX              -- own box data point
EXTERNAL $NM10N03          -- another box data point
...
LOCAL num AT NN(01)
  ...
  SET A100.SP = D100.SP                    -- this APM
  READ num, NN(02) FROM B100.SP, C100.SP -- read from another APM
                                           -- into this APM
  WRITE $NM10N03.FL(13) FROM !BOX.FL(13) -- write to another APM from
                                           -- this APM
  OPEN Valve1, Valve2                      -- this APM
  INITIATE REACT102 : HOLD                 -- initiate hold of another
                                           -- sequence in this APM
  SET FL(02) = REACT102.FL(02)             -- this APM
  SET REACT101.NN(4) = 14.4                -- INVALID, bound data point
                                           -- named
  SET B100.SP = 10                         -- INVALID, another APM,
                                           -- use WRITE
  WRITE F100.PV from E100.PV               -- write to another APM
& (when error GOTO lab2)                   -- from this APM
  OFF DV_CTRL1                             -- state change for device
                                           -- control point
```

2.5.3 Hardware Addressing Name Form Parameter Access

Digital input hardware in the APM of the bound data point can be referenced without building points to represent those slots. This special name form is valid for only CL/APM and the Data Entity Builder and follows the convention

!DImmSss

where **mm** represents the module number of the digital input (01-40) and **ss** represents the slot number (01-32). These name forms can be used in place of standard point names in executable statements but, if used, the name must appear in the EXTERNAL statement.

Example:

```
EXTERNAL !DI01S01                  -- module 1, slot 1
EXTERNAL !DI01S02, !DI01S03        -- module 1, slots 2 and 3
EXTERNAL REACT101                  -- another process module in
                                   -- this APM
      ...
SET !DI01S02.PVFL, !DI01S03.PVFL = !DI01S01.PVFL
SET FL(01), REACT101.FL(10) = !DI02S03.PVFL
IF !DI01S02.PVFL <> REACT101.FL(11) THEN GOTO label1
```

The only parameter valid for this name form is PVFL. This is a logical parameter representing the state of the PV.

2.5.4 Box Data Point Parameter Access

APM-resident parameters (flags, numerics, and timers) of the box data point in which the bound data point resides can be referenced as long as the box data point is named in an EXTERNAL statement. This can be accomplished either by using the reserved point name form (for example, $NMxxNyy) or by using the !BOX name form.

In addition, node-resident parameters of box data points in other nodes on the same UCN can also be referenced, as long as the box data point is named in an EXTERNAL statement. In this case, only the reserved point name form is valid. (These parameters can be referenced in only READ/WRITE statements.)

Example:

```
EXTERNAL !BOX                 -- own box data point
EXTERNAL $NM10N03, $NM10N05   -- box data points of other APMs on the
                              -- same UCN as the bound data point
 ...
SET !BOX.NN(3), !BOX.NN(4), !BOX.NN(5) = 0
CALL Set_Bad (!BOX.nn(10) )
IF Badval (!BOX.NN(11) ) THEN GOTO label3
READ !BOX.NN(02) FROM $NM10N03.NN(13)
WRITE $NM10N03.FL(9), $NM10N05.FL(9)
& FROM !BOX.FL(10), !BOX.FL(11)
```

2.5.5 I/O Module Prefetch Limits

Except for PVs of digital input points, all I/O module parameters of the local APM that are accessed (read) by a sequence program within a step are prefetched just before the execution of the process module. This includes any prefetches that may be required for evaluation of the processing conditions for currently enabled abnormal condition handlers.

There is a compile time limit of 12 prefetches per step, or per WHEN condition on an abnormal condition handler. That's the easy case. There is also a run time limit of 12 prefetches per step, including the WHEN conditions of all currently enabled abnormal condition handlers. If this limit is exceeded, a run time error (Error 109) is reported and the slot goes to failed state. It is important to note that if the same I/O module parameter is needed more than once (for example, used by both the step and one or more of the handlers, or just used by multiple handlers), each use counts as one prefetch for the run time count. Depending on the mix of prefetches between the step and the enabled abnormal condition handlers, the step can fail either before execution or after partial execution. Once the sequence has been failed, it must be turned off before another sequence can be loaded.

If the first step of a subroutine or abnormal condition handler does not require prefetches, the subroutine/handler is NOT a preemption point. If the step that called the subroutine did not require prefetches, the return from the subroutine is NOT a preemption point. A return from a handler requires a RESUME statement that is a preemption point because it returns to a phase.

2.5.6 I/O Module Poststore

Value stores by CL/APM to I/O module parameters are not performed until the next preemption point is reached. This is known as poststore. See subsection 6.3 of the *Control Language/Advanced Process Manager Data Entry* manual for an explanation of how to locate the source of a runtime error when poststore is involved.

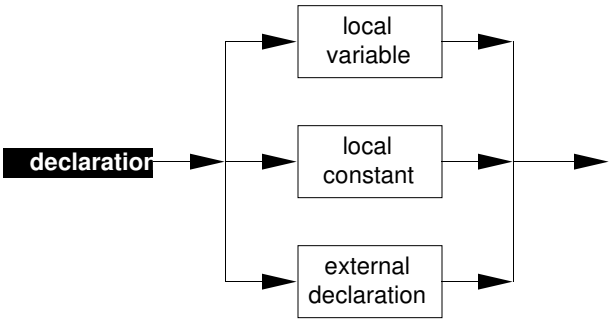
2.6 VARIABLES AND DECLARATIONS

This section describes the declaration, use, and scope of local variables, local constants, and external variables (data points).

Local variables and local constants are owned by a CL program and are defined (declared) at the beginning of the CL program through the LOCAL statement. External variables (data points) are defined outside CL and must be declared at the beginning of the sequence program through the EXTERNAL statement.

All declarations must precede any executable statements in the program.

2.6.1 Variables and Declarations Syntax



2.6.2 Local Variables

Local variables are declared at the beginning of the sequence program. All CL/APM local variables are mapped to parameters of Process Module Data Points, or of APM Box Data Points. This mapping, and the amount of space available for each variable type, is summarized in Table 2-4.

Local variables declared in a program are visible only to the program they are declared in and to any local subroutines belonging to that program; however, the values assigned to these variables are visible outside the program because each such variable is mapped to a parameter of either a Process Module Data Point, a Box Data Point, or an Array point.

Table 2-4 — Mapping of Local Variables

Data Type	Parameter name	Maximum per PMDP	Maximum per Box	Maximum per Array Point
Numeric	NN	80	16384	240
Logical or State_List	FL	127	16384	1023
Time	TIME	4	4096	240
String	STRn	varies by string length (see below)	16384 (STR8 only)	240

Each Box string is restricted to 8-characters or less (STR8). Strings in the PMDP also can have the longer lengths (STR16, STR32, STR64). However, the total space for strings in each PMDP is limited to 128 characters mapped as follows:

STR8(1)	STR8(2)	STR8(3)	STR8(4)	STR8(5)	STR8(6)	STR8(7)	STR8(8)
STR16(1)		STR16(2)		STR16(3)		STR16(4)	
STR32(1)				STR32(2)			
STR64(1)							

STR8(9)	STR8(10)	STR8(11)	STR8(12)	STR8(13)	STR8(14)	STR8(15)	STR8(16)
STR16(5)		STR16(6)		STR16(7)		STR16(8)	
STR32(3)				STR32(4)			
STR64(2)							

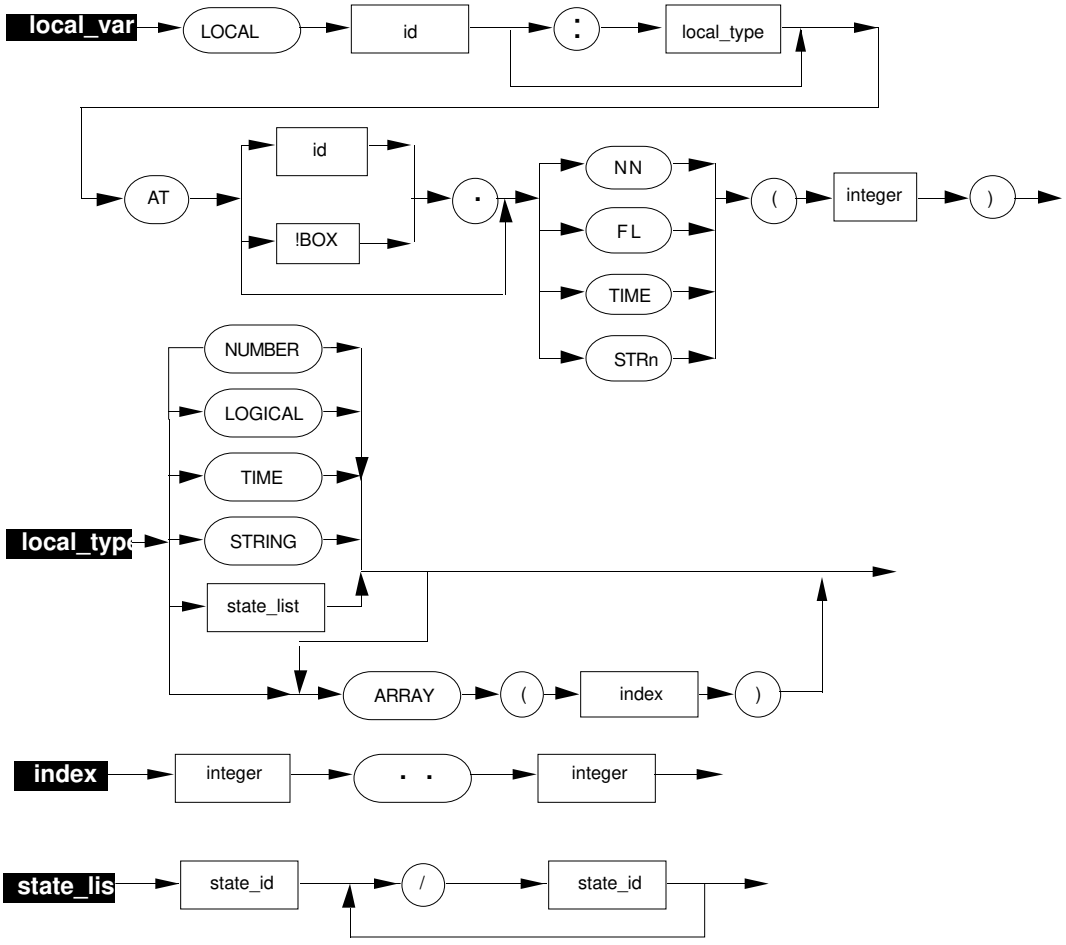
Note that the longer strings are not separately allocated within the PMDP database. Instead they are created through the concatenation of a set of 8-character strings. Thus, if you change the contents of STR8(10), you also are changing the contents of STR16(5), STR32(3), and STR64(2).

Array Point strings can be referenced by string lengths other than the configured length so long as they fall within the configured total number of characters. For example, if an Array Point named ARRAYPT1 is configured to have an array of 10 string elements of string length 64 (thus 640 characters), each of the following is valid:

```
LOCAL str_ary : STRING ARRAY (1..80) AT ARRAYPT1.STR8(1)
LOCAL str_ary : STRING ARRAY (1..40) AT ARRAYPT1.STR16(1)
LOCAL str_ary : STRING ARRAY (1..20) AT ARRAYPT1.STR32(1)
LOCAL str_ary : STRING ARRAY (1..10) AT ARRAYPT1.STR64(1)
```

Statements that exceed the configured string length result in compile-time errors or string truncation.

2.6.2.1 Local Variables Syntax



NOTE

In LOCAL Variable declarations, any state_list clause is restricted to only two states.

2.6.2.2 Local Variables Description

LOCAL variables can be of type Number (default if no data type is specified), Logical, Time, String or a 2-state Enumeration. The Enumeration is specified by listing the two states separated by a slash (/). Note that a variable of 2-state Enumeration type is not equivalent to a logical, even though its hardware location is a flag location. When the variable is used in a SEND statement, the values are displayed in terms of the two state names.

One-dimensional arrays of LOCAL variables can be specified. The array index must be a number declared by naming lower and upper bounds (e.g., 1..5). The lower bound is the left-most integer in the index. The upper bound is the right-most integer in the index. The value of the lower bound must be less than the value of the upper bound.

If a point id is not specified in the required AT clause, the local variable maps to the Bound Data Point's variable. If a point id is specified, it must name either another Process Module Data Point in the same APM as the Bound Data Point, the local box data point (as specified by either !BOX or $NMnnNmm), or an Array Point. Any other point type is a compile-time error. LOCAL declarations of variables outside the bound data point must follow the EXTERNAL declaration of the data point that contains the variable (see subsection 2.6.4).

If the local variable is a scalar, the compiler places the local variable at the named numeric, flag, string, or time location. If the local variable is an array, its first element is placed at the named location and the remaining elements occupy contiguous variables in ascending order from the named location.

Array points allow you to fetch and store arrays of numeric, flag, string, or time variables from an APM. Array points also can be used to access arrays of numeric, flag or string variables (but not time variables) from a different device through a Serial Interface IOP connection. The connection is initially set up in the Data Entity Builder when the Array Point parameter External Data Option (EXTDATA) is set to either IO_NN, IO_FL, or IO_STR. As in an example shown below, CL prints a warning message when a local variable is mapped to a Serial Interface IOP parameter.

2.6.2.3 Local Variables Examples

```
EXTERNAL REACT102, REACT103, REACT1O4, REACT105 !BOX
EXTERNAL arraypt1, arraypt2

LOCAL num AT NN(01)                  -- bound data point numeric
LOCAL log: LOGICAL AT FL(03)         -- bound data point flag
LOCAL num1: NUMBER AT REACT102.NN(10)
                                     -- numeric of another process
                                     -- module data point in this APM
LOCAL log1: LOGICAL AT !BOX.FL(25)   -- this box's flag 25
LOCAL log2: LOGICAL AT !BOX.FL(26)   -- next flag in this box
LOCAL numarr: ARRAY(3..5) AT NN(10)  -- array using bound data point
                                     -- numerics
LOCAL pm3logarr: LOGICAL ARRAY (1..2) AT REACT102.FL(15)
                                     -- logical array using flags of
                                     -- another process module data pt
LOCAL boxnmarr: NUMBER ARRAY (5..10) AT !BOX.NN(25)
                                     -- number array of this box's
                                     -- numerics
LOCAL var1: one/two AT FL(1)         -- state name list using a flag
```

```
LOCAL var2: three/four ARRAY (1..2) AT !BOX.FL(2)   -- state name
                                                    -- list array
LOCAL var3:open/close ARRAY (5..10) AT REACT102.FL(7)
LOCAL arr_time : TIME ARRAY (1..3) AT arraypt1.TIME(1)
                          -- array of times from an APM array point
LOCAL arr_str : STRING ARRAY (2..5) AT arraypt1.STR64(2)
                          -- array of strings from an APM array point
LOCAL arr_flag : FLAG ARRAY (1..10) AT arraypt1.FL(1)
         ^P
**WARNING** This LOCAL variable is mapped to a Serial Interface IOP
parameter
                          -- array of flags from an SIO array point
LOCAL elapsed_time : TIME AT REACT103.TIME(4)
LOCAL time_arr : TIME ARRAY (1..3) AT REACT104.TIME(2)
LOCAL message_string : STRING AT APMS22.STR16(2)
LOCAL str_arr : STRING ARRAY (3..4) AT !BOX.STR8(1)

SET arr_time(1) = NOW
IF var2(2) = four THEN SET arr_str(3) = "ABORT"
SET arr_flag(2) = OFF
SET var1 = two
SET var3(8) = open
IF var2(2) = four THEN ABORT
SEND : var1, var3(8), var2(2)
```

Be careful when naming LOCAL variables. **Do not** use the same name for a LOCAL variable that already exists as a parameter on any APM point referenced by the program. Note that if you do make this error, the APM tagname.parameter, not the LOCAL variable, gets flagged with the compile error making your mistake difficult to track down. For example:

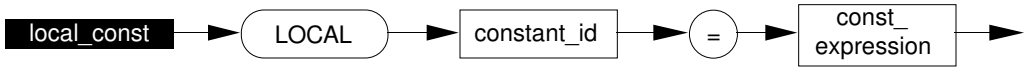
```
LOCAL start: TIME AT TIME(01)
.
.
.
SET FY21000.COMMAND = START       -- FY21000 is a REG PV totalizer
                                  -- with START as a valid parameter
```

The LOCAL definition will compile without comment, but the statement SET FY21000.COMAND = START will be flagged with a "TYPE MISMATCH" compile time error.

2.6.3 Local Constants

Local constants of type Number, Time or String can be declared.

2.6.3.1 Local Constants Syntax



2.6.3.2 Local Constants Description

Local constants cannot be modified.

Constant expressions of Number type must be composed of arithmetic operators, the built-in function ABS, numeric literals, and identifiers that have been previously declared as LOCAL numeric constants. No other functions are permitted. The local constant expression cannot contain divide (/) or remainder (MOD) operators.

Time constants can contain any mixture of days, hours, minutes, and seconds and can be used in any CL/APM statement where time variables can appear. See subsection 2.7.3 for a description of possible Time Expressions. A time constant defined with a value larger than the supported maximum time value defaults to the maximum time value.

The constant expression for String type consists of a string literal. A string constant can be used in any CL/APM statement where a string variable can appear.

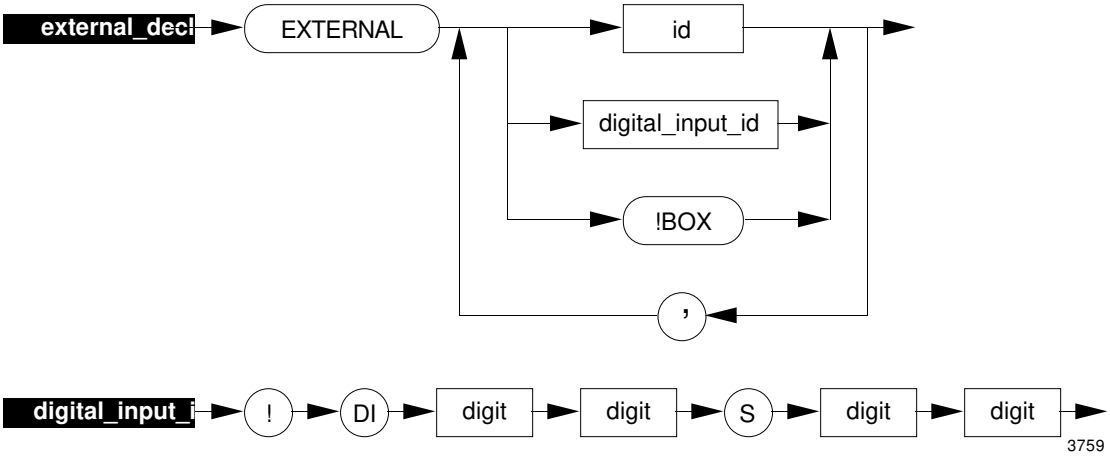
2.6.3.3 Local Constants Examples

```
LOCAL pi = 3.14159265           -- a numeric constant
LOCAL ten_K = 1.0e4             -- another
LOCAL 2_pi = pi * 2             -- a constant expression
LOCAL pi_2 = pi/2               -- illegal: divide operator
LOCAL const1 = 3.0              -- a numeric constant
LOCAL const2 = ABS(const1 * const1 - 4.0)
                                -- numeric constant using ABS
LOCAL time_const1 = 3 MINS 10 SECS
LOCAL time_const2 = 12 HOURS 44 MINS 2 SECS

LOCAL string_const1 = "problem"
LOCAL cascade_string = "cascade"
EXTERNAL a100
        . . .
WAIT time_const1
IF TIME(3) > time_const2 THEN GOTO PHASE time_up
       . . .
SEND : string_const1
IF a100.MODE = CAS THEN SEND : cascade_string
```

2.6.4 External Data Points

Data points other than the bound data point can be accessed by a CL program only if they are named in an EXTERNAL declaration.

2.6.4.1 External Data Points Syntax



2.6.4.2 External Data Points Description

The EXTERNAL declaration introduces the name of one or more data points. None of these can be the bound data point. Each external data point must already exist in the system at the time the program is compiled; otherwise, the compiler reports an error. External data points can be declared in sequence programs, but NOT in Subroutines.

EXTERNAL declarations can name only data points in nodes on the same UCN as the Bound Data Point. Data points in the NIM and other LCN modules cannot be referenced. Refer to heading 2.5 for more information.

Data points must have been previously built by the Data Entity Builder to be used in an EXTERNAL statement; otherwise, the compiler reports an error. The only exceptions to this rule are the two special name forms !DImmSss (see heading 2.5.3) and !BOX (see heading 2.4.3.3).

2.6.4.3 External Data Points Examples

