Honeywell

Control Language Multifunction Controller Reference Manual

PC27-510

Total Plant

Honeywell

Implementation
Hiway Gateway - 2

Control Language Mulifunction Controller Reference Manual

PC27-510
Release 520
10/96

TotalPlant

Copyright, Trademarks, and Notices

Revision 03 - October 20, 1996

About This Publication

This publication provides reference information about the Honeywell Control Language for the Multifunction Controller and Advanced Multifunction Controller (CL/MC).

This publication supports **TotalPlant** Solution (TPS) System network Release 520. TPS is the evolution of TDC 3000^{X}.

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 **TotalPlant** *Solution (TPS) System publications for information related to CL/MC.*

1.1 PURPOSE/BACKGROUND

This publication provides reference information about Honeywell's Control Language for the Multifunction Controller (and Advanced Multifunction Controller). CL/MC is used to build custom-control strategies that cannot be accommodated by standard TPS PV/Control Algorithms. This manual **does not** provide instruction on how to transform a particular control strategy into a CL/MC structure; rather, it outlines the rules of CL/MC and describes all the components that can be used to build a CL/MC structure that will execute your control strategy.

This manual assumes that you are a practicing control engineer with knowledge of TPS product capabilities—specifically, the Multifunction Controller (MC) or Advanced Multifunction Controller (A-MC) and the data points that reside in them. You should also have an operational knowledge of the Universal Station's TEXT EDITOR, CL/MC PROGRAMS, and DATA ENTITY BUILDER functions that are 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/MC.

1.2 REFERENCES

Because this manual primarily describes the elements and rules with which CL/MC structures are built, other publications are necessary to gain a complete knowledge of how CL/MC relates to the rest of TPS (in other words, how to implement a CL/MC 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/MC Information

System Overview, SW70-500, in the *System Summary - 1* binder—Describes what can be done with CL/MC and briefly describes the major components of some CL/MC structures.

Configuration Data Collection Guide, SW12-500, in the *Implementation/Startup & Reconfiguration - 2* binder—Information here ensures that you have collected the required data for implementing a CL/MC structure.

1.2.2 Publications with CL/MC-Specific Information

Text Editor Operation, SW11-506, in the *Implementation/Engineering Operations - 3* binder—Describes how to use the Text Editor to enter CL/MC structures (CL/MC source files).

Data Entity Builder Manual, SW11-511, in the *Implementation/Engineering Operations - 1* binder—Describes how to use the Data Entity Builder to configure a CL/MC Structure into a data point in an MC.

Control Language/Multifunction Controller Data Entry, PC11-585, in the *Implementation/Hiway Gateway - 2* binder—Tells how a CL/MC sequence program is compiled and added to the TPS System.

Process Operations Manual, SW11-501, in the *Operation/Process Operations* binder—Describes procedure to link and load a compiled CL/MC Sequence program to a Process Module Data Point residing in a Multifunction Controller.

1.3 CL/MC OVERVIEW

CL/MC structures can be used to build a custom-control strategy; they are used to augment or replace the standard TPS algorithms. The CL/MC structures are the Sequence Programs, Abnormal Condition Handlers, and Subroutines.

> **NOTE**
>
> Structures can be independently compiled and, therefore, you may hear them referred to as compilation units.

The following are the general steps required to build and implement a CL/MC structure (refer to heading 1.2, REFERENCES, for publications associated with the **BOLD-FACED** functions in the following steps). A more complete description of the steps is given in the *Control Language/Multifunction Controller Data Entry* manual.

1. Use the **DATA ENTITY BUILDER** (DEB) to configure (build) at least the data point ID (name) of the data point that is associated with the CL/MC structure(s).

2. Use the Universal Station's **TEXT EDITOR** to write/enter the CL/MC structure(s); in other words, create the source file(s).

3. Use **CL/MC** in the Command Processor on the US to compile the CL/MC structure. Compiling your source code turns it into an object file that can be executed in the MC (refer to heading 3.2.2 in the *Control Language/MC Data Entry* manual for information on where to put your object file so that loading to a Process Module can be carried out successfully).

4. Use the **DEB** to configure the CL/MC structure into the data point.

5. Use the Process Operations Personality (Process Module Detail Display) to link/load the Sequence program to a Process Module data point in the Multifunction Controller.

RULES AND ELEMENTS OF CL/MC
Section 2

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

2.1 INTRODUCTION TO CL/MC RULES AND ELEMENTS

CL/MC, 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/MC, one arrives (roughly) at the following:

ENGLISH	CL/MC
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

The part of this section under heading 2.2 describes the basic rules (grammar) and elements (words) of CL/MC. The remainder of the section (headings 2.3, 2.4, and 2.5) describes more complex elements of CL/MC (that is, the "phrases" and "clauses" of CL/MC). 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 particular 4-part format, as follows:

Definition—a brief "what is it" discussion.

Syntax—elements, statements, and structures are built following a specific form; the form specification is called syntax. Another word for syntax is grammar, as previously mentioned. The form of anything you want to build in CL/MC must EXACTLY follow the syntax, so that your 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 nonobvious 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/MC 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/MC Statements and Structures.

2.2.1 Character Set Definition

The CL/MC 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 2.2.7.6—Reserved Words Definition)
- Numbers
- Quotes
- Special symbols (such as +, -, /, *)

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 not required between elements except 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 Sections 4 and 5 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:

```
IF hi > foo THEN                (SET foo = hi;
&           GOTO jail)
ELSE SET foo = low              -- NOTE: no continuation
jail:
&    EXIT
END good
```

The following examples are all invalid:

```
gaol:
       EXIT                     -- must use continuation here
IF fred > 3 THEN (SET x = y;
          GOTO gaol)            -- 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 diagram is entered on the left side with the item you want to build 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 diagram in another part of the manual. Rectangles with curved ends are reserved words in CL/MC 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/MC Statement.

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

2.2.5 CL/MC Restrictions

This section lists the restrictions placed on the language by either the CL/MC compiler or some other element of the **TotalPlant** Solution (TPS) System.

2.2.5.1 Length of Identifiers

The following restrictions on identifiers (see heading 2.2.7) are imposed by the TPS Universal Station:

- 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.

- Parameter identifiers can be no longer than 8 characters.

- Sequence program identifiers can be no longer than 8 characters.

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

- Enumeration-state identifiers can be no longer than 8 characters.

The following restrictions are imposed by the File System:

- Enumeration type identifiers can be no longer than 8 characters, must begin with an alphabetic character, and cannot contain ISO 646 foreign characters (refer to heading 2.2.5.2).

- Words in messages must be separated by blanks. Words can be a maximum of 8 characters long. There can be a maximum of 7 words in a message.

2.2.5.2 ISO 646 Compatibility

The CL/MC 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/MC 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/MC. (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/MC 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.
```

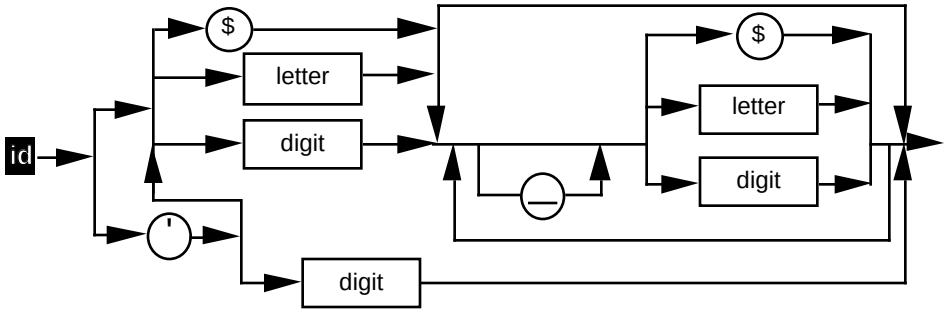
2.2.6.2 Incorrect Examples of Comments

```
-- This is a long comment such as you might use
&  to prefix a subroutine.  It is faulty because a comment cannot
-- span source lines.
```

2.2.7 Identifiers

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

2.2.7.1 Identifiers Syntax



3825

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 2.2.7.7—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.

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 use of Honeywell-supplied standard identifiers and other objects whose names begin with or contain dollar signs. Within a CL/MC program, there is no restriction on using identifiers that begin with dollar signs; however, the CL/MC compiler does not let you create any such object that is visible outside the program (as, for example, the sequence program name).

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)
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
```

2.2.7.4 Box Data Point Identifiers

A box data point is a data point associated with a process-connected box, such as a Multifunction Controller (MC). It represents box parameters that are visible to TPS components, including CL/MC programs that are running in the box. These parameters can include internal variables of the box.

A program's view of box data-point parameters varies, depending on whether the program executes inside the box (MC sequence program) or outside the box (block in the AM). Using an MC Box data point as an example, the parameter DESC (an HG parameter, described in the *Hiway Gateway Parameter Reference Dictionary)* is visible from outside the MC, but not from inside the MC; the MC box parameter DI(nn) (described in Table 4-1) is visible from inside, but not from outside the MC; and the State of an internal MC timer is seen from inside as TM(nn).STATE (described in Table 4-1) but from outside as TMST(nn). In general, parameters visible from outside the box are described in the *Hiway Gateway Parameter Reference Dictionary*. Parameters visible from inside the box are described at heading 4.5. Also, see the examples at heading 2.2.7.5.

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

$HYnnBmm

where **nn** is the hiway number, and **mm** is the box number on the hiway.

2.2.7.5 Box Data Point Identifier Example

```
TM(03).STATE           -- internal (MC) view of timer state
$HY01B25.TM(03).STATE  -- one MC accessing a timer's state
                       -- in a different MC through the C-LINK
```

2.2.7.6 Reserved Words Definition

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

Table 2-2 — CL/MC Reserved Words

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

There are also a number of predefined identifiers in CL/MC. 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, insertion-point names, and built-in function and subroutine names.

Predefined Discrete Types—The following type and its two 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
Day_Time	-- Built-in Function
Exp	-- Built-in Function
Int	-- 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
Number	-- Type name
Off	-- Logical state name
On	-- Logical state name
Round	-- Built-in Function
Set_Time	-- Built-in Subroutine
Sin	-- Built-in Function
Sqrt	-- Built-in Function
Sum	-- Built-in Function
Tan	-- Built-in Function

2.2.7.7 Special Identifiers

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. 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/value.

2.2.7.8 Special Identifiers Examples

```
EXTERNAL 7              -- invalid
EXTERNAL '7             -- OK
EXTERNAL ' xyz          -- invalid, space follows the apostrophe
EXTERNAL 'xyz_          -- invalid, break character, "_", cannot
                        -- be the last character of an identifier
```

2.2.7.9 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 four groups of identifiers that can be named: custom data, 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
   - Logical
   - Enumerations

   Objects are
   - Local variables
   - 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 can't.

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

   Scopes are
   - Sequence Programs
   - Subroutines
   - Phases
   - Abnormal Condition Handlers
   - Steps
   - Functions

   Most things can be declared in only a few of these scopes. For example, a Step can contain only label declarations and a Function can contain only argument declarations.

   Scopes can sometimes be nested. A Step must be within a Phase, and a Phase within a Sequence Program. Local Subroutines of a Block or Sequence Program are considered to be nested within that program unit; a Subroutine can, in turn, contain Functions.

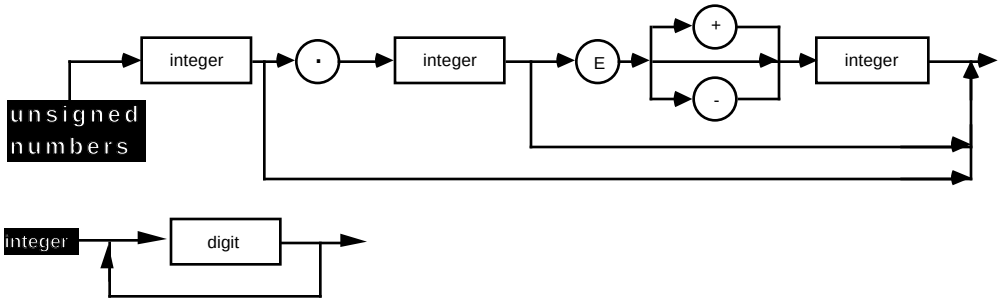
When two scopes are nested, an identifier in an inner scope hides anything in an outer scope that has the same identifier and the same class (Program Unit, Data Type, or Object). For example, a Function argument called X hides any local variable called X in the main program.

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, except Parameter Lists, 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.

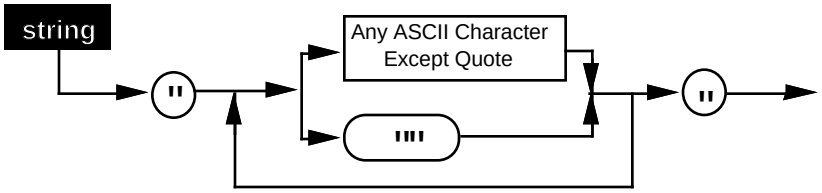
2.2.8.3 Numbers Examples

```
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 one or more characters enclosed at each end by quotation marks ("). In CL/MC, strings are used in only the SEND statement (see heading 3.2.14).

2.2.9.1 String Syntax



2.2.9.2 String Description

Any printable character can appear in a String (String of legal CL/MC characters). If a quotation mark appears in a String, it must be written twice. Spaces are used as word delimiters only.

Subject to the restriction that no basic element can overlap the end of a line of source text, a String can be of any length; however, individual words within a string are limited to eight characters or less since each word in the SEND statement string is assigned as an entry in the HG Library (see heading 4.6).

2.2.9.3 String Examples

```
"This is a String"  -- a String
"&@$?*! system"     -- can contain any printable characters
"He said ""hello""" -- he said "hello"
"A"   " "   """"    -- three Strings of length 1
```

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

2.3 CL/MC DATA TYPES

This subsection describes the types of data that CL/MC can manipulate.

Two kinds of data types are built into the language: 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/MC are of the single type, Number. This type is conceptually a subset of the real numbers and is internally implemented in floating point.

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

Numeric data point parameters are represented in single-precision floating point.

2.3.2 Time

The only use of Time values in CL/MC are in the WAIT statement, the PHASE heading's ALARM clause, and the built-in subroutine Set_time. Refer to Section 4 for details.

The minimum time resolution is one second, and noninteger time expressions are truncated to the next lower integer value (for example, 2/3 MINS becomes 0 minutes). Time expressions can be in MINS or SECS, but not both.

2.3.2.1 Time Expressions Examples

```
SET t1 = AVG (n,m) HOURS
Local sked at NN(1)
   ...
SET sked = 20         -- set a numeric variable
WAIT (sked-3) SECS    -- wait 17 seconds
WAIT (sked/60) MINS   -- wait 0 minutes
```

2.3.3 Discrete Data Types

Real-world values are either continuous or discrete. CL/MC expresses all continuous values with the type, Number. Discrete values are expressed by the type Logical, and by 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 types

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

A few Enumeration types are known to the CL/MC compiler and are predefined by the compiler, rather than by the system; however, there is no visible difference between a compiler-defined type (i.e., the predefined discrete-type Logical) and system-defined type with states Off/On.

The Compiler-defined Enumeration type (predefined discrete-type Logical), is listed under heading 2.2.7.6. Variables of Enumeration Types, can be declared in CL/MC 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.

The only operations defined on Enumeration types are assignment and comparison for equality and inequality.

2.3.3.3 Logical 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 only represent the state of a discrete variable. It does not represent truth or falsity. In CL/MC, 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/MC
	VAR flag: Boolean; x: real;	LOCAL flag: LOGICAL LOCAL x: NUMBER
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/MC 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/MC method A is just as efficient as Pascal method A and is easier to read.

Pascal method B and CL/MC 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 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/MC. A data point is like a Pascal RECORD. Parameters are identified by dot notation; that is, the point name is followed by a dot, which is then followed by the parameter name.

2.3.4.1 Data Points Example

```
A100.PV — parameter PV of data point A100
```

2.3.4.2 Bound Data Point

Every CL/MC program must be bound to a particular data point in order to execute. Sequence Programs (for discontinuous control) are bound to process module data points. The bound data point is identified in the heading of the CL/MC program to which it is to be bound, by specifying the point ID.

CL/MC programs can refer to data points other than the bound data point, as long as those other data points are declared in external declarations, or are indirectly referenced. Parameters of such points are accessed by dot notation, as previously described; however, parameters of the 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.

2.3.4.3 Aliasing

Aliasing occurs when the same datum is accessed through different names. Aliasing may arise in CL/MC because data points can be referred to in several ways: directly (by being named in an EXTERNAL declaration) or implicitly (the bound data point).

CL/MC forbids aliasing. No program can be bound to any data point that is declared in an EXTERNAL declaration, or to which the program indirectly refers.

2.3.5 Arrays Data Type

In the MC, arrays can be of only type Logical or Number, and can have only one dimension. The index type must be Number.

A local variable of type Array can be indexed by any arithmetic expression. If the result of the index expression is not an integer, it is truncated to the next integer. Parameter arrays can be indexed only by a constant.

2.3.5.1 Arrays Examples

```
coeff(6)     -- one dimension
pump(x+y*z)  -- indexed by expression
```

2.3.6 String Data Type

String variables are not supported; string literals are allowed only in the SEND statement.

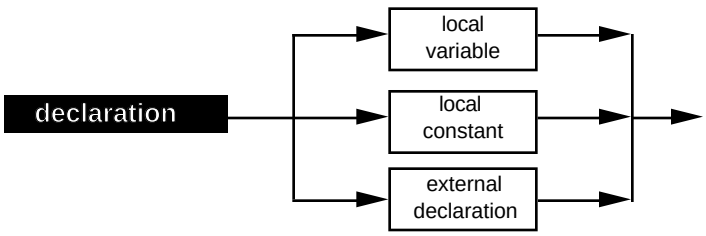
2.4 VARIABLES AND DECLARATIONS

This section describes the declaration, use, and scope of functions, local constants, and local, external, and parameter variables.

Local variables are owned and defined by a CL/MC program. External variables are data points that are defined outside CL/MC. Parameter variables are parameters of the bound data point or of some external variable.

All declarations must precede any executable statements in their program. In addition, function definitions must follow the declarations of any variables that they use.

2.4.1 Variables and Declarations Syntax

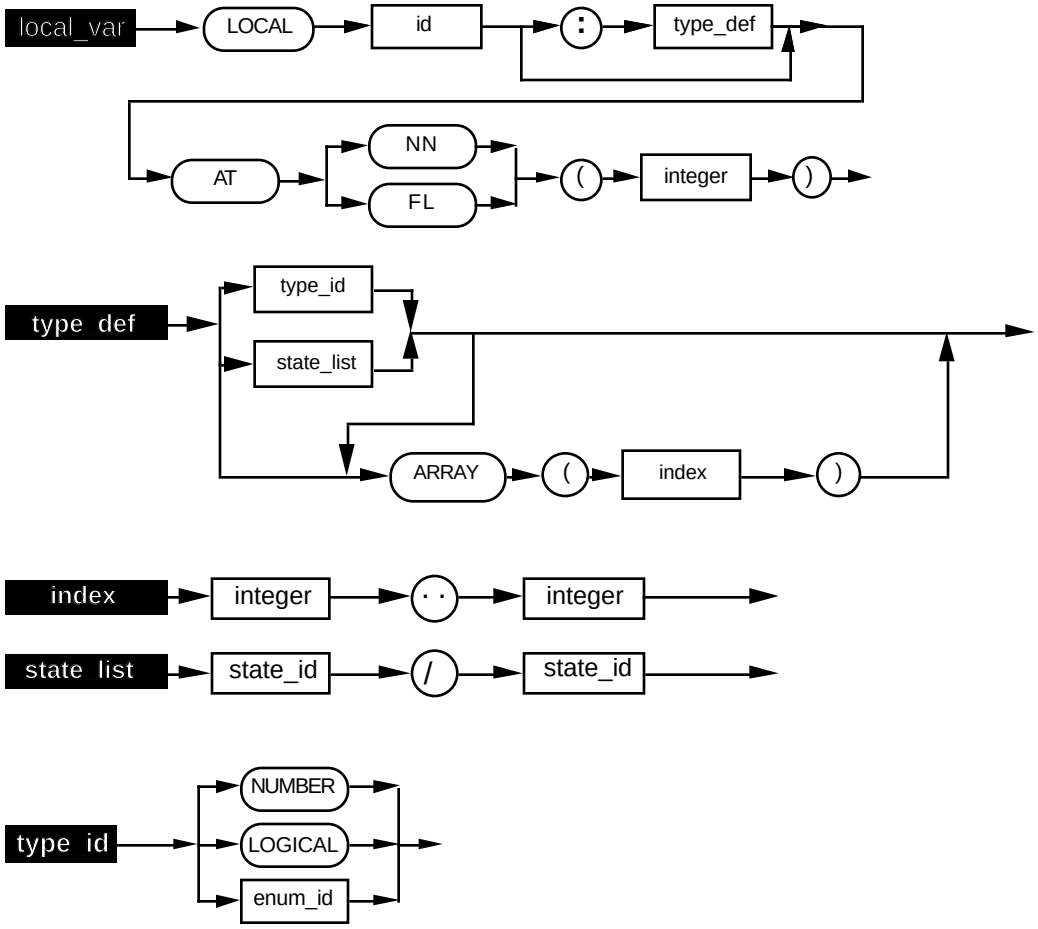


2.4.2 Local Variables

Local variables are declared in the heading of a sequence program. The local variable is an MC numeric or flag, and therefore exists outside the sequence.

Local variables are visible to only the program they are declared in and to any local subroutines belonging to that program.

2.4.2.1 Local Variables Syntax



2.4.2.2 Local Variables Description

LOCAL variables are of only type number (default), logical, or 2-state enumeration. The enumeration type can be located in only a flag variable and is restricted to either a state name list containing two states or the predefined enumeration types used for MC data point parameters (see heading 4.3.3.10) that have two states.

One-dimensional arrays of LOCAL variables can be specified. The array index must be a number declared by naming lower and upper bounds. The lower bound is the left-most integer in the index. The upper bound is the right-most integer in the index and its value must be greater than that of the lower bound.

The AT clause defines the location of the local variable and must name one of the MC's numeric (NN) or flag (FL) variables, using its box data-point-parameter identifier; the compiler places the local variable at that location. If the local variable is an array, its first element is placed at the named location and the remainder of its elements occupy contiguous variables in ascending order from that named.

Note that when a type other than logical is given to a local variable, it is no longer compatible with logical, even though its hardware location may be a flag location. When the variable is used in a SEND statement, the values are displayed in terms of the state names of the type in the declaration.

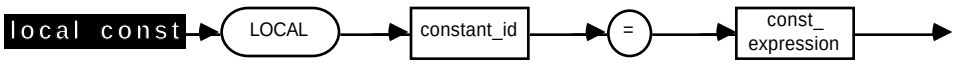
2.4.2.3 Local Variables Examples

```
LOCAL mu: LOGICAL AT FL(12)           -- MC flag variable
LOCAL ka AT NN(04)                    -- MC numeric variable
LOCAL jo: ARRAY(5..9) AT NN(80)       -- MC numeric array
LOCAL temp1 : open/close AT FL(01)    -- State name lists and
LOCAL temp2 : STATE AT FL(02)         -- enumeration-type IDs in
                                      -- local declarations
```

2.4.3 Local Constants

Local constants of the data type Number can be declared.

2.4.3.1 Local Constants Syntax



2.4.3.2 Local Constants Description

Local constants cannot be modified.

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

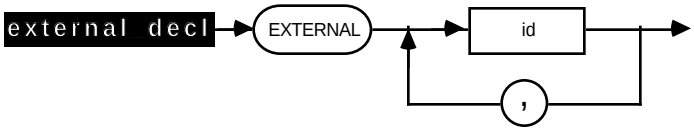
2.4.3.3 Local Constants Examples

```
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
```

2.4.4 External Data Points

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

2.4.4.1 External Data Points Syntax



2.4.4.2 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 MC data points (including points in other MCs on the same C Link). Data points in the HG and other LCN modules cannot be referenced.

2.4.4.3 External Data Points Examples

```
EXTERNAL anp049hx, AX_001
EXTERNAL $HY03B22             -- Box Data Point
```

2.4.5 Process Modules Definition

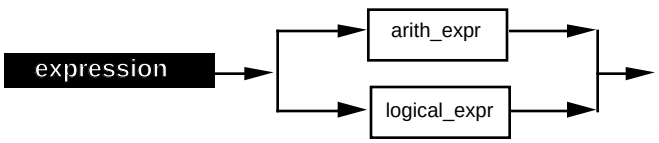
A process module is a data point that is used as the platform for sequence execution. Each sequence program, before it can be executed, is bound (loaded) to a specific process module. The process module is the bound data point of a sequence program. Refer also to heading 2.3.4.

Certain parameters of the related box data point are made to appear (to the CL/MC compiler) as parameters of the process-module data point. This enables convenient access to box numerics, timers, flags, I/O slots, and controller slots. Refer to Section 4 for details.

2.5 EXPRESSIONS AND CONDITIONS

This section describes the formation of arithmetic expressions, logical expressions, and conditions, which are an extended variety of logical expressions used in conditional tests.