```
EXTERNAL anp049hx, AX_001
EXTERNAL APM01S03, !BOX
EXTERNAL !DI01S02, !DI02S03, A100
EXTERNAL DV_CTRL1
```

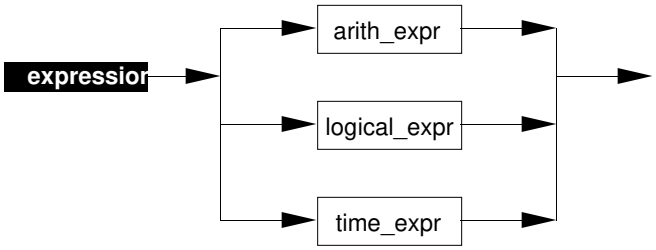
2.7 EXPRESSIONS AND CONDITIONS

This section describes the formation of arithmetic expressions, logical expressions, and time expressions available in CL/APM. It also introduces the use of conditions to test relations (equality, relative magnitude, etc.) of expression values.

2.7.1 Expressions

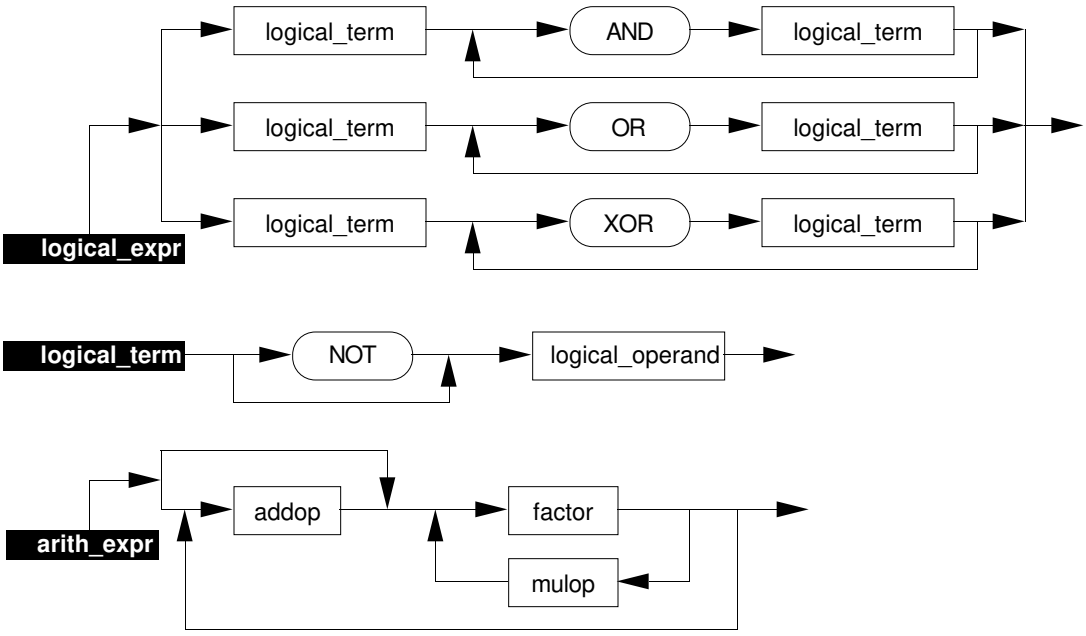
An expression is a formula that defines the computation of a value. The components of an expression are operands and operators.

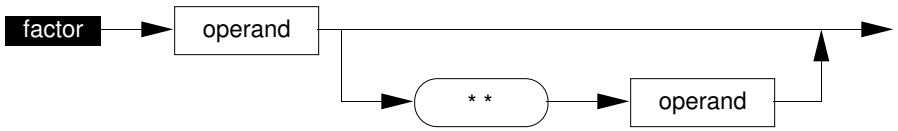
2.7.1.1 Expressions Syntax



2.7.2 Arithmetic and Logical Expressions

2.7.2.1 Arithmetic and Logical Expressions Syntax

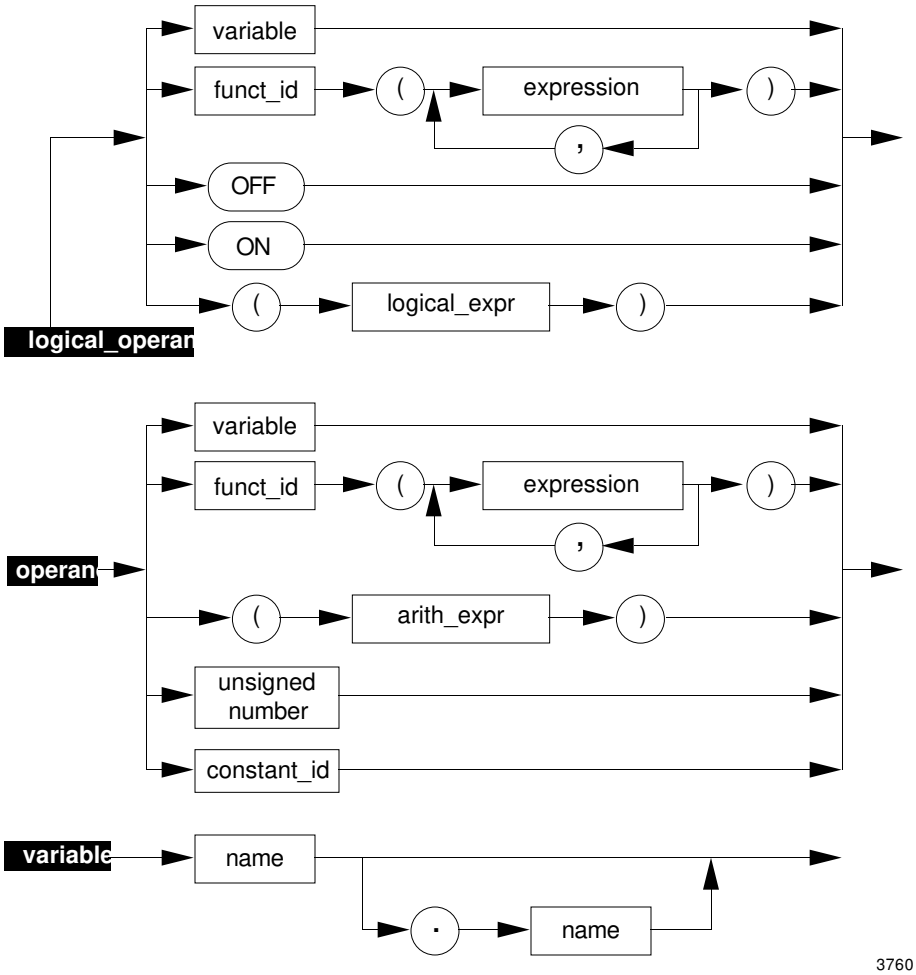


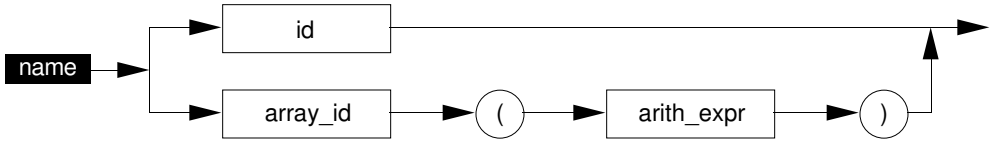


2.7.2.2 Operand Definition

An operand can be a variable, a parameter of a data point, an array element, a number, a quoted String, a discrete-state identifier, a function call, or an expression enclosed in parentheses.

2.7.2.3 Operand Syntax





2.7.2.4 Operators Definition

Operators operate on one or two operands and produce a value that can itself be operated on. Operators can be monadic (take a single operand) or dyadic (take two operands). Operators bind according to the priority order given in Table 2-5. Higher priority means closer binding.

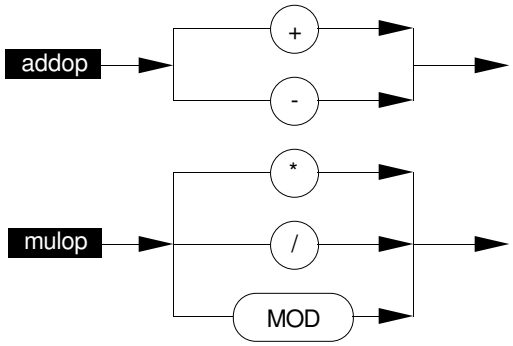
For example, in the expression a + b * c, b * c is evaluated first because the * operator has a higher priority. Operators with the same priority are evaluated left to right.

Table 2-5 — Operator Priorities

Priority	Operator	Meaning
3	**	Exponentiation
2	*	Multiplication
	/	Division
	MOD	Remainder (see heading 2.3.1)
	NOT	Logical complement
1	+	Sum
	-	Difference, Negation
	AND	Logical And
	OR	Logical Or
	XOR	Logical Exclusive Or

Arithmetic Operators Definition—The arithmetic operators are +, -, *, /, MOD and **; their usual meanings are as listed in Table 2-5. Arithmetic operators can take only operands with data type NUMBER (also see heading 2.7.3.7).

Arithmetic Operators Syntax



Arithmetic Operators Description—The minus sign can be used to indicate subtraction (e.g., x-y), or to indicate negation (e.g., -x). As a negation operator, the minus sign cannot appear twice in a row.

The exponential operator is not associative; **a ** b ** c** is invalid and must be rewritten as **a**(b ** c)** or **(a **b) **c**.

Logical Operators Definition—The Logical operators are NOT, AND, OR, and XOR (exclusive or), with their usual meanings, as shown in Table 2-6. Logical operators can take operands of only type LOGICAL (also see Section 2.7.4.6).

Table 2-6 — Logical Operators Truth Table

a	b	NOT a	a AND b	a OR b	a XOR b
On	On	Off	On	On	Off
On	Off	Off	Off	On	On
Off	On	On	Off	On	On
Off	Off	On	Off	Off	Off

When different Logical operators are used in the same expression, parentheses must be used to show grouping. In other words, the phrase **a AND b AND c** is valid, but **a AND b OR c** is not, and must be rewritten as **a AND (b OR c)** or **(a AND b) OR c**.

If the condition in an IF (condition) THEN (consequent) statement involves both OR and XOR, an incorrect error message may be given. For example,

```
IF (x = 1 OR y = 1) XOR z = 1 THEN .......... gives the error message
                    ^
       Type logical expected
```

A work-around can be used; write the XOR in terms of AND and OR. For example, the above statement could be rewritten as follows:

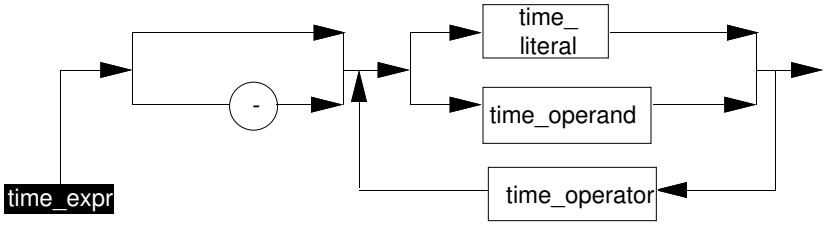
```
      IF ((x = 1 OR y = 1) OR z = 1) AND NOT
&          ((x = 1 OR y = 1) AND z = 1) THEN ............
```

2.7.3 Time Expressions

A time expression is a special form of expression—composed of time operands and time operators—that produces a result of the Time data type. It is used to express either a duration or an absolute time.

A negative result from a time expression is displayed as an unpredictable absolute date. If you need to see the actual value of a negative result, use the NUMBER function to convert the value to a Number data type, and send that result to the Universal Station.

2.7.3.1 Time Expression Syntax



2.7.3.2 Time Operands Definition

A time_operand can be a

- variable,
- parameter,
- constant,
- array element,
- literal, or
- function call with a Time data type result.

2.7.3.3 Time Operators Definition

The Time operators are a subset of the arithmetic operators, and they have the same priority. The types of operands required for these operators are shown in Table 2-7.

Table 2-7 — Time Operators

Operator	Left Hand Side	Right Hand Side	Result
+	Time	Time	Time
-	Time	Time (or none)	Time
*	Time	Number	Time
*	Number	Time	Time

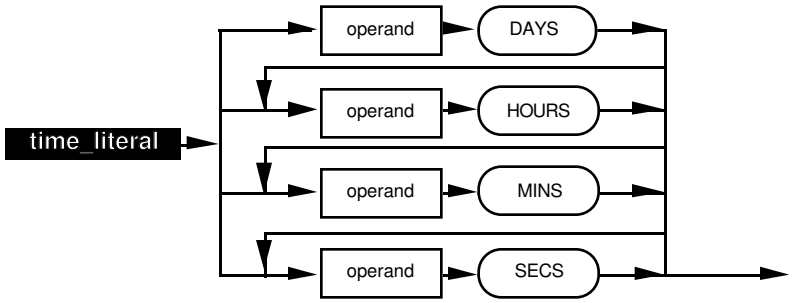
2.7.3.4 Time Expression Examples

```
EXTERNAL arraypt3
LOCAL n at NN(18)
LOCAL t1 : TIME AT Time(1)  -- TIME defines the data type, and
LOCAL t2 : TIME AT Time(2)  -- Time(n) specifies an APM parameter
LOCAL sked : TIME AT arraypt3.Time(2)
   ...
SET t1 = 1 DAYS 2 MINS      -- set a time variable
SET sked = 20 SECS          -- set another time variable
SET t1 = n SECS + sked
SET t1 = n * t2
```

2.7.3.5 Time Literal Definition

A Time Literal represents a defined amount of time.

2.7.3.5.1 Time Literal Syntax



2.7.3.5.2 Time Literal Description

The operands in a time literal must be of data-type Number but do not need to have integer values. The value of any operand can be positive, negative, or zero. The whole time literal can have a positive, negative, or zero value; however, negative time values can have unexpected results.

Using the multiplier appropriate to its suffix, each operand in the time literal is computed as required and converted into an integer number of seconds (see Table 2-8). Rounding takes place after multiplication. The results of these conversions are then added to obtain the final time literal.

Table 2-8 — Time Literal Multipliers

Suffix	Multiplier
DAYS	86,400
HOURS	3,600
MINS	60
SECS	1

Time has a greater precision than Number. Values of data-type Time from -2,147,483,648 to +2,147,483,647 can be represented exactly. Values of data-type Number from -16,777,215 to +16,777,215 can be represented exactly. Values of data-type Number, which are of magnitude greater than 16,777,215, are approximately represented; therefore, if the arithmetic operand requires any calculation, or involves any variables of Number type, round-off can cause the precision to be less than if the same calculation were done with Time variables. All numeric literals are carried to the full precision necessary to accurately represent Time values.

Although each component of a time literal is optional, at least one component must be present.

Note that time is maintained on the APM down to tenths of milliseconds. Consequently, a comparison of two time parameters may fail—because of differences in tenths of milliseconds—even though a display of the two parameters shows identical values.

The tenths of milliseconds value for time literals defined in CL/APM is always zero.

2.7.3.5.3 Time Literal Examples

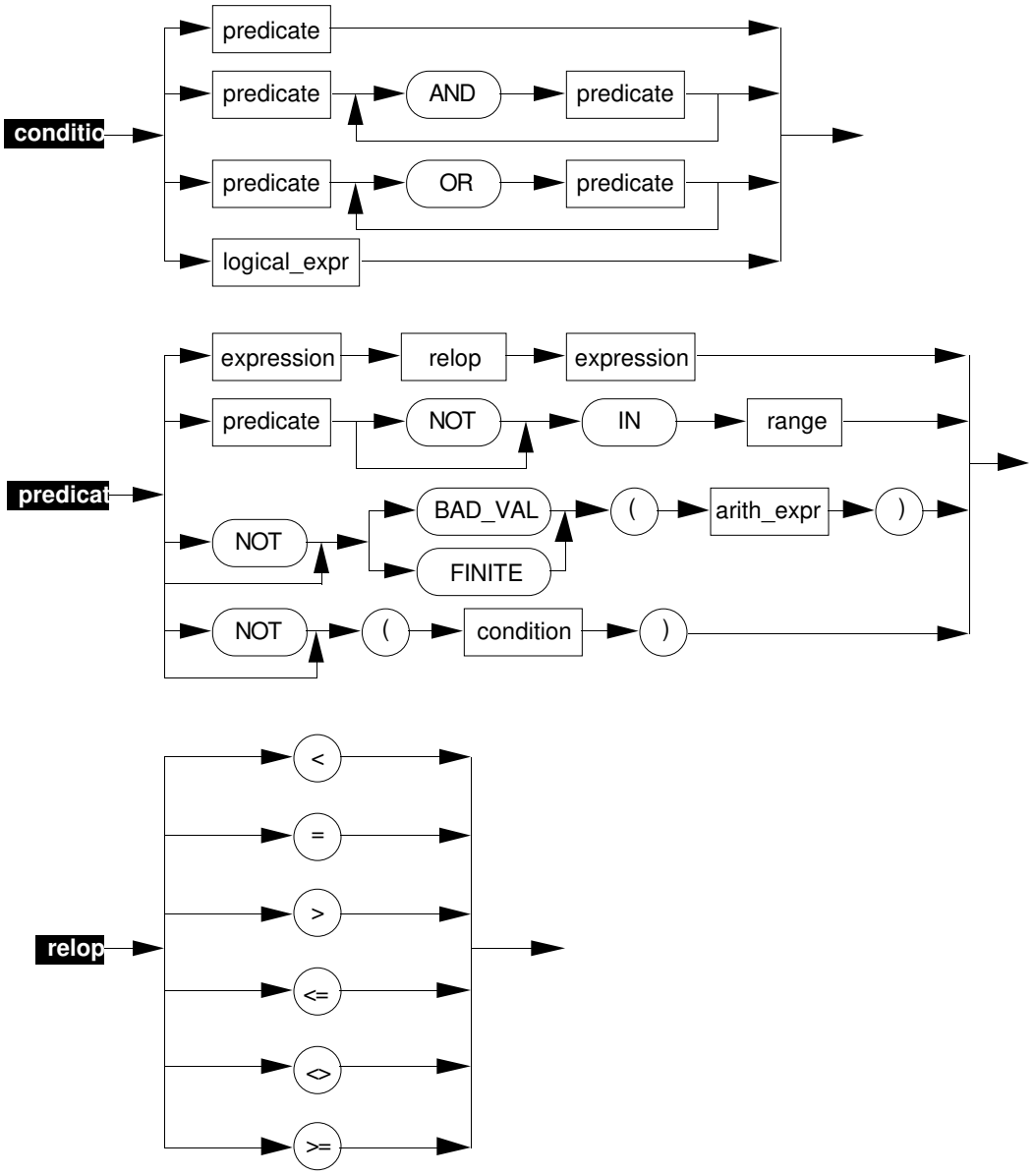
```
5 MINS                          -- five minutes
300 SECS                        -- five minutes
(1/12) HOURS                    -- five minutes
4 MINS 60 SECS                  -- five minutes
6 MINS (-60) SECS               -- five minutes
2 MINS 3 MINS                   -- NOT VALID (MINS twice)
10 SECS 5 MINS                  -- NOT VALID (out of order)
1 DAYS 6.5 HOURS                -- valid
(x * y + z) SECS                -- valid
h HOURS  m MINS  s SECS         -- valid
Max (x,y) DAYS                  -- valid
```

In the third example, (1/12) HOURS is precisely equal to 300 seconds because the round-off error was not large enough to cause problems. In the last three examples, however, round-off error may cause some inaccuracy if the evaluated arithmetic expression creates a value greater than + or - 16,777,215.

2.7.4 Conditions

A condition is a formula that expresses a truth or falsehood. It is an enhanced form of expression, used where a truth value is to be tested, as in an IF statement or in a WHEN clause. A predicate is an expression that returns a truth value. Its result cannot be stored.

2.7.4.1 Conditions Syntax



2.7.4.2 Conditions Description

A condition can be a logical expression, a relation between two expressions (including time expressions), a range test, an application of a predicate, two conditions joined by AND or OR, or any condition prefixed by NOT.

When a condition is a logical expression, it is implicitly tested for equality to On. For example,

IF x AND y AND NOT z THEN ...

is equivalent to

IF (x = On) AND (y = On) AND NOT (z = On) THEN ...

2.7.4.3 Relations Definition

The relational operators are shown in Table 2-9.

Table 2-9 — Relational Operators

Operator	Meaning
<	Less than
=	Equal to
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
<>	Not equal to

Not all relations are defined on every data type. The full set of six relations is defined for expressions of types Number and Time. Only the relations of equality and inequality are defined for discrete types and for Strings. No relations are defined for arrays taken as a whole.

2.7.4.4 Range Tests Definition

Range tests (x IN y..z, a NOT IN b..x) are defined on only Number data type, not on Enumeration types. The test is inclusive; for example, **5 IN 5..10** is true. The value of the left-most expression in the range must be less than the value in the right-most expression.

2.7.4.5 Testing for Bad Values and Finite Values

A predicate is a property of a subject that can be affirmed or denied. The built-in predicates Badval and Finite can be applied to expressions of type Number (the subject). Badval(X) returns ON if, and only if, X is a bad value.

Example:

```
IF Badval ( NN(03) * NN(02) - NN(01) + 14.4 ) THEN
```

Finite(x) returns ON if, and only if, x is Finite; that is, neither a bad value nor an infinite value.

See heading 2.3.1.1 for more information on Bad Values and heading 2.3.1.2 for information on creation and propagation of infinities.

NOTE

Badval and Finite, which can appear in the same contexts as Logical functions, are predicates returning truth values, not functions returning Logical values. Their results can be used in conditions within IF statements and WHEN clauses, but cannot be stored, compared, returned as the results of functions, used as array indices, passed to subroutines or functions, or sent by a SEND statement.

2.7.4.6 Connecting Conditions with OR and AND

The Logical operators AND and OR, but not XOR, can be used to connect conditions.

When both AND and OR are used in a compound condition, parentheses must be used to show grouping, just as when AND and OR are used as Logical operators. For example,

```
a < 5 AND b = 6 OR c > 7
```

is ambiguous and must be rewritten as one of the following:

```
(a < 5 AND b = 6) OR c > 7
a < 5 AND (b = 6 OR c > 7)
```

CL STATEMENTS
Section 3

This section describes the statements that you can use when building CL structures.

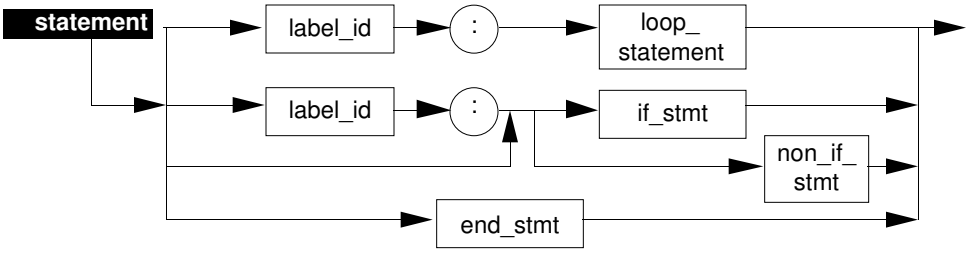
3.1 INTRODUCTION

This section describes CL statements. A statement defines a single, simple action to be performed within a CL structure. CL statements can be grouped into two major categories, according to function: PROGRAM STATEMENTS, and EMBEDDED COMPILER DIRECTIVES.

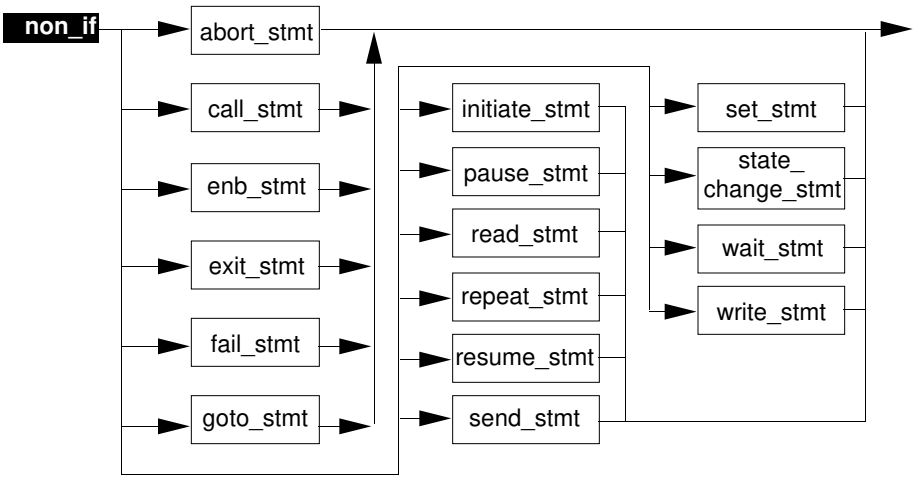
3.2 PROGRAM STATEMENTS DEFINITION

CL Program statements can be categorized as follows:

- Assignment statements, whose purpose is to change the value of one or more variables: SET, READ, WRITE, and STATE CHANGE.
- Control statements, which establish program conditions or direct the flow within a program: GOTO, IF/THEN/ELSE, LOOP/REPEAT, CALL, ENB, INITIATE, and RESUME.
- Delaying statements, which cause the program to wait for some event to occur or for a time delay: PAUSE and WAIT.
- Termination statements, which signify the termination of the program or a part of it: FAIL, EXIT, ABORT, and END.
- Communication statements, which communicate with an operator or a Computing Module: SEND.



3.2.1 Program Statements Syntax



3.2.2 Statement Labels

Labels are used as the targets of GOTO and REPEAT statements. GOTO and REPEAT are used to transfer control to another part of a program, which is identified by the label referred to in the GOTO and REPEAT statements. A label is an identifier followed by a colon. A label can precede any executable statement but cannot appear on a continuation line; therefore, a statement that is embedded within another statement cannot have a label. A LOOP statement must have a label; therefore, a LOOP statement cannot be embedded within another statement.

3.2.2.1 Statement Labels Examples

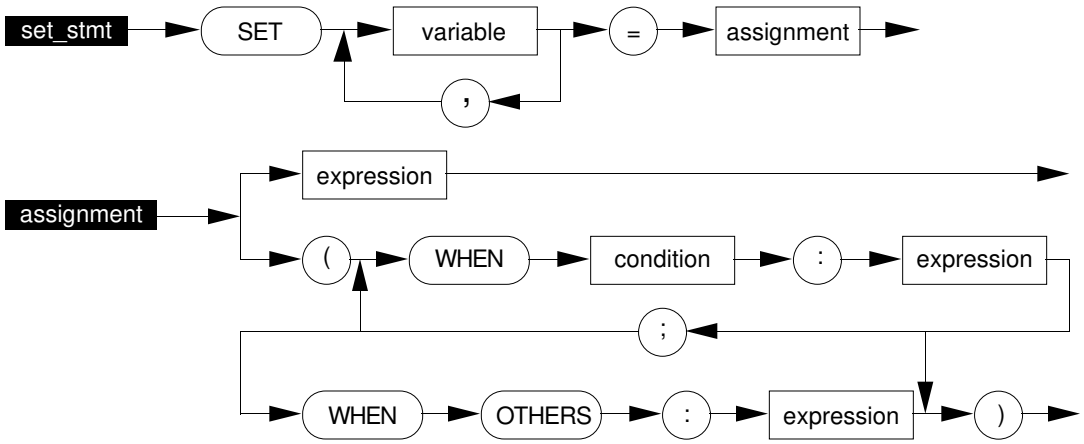
```
lab_01:           CALL test (x, y, z)
      IF x > y THEN    (SET x = z;
& badlabel:           SET z = y)      -- NOT VALID
```

In this example, **badlabel** appears on a continuation line; therefore, it is invalid.

3.2.3 SET Statement

This statement modifies the value of one or more variables, possibly depending on the result of one or more conditions.

3.2.3.1 SET Syntax



3.2.3.2 SET Description

A SET statement with a simple expression on the right-hand side of the equal sign unconditionally assigns the value of that expression to each of the variables on the left-hand side of the equal sign. The assignments are executed in reverse order from the order in which the variables are declared.

A SET statement whose right-hand side begins with **WHEN...** is called a conditional SET statement. When this statement is executed, its WHEN conditions are successively evaluated, until a true condition is found. The expression corresponding to the true condition is then evaluated and its value is assigned to the variables on the left-hand side of the equal sign. After one true condition is found, no other conditions are evaluated. Execution proceeds with the next statement. The data type of the assignment on the right-hand side must match the data type of the variable(s) on the left-hand side.

A conditional SET statement can have any number of WHEN clauses. The last WHEN clause can name the special condition OTHERS, which is always true.

A conditional SET statement that does not name WHEN OTHERS can fail; that is, none of the conditions may be true. If this occurs, it is a run time error with results as if a FAIL statement had been coded. A maximum of 16 parameters or local variables can be referenced with one SET statement.

A SET statement cannot reference parameters of points in other nodes. Use READ/WRITE statements (Section 3.2.4) if you need to reference parameters in nodes on the same UCN.

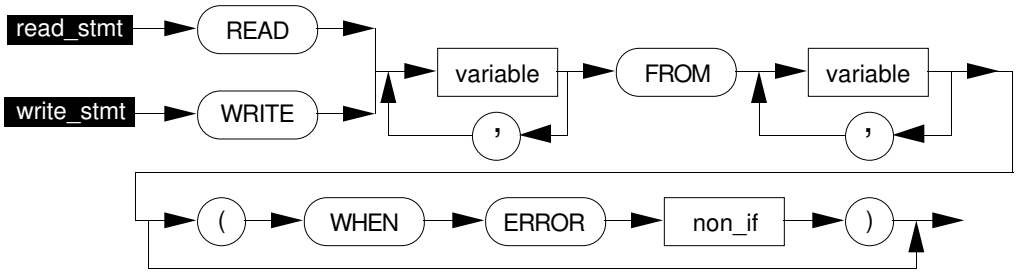
3.2.3.3 SET Examples