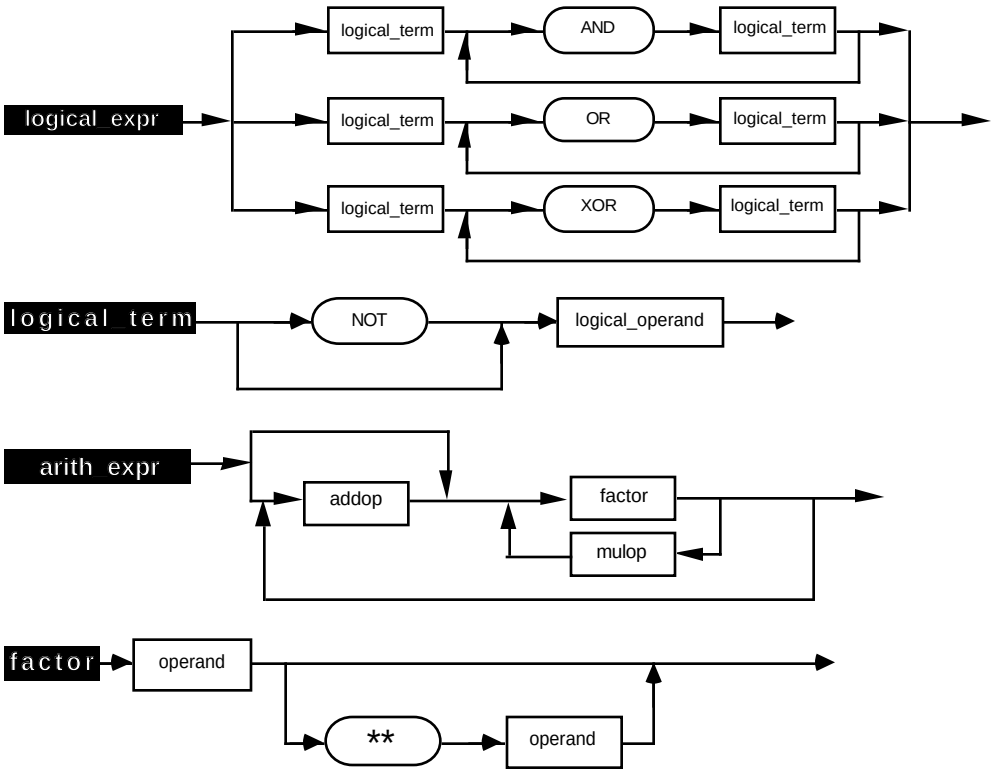
2.5.1 Expressions and Conditions Syntax



2.5.2 Arithmetic and Logical Expressions

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

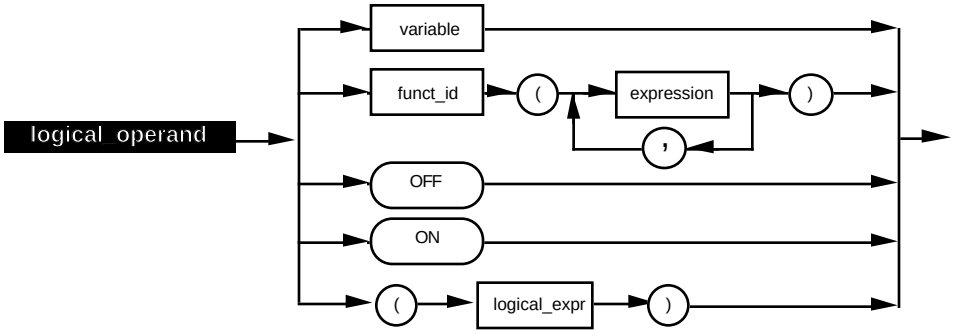
2.5.2.1 Arithmetic and Logical Expressions Syntax

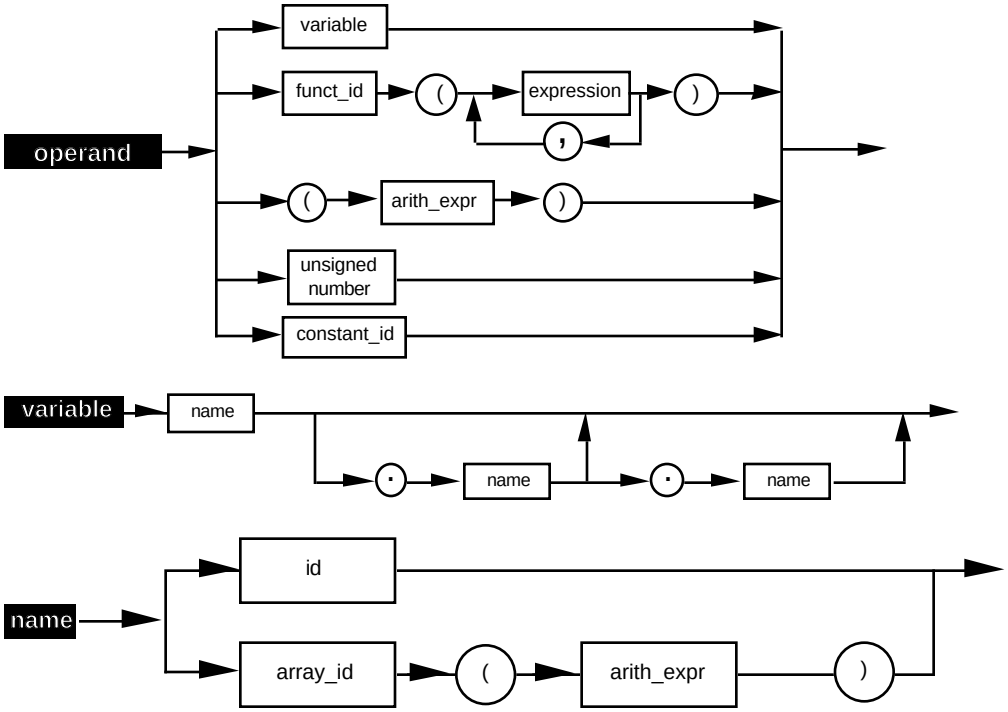


2.5.2.2 Operand Definition

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.

2.5.2.3 Operand Syntax





2.5.2.4 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 2-4; higher priority means closer binding.

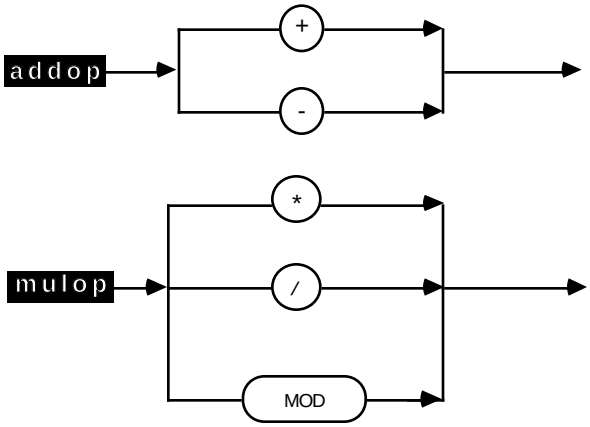
Table 2-4 — Operator Priorities

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

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.

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 (see also heading 2.5.3.7).

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 exponentiation, or power operator (**) works as follows:

If X > 0, the result is as expected
If X = 0, the result is always 1; the expected result would be 0, but the MC's calculation uses the formula: $X^{**}Y = X^Y = e^{y * \ln x}$, X>0.
If X < 0, the result is |X| ** Y.

Exponentiation also produces a small difference in the calculated result if both X and Y are integers; for example $2^{**}10 = 2^{10} = 1023.9999$, instead of 1024.

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 2-5. Logical operators can only take operands of type LOGICAL (also see Section 2.5.4.6).

Table 2-5 — 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
```

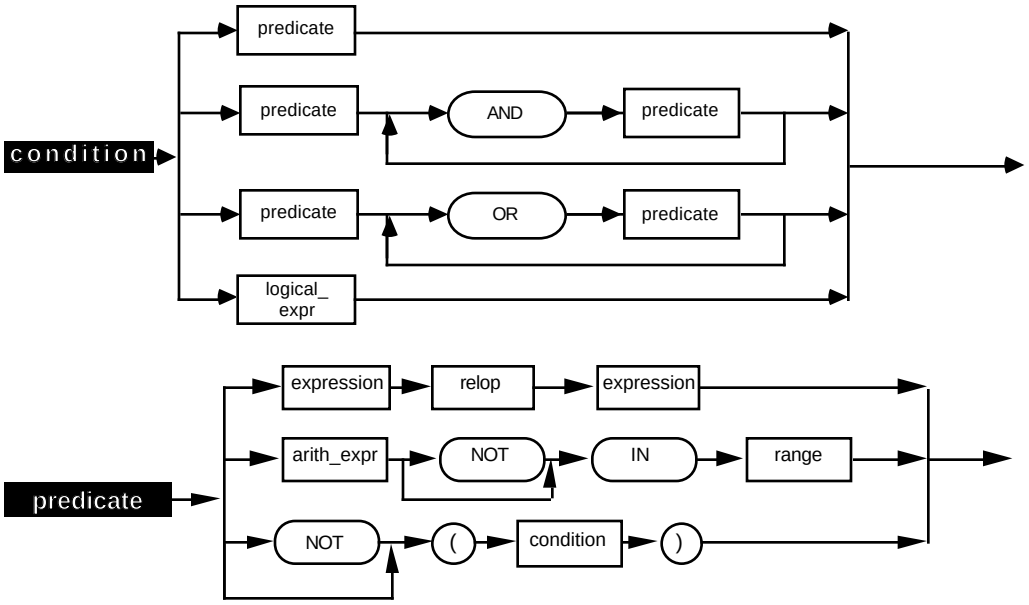
A work-around can be used; write the XOR in terms of AND and OR. For example, the above statement could be rewritten as follows:

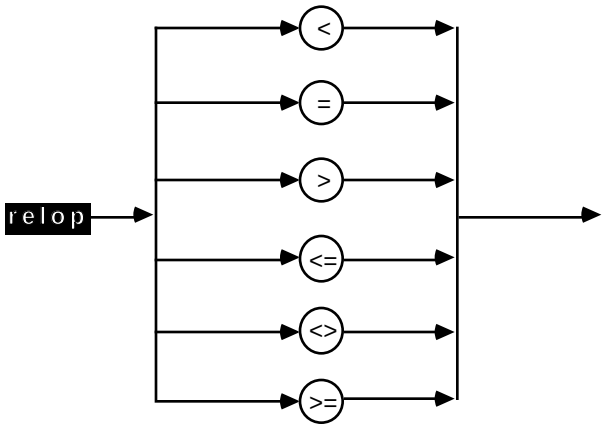
```
      IF ((x = 1 OR y = 1) OR z = 1) AND NOT
&          ((x = 1 OR y = 1) AND z = 1) THEN ............
```

2.5.3 Conditions

A condition is a formula that expresses a truth or falsehood. It is an enhanced form of an 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.

2.5.3.1 Conditions Syntax





2.5.3.2 Conditions Description

A condition can be a logical expression, a relation between two expressions, a range test, 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 ...
```

2.5.3.3 Relations Definition

The relational operators are shown in Table 2-6.

Table 2-6 — 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 only for expressions of type Number. The relations of equality and inequality are defined for discrete types. No relations are defined for arrays taken as a whole.

2.5.3.4 Range Tests Definition

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

2.5.3.5 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/MC STATEMENTS
Section 3

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

3.1 INTRODUCTION

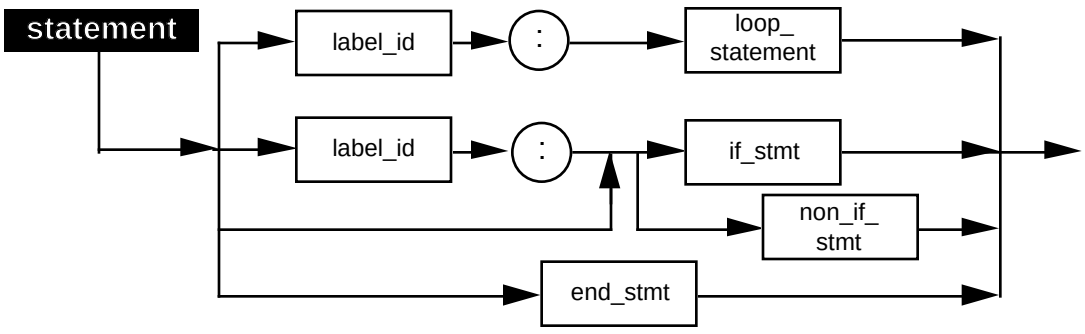
This section describes CL/MC statements. A statement defines a single, simple action to be performed within a CL/MC structure. CL/MC statements can be grouped into two major categories, according to function: **Program Statements** and **Embedded Compiler Directives**.

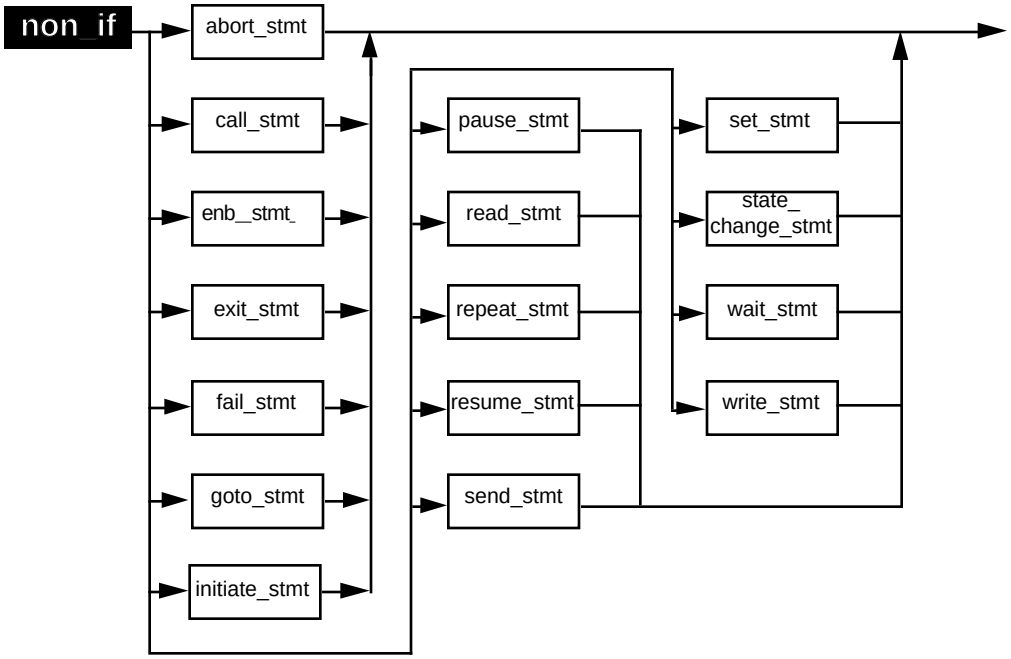
3.2 PROGRAM STATEMENTS DEFINITION

CL/MC 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 the state-change statement.
- Control statements, which establish program conditions or direct the flow of control 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

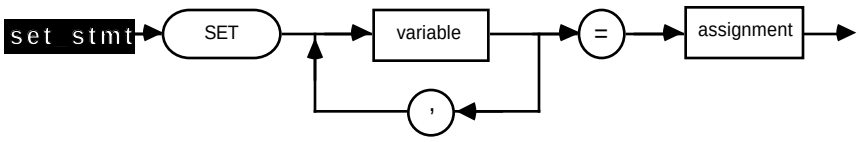
```
lab_01:          CALL test (x, y, z)
     IF x > y THEN    (SET x = z;
& badlabel:           SET z = y)     -- NOT VALID
```

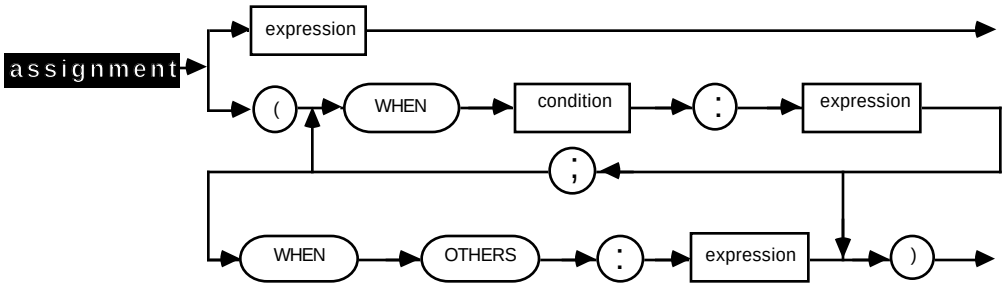
In this example, **badlabel** appears on a continuation line; therefore, it is invalid.

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 runtime error. A maximum of 16 parameters can be referenced with one SET statement.

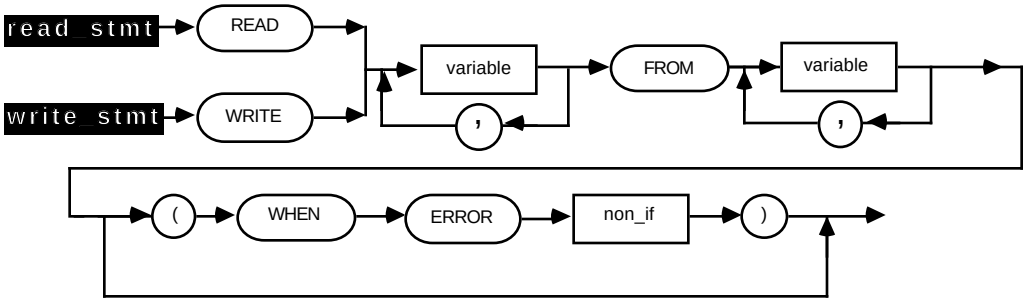
3.2.3.3 SET Examples

```
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 color = (WHEN x > lim: red;     -- conditional SET
&         WHEN x < lim-db: green;
&         WHEN OTHERS: yellow)
IF A100.MODE <> PMAN                -- testing mode before change avoids
& THEN SET A100.MODE = PMAN         -- unnecessary set of slot change
                                    -- flag
```

3.2.4 READ and WRITE Statements

These statements are used to access remote variables in other MCs on the same C-Link and test to see 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 to local variables (in this MC) from remote variables (in other MCs). The WRITE statement writes to remote variables (in other MCs) from local variables and/or numeric constants. The number of items on the left-hand and right-hand sides of the FROM must be equal.

The number of variables allowed in a READ statement is limited to 16. The number of variables allowed in a WRITE statement depends on the type of the variables: numeric variables, 15; flag variables 16; combined numerics and flags in a single WRITE 16 with no more than 14 numerics.

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—so that one or more variable cannot be correctly transferred—the statement in the error clause is executed. If there is no error clause and a communication error prevents a variable transmission, the calling sequence program is failed.

These statements are preemption points. A READ or WRITE statement without an error clause differs from the corresponding SET statement only in that it is a preemption point.

If variables in the other MCs are accessed by their tagnames, these tagnames must be listed in the EXTERNAL declarations. If the variables are accessed by their system IDs, the box data point ID (i.e., $HyxxByy) must be included in the EXTERNAL declarations. You should read all of the topics under heading 4.5.3 to understand methods of parameter access with CL/MC.

Only the parameters shown in Table 3-1 can be accessed through the Read/Write statements. In the table, R indicates that read access is allowed, W indicates that write access is allowed, X indicates that no access is allowed, and — means that the point has no such parameter.

Table 3-1 — Parameters Accessible through the C-Link

Parameter Kind of point	OP	PV	PVP	STATE	SP	SPP
Analog Input	—	R	R	X	X	X
Analog Output	R	—	—	—	—	—
Counter Input	—	X	—	R	—	—
Digital Input	—	R	—	—	—	—
Digital Output	R	X	—	—	—	—
Flag	—	R/W	—	—	—	—
Logic Block	—	R	—	—	—	—
Numeric	—	R/W	—	—	—	—
Regulatory Ctrl	X	R	R	—	R	R
Timer	—	X	—	R	X	—

3.2.4.3 READ and WRITE Examples

```
LOCAL a AT NN (01)
LOCAL b AT NN (02)
LOCAL c AT NN (03)
LOCAL d AT NN (04)
LOCAL e AT NN (05)
EXTERNAL A100, B100, timer1
LOCAL temp1 : OPEN/CLOSE AT f1(01)
LOCAL temp 2 : STATE AT f1(02)
  ...
READ a, b FROM A100.PV, B100.PV (WHEN ERROR GOTO foo)
READ c, d FROM A100.SP, B100.SP
READ temp2 FROM timer1.state      -- reading a timer state into a
                                  -- local variable
WRITE B100.PV FROM e (WHEN ERROR FAIL)
WRITE B100.PV FROM 27.2
```

3.2.4.4 READ and WRITE Communication Error Handling

A communication error on a WRITE, or any other store, indicates that one or more of the local variables is not actually stored into its corresponding remote variable.

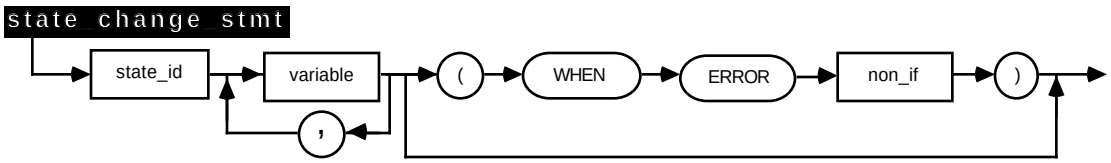
In the MC, if a communication error occurs during a READ, one or more destination variables remain unchanged.

In event of a communications error, the WHEN ERROR clause can be used to start an appropriate action; if there is no error clause, the calling sequence is failed.

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.

3.2.5.1 STATE CHANGE Syntax



3.2.5.2 STATE CHANGE Description

The variables must identify data points that have an OP parameter of a discrete type. The STATE_ID (which is the descriptor you used when you built the point) can be of any discrete type, including 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 Feedback

The WHEN ERROR clause applies to only Digital Composite points 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 only on feedback error; for other types of state change command failures the sequence is failed. The WHEN ERROR clause is executed if any variable in the variable list has a feedback error.

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
Feedback Timeout	WHEN ERROR path is executed	Seq. continues to next statement	Not applicable	Not applicable
Command Failure	Sequence fails	Sequence fails	Sequence fails	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.

Then,

```
stop MOTOR1
open Valve2, Valve3
high 37SW (WHEN ERROR GOTO STEP retry)
```

are valid STATE CHANGE statements.

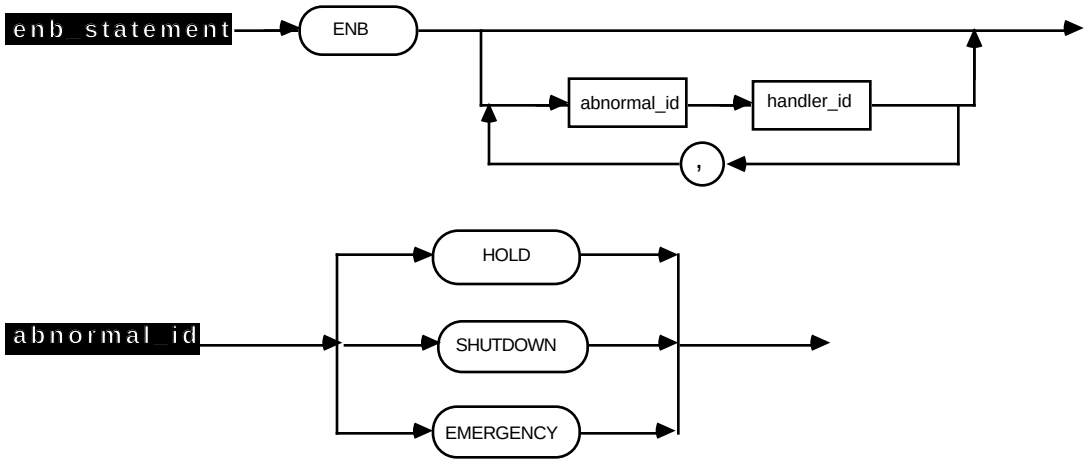
3.2.6 ENB Statement

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 not a preemption point and the new handler list does not take effect until the next preemption point.

3.2.6.1 ENB Syntax



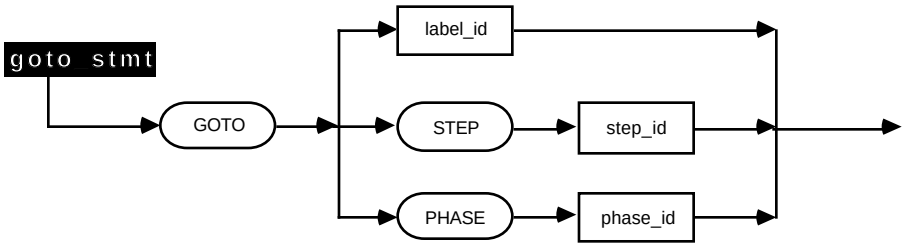
3.2.6.2 ENB Examples

```
ENB HOLD fillhold, EMERGENCY ovenstop
ENB SHUTDOWN tankck
IF 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

In a sequence program, 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 statement cannot be used to exit a subroutine; use EXIT instead.

NOTE

In the MC, 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.

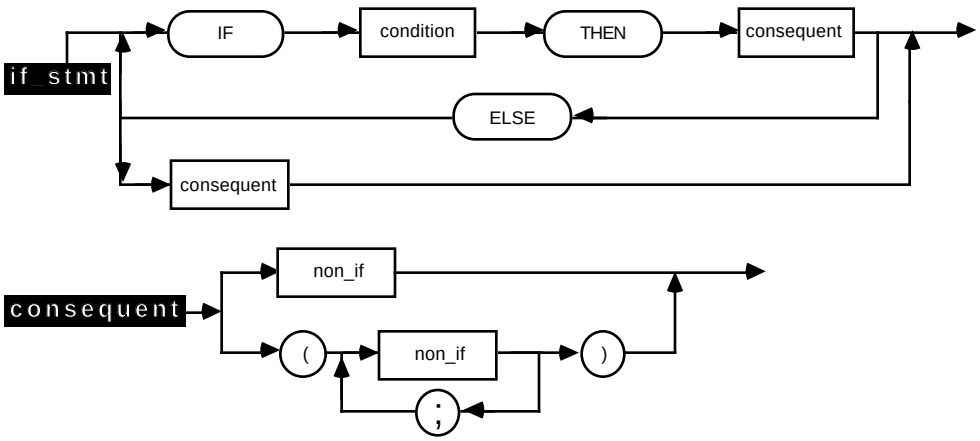
3.2.7.3 GOTO Examples

```
GOTO label            -- simple GOTO
    ...
PHASE test            -- each execution of 'GOTO test' causes a phase
    ...               -- change to be placed in the MC's Sequence
GOTO test             -- Circular List

PHASE test            -- this brings you back to beginning of the code
label:                -- for this phase without adding to the list
    ...
GOTO label
```

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.

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

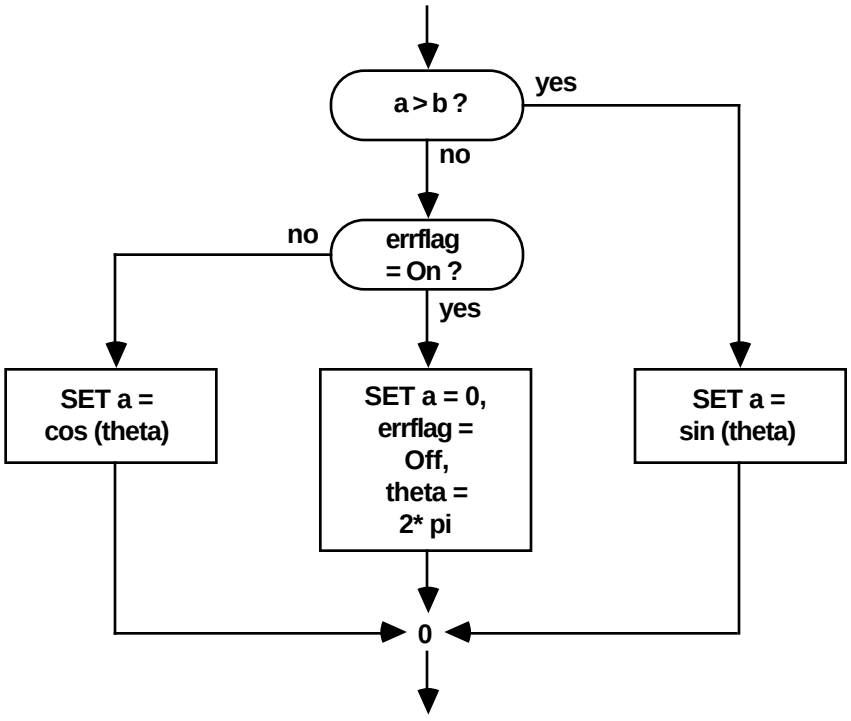
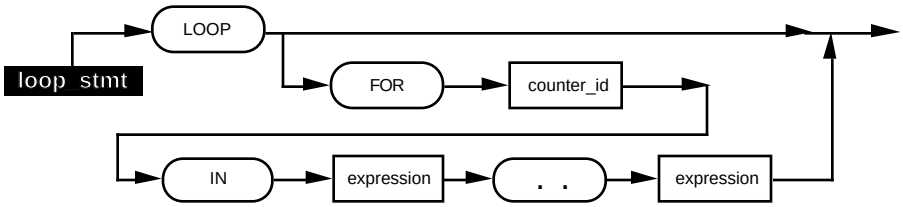


Figure 3-1 — If Statement Flowchart 1528

3.2.9 LOOP Statement

This statement provides loop control in a step or subroutine.

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.

On the MC, the upper bound of the FOR range is dynamically re-evaluated each time the REPEAT statement is executed. 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. Each backward branch on the REPEAT statement causes preemption.

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)
```

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 step, block, or subroutine.