```
SET x = x + 1                          -- simple SET
SET x, y, z = 0                        -- multiple SET
                                       -- (first z is set to 0,
                                       -- then y is set to 0
                                       -- then x is set to zero last)
SET NN(03) = (WHEN NN(04) = 10.0 : 10.0;  -- conditional SET
&             WHEN NN(04) > 10.0 : 100.0;
&             WHEN OTHERS        : 1.0)
```

3.2.4 READ and WRITE Statements

These statements are used to access variables in other nodes on the same UCN and to test whether the access was correctly performed.

3.2.4.1 READ and WRITE Syntax



3.2.4.2 READ and WRITE Description

The READ statement reads into variables in this APM from remote variables in other nodes on the same UCN. The WRITE statement writes to remote variables in other nodes on the same UCN from variables—and/or numeric constants—in this APM. The number of items on the left-hand and right-hand sides of the FROM must be equal. These statements are preemption points.

Any local variable or APM-resident point.parameter can appear in the source of a WRITE statement, including the APM-resident box data-point parameters and the PVFL parameter of the digital input hardware-addressing name form.

I/O points cannot be the destination parameters in a READ statement (the variable or variables appearing before FROM). This restriction applies to AO, AI, DO, and DI points as well as digital input PVs. In order to transfer data from a remote node to local I/O points, you must first READ the data to local flags or numerics, then transfer the data to I/O point parameters by the SET statement.

The program is suspended until the system confirms transmission of all variables. If all are correctly transmitted, the program proceeds to the next sequential statement. If there is any communication error or Data Owner error such that any store or fetch could not be completed, the statement in the error clause (if any) is executed.

The number of variables allowed in a READ or WRITE statement is 16.

Whole array fetches and stores are not allowed in a READ or WRITE statement. If an array parameter appears in a READ or WRITE statement, its index must be a constant or a computed subscript.

3.2.4.3 READ and WRITE Examples

```
EXTERNAL REACT101           -- process module in this APM
EXTERNAL $NM01N01           -- box data point of another APM
EXTERNAL device2            -- device control point in another APM
EXTERNAL !BOX
EXTERNAL A100, B100, C100, D100, E100   -- points in another APM
EXTERNAL arraypt1           -- an Array Point in another APM
```

```
LOCAL a AT NN(01)
LOCAL b : NUMBER ARRAY (1..5) at NN(10)
LOCAL c : ARRAY (1..10) AT REACT101.NN(20)
LOCAL temp1 : LOGICAL AT !BOX.FL(18)
LOCAL temp2 : running/stopped AT FL(10)
--
READ NN(1), FL(1) FROM A100.PV, $NM10N01.FL(3)
READ a, b(3) FROM A100.PV, B100.PV
READ c(10) FROM C100.PV (WHEN ERROR GOTO labl)
READ temp1 FROM $NM01N01.FL(02)
READ a FROM arraypt1.NN(6)
WRITE E100.PV FROM temp2
WRITE D100.PV FROM !BOX.NN(3)
WRITE B100.PV FROM 27.2
--
READ c(a) FROM D100.PV
READ A100.OP FROM $NM01N01.NN(04) -- INVALID, destination parameter
                                  -- resides in an I/O point
WRITE device2.NN(1) FROM a
```

3.2.4.4 READ and WRITE Error Handling

An error on a READ or WRITE indicates that one or more of the variables could not be stored into its destination variable. It is not possible to identify which or how many variables are affected; therefore, the use of single-variable READs and WRITEs is recommended for more flexible determination and handling of error conditions.

The READ/WRITE can fail for a number of reasons. For example, a READ statement may attempt to fetch a numeric from another APM, which has failed; or, a WRITE to a setpoint of a PID will fail if the PID is not in the correct mode. There is no explicit way to tell the difference between the two possible types of errors (communication or Data Owner) after the fact; therefore, before-the-fact checks (such as reading the mode before doing a store to a setpoint for example) may be good procedure.

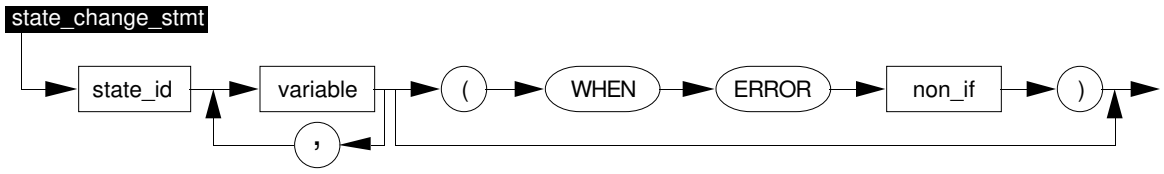
The WHEN ERROR clause specifies an error path to be executed if the READ/WRITE statement cannot be completed. All failures cause the error path to be taken. (The compiler no longer allows READ of IOL poststore parameters.)

When a READ/WRITE statement without an error path fails for other than a poststore error, the sequence fails with Failure Code F170 (Communication error detected in READ/WRITE statement).

3.2.5 STATE CHANGE Statement

This statement sets the state of the Output (OP) parameter of one or more Digital Output, Digital Composite, or Device Control data points that have two or more discrete output states, and optionally verifies that the state has been properly set. This is the only CL/APM statement that does not begin with a reserved word.

3.2.5.1 STATE CHANGE Syntax



3.2.5.2 STATE CHANGE Description

The variables must identify Digital Output, Digital Composite, or Device Control data points that have an OP parameter of a discrete type. The state IDs of the OP can be of any name, or the OP can be of type Logical, but if more than one data point is named, each OP must be of the same type.

This statement sets the data points' OP parameters equal to the named state; therefore, **close A100** is the same as **SET A100.OP = close**.

One State Change statement can reference the OP parameter of a maximum of 16 points. All data points must reside in the same node as the bound data point.

3.2.5.3 STATE CHANGE With WHEN ERROR Clause

The WHEN ERROR clause applies to only Digital Composite and Device Control points and when Command Disagree alarming is enabled. For other point types and when Command Disagree alarming is not enabled, the WHEN ERROR clause is ignored.

The WHEN ERROR clause is executed for any type of state change command failure.

The following table describes the sequence program action on various conditions of the STATE CHANGE statement:

	Command Disagree Is Configured		Command Disagree NOT Configured	
	WHEN ERROR path is coded	WHEN ERROR path not coded	WHEN ERROR path is coded	WHEN ERROR path not coded
Normal Execution	Seq. continues to next statement	Seq. continues to next statement	Seq. continues to next statement	Seq. continues to next statement
Command Disagree Timeout	WHEN ERROR path is executed	Seq. continues to next statement	Not applicable	Not applicable
Command Failure	WHEN ERROR path is executed	Sequence Fails	WHEN ERROR path is executed	Sequence fails

A STATE CHANGE statement with an error clause is a preemption point.

3.2.5.4 STATE CHANGE Examples

Suppose: Motor1's output states are start/stop;
Valve2 and Valve3's output states are open/close;
37SW's output states are low/high.
A device control point , DEV_CTL, has output states of up/down

Then,
```
stop motor1
open valve2, valve3
high 37SW (WHEN ERROR GOTO STEP retry)
down DEV_CTL
```

are valid STATE CHANGE statements.

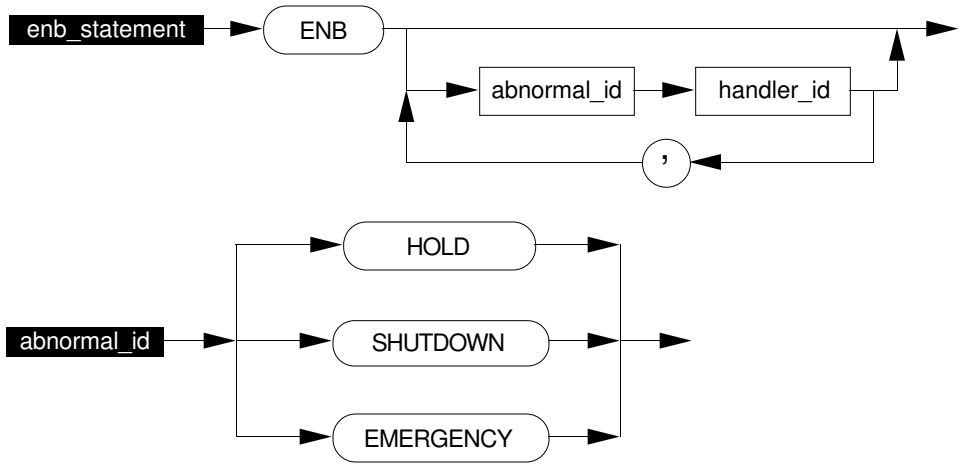
3.2.6 ENB Statement

This statement enables a new set of abnormal condition handlers, or disables all currently enabled abnormal condition handlers.

When this statement is executed, all abnormal condition handlers that are currently enabled are disabled and only the specified handlers are enabled. The same handler type (e.g., HOLD handler) cannot appear twice in the same ENB statement. If no handler names are specified, all handlers are disabled.

This statement is a preemption point.

3.2.6.1 ENB Syntax



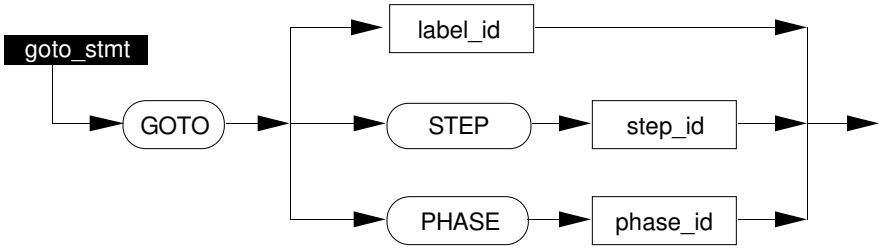
3.2.6.2 ENB Examples

```
ENB HOLD fillhold, EMERGENCY ovenstop
ENB SHUTDOWN tankck
IF !BOX.NN(3) > 55.0 THEN ENB HOLD hold1, EMERGENCY emer2
ELSE ENB HOLD hold2, EMERGENCY emer2
ENB
```

3.2.7 GOTO Statement

This statement unconditionally branches to another place in the program.

3.2.7.1 GOTO Syntax



3.2.7.2 GOTO Description

The target of a GOTO statement can be any label in the present step, the heading of any step in the current phase, or the heading of any phase in the program. A GOTO cannot be used to branch from the main body of a handler into the RESTART section, nor can it be used to branch from the RESTART section to the main body of the handler. These are detected as compile time errors.

A GOTO statement cannot be used to exit a subroutine; use EXIT instead.

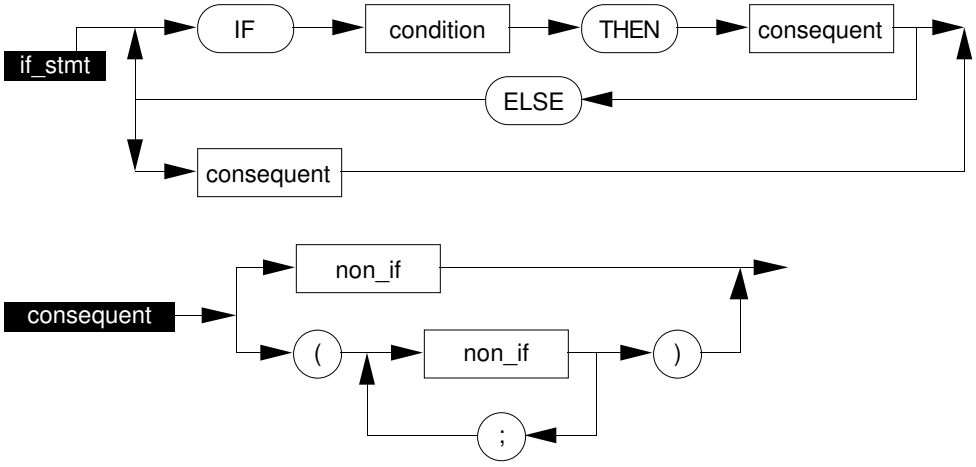
NOTE

A backward-branching GOTO causes preemption. Although a forward GOTO statement is not a preemption point, a GOTO STEP or GOTO PHASE statement always causes preemption, because STEP and PHASE headings are preemption points.

3.2.8 IF, THEN, ELSE Statement

These statements cause the conditional execution of another statement or statements.

3.2.8.1 IF, THEN, ELSE Syntax



3.2.8.2 IF, THEN, ELSE Description

Any ELSE or ELSE IF statement that does not directly follow an IF or ELSE IF statement is an error.

The **consequent** gives the statement(s) to be conditionally executed. The first form of the consequent indicates the conditional execution of a single statement; the second form indicates conditional execution of multiple statements. Consequents are considered one statement for syntax checking; therefore, if consequents are on a separate line from the IF, a continuation character is required for each line.

A sequence of IF ... ELSE IF statements is evaluated until one of the IF conditions is true. If this occurs, the consequent of the statement that has the true condition is executed. Any succeeding ELSE IF or ELSE statements in the sequence are ignored.

If none of the conditions in the IF and ELSE IF statements are true, the consequent of the ELSE statement (if any) is executed.

An IF, ELSE IF, or ELSE statement is not a preemption point of itself; however, the consequent of an IF, ELSE IF, or ELSE can contain one or more preemption points.

An IF or ELSE statement must always appear on a new line (you need an & for a THEN... that appears on a new line but not for ELSE...). It can be indented like any non-IF statement. It can never appear in the consequent of an IF or ELSE statement, in a WAIT statement's WHEN clause, or in the error clause of a READ, WRITE, state change, or INITIATE statement.

An attempt to insert comment statements between consecutive non_if statements in the consequent clause will force a compiler syntax error.

3.2.8.3 IF, THEN, ELSE Examples

```
IF 2b OR NOT 2b <> the_question THEN FAIL
IF x < y THEN SET x = y
IF x NOT IN 1..10 THEN (SEND: "range error", x;
&                  GOTO retry)

IF a > b THEN SET a = sin (theta)
ELSE IF errflag THEN (SET a = 0;
&                SET errflag = Off;
&                SET theta = 2 * pi)
ELSE SET a = cos (theta)
```

The second example above corresponds to the flowchart in Figure 3-1.

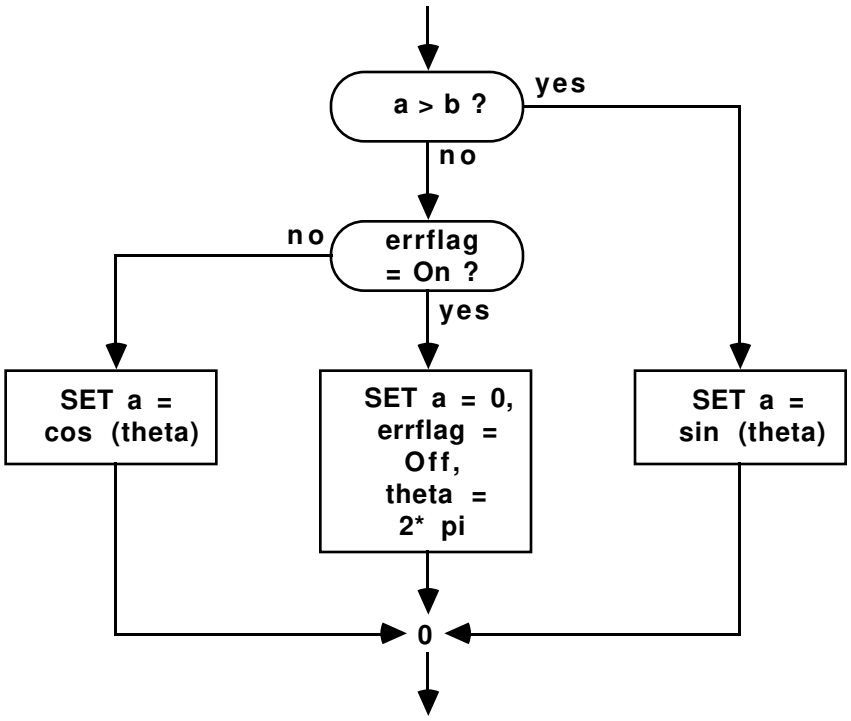
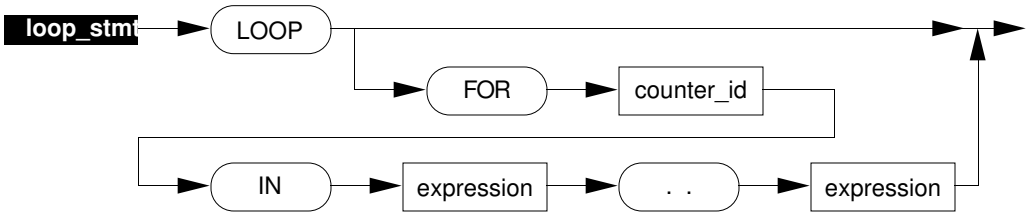


Figure 3-1 — If Statement Flowchart 1528

3.2.9 LOOP Statement

This statement provides loop control within a step, subroutine, or abnormal condition handler or in a phase without steps.

3.2.9.1 LOOP Syntax



3.2.9.2 LOOP Description

The LOOP statement's FOR clause names a counter variable. This variable must be a scalar local variable or scalar subroutine argument of type Number. The upper and lower bounds of the range are evaluated. If their values are not integers, they are rounded to the nearest integer.

The counter variable is initialized to the value of the lower bound. Each time that loop's REPEAT statement is executed, the counter variable is incremented by 1. If that value does not exceed the range's upper bound (previously computed), the loop is repeated; otherwise the loop terminates. On normal exit from the loop, the counter variable is equal to the final expression plus 1.

The upper bound of the range is dynamically re-evaluated each time the REPEAT statement is executed. Each backward branch on the REPEAT statement causes preemption. Lack of a REPEAT statement following a LOOP forces a compile-time warning.

If the LOOP statement does not contain a FOR clause, it never normally terminates. It can be exited by a GOTO or EXIT statement, or by the occurrence of an abnormal condition.

A LOOP statement must have a label.

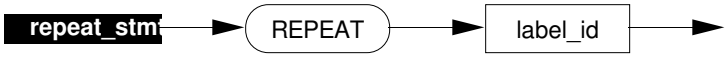
3.2.9.3 LOOP Examples

```
label: LOOP
label: LOOP FOR count IN 10..20
label: LOOP FOR count IN 20..10    -- invalid (decrement not supported)
label2: LOOP for index IN x * y .. z+3
Label3: LOOP for i IN NN(2) * 3..NN(4) - 5
```

3.2.10 REPEAT Statement

This statement causes a loop to be repeated.

3.2.10.1 REPEAT Syntax



3.2.10.2 REPEAT Description

The target of a REPEAT statement must be a label:

- in the current phase if the phase has no steps
- in the current step in a subroutine or abnormal condition handler
- in the current subroutine or abnormal condition handler if it has no steps

The label_ID must define a loop; that is, the label referred to must have a LOOP statement attached to it.

A loop can have only one REPEAT statement.

The REPEAT statement causes the loop's counter variable (if any) to be incremented by 1. If the counter variable is then less than or equal to its final value, the program branches back to the first statement in the loop. If the counter variable exceeds its final value, the branch is not taken and execution proceeds sequentially, following the REPEAT statement.

If the loop does not define a counter variable, the REPEAT statement causes an unconditional branch to the first statement in the loop. A loop need not be executed the maximum number of times. Instead, it can be exited by a GOTO or by embedding the REPEAT in a conditional statement and failing to execute it.

Loops can be nested to any depth. Whenever a loop is entered through its heading (or its beginning), its loop counter is reset, and it again begins counting towards its maximum.

NOTE

A REPEAT statement is a preemption point.

3.2.10.3 REPEAT Examples

The following example demonstrates conditional execution of a REPEAT statement.

```
 setx: LOOP FOR i IN 1..5
      SET x.SP = 2
      WAIT 30 SECS
      IF x.PV <> 2 THEN (REPEAT setx;
&                        SEND: "x.PV bad";
&                        FAIL)
```

In this example, the REPEAT statement is executed if x.PV is unequal to 2. The first five times it is executed, the loop is repeated; the program again stores into x.SP and waits thirty seconds. The sixth time, however, the REPEAT statement does not cause a reinvocation of the Repeat loop, and the SEND and FAIL statements are executed.

The following example demonstrates nested loops:

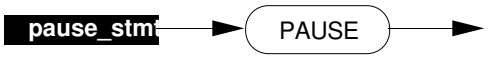
```
outer:  LOOP FOR i IN 1 .. 10
inner:  LOOP FOR j IN 1 .. i
                SET a(i*j) = -1.0
            REPEAT inner
          REPEAT outer
```

3.2.11 PAUSE Statement

If a sequence program is in semiautomatic mode, this statement causes it to pause until it is resumed by the operator. In fully automatic mode, the PAUSE statement is ignored.

This statement is a preemption point when the sequence is executed in the semiautomatic mode.

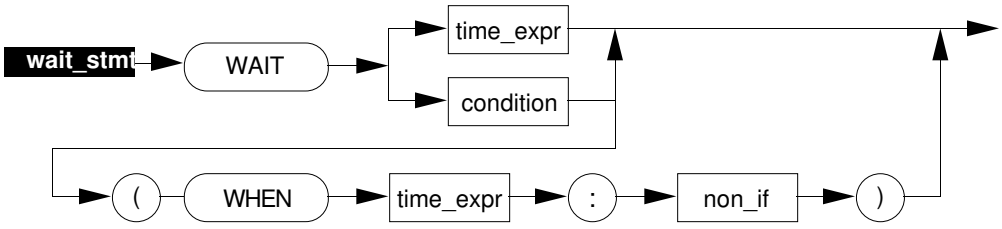
3.2.11.1 PAUSE Syntax



3.2.12 WAIT Statement

This statement causes the program to wait until some condition is fulfilled, or until a time-out occurs. This statement is a preemption point unless the time expression evaluates to zero (0).

3.2.12.1 WAIT Syntax



3.2.12.2 WAIT Description

The "WAIT time_expr" form of the WAIT statement defines a timed delay of any duration. See subsection 2.7.3 for the definition of time expressions.

For the "WAIT condition" form of the WAIT statement, the program is suspended until the named condition becomes true. If the "WAIT condition" statement contains a "WHEN time_expression" clause, and the time expires while the WAIT condition is still false, the non-if statement following the colon is executed.

To wait until a particular date, use the conditional wait (comparing two dates) as shown in the examples below. If you use a time expression that resolves to a date (for example, using the built in function Date_Time) in a WAIT, it is interpreted as a very large value which causes an indefinite delay that the operator will need to step past.

Negative time values in a WAIT statement do not cause delays; instead, the program "falls through" to the next statement.

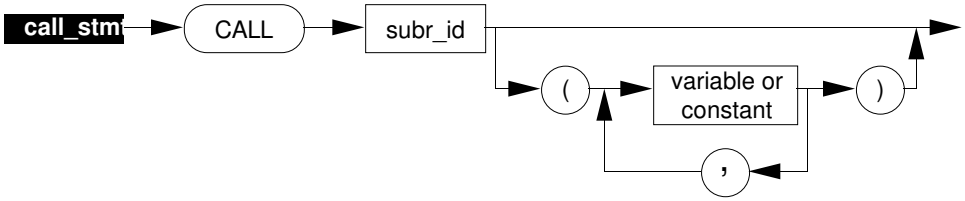
3.2.12.3 WAIT Examples

```
LOCAL elapsed_time : TIME AT APM02S03.TIME(4)  -- Time variable
LOCAL x : TIME AT pm03.TIME(2)                 -- another Time variable
LOCAL time_const = 3 MINS 10 SECS              -- Time constant
LOCAL j AT NN(3)                               -- Number variable
WAIT elapsed_time
WAIT time_const
WAIT TIME(1)
WAIT 2 HOURS j MINS 3 SECS
WAIT 1 DAYS 10 MINS
WAIT 24 HOURS                      -- wait until 24 hours from now
WAIT Now > 8 HOURS                 -- wait until 8:00 a.m.today
SET X = Date_Time + 5 DAYS
WAIT Date_Time > x                 -- wait until 5 days from now
WAIT arraypt1.TIME(2)              -- wait value from an Array Point
WAIT A100.PV > 50.0 (WHEN 5 MINS: GOTO stop)   -- wait for PV to exceed
                                               -- 50; if it takes more
                                               -- than 5 mins goto stop
```

3.2.13 CALL Statement

This statement invokes (**calls**) a subroutine.

3.2.13.1 CALL Syntax



3.2.13.2 CALL Description

The arguments of the CALL statement must match the arguments of the subroutine declaration in both data type and mode (IN, OUT, or IN OUT). A variable must appear in each place where the SUBROUTINE heading names an OUT or IN OUT argument; a variable or constant can be used in the place of an IN argument.

Arguments passed in the CALL statement to a subroutine can be only the following:

- scalar local variables
- entire array local variables and entire array parameters
- scalar parameters (Limitation: IOL parameters requiring prefetch are not permitted with the exception of DO parameters SO and INITREQ.)
- digital input PVs (PV or PVFL parameter)
- array parameters indexed by a constant
- element of a local array indexed by a constant
- numeric literals
- enumeration state identifiers
- expressions (with built-in subroutines only)

A maximum of 32 arguments can be passed to a subroutine. When an expression is passed to a built-in subroutine, it can contain a computed index; however, off-node references cannot appear either in a subscript or as an argument itself.

An entire array can be passed into a subroutine, and in turn passed to another subroutine. However, a subroutine cannot pass an individual element of an array argument to another subroutine. An example of an attempt to do this is included in the examples at heading 3.2.13.2.

A CALL statement can appear anywhere in the sequence program, including within a HANDLER or SUBROUTINE; however it is a run time error to nest subroutine calls more than one level deep. That is, subroutine A can call subroutine B, but subroutine B cannot in turn call subroutine C.

If the first step of the subroutine does not require prefetches, the subroutine call is NOT a preemption point. If the step that called the subroutine did not require prefetches, the return from the subroutine is NOT a preemption point.

3.2.13.3 CALL Examples

The following CALL examples match the subroutine header definitions at heading 4.4.1.5, Subroutine Arguments Examples.