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. REPEAT statements are contextually, although not syntactically, bound to the LOOP statement having the given label.

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 in the MC.

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 four times it is executed, the loop is repeated; the program again stores into x.SP and waits thirty seconds. The fifth 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.

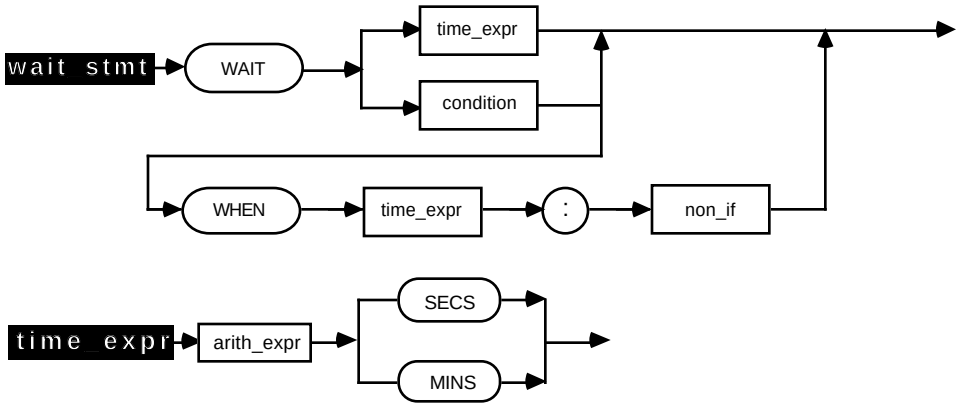
This statement is a preemption point.

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.

3.2.12.1 WAIT Syntax



3.2.12.2 WAIT Description

The "WAIT time_expr" form of the WAIT statement defines a simple timed delay. If the truncated integer value of the time expression is less than 1 or greater than 9999, 1 or 9999, respectively, is assigned.

The time expression cannot contain mixed units; it can be MINS or SECS but not both. Because of asynchronous MC timekeeping, the delay can terminate as early as one time unit before the specified time.

For the "WAIT condition" form of the WAIT statement, the program is suspended until the named condition becomes true. If the WAIT statement contains a WHEN clause, the WAIT is timed out as specified. If the time expires while the condition is still false, the statement in the WHEN clause is executed.

3.2.12.3 WAIT Examples

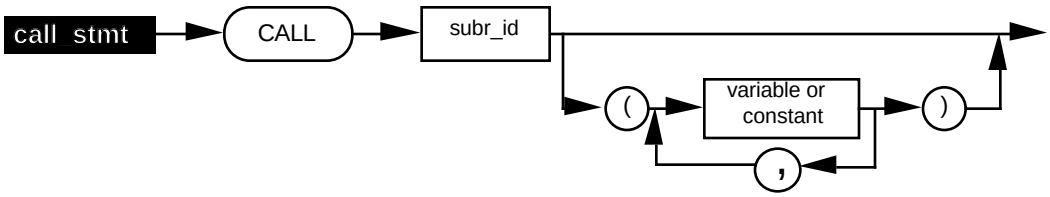
```
WAIT 5 MINS
WAIT x > 5.5
WAIT A_001.PV IN A_001.lolim .. A_001.hilim

boil: LOOP FOR tick IN 1..60  -- minutes
     IF temp >= 100 THEN GOTO done
     WAIT press > 1 (WHEN 1 MINS: GOTO next)
          ...                -- do something to adjust pressure
next: REPEAT boil                -- go test temp again
     SEND: "this watched pot never boiled"
     FAIL
done: ...                       -- ok, continue with process
```

In the last example, note that **temp** is tested once every minute, as long as **press** is less than 1; but **press** is continuously tested.

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 are limited to variables or constants that 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.

In CL/MC, this statement cannot occur within a handler or subroutine.

3.2.13.3 CALL Examples

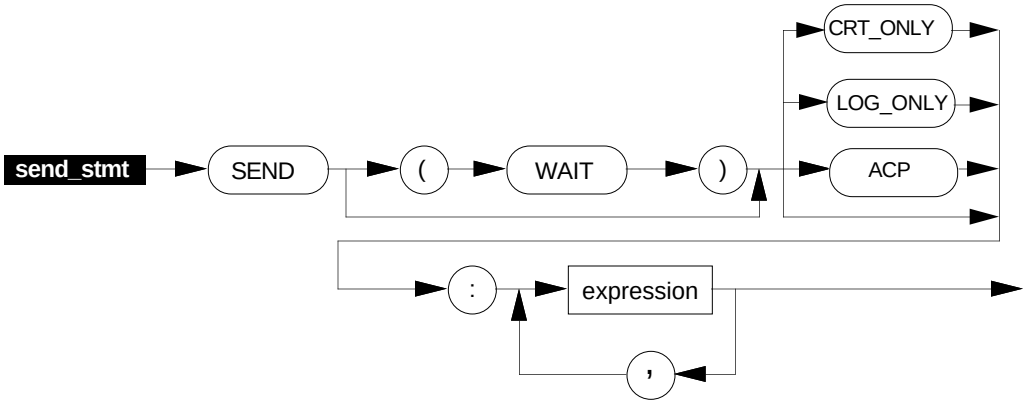
The following CALL examples match the subroutine header definitions at heading 4.5.7.1, Subroutine Arguments Examples.

```
EXTERNAL HG050601                    -- HG digital input point
EXTERNAL HG0502                      -- HG regulatory point
EXTERNAL $HY02B05                    -- HG BOX data point
LOCAL num AT NN(3)
LOCAL flagarr : Logical ARRAY (1..3) AT FL(10)
     . . .
CALL sub1 (HG050601.PV, DI(116).PV   -- digital input PVs
CALL sub2 (num, flagarr)             -- scalar local variable, entire
                                     -- array local variable
CALL sub3 (flagarr(2), 5.5, $HY02B05.NN(19)
                                     -- element of local array, numeric
                                     -- literal, array parameter indexed
                                     -- by a constant
CALL sub4 (HG0502.SP)                -- scalar parameter
CALL sub5 (flagarr(num))             -- ILLEGAL the local array is not
                                     -- indexed by a constant
CALL sub6 (state1)                   -- enumeration state identifier
```

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 four destination options at the left of the colon establishes the message destination as follows:

(none)	-- to home unit MMI CRT and LOG
CRT_Only	-- to home unit MMI CRT only
LOG_Only	-- to home unit MMI LOG only
ACP	-- to assigned Advanced Control Program

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. The WAIT option can be selected with only (none) and CRT_Only.

The "expressions" in the SEND statement are the message items to be sent. Each message item can have a value of any of the types: Number, Logical, String literal, or 2-state enumeration. An array cannot be sent as a whole, but its elements can be individually sent. Tag names cannot be sent in the SEND statement.

SENDing only a null string or string of all blanks is not allowed; a SEND that sends other items in addition to a blanks string is allowed. Each word in a SEND string literal must be eight characters or less (see heading 4.6).

For Destinations (none), CRT_Only, and Log_Only, there can be no more than seven Send items, where a Logical counts as one item, a Number counts as two items, and a String (see heading 4.6) counts as one item. Messages with seven Strings sent to US CRT have the last two characters of the last String truncated. The message correctly appears at the LOG.

The Process Module Data Point parameter ACP identifies an Advanced Control Interface Data Point (resident in a Computer Gateway) that can receive messages from this program. For Destination ACP, there can be no more than 12 Send items, where Logical counts as one item, an integer constant in the range 1..2047 counts as one item, any other number counts as two items, and a String (see heading 4.6) counts as one item.

3.2.14.3 SEND Statement Examples

```
SEND: "Send examples"
SEND (WAIT) CRT_Only: "Value is", LP(1).PV
SEND (WAIT): "Error ", A100.PV, "out of range"
SEND Log_Only: "values are", a, b, "and", c
SEND CRT_Only: "This is a CRT message"
SEND ACP: 1, 37, A101.PV
```

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. Any such statements are aborted when the handler begins to execute.

This is done for two reasons. First, the handler may need to execute a SEND with WAIT. Second, it is likely that when the handler is done, the program will no longer need the confirmation that it was waiting for.

3.2.14.5 Event Initiated Reports from CL/MC

Two types of Event Initiated Reports can be invoked by specially formatted CL/MC 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.

3.2.14.6 Button Primmod Assignment

With Release 520, CL programs and schematic buttons will be able to change the primmod assignments on any of the 40 configurable LED buttons.

A new predefined form of the SEND statement allows the user to assign a Primmod, or clear a previous assignment on a specified configurable LED button.

3.2.14.7 Button LED Control

For Release 520, the Button LED Control function permits US Button LED states to be dynamically changed via an AM-, MC-, or PM-type CL Send statement.

This function allows the red and yellow LEDs on each of the 40 configurable LED buttons to be set to three possible modes (on/off/blink) by using the new predefined form of the CL SEND statement.

CAUTION

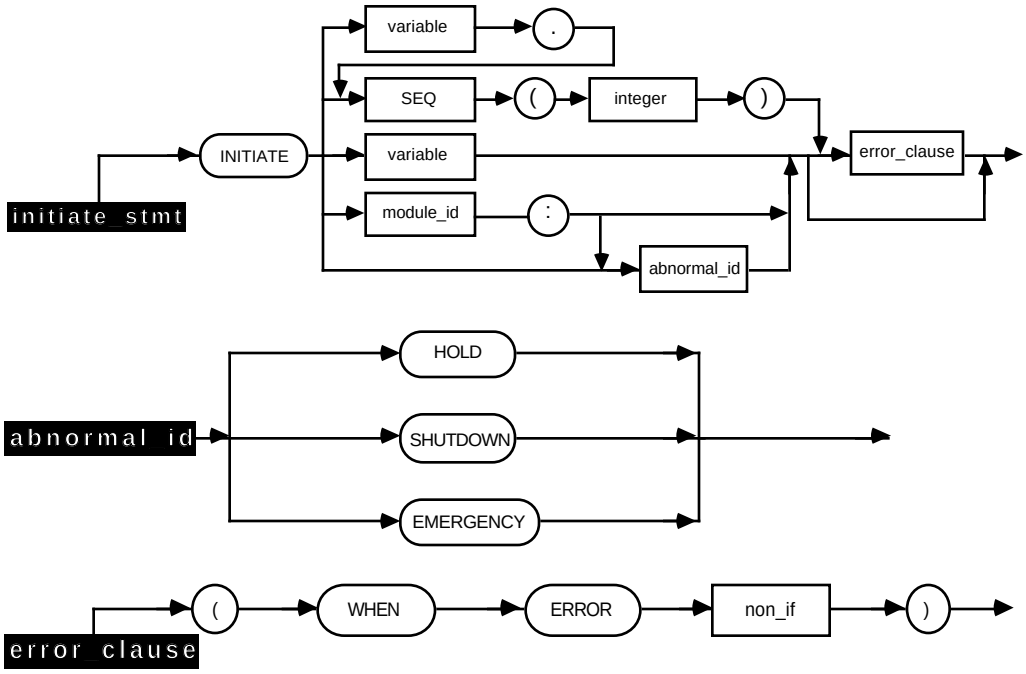
The state of the LEDs on configurable buttons can be changed on the operator's keyboard by use of the new CL message functionality. The Primmod value assigned to configurable buttons can also be changed by using the new actor and CL message functionality. However, whenever a station is reloaded or an Area Database change is made, all configurable buttons will be initialized to the configured values in the Button File.

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 communication errors when initiating a process module in another MC.

3.2.15.1 INITIATE Syntax



3.2.15.2 INITIATE Description: Initiating Data Points and Programs

"INITIATE variable" starts another sequence in this MC; the variable must be a process module data point identifier, and must be declared in an EXTERNAL statement.

"INITIATE SEQ(n)," where n is an integer corresponding to the sequence slot number, starts a sequence in this MC.

"INITIATE variable.SEQ(n)" starts a sequence in another MC; the variable must be an MC box data-point identifier (of the form $HYnnBmm—refer to heading 2.2.7.4) and must be declared in an EXTERNAL statement.

"INITIATE module_id," where the module_id must name a process-module data point, starts the sequence program that is bound to that 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.

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 MCs. In most instances, in case of such error, the requesting sequence is failed.

The WHEN ERROR clause is activated only when a communication error between MCs occurs (the statement in the error clause is executed and the requesting sequence is not failed).

It is a compile-time error to use a WHEN ERROR clause on an INITIATE of your own abnormal condition handler. The compiler will accept a WHEN ERROR clause on an INITIATE of another Process Module in the same MC, but it will have no effect.

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

3.2.15.5 INITIATE Examples

```
EXTERNAL mixer, reactor -- Process module data points
EXTERNAL boiler, oven   -- Process module data points
EXTERNAL $HY02B10       -- Box data point identifier of another MC

INITIATE mixer          -- Starts another sequence in this MC
INITIATE mixer:hold     -- Starts handler of another sequence in this MC
INITIATE EMERGENCY      -- Starts handler in this sequence in this MC

INITIATE $HY02B10.SEQ(10) (WHEN ERROR GOTO retry)
                        -- Starts sequence in sequence slot 10
                        -- in another MC with goto action when error

INITIATE $HY02B10.SEQ(6):EMERGENCY (WHEN ERROR INITIATE SHUTDOWN)
                        -- Starts handler of program in sequence slot 6
                        -- in another MC with handler of this sequence
                        -- to start when error

INITIATE SEQ(2)         -- Starts sequence in slot 2 in this MC
```

3.2.16 FAIL Statement

This statement causes the program to be suspended and enter a soft failure state. 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

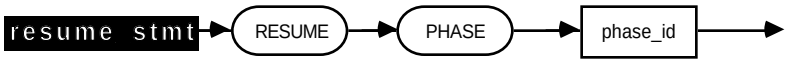
3.2.16.2 FAIL Examples

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

3.2.17 RESUME Statement

This statement can be executed by only the Restart routine of an Abnormal Condition Handler. It causes resumption of the normal sequence at the beginning of a specified phase (which allows preemption to occur).

3.2.17.1 RESUME Syntax



3.2.17.2 RESUME Description

The named phase is resumed at its head.

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

3.2.18.2 EXIT Description

A program termination resulting from the execution of an EXIT statement is considered a normal termination (as opposed to that caused by the ABORT statement). Executing an EXIT statement is equivalent to **falling off the end** of a program.

3.2.19 ABORT Statement

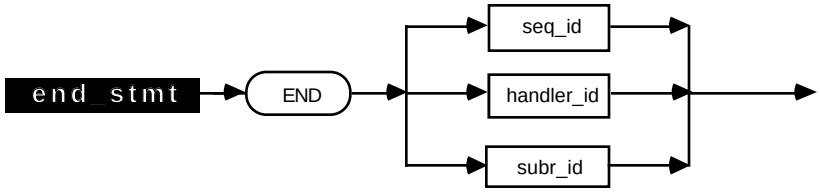
This statement causes program termination. When executed in a subroutine, it terminates both the subroutine and its caller.

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/MC 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/MC 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 effects conditional compilation. There is a global compiler variable called the compiler debug switch, that can be turned on or off when you compile your structure, but cannot be changed during a compilation. See heading 5.3.1.2 in the *Control Language/Multifunction Controller Data Entry* manual.

When the compiler debug switch is off, any source line that begins with %DEBUG is ignored.

When the compiler debug switch is on, each source line that begins with %DEBUG is treated as an ordinary source line with the %DEBUG stripped off.

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 debug switch is on at the time 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 Equipment List object file name includes a .QO suffix, but its entry here is optional. 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/MC Source Example

```
SEQUENCE testmc(point hg05pm06)
%INCLUDE_SOURCE phase1
%INCLUDE_SOURCE phase2
%INCLUDE_SOURCE step3
END testmc
%INCLUDE_SOURCE pmsubrtn
```

3.3.5.2 %INCLUDE_SOURCE CL/MC Listings Example

```
CL V41.00   TESTMC            11/15/91 13:35:48:8946  Page  1

File Line Loc Text

      1       SEQUENCE testmc(point hg05pm06)
      2       %INCLUDE_SOURCE phase1
  1   3       PHASE one
            ^
**NOTE  **  All enabled abnormal handlers are disabled

  1   4       STEP one
  1   5   11  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 testmc
     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  512 blocks

New MC object file:
     File 02055119.MO created

Compilation was on a UP.  Memory limit in words   =  320,000
Words of heap memory used:  31,096

Options Selected:  -NoXRef -UpdateLib

The following source file(s) were referenced:

File          File Path
-----    ----------------------
         NET>DO>TESTMC.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/MC STRUCTURES
Section 4

This section has information specifically about CL for the Multifunction Controller;

4.1 CL/MC STRUCTURES—GENERAL ORIENTATION

CL/MC structures can be independently compiled and therefore are referred to as compilation units. They are composed of a series of headings, data definitions, and/or statements that execute a custom control strategy. The CL/MC structure (sequence program) executes on only the Multifunction Controller or Advanced Multifunction Controller.

The smallest executable instruction in a CL structure is the statement. A program statement is a command to perform a simple action, and is used in CL sequence programs, subroutines, and Abnormal Condition Handlers. Some program statements have the side effect of preempting (suspending) program execution. All program statements are described in Section 3.

4.2 SEQUENCE PROGRAM DEFINITION

A Sequence Program is a set of instructions that details a complete sequence of events in the production of some product. A sequence program is loaded (after compiling without errors) into a process module, which is a named data point in the MC. Sequence programs are loaded from the Process Module Detail Display, using the US Operator's Personality. Heading 3.2.2 in the *Control Language/Multifunction Controller Data Entry* manual specifies which volume (name = &Enn, where nn = hiway number) your program's object file (.MO file) must be in, so that loading the program to a Process Module can be carried out successfully. Refer also to the following headings in the *Control Language/Multifunction Controller Data Entry* manual: 2.3 (Table 2-1), 2.6, 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. The aspects of subroutines that apply to CL/MC are discussed under the general topic of subroutines in Section 4.

Abnormal Condition Handlers are CL/MC 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 Multifunction Controller.

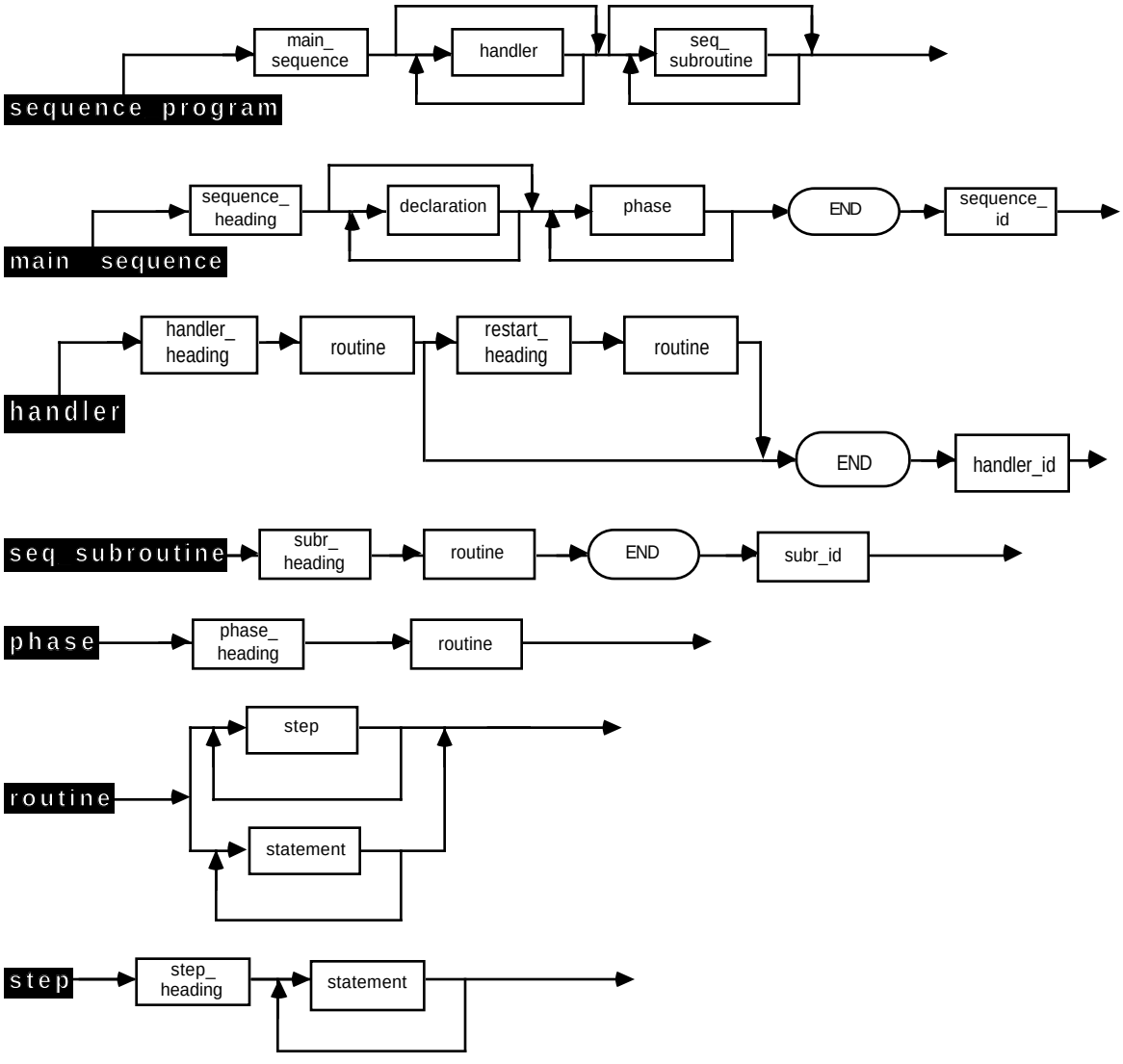
The suspension of a sequence program to allow another program to run is called **preemption**.

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

- Any statement that causes a delay (individually described in Chapter 4)
- Any backward GOTO or REPEAT statement
- The crossing of any STEP or PHASE 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.

4.2.1 Sequence Program Syntax



4.2.2 Sequence Program Description

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

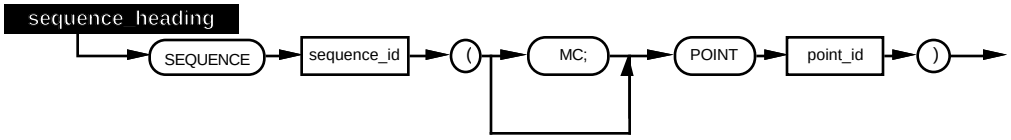
The main sequence is made up of phases; a phase or subroutine consists of one routine. An Abnormal Condition Handler consists of one or two routines, the second (if present) being its Restart routine. Routines (phases, handlers, and subroutines) are made up of steps; however, if a routine consists of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

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

4.2.3 SEQUENCE Heading

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

4.2.3.1 SEQUENCE-Heading Syntax



4.2.3.2 SEQUENCE-Heading Description

The sequence ID is an identifier by which the program is externally known. The point description identifies the bound data point, which must be a process module data point.

4.2.3.3 SEQUENCE-Heading Examples

```
SEQUENCE fred (POINT PM47B)    or
SEQUENCE fred (MC; POINT PM47B)
```

4.2.4 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.

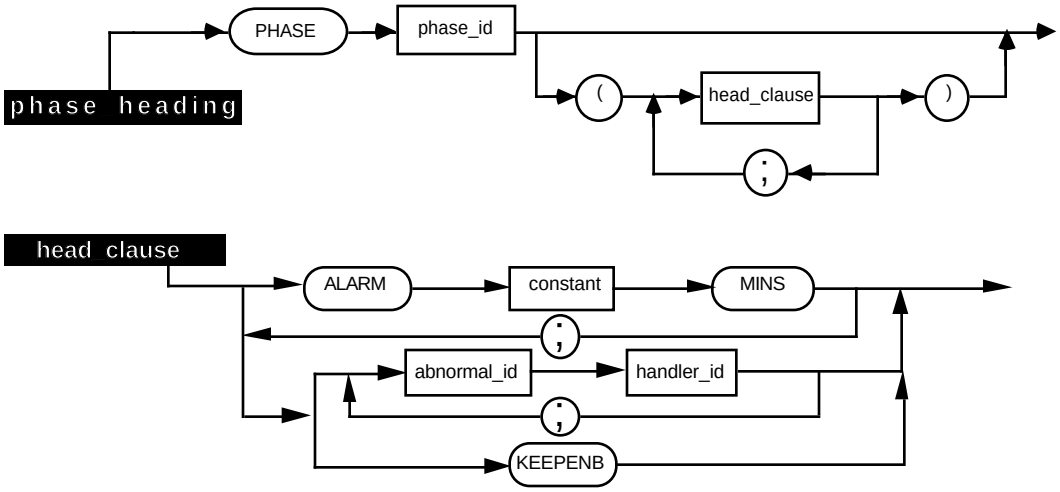
During a phase, a check is made at each preemption point for the occurrence of abnormal conditions. When such a condition occurs, all activities cease and an abnormal condition handler is activated.

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 may begin. Each time a phase boundary is crossed, a phase change is noted in the

MC's Sequence Circular List. (This phase change can be displayed by the Universal Station.) In the new phase heading, the phase-alarm timer can be restarted at a newly specified setting and new abnormal condition handlers can be activated.

4.2.4.1 PHASE-Heading Syntax



4.2.4.2 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. The alarm constant must be an integer in the range of 1 to 9999.

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. A PHASE heading can contain either the KEEPENB clause or named handlers, but not both.

Note that based on program flow, you may not 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.

4.2.4.3 PHASE-Heading Examples

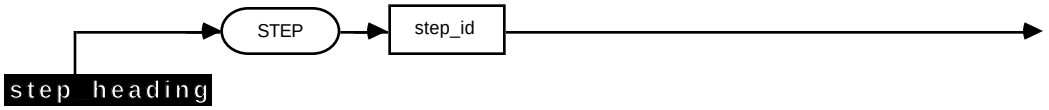
```
PHASE fill_up (ALARM 2 MINS;
&       HOLD fillhold)

PHASE baking (ALARM 30 MINS;
&       SHUTDOWN oven_clr
&       EMERGENCY oven_stp)
```

4.2.5 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.

4.2.5.1 STEP-Heading Syntax



4.2.5.2 STEP-Heading Description

A step change is marked for the sequence.

The STEP heading is a preemption point.

4.2.5.3 STEP-Heading Examples