```
EXTERNAL !DI01S02, A200      -- digital input points
EXTERNAL D300                -- regulatory PV point
EXTERNAL DOPT                -- digital output point
LOCAL num AT NN(3)
LOCAL flagarr : LOGICAL ARRAY (1..3) AT FL(10)
     ...
CALL sub1 (!DI01S02.PVFL, A200.PV)  -- digital input PVs
CALL sub2 (num, flagarr)            -- scalar local variable, entire
                                    -- array local variable
CALL sub3 (flagarr(2), 5.5, NN(19)) -- element of local array, numeric
                                    -- literal, array parameter
                                    -- indexed by a constant
CALL sub4 (D300.C)                  -- scalar parameter
CALL sub5 (flagarr(num))            -- ILLEGAL the local array is not
                                    -- indexed by a constant
CALL sub6 (DOPT.SO)                 -- scalar parameter not requiring
                                    -- prefetch
CALL sub7 (open)                    -- enumeration state identifier
```

The following example shows an improper CALL statement.

```
SUBROUTINE suba (arr1: IN LOGICAL ARRAY (1..3))
CALL subb (arr1(2))                 -- ILLEGAL—you cannot pass an
                                    -- element of an incoming array
                                    -- argument to another subroutine
END suba
```

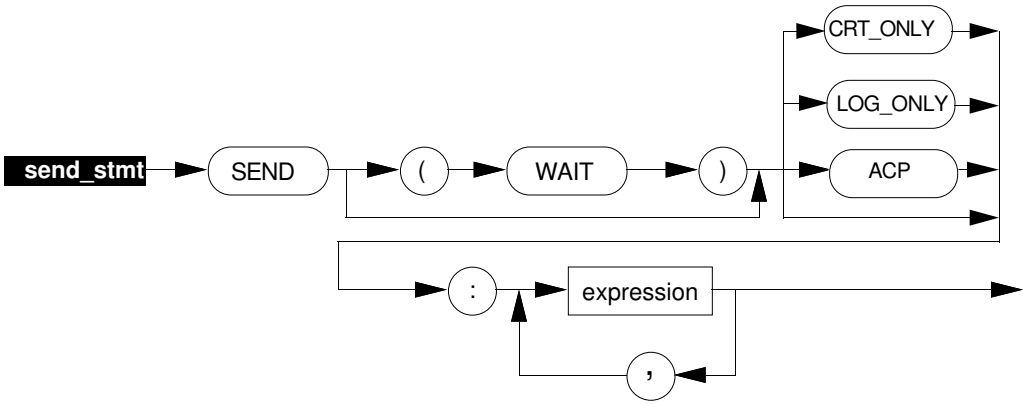
The following examples illustrate the use of expressions in CALLs to the built-in subroutine Modify_String.

```
EXTERNAL APM02S01     -- a process module data point in this node
EXTERNAL APM03S01     -- a process module data point in another node
LOCAL a AT NN(18)
LOCAL b AT NN(19)
LOCAL c AT NN(20)
               . . .
CALL Modify_String (NN(a+1), STR8(2), NN(b)+NN(c), APM02S01.NN(1),
&                   STR8(1), NN(8))      -- valid call

CALL Modify_String (NN(a+1), STR8(2), NN(b)+NN(c), NN(APM03S01.NN(1)),
&                   STR8(1), NN(8))
                    -- invalid call, off-node reference in subscript

CALL Modify_String (APM03S01.NN(1), STR8(2), NN(b)+NN(c),
&                   APM02S01.NN(1), STR8(1), NN(8))
                    -- invalid call, off-node ref. passed as argument
```

3.2.14 SEND Statement

This statement sends a message to the operator or to another data point. It can optionally wait for operator confirmation.

3.2.14.1 SEND Syntax



3.2.14.2 SEND Description

The WAIT option suspends the program until the message is confirmed by the operator. If WAIT is specified, the SEND statement is a preemption point (refer to heading 3.2.14.4). The WAIT option can be selected for only the default destination (CRT and LOG) or CRT_ONLY. Note that there is no timeout of a SEND with WAIT.

If included, the destination specified at the left of the colon is one of the special destinations CRT_ONLY, LOG_ONLY, or ACP.

- If the destination is CRT_ONLY, the message goes to the operator CRT of the bound data point's Unit.
- If the destination is LOG_ONLY, the message goes to the operator's printing (log) device.
- If the destination is omitted, the message goes to both the operator's CRT and Log device.
- If the destination is ACP, the message goes to an Advanced Control Interface Data Point (ACIDP) in a CG. (ACP is a parameter of the process module data point that is configured to specify the name of an ACIDP that is to receive sequence messages.)

The "expressions" in the SEND statement are the items to be sent. Each item can have a value of any of the types: Number, Logical, or self-defined enumeration, plus String and Time variables, constants, literals or parameters. Point names cannot be sent in the SEND statement. An array cannot be sent as a whole, but its elements can be individually sent.

All printable characters are valid in SEND strings. Note that the maximum number of characters that can be displayed on a CRT is 60, and the maximum number of characters that the LOG can print is 71. Messages longer than these limits are truncated without warning.

SENDing only a null or blank string is not allowed; a SEND that sends other items in addition to a null or blank string is allowed.

The external (human readable) format used for time values in SEND messages depends on the duration.

- When the duration is less than one day, the format is HH:MM:SS.
- When the duration is > 1 and < 1000 days, the format is DDD HH:MM:SS.
- When the duration is >= 1000 days, the format is DDMMMYY HH:MM:SS.

3.2.14.3 SEND Examples

```
SEND : "Begin cleanup"
SEND (WAIT): "Error ", A100.PV, "out of range"
SEND Log_Only: "values are", a, b, "and", c
SEND (WAIT) CRT_Only: "This is a CRT message"
SEND ACP: 1, 37, A101.PV
SEND : "Mode value is ", B2002.mode
SEND : arraypt1.STR64(1)    -- arraypt1 is an Array Point

LOCAL elapsed_time : TIME AT pm03.TIME(4)     -- Time variable
LOCAL time_const = 3 MINS 10 SECS             -- Time constant
LOCAL string1 : STRING AT STR8(3)             -- String variable
LOCAL error_message = "Error has occurred"    -- String constant
      . . .
SEND : "Elapsed time is ", elapsed_time       -- String literal and
                                              -- a Time variable
SEND : "Constant value is ", time_const       -- String literal and
                                              -- a Time constant
SEND : "Program will wait ", 15 MINS 10 SECS  -- String literal and
                                              -- a Time literal
SEND : "Time value is ", TIME(2)              -- String literal and
                                              -- a Bound Data Point
                                              -- Time parameter
SEND : "String values are ", STR8(3), STR8(4) -- String literal and
                                              -- Bound Data Point
                                              -- String parameters
SEND : string1                                -- String variable
SEND : error_message                          -- String constant
```

3.2.14.4 Preemption of SEND Statement

When an abnormal condition occurs and a handler is entered, the main sequence might have been waiting for operator response on an outstanding SEND statement with the WAIT option. The message is still available for confirmation, but the handler proceeds. The handler itself may execute a SEND with WAIT, and at that point the operator can confirm either of the outstanding SENDS (both can be confirmed, one at a time).

3.2.14.5 Event Initiated Reports from CL

Two types of Event Initiated Reports can be invoked by specially formatted CL/APM messages:

- Logs, reports, journals, and trends configured in the Area Database.
- Event History reports.

Details of message requirements are given in Section 30 of the *Engineer's Reference Manual* located in the *Implementation/Startup & Reconfiguration - 2* binder.

NOTE

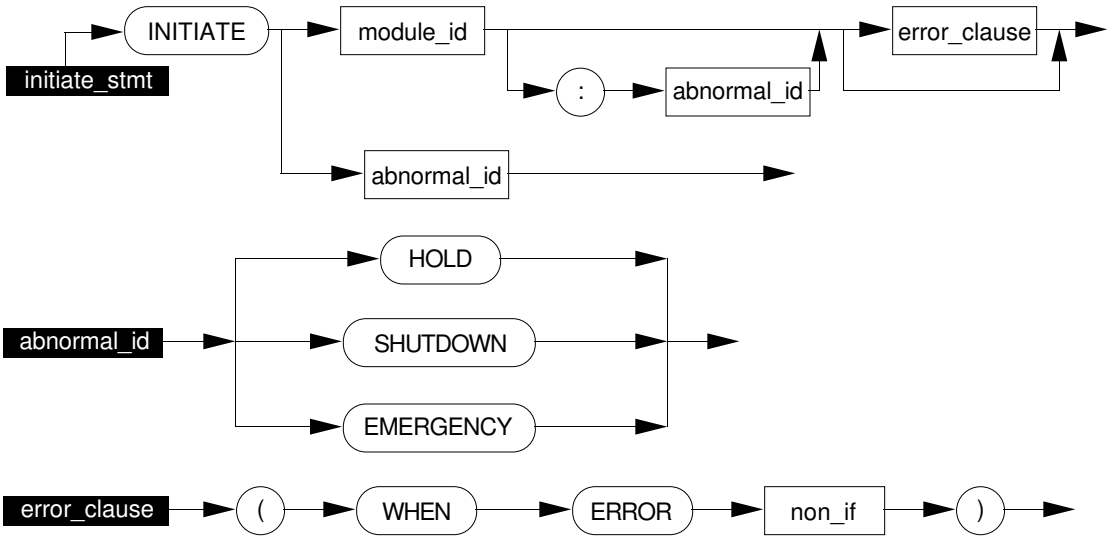
If the report name is not found in the Area Database, no error is reported.

3.2.15 INITIATE Statement

This statement causes the initiation of an abnormal sequence (an Abnormal Condition Handler) of the sequence program executing the INITIATE statement; or it can start a separate sequence program, or an abnormal sequence of a separate sequence program.

It can optionally wait for confirmation that the requested action has occurred and take action on errors.

3.2.15.1 INITIATE Syntax



3.2.15.2 INITIATE Description: Initiating Programs

`INITIATE module_id` starts the sequence program that is loaded into the process module data point identified by "module_id" by changing its PROCMOD parameter value. This form of the INITIATE statement can be used to initiate a sequence in this APM or in another APM or PM on the same UCN.

If the INITIATE statement is initiating a sequence in another APM or PM, it is a preemption point.

3.2.15.3 INITIATE Description: Initiating Abnormal Condition Handlers

"INITIATE module_id : abnormal_id" starts the named Abnormal Condition Handler of the sequence program executing in the named module, as soon as the sequence reaches its next preemption point.

"INITIATE abnormal_id" starts the named handler of the present sequence program. That handler must be higher in priority than the one executing this statement. Control is directly transferred to the head of the handler; no further statements are executed by the present sequence program. It is a run time error to initiate a handler that is not enabled.

An INITIATE statement that starts an Abnormal Condition Handler of the same sequence program, while not a preemption point itself, transfers control to a preemption point, thus causing preemption.

3.2.15.4 INITIATE Description: WHEN ERROR Clause

The INITIATE request can result in an error condition for several reasons, including the target process module is in the wrong state; the target process module is not loaded; the requested handler is not enabled; or there is a communications failure between nodes. If the INITIATE statement has an error clause and an error condition occurs, the statement in the error clause is executed; however, in the event of an error, if the INITIATE statement has no error clause the **requesting sequence** is failed.

It is a compile-time error to use a WHEN ERROR clause on an INITIATE of your own abnormal condition handler.

An INITIATE statement with a WHEN ERROR clause is a preemption point.

3.2.15.5 INITIATE Examples

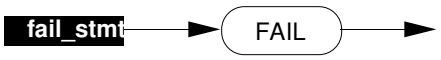
```
EXTERNAL !BOX           -- Box data point of this APM
EXTERNAL mixer, reactor -- Process module data points in this APM
EXTERNAL boiler, oven   -- Process module data points in another APM
EXTERNAL $NM02N10       -- Box data point identifier of another APM
INITIATE mixer          -- Starts another sequence in this APM
INITIATE mixer:hold    -- Starts handler of another sequence in this APM
INITIATE EMERGENCY     -- Starts handler of this sequence in this APM
INITIATE reactor (WHEN ERROR FAIL)  -- Starts another sequence in this
                                    -- APM with fail action when error
INITIATE boiler:HOLD (WHEN ERROR GOTO retry) -- Starts a handler of
                                             -- another sequence in
                                             -- another APM with goto
                                             -- action when error
INITIATE oven (WHEN ERROR INITIATE SHUTDOWN) -- Starts another sequence
                                             -- in another APM with
                                             -- handler of this sequence
                                             -- to start when error
```

3.2.16 FAIL Statement

This statement causes the program to be suspended and enter a soft failure state (error code 112). The operator is informed and can take recovery action. The recovery actions available are system-defined; they include, but are not limited to, resuming execution of the program at the next sequential statement (presumably after having made some changes).

This statement is a preemption point.

3.2.16.1 FAIL Syntax



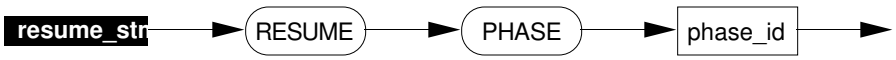
3.2.16.2 FAIL Examples

```
close switch_2 (WHEN ERROR FAIL)
IF val > 10 THEN (FAIL; GOTO recover)
```

3.2.17 RESUME Statement

This statement can be executed only by the Restart routine of an Abnormal Condition Handler. It causes resumption of the normal sequence at the beginning of a specified phase.

3.2.17.1 RESUME Syntax



3.2.17.2 RESUME Description

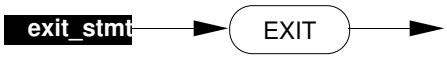
The named phase is resumed at its head.

This statement is a preemption point, because it returns to a phase.

3.2.18 EXIT Statement

When this statement is executed in a subroutine, it returns control to the subroutine's caller; when executed in a main program or Abnormal Condition Handler, it terminates the program.

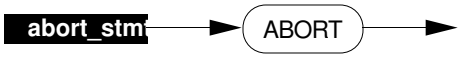
3.2.18.1 EXIT Syntax



3.2.19 ABORT Statement

This statement causes program termination. Its action is identical to the EXIT statement with the exception that, when executed in a subroutine, an ABORT statement terminates both the subroutine and the program.

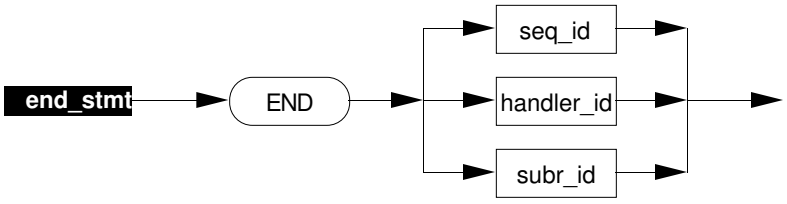
3.2.19.1 ABORT Syntax



3.2.20 END Statement

This must be the last statement of a main program, subroutine or handler. Execution of this statement is identical to the EXIT statement.

3.2.20.1 END Syntax



3.2.20.2 END Description

The END statement is counted as an executable statement even though it is not assigned a statement number on program listings.

The ID in each END statement must match the ID in the corresponding sequence or handler or subroutine heading.

3.3 EMBEDDED COMPILER DIRECTIVES

This section describes compiler directives that can be directly embedded in a CL structure.

3.3.1 Embedded Compiler Directives Syntax

All compiler directives begin with a percent sign (**%**) and must begin in the first column of a source line. Alphabetic characters can be either upper-case or lower-case.

3.3.2 %PAGE Directive

This compiler directive causes a page break when you print your CL listing. It has no further effect. Any characters between %PAGE and the end of the source line are ignored.

3.3.3 %DEBUG Directive

This compiler directive is used to indicate source lines that are subject to conditional compilation.

The Compile command option "-D" (or "-DEBUG") controls how source lines beginning with %DEBUG are to be processed. See heading 5.3.1.2 in the *Control Language/Advanced Process Manager Data Entry* manual for information on the full set of Compile command options.

- When compile option "-D" is specified, each source line that begins with %DEBUG is treated as an ordinary source line with the %DEBUG stripped off.
- When compile option "-D" is **not** specified, any source line that begins with %DEBUG is ignored.

3.3.3.1 %DEBUG Example

```
   SET x.PV = 4
%DEBUG SEND: "x.PV has been set"
```

In this example, the SEND statement is executed only if the compiler command option is selected when the program is compiled.

3.3.4 %INCLUDE_EQUIPMENT_LIST Directive

This compiler directive is used to indicate a single Equipment List object file to be read during compilation of this program. The %INCLUDE_EQUIPMENT_LIST directive must be the first line in the CL Source Program, excluding comment lines. The syntax for this directive is:

```
%INCLUDE_EQUIPMENT_LIST el_object_pathname
```

Where "el_object_pathname" represents the name of the Equipment List object file. This can be either the full pathname or the object file name only (if the object name only is specified, the user default path is used). The file name requires a .QO suffix. The device name and volume portion of the pathname specified here can be overwritten by use of the "-OEP" command when compiling the program.

Note that separate object files are generated by the compiler for each unit instance found within the equipment list.

3.3.5 %INCLUDE_SOURCE Directive

The %INCLUDE_SOURCE compiler directive allows you to include information from another ASCII text file in a CL source file. Conceptually, the statements from the included file replace the %INCLUDE_SOURCE directive. The %INCLUDE_SOURCE directive names a single filename or file pathname. Include Source files must be named with a .CL suffix; however, the .CL is optional when naming the file in the directive. If only the filename is used in the directive, the CL SOURCE/OBJECT path is used to locate the file. Include Source files can be located anywhere on the local LCN.

Include Source files can contain CL statements, data declarations, compiler directives, and comments. In addition, an Include Source file can contain an unlimited number of other Include Source directives. Nesting is limited to five levels deep.

The CL Compiler checks date/time stamps of the main source file and each Include Source file to ensure that none of these files change during a compilation. If any file changes, the compile is deemed invalid, an error message is generated, and the compilation is terminated.

The listings contain a FILE column if a %INCLUDE_SOURCE directive is present in the main source. Each Include Source directive is assigned a unique number. The FILE column displays that unique number next to each line of that particular Include Source file. The main CL source file is not assigned a file number.

The COMPILATION RESULTS section (located before the cross-reference section) shows the full pathname for the main source and each Include Source file.

The COMPILATION RESULTS section also shows the total amount of heap memory used. This indicates the total amount of memory that was required to compile the CL program.

If the %INCLUDE_EQUIPMENT_LIST directive is used, it must be the first noncomment statement in the file. The %INCLUDE_SOURCE directive may appear anywhere afterward.

3.3.5.1 %INCLUDE_SOURCE CL/APM Source Example

```
SEQUENCE testapm(apm;point apm01)
%INCLUDE_SOURCE phase1
%INCLUDE_SOURCE phase2
%INCLUDE_SOURCE step3
END testapm
%INCLUDE_SOURCE pmsubrtn
```

3.3.5.2 %INCLUDE_SOURCE CL/APM Listings Example

```
CL V41.00   TESTAPM           12/10/91 10:17:37:1962  Page  1

File Line Loc Text

       1       SEQUENCE testapm(apm;point apm01)
       2       %INCLUDE_SOURCE phase1
  1    3       PHASE one
             ^
**NOTE  **  All enabled abnormal handlers are disabled

  1    4       STEP one
  1    5    1  SEND:"hi"
  1    6       STEP two
  1    7    1  SEND:"bye"
       8       %INCLUDE_SOURCE phase2
  2    9       PHASE two
             ^
**NOTE  **  All enabled abnormal handlers are disabled

  2   10       STEP one
  2   11    1  SEND:"phase two, step one"
  2   12       STEP two
  2   13    1  SEND:"phase two, step two"
      14       %INCLUDE_SOURCE step3
  3   15       STEP three
  3   16    1  CALL pmsubrtn
      17       END testapm
      18       %INCLUDE_SOURCE pmsubrtn
  4   19       SUBROUTINE PMSubrtn
  4   20    1  SEND:"PMSubrtn"
  4   21       END PMSubrtn

*******No errors detected
```

```
-------COMPILATION RESULTS-------

***     3 BLOCKs of object code out of a maximum of  392 blocks

New APM object file:
     File A0701966.NO created

Compilation was on a UP.  Memory limit in words   =  320,000
Words of heap memory used:  25,599

Options Selected:  -NoXRef -UpdateLib

The following source file(s) were referenced:

File          File Path
-----    ----------------------
         NET>DO>TESTAPM.CL (Main Source File)
   1     NET>DO>PHASE1.CL
   2     NET>DO>PHASE2.CL
   3     NET>DO>STEP3.CL
   4     NET>DO>PMSUBRTN.CL
```

CL/APM STRUCTURES
Section 4

This section describes the structure and components of a Sequence Program written in CL/APM. It also describes data addressing options and explains the available built-in functions and other support features .

4.1 SEQUENCE PROGRAM DEFINITION

A Sequence Program is a set of instructions that details a complete sequence of events in the production of some product. After compiling without errors, a sequence program is loaded into a process module, which is a named data point in the Advanced Process Manager. Sequence programs are loaded from the Process Module Detail Display using the US Operator's Personality. Heading 3.2.2 in the *Control Language/Advanced Process Manager Data Entry* manual specifies which volume your program's object file (.NO file) must be in, so that loading the program to an APM can be carried out successfully. Refer also to the following headings in the *Control Language/Advanced Process Manager Data Entry* manual: 2.3 (Table 2-1), 2.4, and all of heading 3.2, for information on the relationship of sequence programs, process modules and user volumes.

Sequence programs are constructed by combining statements to form steps, combining steps to form phases, then combining phases to form the sequence of tasks you want to perform. Subroutines can exist within sequence programs.

Abnormal Condition Handlers are CL/APM structures that can be used to take control from the sequence program if a certain "abnormal condition" occurs, usually some sort of process upset.

A sequence program's execution can be interleaved with that of other sequence programs running in the same APM.

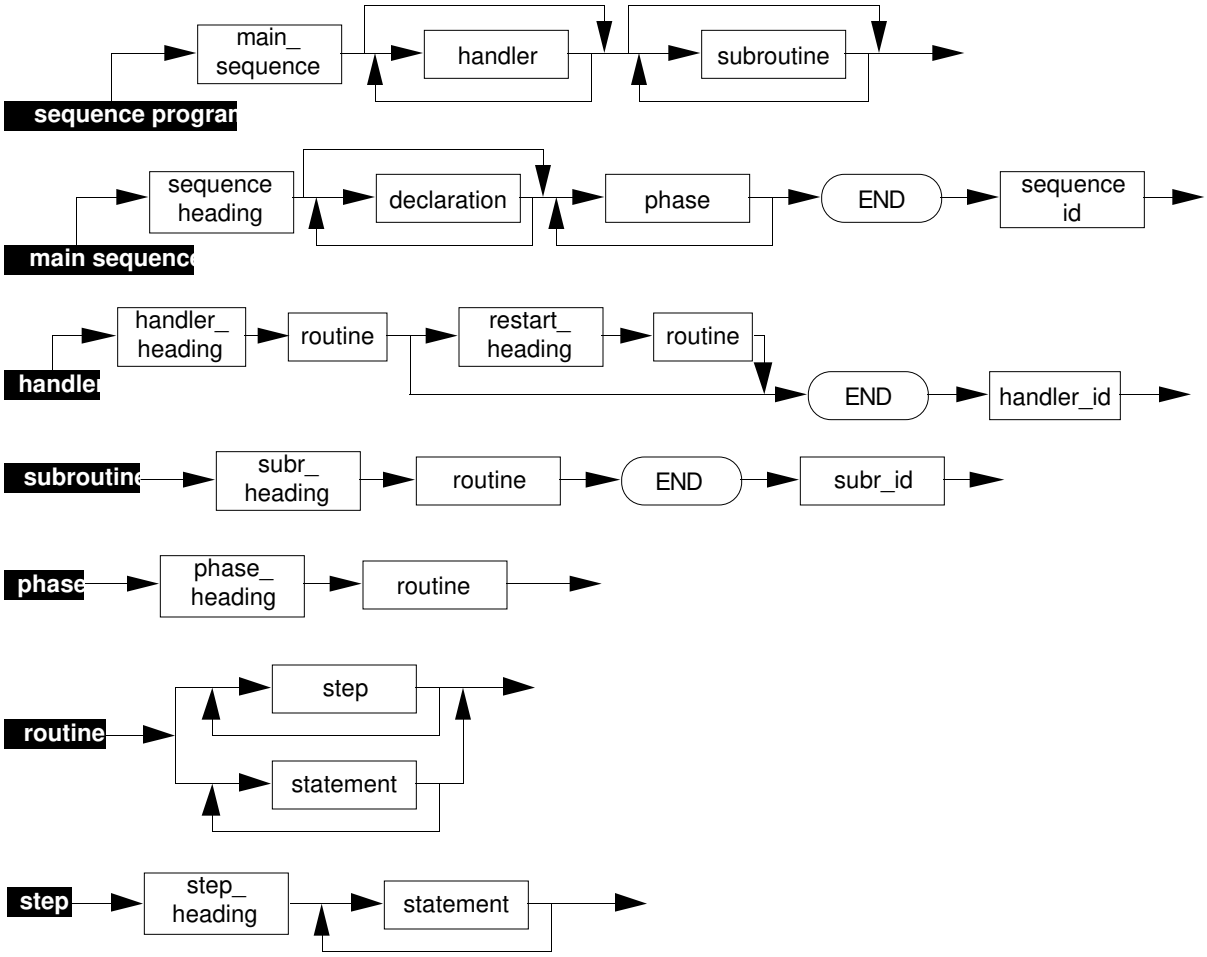
The suspension of a sequence program to allow another program to run is called **preemption**.

A sequence program can be preempted only at specific points, which are as follows:

- Any statement that causes a delay (individually described in Chapter 3)
- Any backward GOTO or REPEAT statement
- The crossing of any STEP, PHASE, or RESTART heading

The statements between two adjacent preemption points are guaranteed to execute without being interrupted by other sequence programs. You can use this guarantee to avoid the need for synchronization on variables shared between separate sequence programs.

4.1.1 Sequence Program Syntax



4.1.2 Sequence Program Description

A sequence program consists of a main sequence, optionally followed by abnormal condition handlers and subroutines.

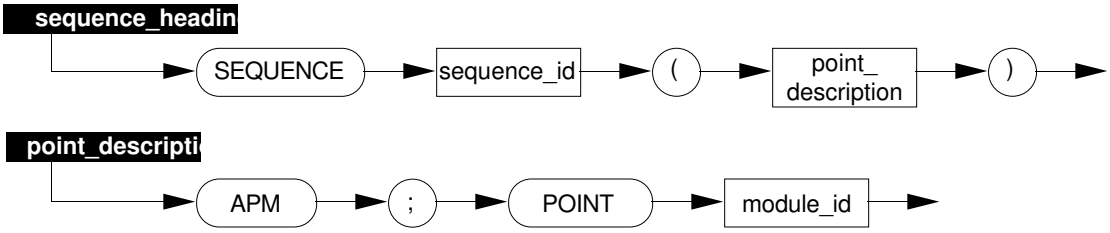
The main sequence is made up of phases; a phase or subroutine consists of one routine. An Abnormal Condition Handler consists of one or two routines, the second (if present) being its Restart routine. Routines (phases, handlers, and subroutines) are made up of steps; however, if a routine consists of only one step, the STEP heading can be omitted.

The sequence ID (handler ID, subroutine ID) in each END statement must match that in its respective heading.

4.1.3 SEQUENCE Heading

The SEQUENCE heading identifies the sequence program, the program destination, and the bound data point.

4.1.3.1 SEQUENCE-Heading Syntax



4.1.3.2 SEQUENCE-Heading Description

The sequence ID is an identifier by which the program is to be externally known. This ID is displayed on the Process Module Detail Display when the sequence is loaded into the module.

The point description identifies that this sequence is destined for an Advanced Process Manager, not a Process Manager or a Multifunction Controller. "Module_ID" is the name of the Process Module Data Point where the sequence program is loaded.

4.1.3.3 SEQUENCE-Heading Example

```
SEQUENCE fred (APM; POINT REACT101)
```

4.1.4 PHASE Heading

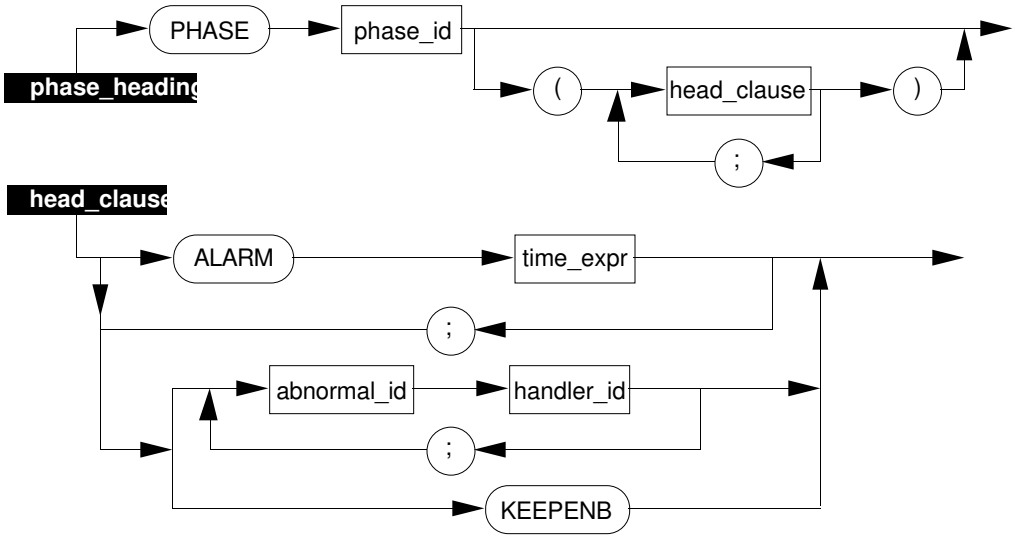
A phase is a major process milestone. Phase boundaries are key synchronization points in the sequence program. The PHASE heading identifies the beginning of a new phase and sets up the operating conditions for the execution of its routines.

During a phase, a check is made at each preemption point for the occurrence of abnormal conditions. When such a condition occurs, all activities cease and an abnormal condition handler is activated.

A phase-alarm timer, if active, checks the total execution time of a phase.

At the end of a phase, all activities, including abnormal condition detection, cease. The phase-alarm timer is stopped. The sequence program is momentarily quiescent, and a new phase can begin. When the heading of the next phase is crossed, a phase change is marked for the sequence. This can be displayed by the Universal Station. In the new phase heading, the phase-alarm timer can be restarted at a newly specified setting. Existing enabled abnormal condition handlers can be retained, or a new set of abnormal condition handlers can be activated, including a null set (no handlers enabled).

4.1.4.1 PHASE-Heading Syntax



4.1.4.2 PHASE-Heading Description

The ALARM-heading clause sets the phase-alarm timer to the value of the time expression. If this clause is omitted, the phase-alarm timer is not started for this phase.

NOTE

The time expression in the ALARM Clause cannot contain a variable requiring prefetch. That is, parameters of Analog Input, Analog Output, Digital Input (except for DI PVs), and Digital Output points cannot be used in the ALARM Clause. The ALARM Clause also cannot contain a reference to an off-node parameter.

If the ALARM time is a TIME variable (e.g., TIME(3)), the value of the time variable is snapshot at the phase boundary. That value then is used for time count down during the phase. Therefore, changing the value of the time variable during the phase execution has no effect on the phase time alarm.

The HOLD-heading, SHUTDOWN-heading, and EMERGENCY-heading clauses enable handlers for their respective abnormal conditions. Activation conditions for these handlers are defined in the handlers' own headings. The same Abnormal Condition Handler-heading clause cannot appear twice in one PHASE heading. If a PHASE heading contains no clause that activates a given handler, that condition is disabled.

The KEEPENB clause specifies that the current set of enabled handlers should not be disabled and should be retained (see heading 4.2.1.2). A PHASE heading can contain either the KEEPENB clause or named handlers, but not both.

Note that based on program flow, you cannot be sure which handlers are enabled when a PHASE heading is encountered. If you need to be certain, name the required handlers in the PHASE heading.

If no handlers and no KEEPENB clause are specified in the PHASE heading, all handlers are disabled. A compiler note is issued for this condition.

A PHASE heading not followed by a STEP heading is a preemption point.

4.1.4.3 PHASE-Heading Examples

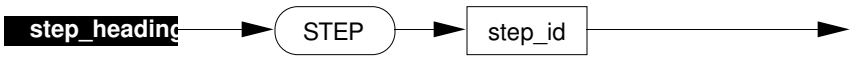
```
PHASE mix_up (ALARM 2 MINS)
PHASE fill_up (ALARM TIME(3); SHUTDOWN cleanup)
PHASE hold_up (ALARM NN(17) SECS; HOLD hold1)
PHASE shut_up (ALARM APM02S03.NN(02) SECS; KEEPENB)
```

4.1.5 STEP Heading

A step is a named minor milestone of the process, that consists of one or more statement groups separated by a STEP heading. As a process milestone, a step is recognized and displayed at the Universal Station.

The STEP heading names a step.

4.1.5.1 STEP-Heading Syntax



4.1.5.2 STEP-Heading Description

The STEP heading is a preemption point.

4.1.5.3 STEP-Heading Examples

```
STEP s1
STEP fill_a
```

4.2 ABNORMAL CONDITION HANDLERS DEFINITION

When a sequence program is not handling an abnormal condition, it is said to be in its **normal sequence**; otherwise, it is in an **abnormal sequence**. Refer to Section 3 for complete information on any of the statements mentioned in this discussion (e.g., SEND).

The abnormal-condition identifiers are HOLD, SHUTDOWN, and EMERGENCY. These are reserved words.

Abnormal Condition Handlers have priority over each other and over normal sequences. These priorities are defined as follows:

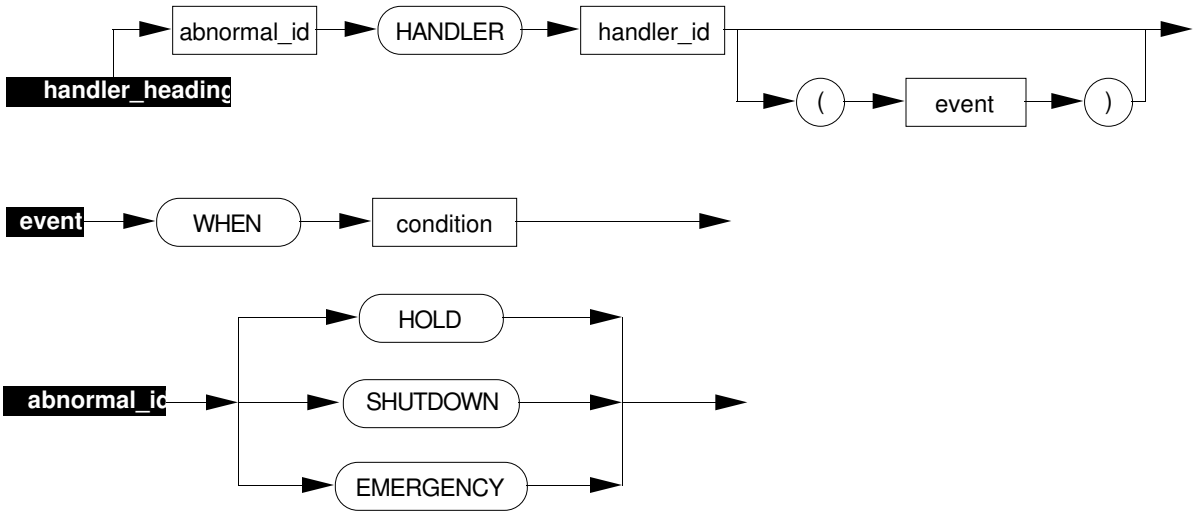
EMERGENCY	Highest
SHUTDOWN	|
HOLD	|
Normal sequence	Lowest

If the operator, or any INITIATE statement in the same or another program, should attempt to start an Abnormal Condition Handler that is not enabled, the Runtime Error E107 (KEYLEVEL ERROR) is generated.

4.2.1 HANDLER Heading

The HANDLER heading specifies the name of the handler, the abnormal condition it handles, and its starting condition (if any).

4.2.1.1 HANDLER-Heading Syntax



4.2.1.2 HANDLER-Heading Description

The handler_ID is the identifier by which the handler is to be known. The abnormal_ID (HOLD, SHUTDOWN, or EMERGENCY) defines the condition to be handled.

An Abnormal Condition Handler must be enabled by a PHASE heading or the ENB statement. The set of enabled handlers is re-evaluated at each PHASE statement; they can be retained (if a KEEPENB clause is found in the PHASE heading), changed (a new set of handlers is named in the PHASE heading), or disabled (no KEEPENB or set of handlers are named in the PHASE heading). Handlers also are re-evaluated at the ENB statement. A handler cannot start until it has been enabled.

Once enabled, a handler can be started in one of three ways:

- by operator action
- by execution of an INITIATE statement in any sequence program (including its own)
- by the occurrence of the event (if any) named in its heading.

The WHEN condition is tested at each preemption point. It also is tested continuously when SEQEXEC is equal to PAUSE, FAIL, ERROR, or END. A PHASE header first enables handlers and then checks for the condition in the WHEN clause; therefore, you must have a preemption point between the condition that you are checking for in the WHEN clause and the next PHASE header. Then, if it is true, the "WHEN" event is deemed to have occurred and the handler begins execution. The following example shows a preemption point "STEP S1", which is between the condition being set to true ("SET X = On") and the next PHASE header ("PHASE P3").

```
SEQUENCE  s  (APM; POINT REACT101)
   LOCAL  x: LOGICAL AT FL(1)
PHASE p1
   SET x = Off
PHASE p2 (HOLD h)
   STEP s0
    SET x = On
   STEP s1
      GOTO PHASE p3
PHASE p3
   SET x = Off
END s
HOLD  HANDLER h (WHEN x = On)
   SEND:  "begin hold handler"
RESTART
   RESUME PHASE p3
END h
```

A handler without a given event can be started by only operator or program action.

Once in an abnormal condition handler, the run time behaves as if that handler and all lower priority handlers are disabled. This is because the condition that triggered the handler may still be true, which if the current handler was still enabled, would re-trigger the handler, causing a loop.

Normally, the highest priority handler that is enabled is executed, and an attempt to INITIATE a lower priority handler, while in a higher priority handler, generates a Failure 173—Store fail due to rights error.

However, situations can occur that transfer control from a higher priority handler to a handler of lower priority. In the following example, control is transferred from the higher priority EMERGENCY handler to the lower priority SHUTDOWN handler. Both handlers are disabled when entering the EMERGENCY handler, but then an ENB statement enables only the SHUTDOWN handler which takes control when its heading event becomes true. Care must be taken to avoid this situation.

```
SEQUENCE example (APM; POINT REACT101)
      LOCAL a AT NN(1)

PHASE one (SHUTDOWN shut1; EMERGENCY emer1)
      SET a = 2         -- emergency handler event becomes true
      WAIT 1 SECS       -- sequence goes to emergency handler
      SEND : "won't get here in normal sequence"
END example

SHUTDOWN HANDLER shut1 (WHEN a = 3)
      SEND : "in shut1"
END shut1

EMERGENCY HANDLER emer1 (WHEN a = 2)
                        -- at this point, both shutdown and
                        -- emergency handlers are disabled
      SEND : "in emer1"
      ENB SHUTDOWN shut1 -- shutdown handler is enabled, but
                        -- the emergency handler is not enabled
      SET a = 3         -- shutdown handler event becomes true
      WAIT 1 SECS       -- sequence goes to the shutdown handler
                        -- from the emergency handler
      SEND : "won't get here in emer1"
END emer1
```

When you exit an abnormal condition handler with a RESUME statement, the set of handlers that will be enabled will depend on the PHASE header statement and whether or not any handlers were enabled within the abnormal condition handler. The following is a general set of rules to help determine which handlers will be in effect:

- If you resume to a phase that has a KEEPENB in its header, the set of handlers that were enabled before the resume will remain in effect. This would include handlers that were enabled prior to entering the abnormal handler and those abnormal handlers that were enabled within the abnormal handler.
- If you resume to a phase with handlers in its header, then those handlers will be enabled. Even if you enabled other handlers prior to the resume to the phase, those handlers in the phase header will be enabled and all previously enabled handlers will be disabled.
- If you resume to a phase that does not have any handlers in its header, then no handlers will be enabled. Any previously enabled handlers will be disabled upon resuming to the phase.

Example:

```
SEQUENCE seq1  (APM; POINT REACT101)
   EXTERNAL  !BOX

PHASE zero
   SET NN(1) = 1             -- Initialize numeric to one so that the
                             -- first time we enter the shutdown
                             -- handler, the first resume condition
                             -- is executed

PHASE one   (SHUTDOWN shut1 ; EMERGENCY emer1)
   STEP sone
      SET !BOX.FL(15) = OFF  -- Set the condition that causes the
                             -- shut1 shutdown handler to be invoked
      WAIT 2 SECS            -- Cause a preemption point

PHASE two (KEEPENB)          -- When entering this phase, all
                             -- previously enabled handlers will
                             -- remain in effect
   PAUSE
   GOTO PHASE one
      ....
PHASE three (HOLD hold1)     -- When entering this phase, all
                             -- previously enabled handlers will be
                             -- disabled and the HOLD hold1 handler
                             -- will be the only handler enabled
   PAUSE
   GOTO PHASE one

PHASE four                   -- When entering this phase, all
                             -- previously enabled handlers will be
                             -- disabled and no handlers will be
                             -- enabled
   PAUSE
   GOTO PHASE one
```

```
PHASE five (HOLD hold1)      -- When entering this phase, all
                             -- previously enabled handlers will be
                             -- disabled and the HOLD hold1 handler
                             -- will be the only handler enabled
   PAUSE
   GOTO PHASE one

PHASE six (KEEPENB)          -- When entering this phase, the
                             -- previously enabled handlers will
                             -- remain in effect
   PAUSE
   GOTO PHASE one

END seq1

HOLD HANDLER hold1
   SEND:"In Hold Handler"
END hold1

EMERGENCY HANDLER emer1
   SEND:"In Emergency Handler"
END emer1

SHUTDOWN HANDLER shut1 (WHEN !BOX.FL(15) = OFF)
   SET !BOX.FL(15) = ON
   SET NN(1) = NN(1) + 1
   RESTART
   IF NN(1) = 2 THEN RESUME PHASE two
   IF NN(1) = 3 THEN RESUME PHASE three
   IF NN(1) = 4 THEN RESUME PHASE four
   IF NN(1) = 5 THEN (ENB EMERGENCY emer1; RESUME PHASE five)
   IF NN(1) = 6 THEN (ENB HOLD hold1; RESUME PHASE six)
END shut1
```

In the above example, phase one enables the handlers shut1 and emer1. The shut1 handler condition is set to OFF which causes the SHUTDOWN handler to be invoked. The first time into the SHUTDOWN handler, the box numeric is incremented to 2 which causes a resume to phase two. Phase two has a KEEPENB header which leaves the previously enabled shut1 and emer1 handlers in effect. Phase two returns to phase one to go through the scenario again.

This time when phase one causes the shut1 handler to be invoked, the box numeric is incremented to 3. This causes a resume to phase three. Phase three has a HOLD hold1 header. This forces all previously enabled handlers to be disabled and the HOLD hold1 handler to be enabled. Phase three returns to phase one to go through the scenario again.

When phase one causes the shut1 handler to be invoked, the box numeric is incremented to 4. This causes a resume to phase four. Phase four does not have any handlers in its header. This causes all previously enabled handlers to be disabled and no handlers will be enabled. Phase four returns to phase one to go through the scenario again.

When phase one causes the shut1 handler to be invoked, the box numeric is incremented to 5. This causes the emer1 EMERGENCY handler to be enabled and then a resume to phase five. Phase five has a HOLD hold1 header which causes the emer1 handler to be disabled and the HOLD hold1 handler to be enabled. Phase five returns to phase one to go through the scenario again.

When phase one causes the shut1 handler to be invoked, the box numeric is incremented to 6. This causes the HOLD hold1 handler to be enabled and then a resume to phase six. Phase six has a KEEPENB header which leaves the previously enabled HOLD hold1 handler enabled. Phase six returns back to phase one and the program runs to its end.

4.2.1.3 HANDLER-Heading Examples