```
STEP s1
STEP FILL_A
```

4.3 ABNORMAL CONDITION HANDLERS

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

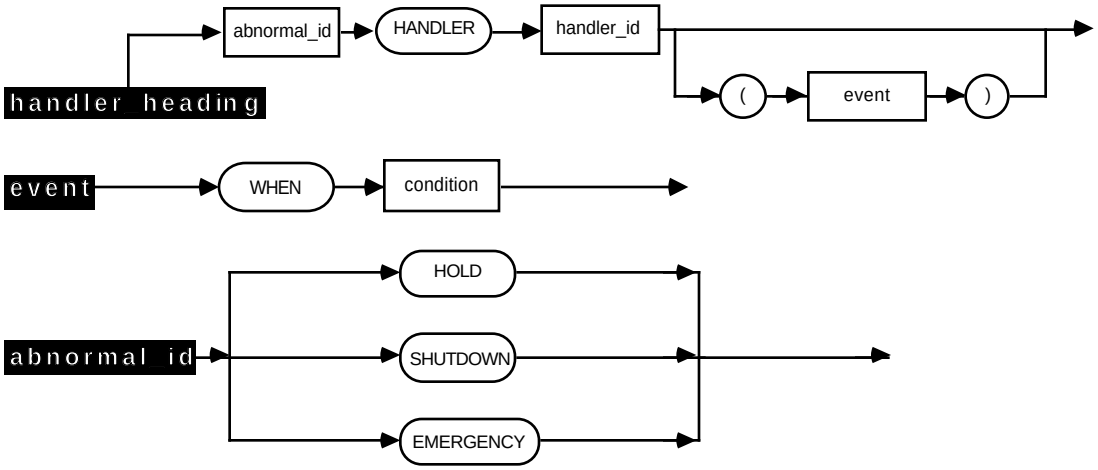
If the operator, or any INITIATE statement (in the same or another program), should attempt to start an Abnormal Condition Handler that is not enabled, the condition is propagated to the next lower priority. That is, if an attempt is made to start EMERGENCY and there is no enabled EMERGENCY handler, the SHUTDOWN handler is started. If that handler in turn is disabled, the HOLD handler is started.

An attempt to start a HOLD handler that is not enabled causes the program to enter the FAIL state, as if a FAIL statement had been executed.

4.3.1 HANDLER Heading

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

4.3.1.1 HANDLER-Heading Syntax



4.3.1.2 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. All handlers are disabled at the end of each phase. 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 ONLY at each preemption point. 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).

Example:

```
SEQUENCE  s  (POINT p)
   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:  "hold handler beginning"
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 runtime 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 retrigger the handler again, causing a loop. (Lower priority handlers are not called while in a higher priority handler.)

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  (POINT MC01S01)

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
      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 Emerg Handler"
END emer1
```

```
SHUTDOWN HANDLER shut1 (WHEN FL(15) = OFF)
   SET 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.

4.3.1.3 HANDLER-Heading Examples

```
HOLD HANDLER cooldown (WHEN Temp.PV > Temp.PVHITP)
```

4.4 RESTART ROUTINES

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.

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.

An INITITATE of a restart section's parent handler from inside the restart section will cause a runtime failure; INITIATING other enabled handlers is allowed.

4.4.1 RESTART Heading

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

4.4.1.1 RESTART-Heading Syntax

4.4.1.2 RESTART-Heading Description

The Restart heading is not a preemption point.

4.5 ACCESSING PARAMETERS

A CL/MC sequence program can access parameters in the MC in which the program is loaded. It can also access some parameters in a different MC, through the C-Link.

Parameters can be accessed in four ways: as a local variable, as a parameter of the bound data point, as a parameter of an EXTERNAL data point, and as a parameter of an EXTERNAL box data point.

4.5.1 Local Variables

NUMBER and LOGICAL variables can be declared as LOCAL variables by using the AT clause to equate them with Numeric and Flag variables of the MC.

4.5.1.1 Local Variables Examples

```
LOCAL tmp AT NN(10)         -- default NUMBER type
LOCAL switch: LOGICAL AT FL(1)
LOCAL table: NUMBER ARRAY(10..20) AT NN(1)
  ...
SET tmp = 15
IF switch = ON THEN SET table(tmp+1) = 100
```

4.5.2 Bound Data Point (BDP) Parameters

All MC parameters are predefined as parameters of the BDP, so that they can be used without any declarations. This method is used only to access parameters in a program's own MC. (See Tables 4-1 through 4-9.)

Some of these parameters behave as data points, having their own parameters in turn. This allows such access expressions as LP(1).SP and AI(3).ALHIFL.

4.5.2.1 BDP Parameters Examples

```
SET NN(10) = 100
SET LP(1).SP = 50
IF PF THEN FAIL
IF DI(1008).PV = On THEN SET FL(05) = Off
```

4.5.3 External Data Point Parameters

Most MC parameters can be accessed as external data point parameters by using the EXTERNAL declaration. This form can be used to access parameters both in the program's own MC and in different MCs on the same C-Link. External data point parameter names are defined in Tables 4-2 through 4-9.

External data point parameters must be in external data points defined by means of the Data Entity Builder. Note that if a Numeric or Flag is defined as an external data point, it is given an artificial PV parameter that contains its value.

Attempts to fetch a PV or OP parameter of a dual-digital obtains only the value of the parameter's upper subslot. Stores to these parameters work correctly.

4.5.3.1 External Data Point Parameters Examples

```
EXTERNAL A100         -- regulatory data point in this MC
EXTERNAL B100         -- regulatory point in different MC
EXTERNAL VALVE    -- digital output
EXTERNAL SubSeq   -- process module in this MC
                  ...
SET A100.OP = 50
READ NN(10) FROM B100.PV
OPEN VALVE            -- state change
INITIATE SUBSEQ: HOLD
```

4.5.3.2 External Box Data Point Parameters

Some parameters in a different MC can be accessed as external box data point parameters by using an EXTERNAL declaration of the predefined box data point name (see headings 2.2.7.4/5—Box Data Point Identifiers/Examples and 2.4.4—External Data Points Definition).

4.5.3.3 External Box Data Point Parameters Examples

```
EXTERNAL $HY02B10     -- Box data point, Hiway 2, Box 10
  ...
WRITE $HY02B10.NN(5) FROM LP(10).PV
READ NN(1) FROM $HY02B10.LP(1).PV
INITIATE $HYO2B10.SEQ(1)
```

For a CL sequence program in the MC to make a direct reference to a PV or OP of a digital point, the code simply names the point and parameter and uses the state text Strings as appropriate.

4.5.4 MC Data Point Parameters

Table 4-1 shows all accessible MC parameters. Bracketed numbers indicate notes at the end of the table.

The MC points ANALOG INPUT, ANALOG OUTPUT, DIGITAL INPUT, DIGITAL OUTPUT, LOOP, and TIMER listed in Table 4-1 (noted with a [1]) have parameters; these are listed in Tables 4-2 through 4-9. Bracketed numbers in Table 4-2 through 4-9 refer to notes at the end of Table 4-9.

The parameters under the SOPL Name heading marked with an * in Tables 4-2 through 4-9 (e.g., PV*) are the default parameters in the SOPL environment; they could be referenced in SOPL without the parameter code (e.g., READ [A1003]), where A1003 is an Analog Input point, would return the PV of A1003 by default). In CL, YOU MUST SPECIFY THE PARAMETER code for all parameters you want to use (e.g., READ A1003.PV). These asterisks are included in these tables as an aid to translating SOPL to CL.

The parameters given in these tables also apply when the MC parameters are defined as external data points with user-defined names. In this case, an artificial PV parameter is added for Numeric (P in SOPL), Flag (ST in SOPL), and Logic Block (ST in SOPL) data points.

Digital and analog composite points are listed in Table 4-1 as if the input and output components were separate points. To find composite point parameters, look in the separate parameter lists for the equivalent input or output point. When a CL program accesses the PRTFL parameter of a composite point, however, it accesses only the PRTFL parameter of the input slot.

CAUTION

MC Data Point Flag Variables (PV)1 through 64 have additional alarming functionality that does not exist for the other flag PVs (65 through 256). The first 64 flags generate a Data Hiway alarm record each time the PV is changed. In addition, if a flag in the first 64 has an LCN HG Flag Point identifier configured, an LCN alarm or return to normal is generated for each PV change. Changing any of the first 64 flag variables at a fast frequency (for example, with CL/MC statements such as `SET FL(64) = ON`, and later, `SET FL(64) = OFF`) could cause excessive alarm traffic and slow down other Hiway/LCN functions. Flags 65 through 256 should be used as boolean variables when alarming is not required.

Table 4-1 — MC Data Point Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds[2]	Description	Remarks
ADFL	PFn	Logical		A/D Slot Fail	View only
AI	AI	ANALOG INPUT[1]	1..16 (p)	Analog Input	p = 1-16 [3]
AO	AO	ANALOG OUTPUT[1]	11..164 (s*10 + p)	Analog Output	s = 1-16, p = 1-4 [3]
CI	CI	COUNTER INPUT[1]	11..164 (s*10 + p)	Counter Input	s = 1-16, p = 1-4 [3]
DI	DI	DIGITAL INPUT[1]	101..1616 (s*100 + p)	Digital Input	s = 1-16, p = 1-16 [3]
DO	DO	DIGITAL OUTPUT[1]	11..168 (s*10 + p)	Digital Output	s = 1-16, p = 1-8 [3]
FL	FL	Logical	1..256	Flag	
IOADFL	PFn	Logical		I/O A/D Fail	View only
IOF1FL	PFn	Logical		I/O file 1 Fail	View only
IOF2FL	PFn	Logical		I/O file 2 Fail	View only
IOFL	PFn	Logical	1..16	I/O Slot Fail	View only
LB	LB	Logical	1..128	Logic Block	View only
LP	LP	LOOP (LP)[1]	1..16 (s)	Regulatory Ctrl	s = 1-16 [3]
LPFL	PFn	Logical	1..16	Reg Slot Fail	View only
NN	NN	Number	1..160	Numeric	
OUT1FL	PFn	Logical		OUT card 1 Fail	View only
OUT2FL	PFn	Logical		OUT card 2 Fail	View only
PRTFL	PF	Logical		Partial Fail	View only
SEQ		Logical	1..16	Process Module	Used only in INITIATE
SFTFL	SF	Logical		Soft Fail	View only
TM	TM	TIMER[1]	1..32	Timer	
TOG	TG	Logical		Time-out Gate	View only

1 These MC parameters have parameters; see Tables 4-2 through 4-8. The names here match the names of Tables 4-2 through 4-8.

2 Array index must be an integer constant. If array bounds are not given, the parameter is scalar.

3 **s** means slot number and **p** means point number. Only indices that represent valid point and slot numbers are allowed.

Table 4-2 — ANALOG INPUT Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
ALHIFL	AH	Logical		Hi Alarm Flag	View only
ALLOFL	AL	Logical		Lo Alarm Flag	View only
AV	AV	Number		Accum Value (EU)	View only
DEV	DV	Number		Deviation (%)	View only
DEVHITP	DH	Number		Dev Hi TripPt(%)	
DEVLOTP	DL	Number		Dev Lo TripPt(%)	
PRTFL	PF	Logical		Partial Failure	View only
PTINAL	AS	Logical		Point in Alarm	View only
PV	PVE	Number		PV (EU)	View only
PVHITP	PHE	Number		PV Hi TripPt(EU)	
PVLOTP	PLE	Number		PV Lo TripPt(EU)	
PVP	PV*	Number		PV (percent)	View only
PVPHITP	PH	Number		PV Hi TripPt(%)	
PVPLOTP	PL	Number		PV Lo TripPt(%)	
RESETCMD	–	Enum [2]			
SP	SPE	Number		SP (EU)	
SPP	SP	Number		SP (percent)	
STATE	ST	Enum [2]		Accum State	View only
STRTSTOP	RS	Enum [3]		Accum Restart	

Table 4-3 — ANALOG OUTPUT Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
MODATTR	–	Enum [2]		Mode attribute	View only
MODE	MD	MCMODE [3]		Mode	
NMODATTR	–	Enum [2]		Normal mode attr	View only
NMODE	–	MCMODE [3]		Normal mode	
OP	OV	Number		Output value (%)	
PRTFL	PF	Logical		Partial Failure	View only

Table 4-4 — COUNTER INPUT Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
AV	AV	Number		Accum Value	View only
PDEVFL	AL	Logical		Pre-alarm Flag	View only
PDEVTP	SL	Number		Pre-alarm TripPt	
PRESET	SH	Number		Alarm limit	
PRESETFL	AH	Logical		Alarm Flag	View only
PRTFL	PF	Logical		Partial Failure	View only
PV	PV	Number		Current pulse	View only
RESETCMD	–	Enum [2]			
STATE	ST	Enum [2]		Counter State	View only
STRTSTOP	RS	Enum [2]		Counter Restart	

Table 4-5 — DIGITAL INPUT Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
PRTFL	PF	Logical		Partial Failure	View only
PTINAL	AS	Logical		Point in Alarm	View only
PV	ST*	Logical		Input State	View only

Table 4-6 — DIGITAL OUTPUT Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
MODATTR	—	Enum [2]		Mode attribute	View only
MODE	MD	MCMODE [3]		Mode	
NMODATTR	—	Enum [2]		Normal mode	View only
NMODE	—	MCMODE [3]		Normal mode	
OP	ST*	Logical		Output state	
PRTFL	PF	Logical		Partial Failure	View only
PV	FB	Logical		Feedback input	View only