```
HOLD HANDLER cooldown (WHEN Temp.PV > Temp.PVHITP)
SHUTDOWN HANDLER shut1 (WHEN (flow = OFF) or (pressure > 35.4) )
EMERGENCY HANDLER emerg1
```

4.3 RESTART ROUTINES DEFINITION

HOLD and SHUTDOWN Condition Handlers can be terminated by a Restart routine, separated by a RESTART heading. The Restart Routine is intended to contain statements that prepare for re-entry into the normal sequence. The optional Resume statement specifies the phase label where normal execution will begin. The EMERGENCY handler cannot have a RESTART routine.

A Restart routine has the same priority as the Abnormal Condition Handler to which it is attached. The Module Operating Status on the Process Module Detail Display DOES NOT change as the Restart routine is entered. In addition, because the restart section is not functionally separate from its parent handler, the detail display does not show that a restart handler has been enabled when the sequence is executing the restart section; therefore, you should inform the operator that the program is executing a restart, using either a descriptive STEP label in the restart section, or SENDing a message.

The RESUME statement is optional and can appear anywhere inside a Restart routine. If you do not include a RESUME statement, execution of the sequence terminates at the end of the Restart routine.

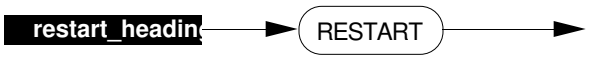
No operator intervention is required to confirm execution of a RESTART routine or a RESUME statement. Step labels cannot be duplicated between an Abnormal Condition Handler and its Restart routine.

A GOTO that tries to branch across a restart keyword in either direction is a compile-time error.

4.3.1 RESTART Heading

The crossing of a RESTART heading causes the handler to enter its Restart routine.

4.3.1.1 RESTART-Heading Syntax



4.3.1.2 RESTART-Heading Description

The Restart heading is a preemption point.

4.4 USER-WRITTEN SUBROUTINES

This section describes user-written CL subroutines. Subroutines can be called by sequence programs, by abnormal condition handlers, or by other subroutines (limited to one level of nesting).

Subroutines can be either user-written or system-supplied; user-written subroutines must use the facility described here, but system-supplied (by Honeywell) subroutines need not be written in CL. User-written subroutines must be compiled together with the sequence program.

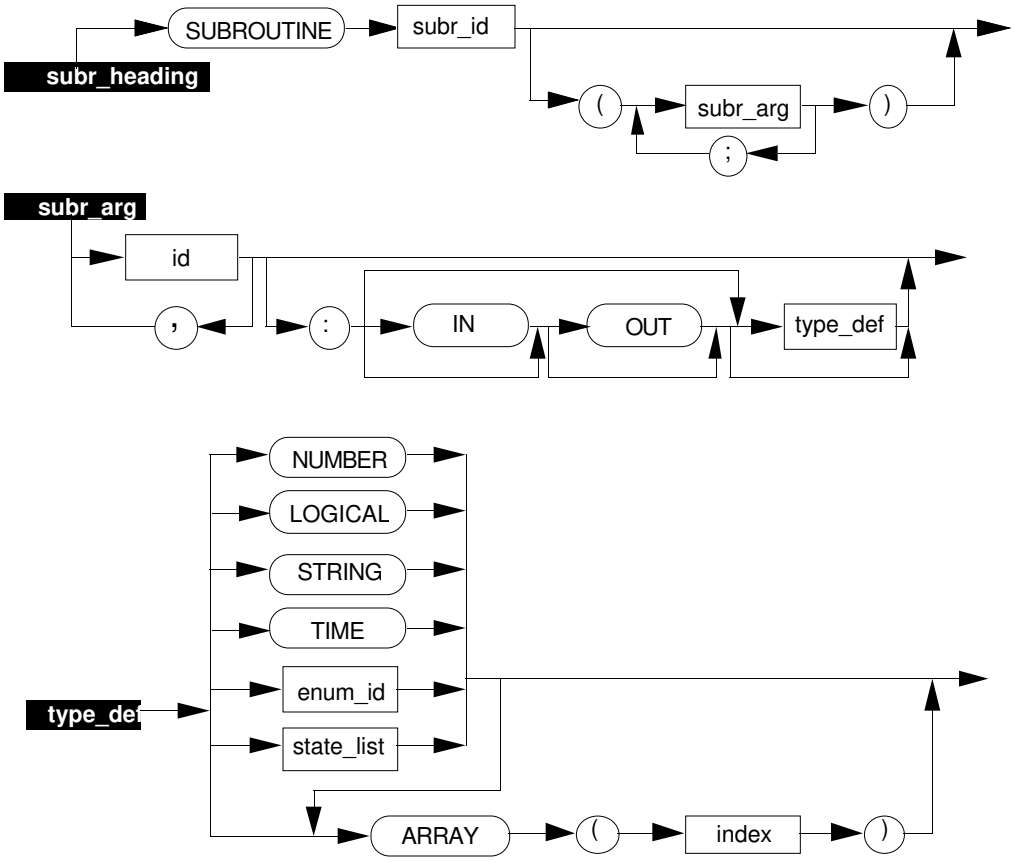
If the first step of the subroutine does not require prefetches, the subroutine call is NOT a preemption point. If the step that called the subroutine did not require prefetches, the return from the subroutine is NOT a preemption point.

Subroutine syntax is shown at Paragraph 4.1.1.

4.4.1 SUBROUTINE Heading

The SUBROUTINE heading identifies it and specifies its arguments, including their type and access mode.

4.4.1.1 SUBROUTINE-Heading Syntax



4.4.1.2 SUBROUTINE-Heading Description

Each argument has an access mode: IN, OUT, or IN OUT. An argument's mode determines whether the subroutine can access that argument, set it, or both. IN arguments can only be accessed. OUT arguments can only be set. IN OUT arguments can be both accessed and set.

> **NOTE**
>
> At present, the compiler does not prevent the READ of OUT-only arguments.

If an argument's mode is omitted, the default is IN. The default data type for all arguments is Number.

4.4.1.3 SUBROUTINE Example

```
SUBROUTINE test (x, y: NUMBER; b: OUT LOGICAL; c : IN OUT)
  IF x>= y THEN SET b = on        -- valid
  IF c> 44.4 THEN SET c = 44.4    -- valid
  SET x = y                       -- NOT VALID, x is IN by default
  IF b THEN SET c = y             -- NOT VALID , b is OUT
END test
```

4.4.1.4 Subroutine Arguments Definition

The following data types, single array elements of these types, and whole arrays of these types can be used as arguments in a subroutine:

- Number
- Logical
- Time
- String
- Enumeration
- State name list

Data point identifiers are not permitted as subroutine arguments.

4.4.1.5 Subroutine Arguments Examples

```
SUBROUTINE test (x,y : NUMBER; b : OUT LOGICAL; c : IN OUT)
SUBROUTINE calc (a,b : OFF/ON; c : IN OUT NUMBER ARRAY (1..3) )
SUBROUTINE calc1 (a,b : IN OUT state1/state2/state3;
&                  val1 : NUMBER ARRAY (2..4))
SUBROUTINE calc2 (setval : mode ; spval : IN OUT NUMBER)
SUBROUTINE calc3 (flags : IN OUT LOGICAL ARRAY (1..4) )
```

The following subroutine header definitions match the calling sequences at heading 3.2.13.3, CALL Examples.

```
SUBROUTINE sub1 (x : LOGICAL; y : IN open/close)
SUBROUTINE sub2 (index; arr1 : IN OUT LOGICAL ARRAY (1..3))
SUBROUTINE sub3 (flag : IN LOGICAL; num1 : IN NUMBER;
&                output : OUT NUMBER)
SUBROUTINE sub4 (a : IN NUMBER)
SUBROUTINE sub5 (a : IN OUT LOGICAL)      -- sub5 will not be called
                                          -- since the calling sequence
                                          -- shown at heading 3.2.13.3
                                          -- is ILLEGAL
SUBROUTINE sub6 (status : IN LOGICAL)
SUBROUTINE sub7 (state : IN open/close/bad/moving/inbtwn)
```

4.5 BUILT-IN FUNCTIONS AND SUBROUTINES

If an enumeration is called for by an External declaration and that enumeration has a state name that is the same as a built-in subroutine or function, the compiler does not let the enumeration be declared. A parameter of that enumeration type cannot be referenced in a CL program.

See paragraph 2.7.4.5 for a discussion of the built-in predicates Badval and Finite which are similar to logical functions but have more restricted uses.

4.5.1 Built-In Arithmetic Functions

All functions listed below accept arguments of type Number and return Number results; in addition, the trigonometric functions listed below must be specified in radians, NOT DEGREES.

Abs (x)	-- absolute value
Atan (x)	-- arc tangent
Avg (x, y, ...)	-- average (maximum of 16 arguments)
Cos (x)	-- cosine
Exp (x)	-- exponential
Int (x)	-- truncate to integer
Ln (x)	-- natural logarithm
Log10 (x)	-- common logarithm
Max (x, y, ...)	-- maximum (maximum of 16 arguments)
Min (x, y, ...)	-- minimum (maximum of 16 arguments)
Round (x)	-- round to integer*
Sin (x)	-- sine
Sqrt (x)	-- square root
Sum (x, y, ...)	-- sum (maximum of 16 arguments)
Tan (x)	-- tangent

* If the value to be rounded is outside the range of -32767.4 to +32766.5, the function returns the value NaN.

4.5.2 Other Built-In Functions

Other CL/APM built-in functions—in alphabetical order—are:

4.5.2.1 Date_Time Function

This function returns TDC 3000 absolute time (seconds since midnight of January 1, 1979).

Example: `SET TIME(4) = Date_Time`

4.5.2.2 Equal_String Function

This logical function is used to do a case insensitive string comparison ("fred" = "Fred" = "FRED"). See subsection 2.3.5 for a discussion of string comparison rules. The calling sequence for this function is

```
Equal_String (str1, str2)
```

where str1 and str2 are two strings to compare. They can be string literals, parameters, local variables or constants. The return status is ON when the strings are equal.

Example:

```
EXTERNAL pm03
LOCAL string1 : STRING AT STR8(1)
LOCAL string2 : STRING AT APM01S03.STR8(2)
LOCAL string_const = "HI"
      . . .
IF Equal_String (string1, string2) THEN EXIT
IF Equal_String ("FRED", STR8(2)) THEN GOTO PHASE two
IF Equal_String (string_const, "George") THEN SEND : string_const
IF NOT (Equal_String (string2, pm03.STR8(3))) THEN FAIL
```

4.5.2.3 Len Function

This function accepts a string and returns the length as a Number value. The calculated string length does not include any trailing blanks. For example, Len of `"abcde    "` is 5, and Len of `""` is 0.

Example: `SET NN(1) = Len (STR8(1))`

4.5.2.4 Now Function

This function returns wall clock time (seconds since midnight of the current day).

Example: `SET TIME(3) = Now`

4.5.2.5 Number Function

This function takes a time expression as input and returns the resulting time duration as a number. The returned value is the number of seconds represented by the time expression. TIME has a greater precision than NUMBER, so the returned value may be less precise than the time expression.

Example: `SET NN(3) = Number (TIME(2) + TIME(3))`

4.5.3 Built-In Subroutines

The CL/APM built-in subroutines—in alphabetical order—are:

4.5.3.1 Modify_String Subroutine

This subroutine changes the value of a substring in a target string to the value of a substring copied from a source string. The calling sequence is

```
CALL Modify_String (s, ts, tp, n, ss, sp)
```

where "s" will contain the return status of the store request — 0 = success, 1= fail
"ts" is the target string to be modified
"tp" is the index to the first character in the target string that will be modified
"n" is the number of characters to move from the source string to the target string
"ss" is the string from which characters will be fetched
"sp" is the index to the first character to be fetched from the source string

If the number of characters (n) is greater than the total number of characters between the source position index (sp) and the defined size of the string, only the number of characters that exist in the source substring, starting at the position index, are moved to the target substring. A status of success (0) is returned.

If the number of characters (n) is greater than the number of characters between the target position index (tp) and the defined size of the target string, only the number of characters that will fit in the target substring will be moved from the source substring. A status of success (0) is returned.

If the target position (tp) is greater than the defined size of the target string or the source position index (sp) is greater than the defined size of the source string, a status of fail (1) is returned.

If the target position (tp) is greater than the current length of the target string, blanks are inserted between the current length position and the target position.

Example:

```
LOCAL target : STRING AT STR8(1)
LOCAL source : STRING AT STR8(2)
LOCAL tindex   AT NN(1)
LOCAL sindex   AT NN(2)
LOCAL chars    AT NN(3)
LOCAL stat   : NUMBER AT NN(09)
             . . .
SET target = "RECIPEAB"
SET tindex = 7.0
SET chars  = 2.0
SET source = "XX"
SET sindex = 1.0

CALL Modify_String (stat, target, tindex, chars, source, sindex)
```

Results: stat = 0 (success)
target = "RECIPEXX"

4.5.3.2 Number_To_String Subroutine

This subroutine creates a human-readable characterization of a Real number. The calling sequence is:

```
CALL Number_to_String (s, str, i, f)
```

where "s" will contain the return status of the store request — 0 = success, 1= fail (the value cannot be converted to a string because of value size or other reason)
"str" will contain the character representation of the specified real value. A blank string is returned when the status = 1 (fail).
"i" identifies the number to be converted
"f" is a format specification describing the desired character representation of the converted number. Format specifications are described in Appendix A of the *Picture Editor Reference Manual.* Number_to_String accepts only format specifications "real," "integer," and "unknown." There is no default; a value must be specified for this argument.

The "f" argument must be a string literal and should be in all upper case. A string variable is not permitted. The CL compiler will generate a compile time error if the format is not legal.

Example:

```
LOCAL i AT NN(1)
LOCAL str : STRING AT STR8(1)
LOCAL stat : NUMBER AT NN(3)
             . . .
SET i = 23.44

CALL Number_To_String (stat, str, i, "G99999")
SEND : "The number is = ", str
```

The following message will appear in the operator message display:

```
The number is = 23.44
```

4.5.3.3 Set_Bad Subroutine

This subroutine attempts to store a bad value into its argument. Success or failure of the bad value store depends on the specific parameter. If "bad" is not a legal value for the parameter being written to, the sequence program fails with an "illegal value" run-time error. The *Advanced Process Manager Parameter Reference Dictionary* describes which parameters of type Real can accept a bad value.

Example: `Call Set_Bad (!BOX NN(2))`

CL/APM SYNTAX SUMMARY
Appendix A

This section provides a quick reference to CL/APM syntax. It is a summary of the rules of form for CL/APM.

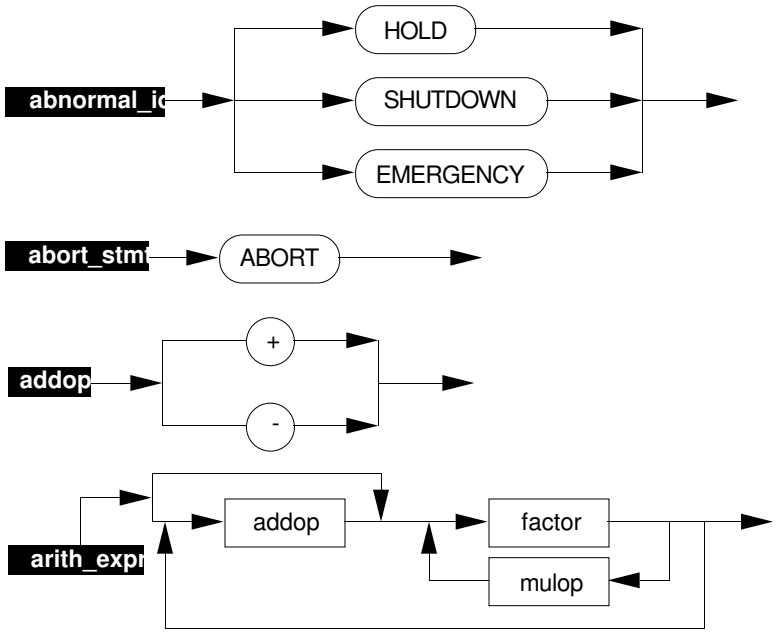
A.1 SYNTAX (GRAMMAR) SUMMARY

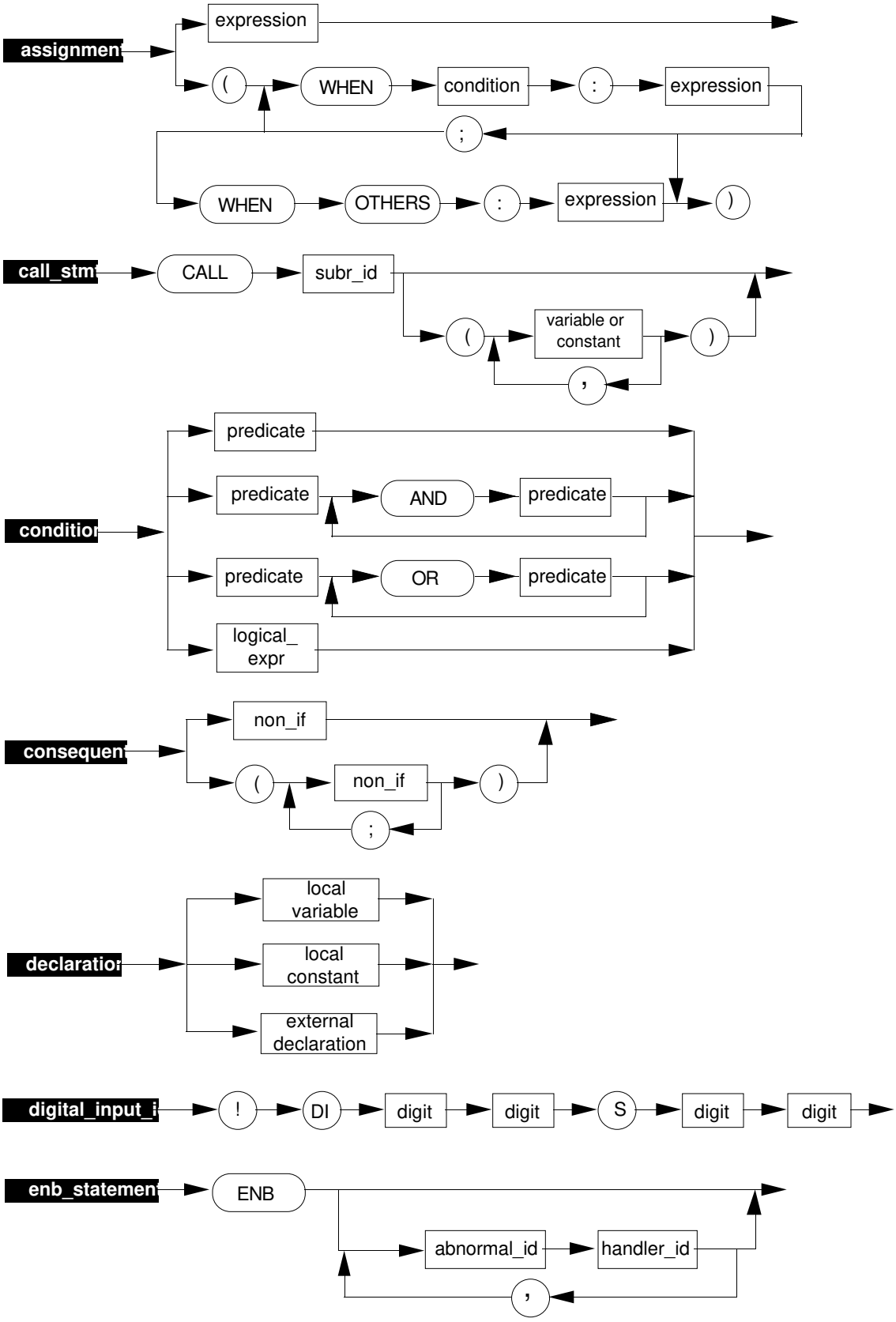
This section is divided into two parts. The first part is a summary of CL/APM syntax in the form of all syntax diagrams that were presented throughout the manual. Each syntax diagram is labeled (in reverse-video), and they are arranged in alphabetical order.

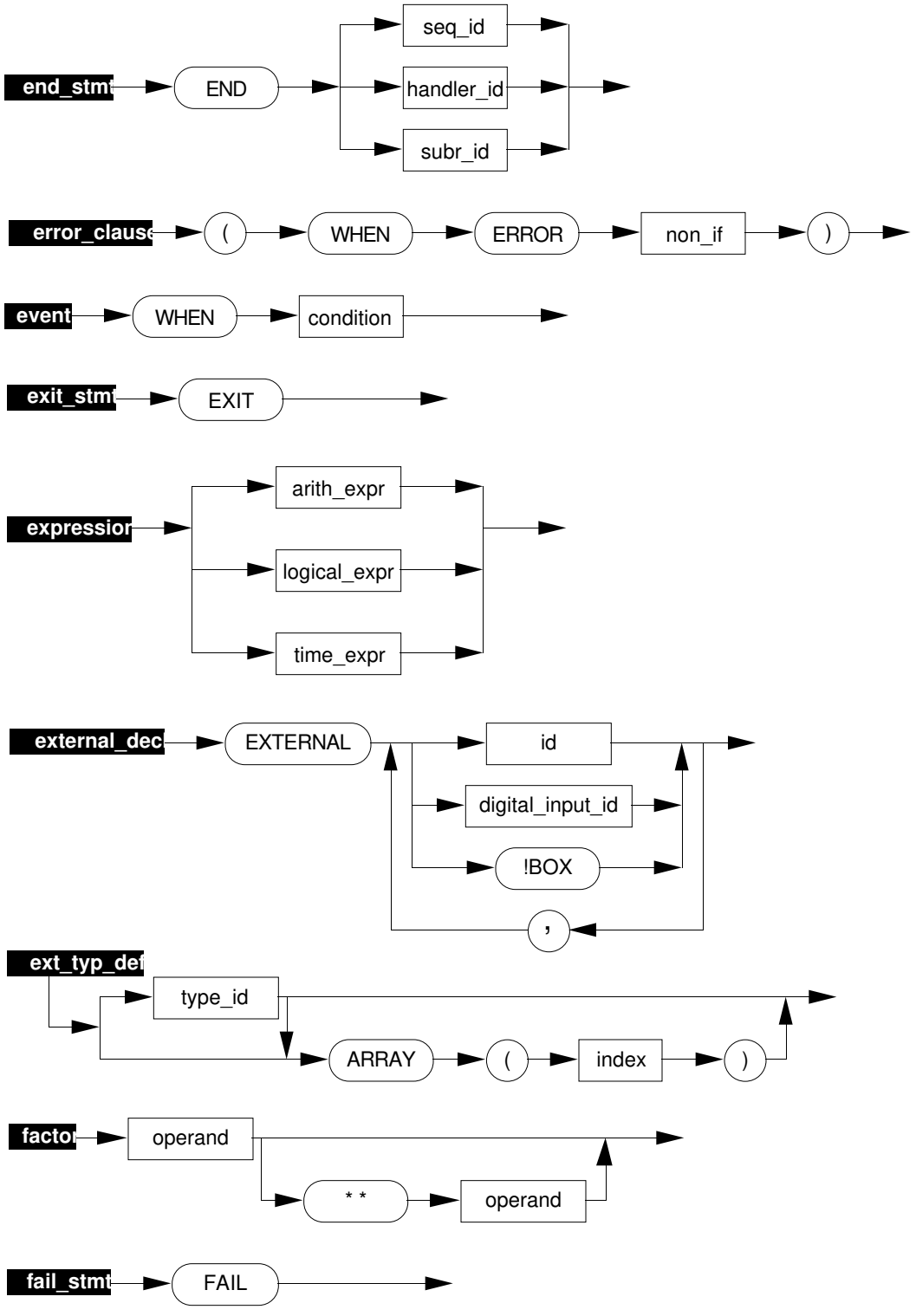
The second part is a summary of CL/APM syntax production rules in BNF notation. The order of the production rules is alphabetical.

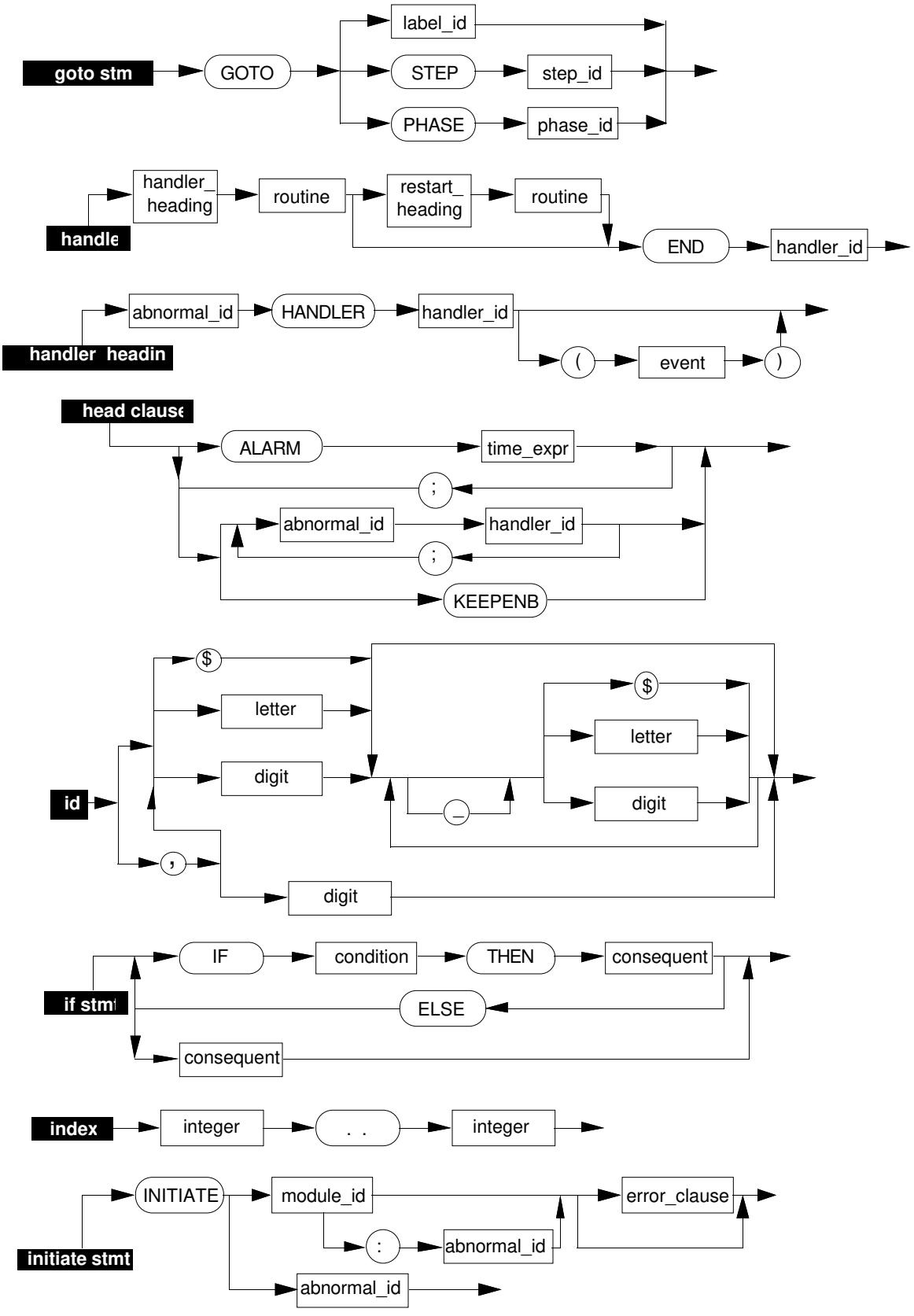
To use either part of this section, decide what item you want to construct and go through either summary (depending on what form you feel most comfortable with) looking for that item on the left-most portion of each page. When you find the item, the way to build it is contained in the diagram or production rule to the right of the item. Note that some simple items that are listed individually in the BNF version (heading A.3) are included WITHIN more complex diagrams.

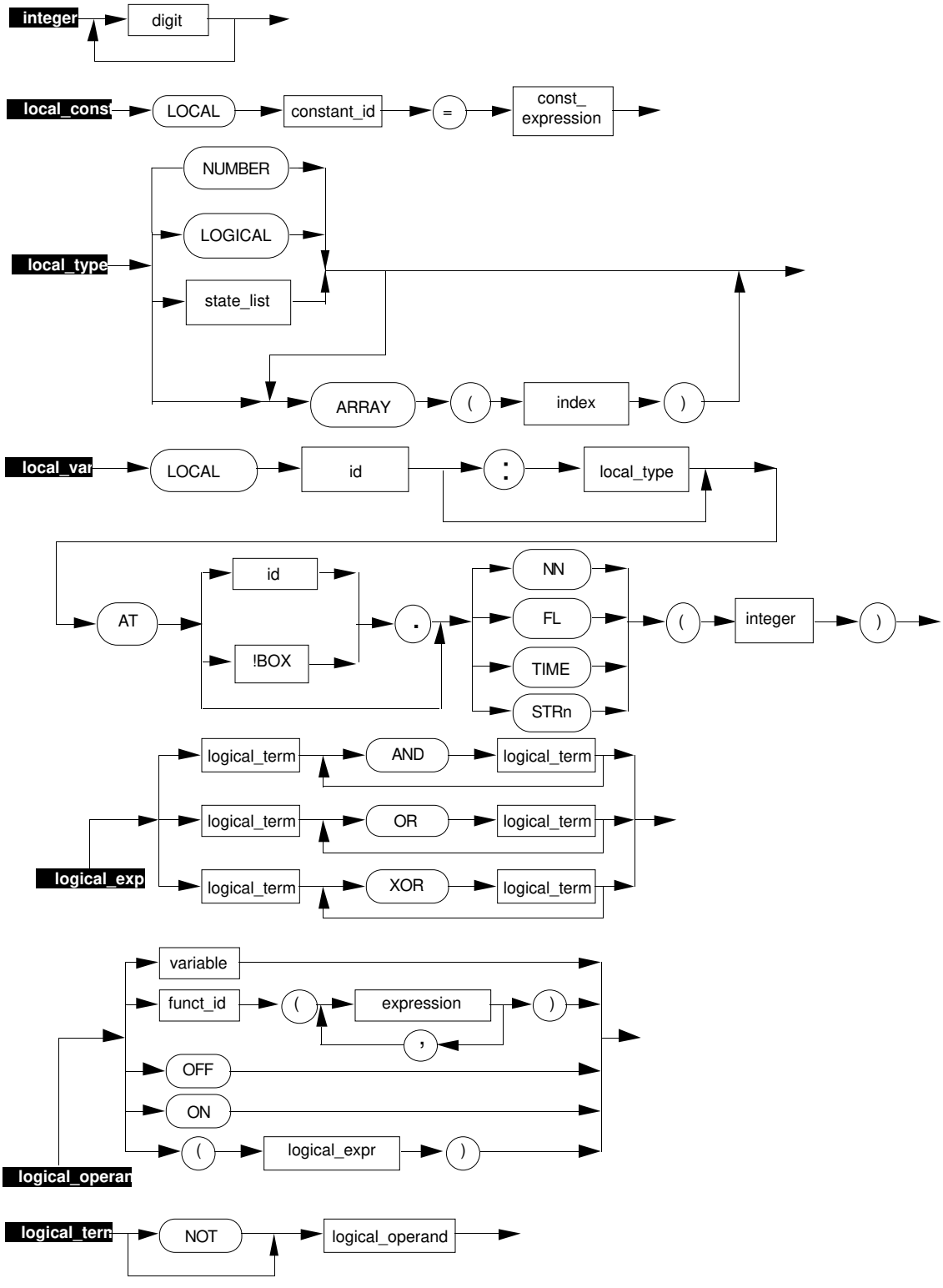
A.2 SYNTAX DIAGRAM SUMMARY

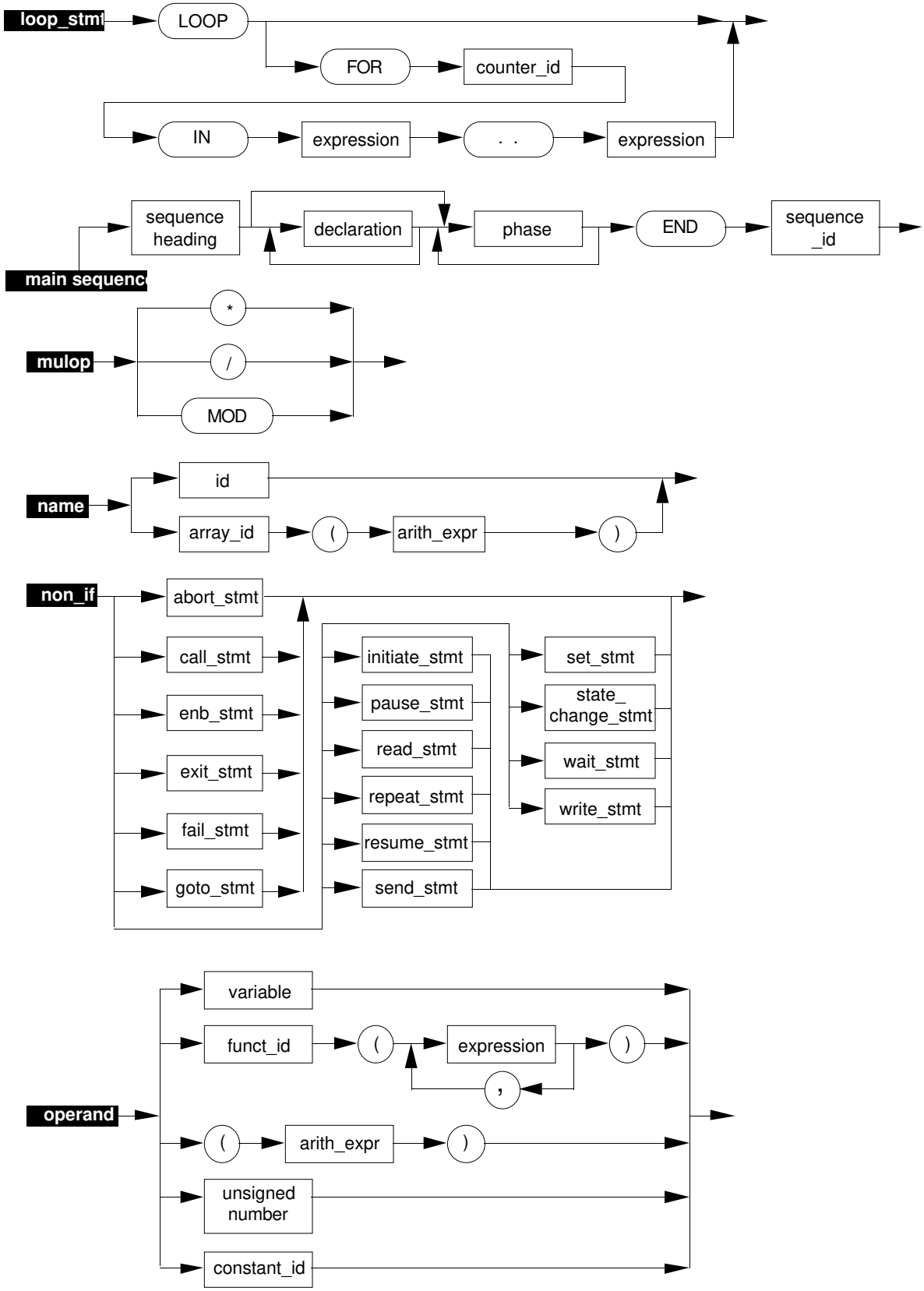


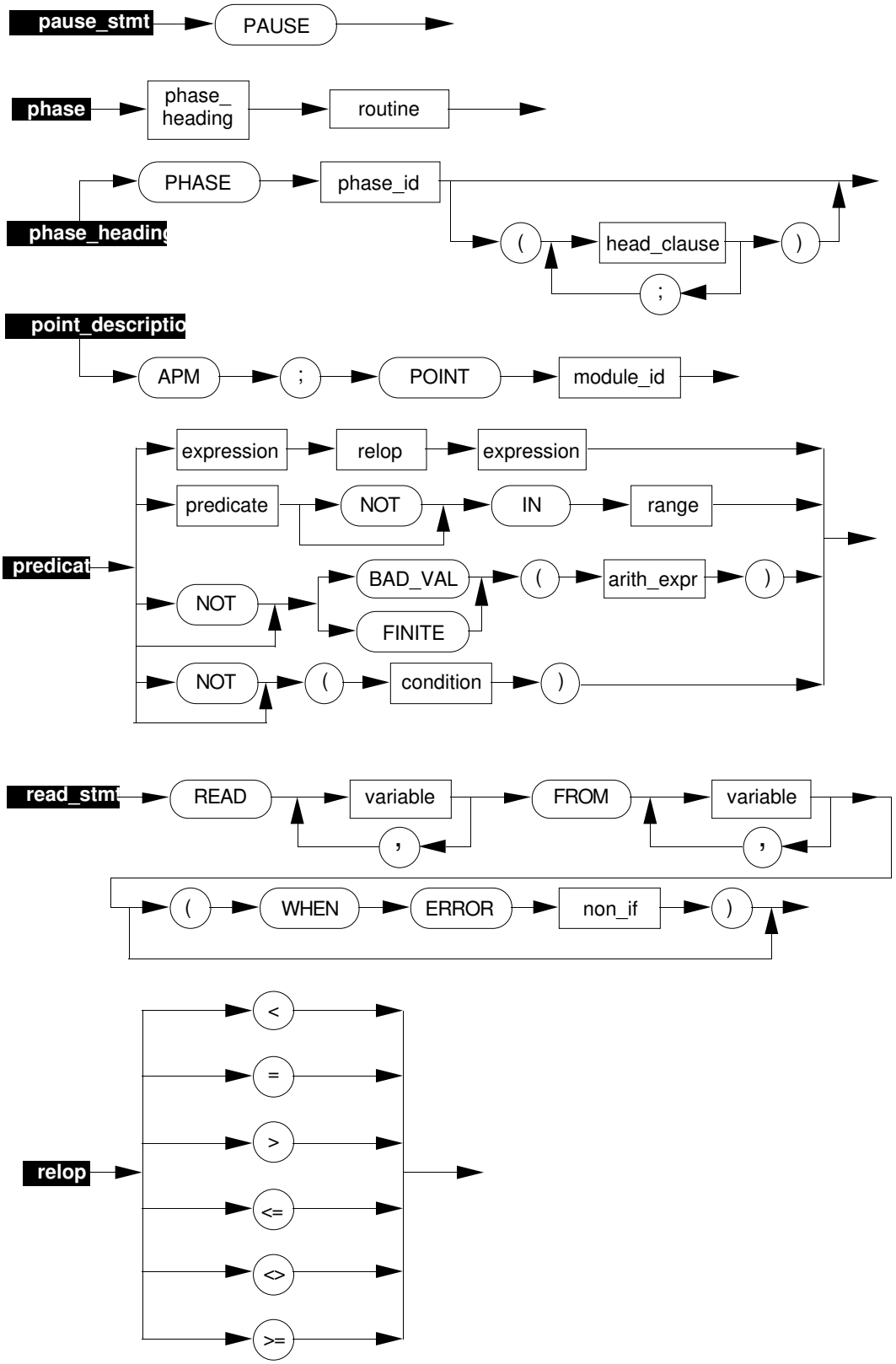

assignmen
expression
(
WHEN
condition
:
expression
;
WHEN
OTHERS
:
expression
)
call_stm
CALL
subr_id
(
variable or constant
)
,
conditio
predicate
predicate
AND
predicate
predicate
OR
predicate
logical_ expr
consequen
non_if
(
non_if
)
;
declaratio
local variable
local constant
external declaration
digital_input_i
!
DI
digit
digit
S
digit
digit
enb_statemen
ENB
abnormal_id
handler_id
,

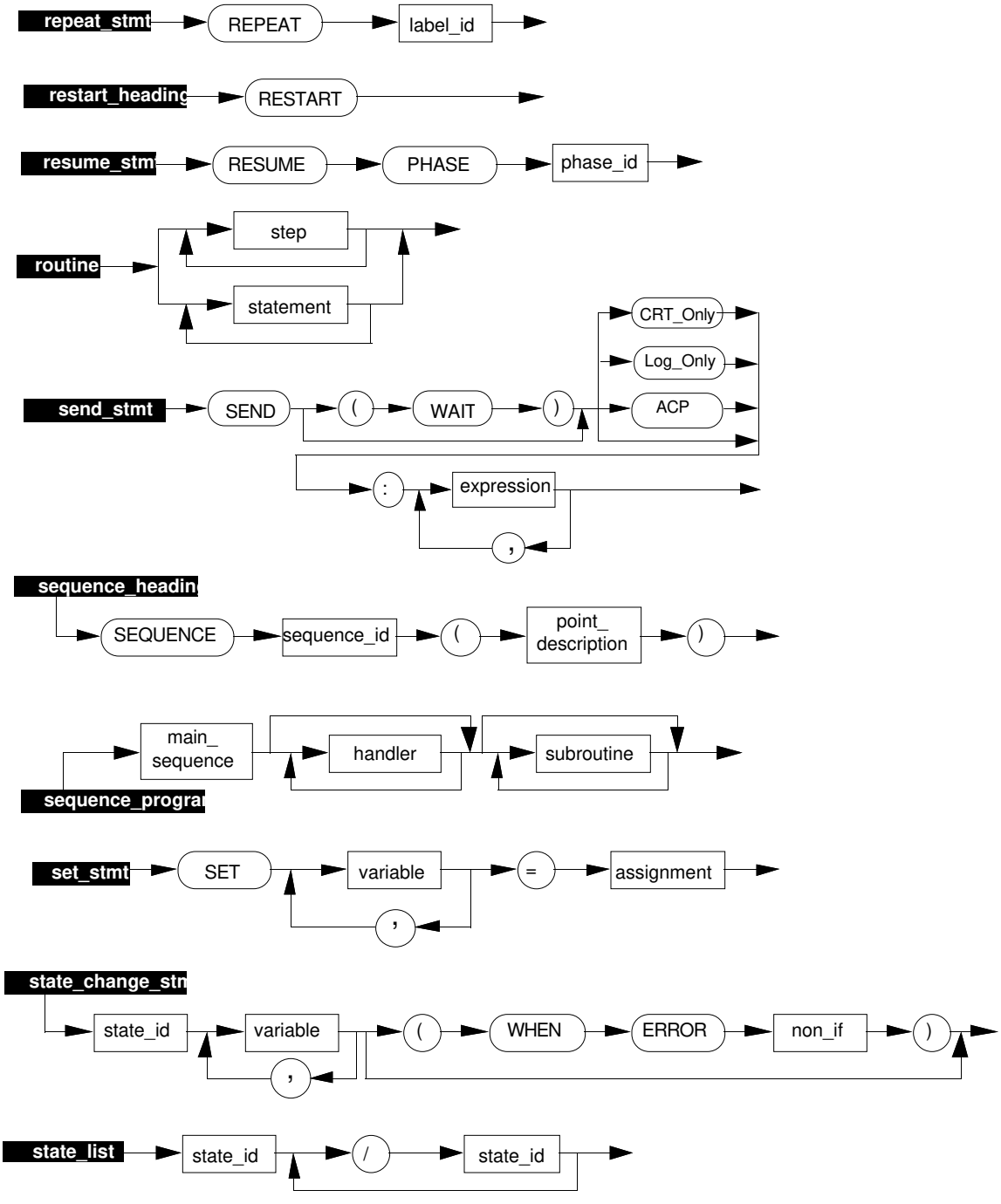

end_stmt
END
seq_id
handler_id
subr_id
error_clause
(
WHEN
ERROR
non_if
)
event
WHEN
condition
exit_stmt
EXIT
expression
arith_expr
logical_expr
time_expr
external_dec
EXTERNAL
id
digital_input_id
!BOX
,
ext_typ_def
type_id
ARRAY
(
index
)
factor
operand
* *
operand
fail_stmt
FAIL

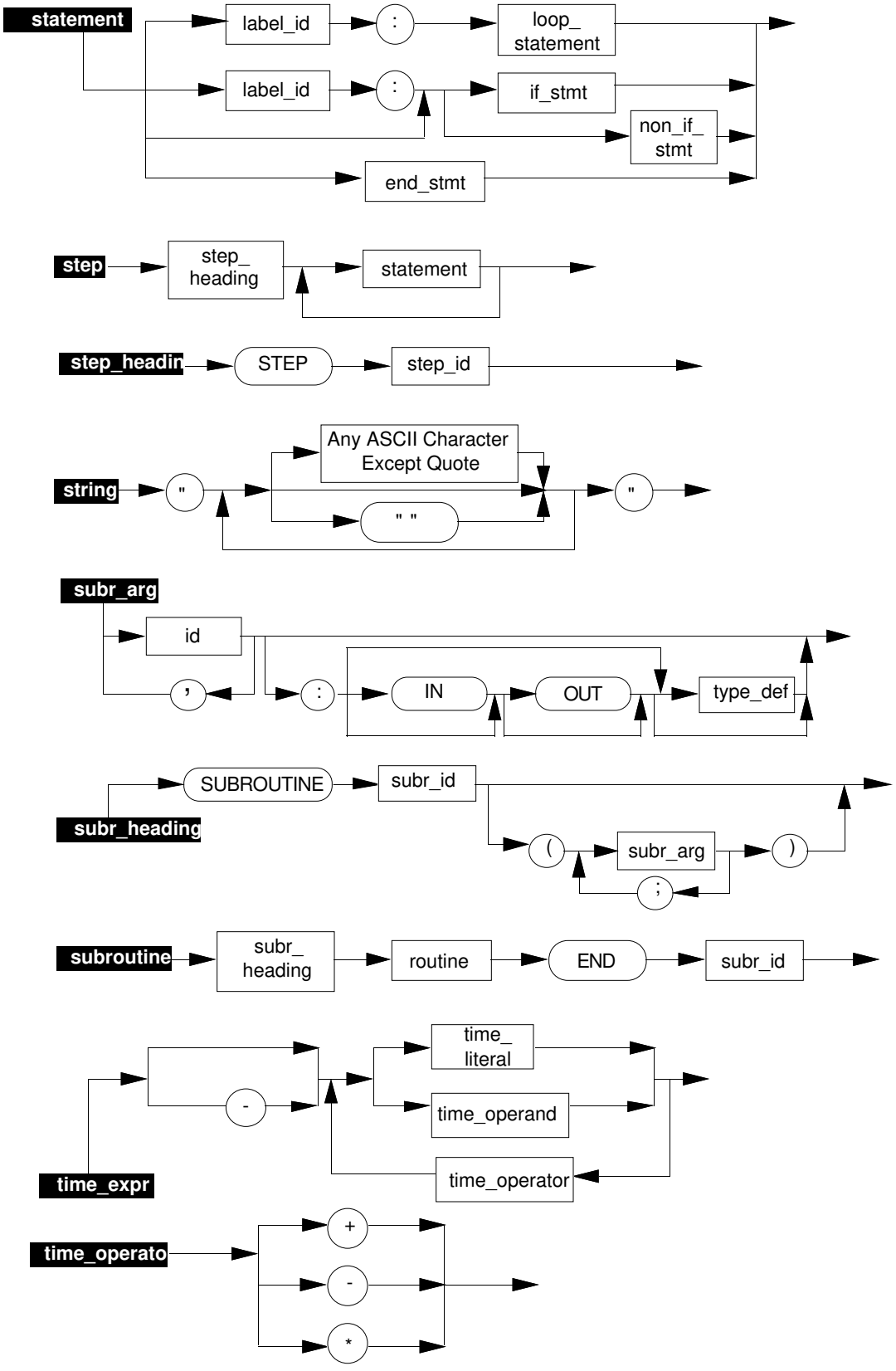

goto stm
GOTO
label_id
STEP
step_id
PHASE
phase_id
handle
handler_ heading
routine
restart_ heading
routine
END
handler_id
handler headin
abnormal_id
HANDLER
handler_id
(
event
)
head clause
ALARM
time_expr
;
abnormal_id
handler_id
;
KEEPENB
id
$
letter
digit
,
_
$
letter
digit
digit
if stm
IF
condition
THEN
consequent
ELSE
consequent
index
integer
. .
integer
initiate stmt
INITIATE
module_id
:
abnormal_id
error_clause
abnormal_id

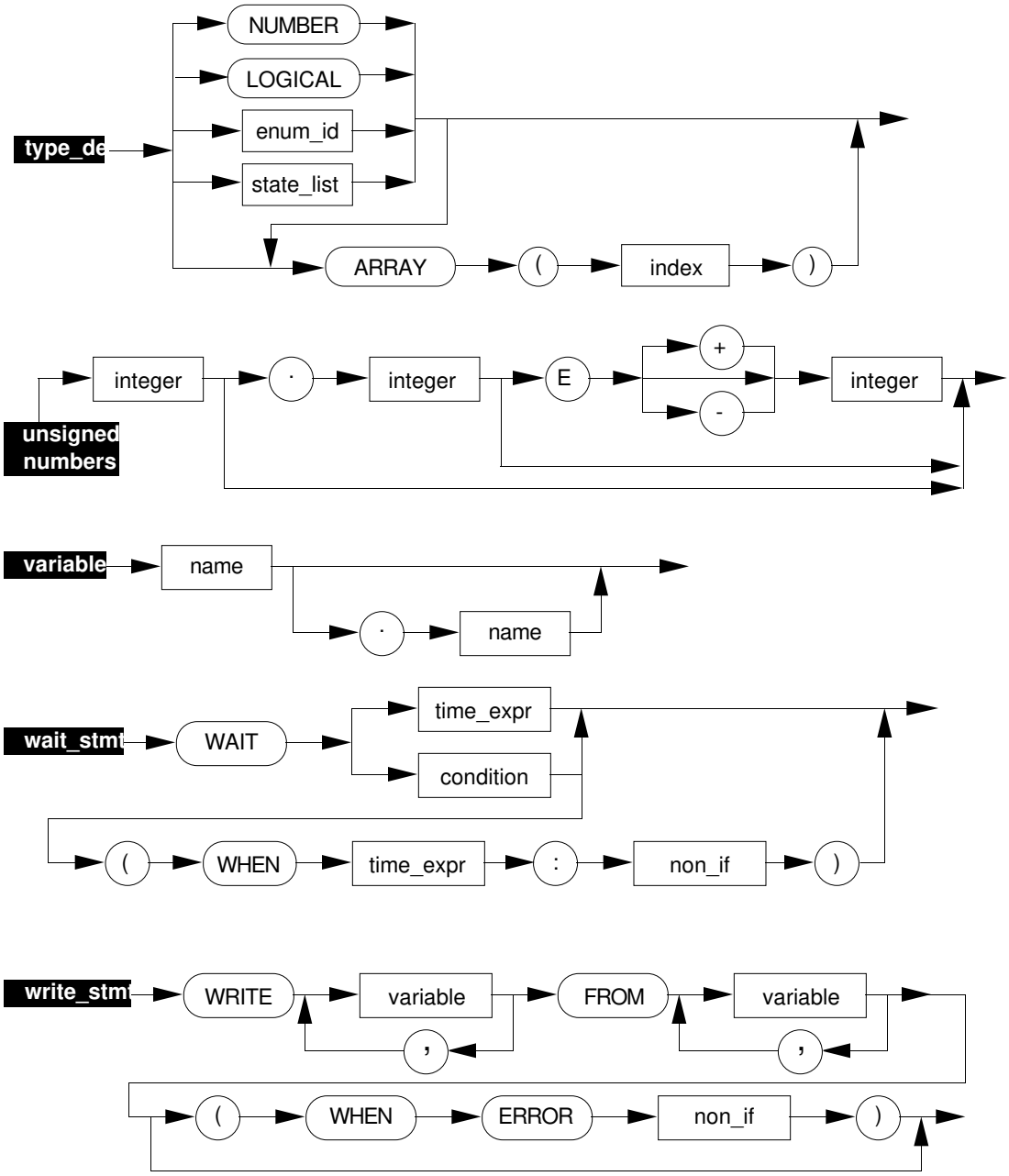

integer
digit
local_cons
LOCAL
constant_id
=
const_
expression
NUMBER
LOGICAL
local_type
state_list
ARRAY
(
index
)
local_var
LOCAL
id
:
local_type
AT
id
!BOX
.
NN
FL
TIME
STRn
(
integer
)
logical_exp
logical_term
AND
logical_term
logical_term
OR
logical_term
logical_term
XOR
logical_term
variable
funct_id
(
expression
)
,
OFF
ON
(
logical_expr
)
logical_operan
logical_tern
NOT
logical_operand


loop_stmt
LOOP
FOR
counter_id
IN
expression
. .
expression
main sequenc
sequence heading
declaration
phase
END
sequence _id
mulop
*
/
MOD
name
id
array_id
(
arith_expr
)
non_if
abort_stmt
call_stmt
enb_stmt
exit_stmt
fail_stmt
goto_stmt
initiate_stmt
pause_stmt
read_stmt
repeat_stmt
resume_stmt
send_stmt
set_stmt
state_ change_stmt
wait_stmt
write_stmt
operand
variable
funct_id
(
expression
,
)
(
arith_expr
)
unsigned number
constant_id


pause_stmt
PAUSE
phase
phase_ heading
routine
PHASE
phase_id
phase_headin
(
head_clause
)
;
point_descriptio
APM
;
POINT
module_id
predicat
expression
relop
expression
predicate
NOT
IN
range
BAD_VAL
(
arith_expr
)
NOT
FINITE
NOT
(
condition
)
read_stm
READ
variable
FROM
variable
,
,
(
WHEN
ERROR
non_if
)
relop
<
=
>
<=
<>
>=


repeat_stmt
REPEAT
label_id
restart_heading
RESTART
resume_stm
RESUME
PHASE
phase_id
routine
step
statement
send_stmt
SEND
(
WAIT
)
CRT_Only
Log_Only
ACP
:
expression
,
sequence_headin
SEQUENCE
sequence_id
(
point_
description
)
main_
sequence
handler
subroutine
sequence_progra
set_stmt
SET
variable
=
assignment
,
state_change_stn
state_id
variable
(
WHEN
ERROR
non_if
)
,
state_list
state_id
/
state_id


statement
label_id
:
loop_ statement
label_id
:
if_stmt
non_if_ stmt
end_stmt
step
step_ heading
statement
step_headin
STEP
step_id
string
"
Any ASCII Character Except Quote
" "
"
subr_arg
id
,
:
IN
OUT
type_def
subr_heading
SUBROUTINE
subr_id
(
subr_arg
)
;
subroutine
subr_ heading
routine
END
subr_id
time_expr
-
time_ literal
time_operand
time_operator
time_operato
+
-
*


type_de
NUMBER
LOGICAL
enum_id
state_list
ARRAY
(
index
)
unsigned numbers
integer
.
integer
E
+
-
integer
variable
name
.
name
wait_stmt
WAIT
time_expr
condition
(
WHEN
time_expr
:
non_if
)
write_stmt
WRITE
variable
,
FROM
variable
,
(
WHEN
ERROR
non_if
)

A.3 NOTATION USED FOR SYNTAX PRODUCTION RULES

This presentation of CL/APM syntax production rules follows BNF notation conventions as follows:

- Sequences of lower-case characters and embedded underscores mean things are to be combined according to the exact form expressed under this Syntax heading.
- Upper-case characters and special characters appear as written, except for the symbols ::=, {, }, [,], and |, which are explained as follows:
  - An item enclosed in braces ({, }) stands for the occurrence of that item zero or more times.
  - An item enclosed in square brackets ([,]) stands for the occurrence of that item zero or one times; i.e., the item is optional.
  - The symbols ::= and | stand for production (how to build, or put together) and alternation, respectively. For example, **x ::= y | z** can be read **x produces y or z**. Another way to explain the ::= symbol is that in order to form x, y or z must be present in the form listed. Many times, a form on the right of the ::= symbol is itself given a syntactic form, using the same example, **x ::= y | z**. A further rule that governs the form of z is specified, indented and just below the form that specifies how to form x. For example,

    x ::= y | z
    y ::= point_param_sp
    z ::= point_param_pv

    This means that to produce x you need to specify y or z; y is produced by specifying a point's setpoint, and z is produced by specifying a point's process variable parameter PV.
- Unless otherwise noted, all symbols ending in **_id** are ordinary identifiers (i.e., **anything_id ::= id**).

A.4 CL/APM SYNTAX PRODUCTION RULES

abnormal_id ::= HOLD | SHUTDOWN | EMERGENCY

abort_stmt ::= ABORT

addop ::= + | -

arith_expr ::= [addop] term {addop term}

array_def ::= ARRAY (index)

```
assignment ::= expression
             | (WHEN condition : expression
               {; WHEN condition : expression}
               [; WHEN OTHERS : expression] )

call_stmt ::= CALL subr_id [ (variable | constant {, variable | constant}) ]

condition ::= predicate {AND predicate}
            | predicate {OR predicate}
            | logical_expr

consequent ::= non_if
             | (non_if {; non_if} )

declaration ::= local_var
              | local_const
              | external_decl

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

digital_input_id ::= !DI digit digit S digit

else_if_stmt ::= ELSE IF condition THEN consequent

else_stmt ::= ELSE consequent

enb_statement ::= ENB [abnormal_id handler_id {, abnormal_id handler_id}]

end_statement ::= END seq_id | END handler_id | END subr_id

error_clause ::= (WHEN ERROR non_if)

event ::= WHEN condition

exit_stmt ::= EXIT

expression ::= arith_expr | time_expr | logical_expr

expressions ::= expression {, expression}

external_decl ::= EXTERNAL ident {, ident}

factor ::= operand [** operand]

fail_stmt ::= FAIL

first_char ::= $ | letter

goto_stmt ::= GOTO lablel_id
            | STEP step_id
            | PHASE phase_id
```

```
handler ::= handler_heading
            routine
            [ restart_heading
            routine ]
            END handler_id

handler_heading ::= abnormal_id HANDLER handler_id [ (event) ]

head_clause ::= [ALARM time_expr]
              | [ALARM time_expr ; ] KEEPENB
              | [ALARM time_expr ; ] abnormal_id handler_id
                        {; abnormal_id handler_id}

ident ::= id | digital_input_id | !BOX

id ::= first_char { [_] letter_or_digit }
     | digit  [_] letter_or_digit  { [_] letter_or_digit }
     | ' letter_or_digit { [_] letter_or_digit }

ids ::= id {, id}

if_stmt ::= IF condition THEN consequent
            { else_if_stmt }
            [ else_stmt ]

index ::= integer .. integer

initiate_stmt ::= INITIATE module_id [ : abnormal_id] [error_clause]
                | INITIATE abnormal_id

integer ::= digit {digit}

letter ::= upper_case_alphabetic
         | lower_case_alphabetic

letter_or_digit ::= letter | digit | $

local_const ::= LOCAL constant_id = const_expression

local_type :: =  NUMBER [array_def]
              | LOGICAL [array_def]
              | TIME [array_def]
              | STRING [array_def]
              | state_list [array_def]
              | array_def

local_var ::= LOCAL typed_id AT [slot_box_id . ] param_id (integer)

logical_expr ::= logical_term {AND logical_term}
              | logical_term {OR logical_term}
              | logical_term {XOR logical_term}
```

```
logical_operand ::= variable
                  | function_id (expressions)
                  | OFF
                  | ON
                  | (logical_expr)

logical_term ::= [NOT] logical_operand

loop_stmt ::= LOOP [ FOR counter_id IN range ]

main_sequence ::= sequence_heading
                  { declaration }
                  phase
                  { phase }
                  END sequence_id

mulop ::= * | / | MOD

name ::= id
       | array_id (arith_expr)

non_if ::= set_stmt          | read_stmt       | write_stmt
         | state_change_stmt | goto_stmt       | repeat_stmt
         | pause_stmt        | wait_stmt       | call_stmt
         | send_stmt         | initiate_stmt   | fail_stmt
         | resume_stmt       | exit_stmt       | abort_stmt
         | enb_stmt

one_d_array ::= (const_expression {, const_expression} )

operand ::= variable
          | function_id (expressions)
          | (arith_expr)
          | unsigned_number
          | String
          | constant_id

own_box_id ::= !BOX

param_id ::= NN | FL | TIME | STR8 | STR16 | STR32 | STR64

pause_stmt ::= PAUSE

phase ::= phase_heading
          routine

phase_heading ::= PHASE phase_id [(head_clause {; head_clause} )]

point_description ::= APM ; POINT point_id
```

```
predicate ::= expression relop expression
            | arith_expr [NOT] IN range
            | [NOT] BAD_VAL (arith_expr)
            | [NOT] FINITE (arith_expr)
            | [NOT] (condition)

range ::= expression .. expression

read_stmt ::= READ variables FROM variables [error_clause]

relop ::= < | = | > | <= | <> | >=

repeat_stmt ::= REPEAT label_id

restart_heading ::= RESTART

resume_stmt ::= RESUME PHASE phase_id

routine ::= step {step}
          | statements

send_stmt ::= SEND [(WAIT)] [variable] : expressions

sequence_heading ::= SEQUENCE sequence_id ( point_description )

sequence_program::= main_sequence
                    { handler }
                    { subroutine }

set_stmt ::= SET variables = assignment

sign ::= + | -

slot_box_id ::= id | !BOX

state_change_stmt ::= state_id [variables] [error_clause]

state_list ::= state_id / state_id {/ state_id}

statement ::= label_id : loop_stmt
            | [label_id :] unlabeled_stmt
            | end_stmt

statements ::= statement {statement}

step ::= step_heading
         statements

step_heading ::= STEP step_id

String ::= " {String_chr} "
```

```
String_chr ::= any ASCII character_except_quote
             | ""

subr_arg ::= ids
           | ids : IN [type_def]
           | ids : OUT [type_def]
           | ids : IN OUT [type_def]
           | ids : type_def

subr_args ::= (subr_arg {; subr_arg} )

subr_heading ::= SUBROUTINE subr_id [subr_args]

subroutine ::= subr_heading
               routine
               END subr_id

term ::= factor {mulop factor}

time_expr ::= [-] time_term {time_operator time_term}

time_literal :: = operand DAYS [operand HOURS] [operand MINS] [operand SECS]
                | operand HOURS [operand MINS] [operand SECS]
                | operand MINS [operand SECS]
                | operand SECS

time_operand ::= variable | parameter | constant | array_element | literal | function_result

time_operator ::= + | - | *

time_term ::= time_literal | time_operand

type_def ::= local_type
           | enum_id [array_def]

typed_id ::= id [ : type_def ]

unlabeled_stmt ::= if_stmt
                 | non_if

unsigned_number ::= integer [. integer  [E [sign] integer]]

variable ::= name [. name]

variables ::= variable {, variable}

wait_stmt ::= WAIT time expr
            | WAIT condition [ (WHEN time_expr : non_if) ]

write_stmt ::= WRITE variables FROM variables [error_clause]
```

CL SOFTWARE ENVIRONMENT
Appendix B

This appendix refers you to the various control functions reference manuals for information on how TDC 3000 control functions support CL (in other words, the CL Run Time Environment). This appendix also details some of the limitations the TDC 3000 CL Run Time Environment places on various aspects of building CL structures, such as memory usage.

B.1 REFERENCES TO CONTROL FUNCTIONS PUBLICATIONS

The relationships between the TDC 3000 Software environment and CL/APM is found in the *System Control Functions,* and the *Advanced Process Manager Control Functions* publications.

Although you should read all of those publications to understand TDC 3000 Data Acquisition & Control, the following subsections contain information specific to CL; they should be read to gain an understanding of the data point types that you will be using, as well as any particular constraints and nuances of CL as it relates to the standard TDC 3000 Control Software.

HEADING	TOPIC
SYSTEM CONTROL FUNCTIONS	
3.3.1.1.2	CL Access (to parameters — general)
3.3.7	Advanced Functions
4.7.4 & 4.7.5	Value Stores (CL corrective action on errors)
APM CONTROL FUNCTIONS & ALGORITHMS	
6.1	Process Module Data Point (CL/APM)

B.2 CL/APM CAPACITIES

NOTE

Under some conditions, memory limits in the Engineering personality may be reached before reaching the following absolute limits.

Max. number of statements per step	255
Max. object size for each sequence (392 blocks of 32 words each)	12544

Max. blocks of code for all sequence slots in APM. (Note that in the APM node configuration the value "Number of Process Module Slots" must be nonzero to be able to build Process Module points.)	12400 (minus blocks used for pids, logic blocks, etc.)

Max. phases and steps in a sequence is only limited by the number of slots available for phases and step identifiers in the NIM Library.

Max. size for expression (or condition). See Table B-1 for guidelines to calculating the size of an expression.	100 items
Max. declarations in a sequence (includes locals, externals, and constants). All declarations count as 1 item each.	approx. 270 items
Max. number of constant declarations in a sequence; only constants that are referenced in the body of the sequence are counted. Declarations for duplicate values are not counted.	Numbers—256 Time—255 Strings—see section B.4.1

Table B-1 — Calculating Expression Size

Item	Number of words
• Operators (e.g., +, -, *, etc.) • All constants except string constants • LOCAL flag and numeric variables with a constant index mapped to the Bound Data Point • Bound Data Point flag and numeric parameter references with a constant index • Subroutine arguments • The built-in functions Now, Date_Time, Number, Equal_String, Len • IOL prefetch reference	One word
• String constants • All other operands, including operands with computed indices. This includes LOCALs on the Bound Data Point or other Process Module data points, regulatory control references, references to off-node parameters, etc.	Two words

EXAMPLES (expression sizes reflect number of words generated on the right-hand side of the equal (=) sign):

```
LOCAL a : TIME AT TIME(01)
LOCAL x AT NN(01)
LOCAL y AT NN(02)
LOCAL z AT !BOX.NN(17)
    . . .
SET x = (y + NN(03))                    -- expression size is 3 words
SET x = (z + NN(03))                    -- expression size is 4 words
SET a = TIME(01) + !BOX.TIME(01) + now -- expression size is 7 words
SET a = 15 SECS + a                     -- expression size is 4 words
```

(Continued)

Table B-1 — Calculating Expression Size (continued)

RULES:
- When a computed index is used on a parameter reference, one word must be added for each computed index.
- When a computed index is used on a local variable or subroutine argument, four words must be added for each computed index.
- When an array subroutine argument is referenced with a constant subscript, three words are generated for the constant index.

EXAMPLES (expression sizes reflect number of words generated on the right-hand side of the equal (=) for SET statements, or the expression following the IF statement and prior to the THEN)

```
LOCAL a : TIME AT TIME(01)
LOCAL b : TIME ARRAY(1..3) AT APM01S01.TIME(1)
LOCAL i AT NN(8)
LOCAL x AT NN(01)
LOCAL y AT NN(02)
LOCAL z AT !BOX.NN(17)
     . . .
SET x = (y + NN(i))                    -- expression size is 6 words
SET x = (z + NN(i + NN(i + 1)))        -- expression size is 14 words
SET a = b(i) + !BOX.TIME(i) + Now      -- expression size is 14 words
CALL sub1 (b)
CALL sub2 (a)
     . . .
SUBROUTINE sub1 (arg1 : TIME ARRAY (1..3))
IF arg1 (i) = 1 SECS THEN EXIT         -- expression size is 8 words
IF arg1 (1) = 1 SECS THEN EXIT         -- expression size is 6 words
END sub1
     . . .
SUBROUTINE sub2 (arg1 : TIME)
IF arg1 = 1 SECS THEN EXIT             -- expression size is 3 words
END sub2
```

(Continued)

Table B-1 — Calculating Expression Size (continued)

RULE: • If a time expression is composed of a computed time (vs a time constant), add a one-word overhead to the expression for each unused time suffix (i.e., DAYS, HOURS, MINS, SECS) in the expression, plus a one-word overhead for a time expression.

EXAMPLES (expression sizes reflect number of words generated on the right-hand side of the equal (=) for each set statement)

```
LOCAL a : TIME AT TIME(01)
LOCAL i AT NN(8)
LOCAL x AT NN(01)
LOCAL z AT !BOX.NN(17)
     . . .
SET i = Number (10 SECS)              -- expression size is 2 words
SET a = x SECS                        -- expression size is 5 words
                                      -- (1 word for "x", 3 words
                                      -- overhead for DAYS, HOURS,
                                      -- MINS not used plus 1 word
                                      -- overhead for a time expr.)
SET i = Number (x SECS)               -- expression size is 6 words
SET a = 1 DAYS 1 HOURS x SECS         -- expression size is 5 words
SET a = 1 MINS NN(1) SECS             -- expression size is 5 words
SET a = 1 HOURS z SECS                -- expression size is 6 words
                                      -- (1 word for the constant 1,
                                      -- 2 words overhead for DAYS,
                                      -- MINS not used, 2 words
                                      -- for"z",plus 1 word overhead
                                      -- for a time expression)
SET a = z DAYS x SECS                 -- expression size is 6 words
                                      -- 2 words for "z",
                                      -- 2 words for HOURS, MINS,
                                      -- not used, 1 word for "x",
                                      -- and 1 word overhead for a
                                      -- time expression)
```

B.3 CL/APM DIFFERENCES FROM CL/PM

A program that compiles correctly for R300 CL/PM can be compiled successfully for R400 CL/APM (however, you must change the sequence heading point description from PM to APM before compiling). The program object size compiled for the APM will be larger than if compiled for the PM. The change in object size has two components.

- String constants—except for the sequence name—for the APM are located in the object program instead of being added to the NIM libraries.
- Some CL/APM statements generate an additional word of code (see Table B-1 for guidelines on calculating expression sizes).

WARNING: There is a difference between CL/PM and CL/APM when a Digital Composite PV or OP is passed to a subroutine. The state list in this situation must include the state NONE. For example, suppose that you have a PM Digital Composite with the OP defined with state0=off and state1=on, and you pass this parameter to a subroutine using the following PM/CL code:

```
CALL SUB1 (DIGCOMP.OP)
...
...
SUBROUTINE SUB1 (A: IN OFF/ON)
...
```

If you attempt to compile this in the APM, you will get a Data Type Mismatch error under the call statement. In the APM, you must add NONE in the state list as shown below:

```
SUBROUTINE SUB1 (A: IN OFF/ON/NONE)
```

The PM to APM translator will not make this change—it is up to the programmer to modify the state list.

The following list highlights other significant differences between CL/APM and CL/PM and gives references to the subsections in this manual where the CL/APM capabilities are discussed.

- Time data type—Support for the data type Time (2.3.2) has been expanded and the time functions Date_Time, Now, and Number have been added (4.5.2). Time expressions and time literals are supported (2.7.3).
- Arrays data type—Elements of array parameters can be accessed with variable or calculated subscripts in any CL/APM statement except CALLs to user-written subroutines. Whole-array parameters are valid arguments in calls to user-written subroutines (2.3.4)
- String data type—The maximum string length has been extended to 64 characters and support for the data type has been expanded (2.3.5).
- Local variables—The data types Time and String are mapped against the Process Module, Box and Array data point parameters TIME and STR8, STR16, STR32, and STR64 (2.6.2).

- Local constants—Time constants can be used in any CL/APM statement where time variables can appear (2.6.3).
- Wait statement—More complex time expressions are permitted in Wait statements (3.2.12).
- Send statement—String and Time variables, constants, literals, and parameters can be included in these messages (3.2.14).
- Sequence heading—The sequence program point description begins with APM;POINT (4.1.3).
- Phase heading—More complex time expressions are permitted in the Alarm clause of Phase headings (4.1.4).
- Object file suffix—The CL compiler identifies CL/APM object files with a .NO suffix.
- Additional built-in functions and subroutines (4.5.2 and 4.5.3).

B.4 ITEMS AFFECTING OBJECT CODE SIZE

B.4.1 String Literals in Object Code

All string literals are generated into the object code. This includes the phase names, step names, subroutine names, handler names (all of which are in upper case in the object), and SEND statement strings. When used in a SEND statement, state names for parameters of the type self-defining enumeration and state names for locally defined enumerations also are generated into the object. The NIM library is updated only with the APM sequence name.

String constants defined at the beginning of the code are not added into the code unless they are used in the program body. If a string constant is used more than once in a program, it is generated in the object only once.

A string literal for an 8-character blank string is automatically added to every object code to handle the case of a PHASE statement without a STEP statement.

The only limitation on string constants is object program size.

B.4.2 Time Constants in Object Code

All time constants are generated into the object code. This includes time literals defined in the code, such as

```
SET x = 15 SECS
```

and time constants defined at the beginning of the program, such as

```
LOCAL time_const = 15 MINS 10 SECS
```

Time constants defined at the beginning of the code are not added into the object code unless they are used in the program body. If a time constant is used more than once in a program, it is generated in the object only once.

A time constant for zero seconds is automatically added to every object code to handle the case of a PHASE statement without an ALARM heading clause.

Up to 255 user-defined time constants can be defined in a CL/APM sequence program.

B.4.3 Number Constants in Object Code

All number constants are generated into the object code. This includes number literals defined in the code, such as

```
SET y = 40.4
```

and number constants defined at the beginning of the program, such as

```
LOCAL num_const = 16.8
```

and number literals used in time expressions, such as the "5" in the following

```
SET time(1) = 5 MINS nn(1) SECS
```

Number constants defined at the beginning of the code are not added into the object code unless they are used in the program body. If a number constant is used more than once in a program, it is generated in the object only once.

Up to 256 user-defined number constants can be defined in a CL/APM sequence program.

Index

Index

Index

Index

Index

Index

Index

Index

Index

READER COMMENTS

Honeywell IAC Automation College welcomes your comments and suggestions to improve future editions of this and other publications.

You can communicate your thoughts to us by fax, mail, or toll-free telephone call. We would like to acknowledge your comments; please include your complet name and address

BY FAX: Use this form; and fax to us at (602) 313-4108

BY TELEPHONE: In the U.S.A. use our toll-free number 1*800-822-7673 (available in the 48 contiguous states except Arizona; in Arizona dial 1-602-313-5558).

BY MAIL: Use this form; detach, fold, tape closed, and mail to us.

Title of Publication: **Control Language/Advanced Process Manager Reference Manual** Issue Date: **7/93**

Publication Number: **AP27-410**

Writer: **J. Kennedy**

COMMENTS: _____

_____

_____

_____

_____

_____

_____

_____

RECOMMENDATIONS: _____

_____

_____

_____

_____

_____

_____

_____

NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

TELEPHONE _____ FAX _____

(If returning by mail, please tape closed; Postal regulations prohibit use of staples.)

Communications concerning technical publications should be directed to:

Automation College
Industrial Automation and Control
Honeywell Inc.
2820 West Kelton Lane
Phoenix, Arizona 85023-3028

FOLD FOLD

From: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE USA

BUSINESS REPLY MAIL

FIRST CLASS **PERMIT NO. 4332** PHOENIX, ARIZONA

POSTAGE WILL BE PAID BY ....

Honeywell

Industrial Automation and Control
2820 West Kelton Lane
Phoenix, Arizona 85023-3028

Attention: Manager, Quality

FOLD FOLD

Cut Along Line

Additional Comments:

Honeywell

Industrial Automation and Control
Honeywell Inc.
16404 North Black Canyon Highway
Phoenix, Arizona 85023-3099

Helping You Control Your World