Table 4-7 — LOGIC BLOCK Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
PV	—	Logical		Current state	View only

Table 4-8 — LOOP (REGULATORY or MODULATING CONTROL SLOT) Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
ALHIFL	AH	Logical		Hi Alarm Flag	View only
ALLOFL	AL	Logical		Lo Alarm Flag	View only
BIAS	BS	Number		Bias	
DEV	DV	Number		Deviation (%)	View only
DEVHITP	DH	Number		Dev Hi TripPt(%)	
DEVLOTP	DL	Number		Dev Lo TripPt(%)	
INITMAN	IM	Logical		IM mode	View only
K	KP	Number		K on PID	
K1	K1	Number		Aux gain on X	
K2	K2	Number		Aux CP bias (%)	
KA	KA	Number		Aux gain on Y	
LM	LM	Logical		LM mode	View only
MODATTR	—	Enum [2]		Mode attribute	View only
MODE	MD	MCMODE [3]		Mode	
NMODATTR	—	Enum [2]		Normal mode attr	View only
NMODE	—	MCMODE [3]		Normal mode	
OAUTO	AUT	Logical		AUTO mode	
OCAS	CAS	Logical		CASC mode	
OCOM	COM	Logical		COMP mode	
OMAN	MAN	Logical		MAN mode	
OP	OV	Number		Output value (%)	
OPHILM	OH	Number		OP Hi Limit (%)	
OPLOLM	OL	Number		OP Lo Limit (%)	
OPU	OVX	Number		OP unlimited (%)	
PAUTO	SAU	Logical		SEQ-AUTO mode	
PCAS	SCA	Logical		SEQ-CASC mode	
PCOM	SCO	Logical		SEQ-COMP mode	
PMAN	SMA	Logical		SEQ-MAN mode	
PROG	SEQ	Logical		SEQ mode	View only
PROGVAL	SQV	Number		SEQ input (%)	
PRTFL	PF	Logical		Partial Failure	View only
PTINAL	AS	Logical		Point in Alarm	View only
PV	PVE	Number		PV (EU)	View only
PVEUHI	RG1	Number		Range 100% (EU)	
PVEULO	RG0	Number		Range 0% (EU)	
PVHITP	PHE	Number		PV Hi TripPt(EU)	
PVLOTP	PLE	Number		PV Lo TripPt(EU)	
PVP	PV *	Number		PV (%)View only	
PVPHITP	PH	Number		PV Hi TripPt(%)	
PVPLOTP	PL	Number		PV Lo TripPt(%)	
PVRAW	APV	Number		Analog PV	View only
PVROCNTP	CL	Number		ROC - TripPt(%)	
PVROCPTP	CH	Number		ROC + TripPt(%)	
RATIO	RT	Number		Ratio	
SOA	OA	Logical		Status out (SOA)	
SOB	OB	Logical		Status out (SOB)	
SP	SPE	Number		SP (EU)	
SPHILM	SHE	Number		SP Hi Limit (EU)	
SPLOLM	SLE	Number		SP Lo Limit (EU)	
SPP	SP	Number		SP (%)	

(Continued)

Table 4-8 — LOOP (REGULATORY or MODULATING CONTROL SLOT) Parameters (Continued)

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
SPPHILM	SH	Number		SP Hi Limit (%)	
SPPLOLM	SL	Number		SP Lo Limit (%)	
T1	T1	Number		TI on PID	
T1L	T1L	Number		Lead Time	
T2	T2	Number		TD on PID	
T2L	T2L	Number		Lag Time	

Table 4-9 — TIMER Parameters

Parameter Name	SOPL Name	Data Type	Array Bounds [1]	Description	Remarks
PV	PV	Number		Current time	View only
RESETCMD	—	Enum [2]			
SP	SP	Number		Setting time(secs)	
SPMIN	—	Number		Setting Time(mins)	
STATE	ST*	Enum [2]		Timer State	View only
STRTSTOP	RS	Enum [2]		Timer Restart	
TIMELEFT	RV	Number		Remaining time	View only
TIMOUTFL	AS	Logical [4]		Time-out Flag	View only

NOTES FOR TABLES 4-2 THROUGH 4-9

[**1**] Array bounds are given for only array parameters. Where array bounds are omitted, the parameter is scalar.

[**2**] Enumeration parameters have the following data types:

Parameter Name	Enumeration Name	Enumeration States
MODATTR, NMODATTR	MODATTR	OPERATOR/PROGRAM
STATE	STATE	STOPPED/RUNNING
RESETCMD	RESET	RESET/RESTSTRT
STRTSTOP	STRTSTOP	STOP/START

Reading of RESET or STRTSTOP parameters returns RESET or STOP value if STATE = STOPPED, and RESTSTRT or START if STATE = RUNNING.

A variable or subroutine argument that was declared with a state name list as its data type is not compatible with variables, arguments or parameters whose data type is in the above list.

(Continued)

NOTES FOR TABLES 4-2 THROUGH 4-9 (Continued)

This is because the CL compiler uses the LCN view of system-defined Enumeration types, all of which have more states than the MC view of these types described above. For example a local variable declared as

LOCAL var1:stopped/running at FL(01)

cannot be used in a set statement with the state parameter of a timer as in

SET Var1=TM(01).STATE

If var1 was declared as

Local Var1:STATE at FL(01)

the SET statement would be accepted by the compiler.

[**3**] Parameters MODE and NMODE are the enumeration type MCMODE, which has the following states:

Parameter Name	Enumeration Name	Enumeration States
MODE, NMODE	MCMODE	OMAN/OAUTO/OCAS/OCOM/ PMAN/PAUTO/PCAS/PCOM/ MAN/AUTO/CAS/COM

Do not confuse this with the type MODE, which is available on other LCN-connected boxes and which has two different states.

The values MAN, AUTO, CAS, and COM are used only for storing into MODE or NMODE; they cannot be compared with MODE or NMODE. These values are valid only when MODATTR or NMODATTR, respectively, is already equal to PROGRAM.

The values OAUTO, OCAS, PAUTO, PCAS, and CAS cannot be stored into or compared with MODE or NMODE of Digital Output, Digital Composite, Analog Output, or Analog Composite data points.

Reading of MODE (NMODE) parameter returns OMAN, OAUTO, OCAS, or OCOM values if MODATTR (NMODATTR) = OPERATOR, and PMAN, PAUTO, PCAS, or PCOM values if MODATTR (NMODATTR) = PROGRAM.

[**4**] The LCN view of the parameter TIMOUTFL is Normal/TimeUp. The MC view is Off/On.

The following are examples of gaining access to MC Parameters:

```
EXTERNAL My_AO          -- Analog output point
SET My_AO.OP = 5        -- Parameter of external point
SET AO(11).OP = 5       -- Parameter of BDP parameter

EXTERNAL My NUM         -- Numeric data point
SET My_NUM.PV = 100     -- Parameter of external point
SET NN(15) = 100        -- Parameter of bound data point
SET My_NUM = 100        -- INVALID
```

4.5.5 Self-Defining Enumerations

Some parameters of MC points are defined as type Logical if accessed as parameters of the BDP, but if accessed as parameters of an EXTERNAL, the parameters are defined as self-defining Enumerations. This applies to the following parameters:

PV of Digital Input data point
PV and OP of Flag data point
PV of Digital Composite point
OP of Digital Composite point
PV of Logic Block point

The state names for the self-defining Enumeration are found in the STATE1 and STATE2 parameters of the EXTERNAL data point.

4.5.5.1 Self-Defining Enumeration Examples

```
EXTERNAL VALVE              -- DO point, OPEN/CLOSE
EXTERNAL SWITCH       -- DI point, OK/NG
EXTERNAL FULLFL       -- Flag point, FULL/EMPTY
  ...
SET VALVE.OP = CLOSE
OPEN VALVE
IF SWITCH.PV = OK THEN SET FULLFL.PV = EMPTY
SET FULLFL.PV = FULL
```

4.6 HIWAY GATEWAY (HG) LIBRARY

The HG maintains four libraries; MC is assigned to one of these four libraries. A library consists of 576 entries of 8-character Strings. In CL/MC, the Strings listed in Table 4-10 must be defined in the HG library that is assigned to the MC on which the program is to run.

Table 4-10 — Strings in HG Library

String	Available Entry Number
Sequence ID	129-256, 513-576
Phase ID	129-256, 513-576
Step ID	257-512
SEND String-item	1-576

Each word in the String of a SEND statement is assigned as an entry in the HG Library. Words in a SEND statement String can be eight characters maximum. Spaces are the delimiting characters between words. For example:

```
SEND: "send String items"
```

is treated as

```
SEND: "send", "String", "items"
```

The HG library can be automatically generated by the CL compiler when automatic library generation is specified at compile time (refer to *Control Language/Application Module Data Entry*). String entries are made in decreasing order, from 576 towards 1. If a program has no Step ID, the 257th entry is used for the step; therefore, the 257th library entry should be blank. SEND string items first fill up entries in the Sequence ID, Phase ID, and Step ID; if this becomes a problem, manually enter the SEND string items into the HG libraries using the Engineering Personality.

4.7 USER-WRITTEN SUBROUTINES

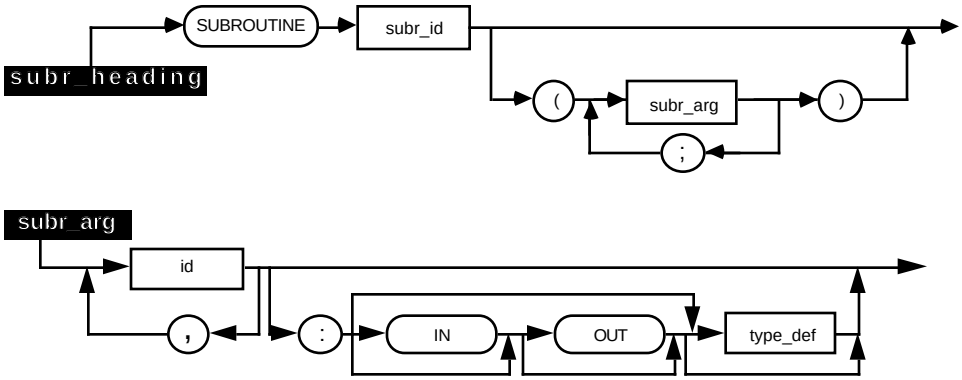
This section describes user-written CL subroutines. A program can call subroutines that are user-written or system-supplied; user-written subroutines must use the facility described here, but system-supplied subroutines need not be written in CL.

User-written CL/MC subroutines are referred to as "sequence subroutines"; see heading 4.2.1 for "sequence subroutine" syntax.

4.7.1 SUBROUTINE Heading

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

4.7.2 SUBROUTINE-Heading Syntax



4.7.3 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.

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

4.7.4 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
```

4.7.5 Subroutine Arguments

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

- Number
- Logical
- Enumeration
- State name list

Whole arrays can be used as subroutine arguments only if they are of data type Number or data type Logical.

Data point identifiers are not permitted as subroutine arguments.

Enumeration types that can be passed as subroutine arguments are limited as follows: state name lists can contain only two states, and enumerations are limited to those shown in Tables 4-2 through 4-8.

4.7.5.1 Subroutine Arguments Examples

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

```
SUBROUTINE sub1 (x : IN state1/state2; y : IN LOGICAL)
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
                                          -- at 3.2.13.3 is ILLEGAL
SUBROUTINE sub6 (state : IN state1/state2)
```

4.8 BUILT-IN FUNCTIONS AND SUBROUTINES

If an enumeration is called for by a parameter list file or 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.

4.8.1 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.

The built-in function ROUND does not work correctly on negative numbers in the MC; write in-line code or a subroutine that will perform the ROUND function.

Abs (x)	-- absolute value
Exp (x)	-- exponential
Int (x)	-- truncate to integer
Ln (x)	-- natural logarithm
Log10 (x)	-- common logarithm
Max (x, y, ...)	-- maximum (5 or fewer arguments)
Min (x, y, ...)	-- minimum (5 or fewer arguments)
Round (x)	-- round to integer
Sqrt (x)	-- square root
Sum (x, y, ...)	-- sum

4.8.2 Set_Time Subroutine

The built-in subroutine Set_Time is provided to start a timer. The timer must be in the same MC as the program calling Set_Time. Its syntax is

```
Set_Time (      x: a timer point;
&       count: Time;
&       start: LOGICAL)
```

This subroutine sets the timer **x** to count for **count** time. If **start** is equal to On, the timer is started; otherwise, it is left inactive.

The Count specification cannot be a generalized Time expression, but is restricted as in the WAIT statement; i.e., it can be one of

```
arith_operand MINS
arith_operand SECS
```

If the (truncated) value of the arith_operand is less than zero or greater than 9999, 0 or 9999, respectively, is used instead.

4.8.2.1 Set_Time Example

```
CALL SET_TIME (TM(01), 15 SECS, ON)
```

4.8.3 Day_Time Function

The syntax of the Function Day_Time is

```
Day_Time: LOGICAL(H, M : NUMBER)
```

Day_Time returns On if the current time of day is H hours M minutes.

4.8.3.1 Day_Time Example

```
IF DAY_TIME (16,30) THEN GOTO PHASE BEGIN
```

4.9 INTERCOMMUNICATIONS BETWEEN MULTIFUNCTION CONTROLLERS

There are two methods of configuration for intercommunications between Multifunction Controller boxes that are connected together with the same C-Link cable. These intercomunications include the ability to Read or Write values or to initiate a sequence in another box.

The first (normal) configuration method allows a maximum of eight C-Links, each containing up to eight Multifunction Controllers.

The other (extended) configuration method allows up to four C-Links, each containing up to 16 Multifunction Controllers. (Note that this method is only available for Advanced Multifunction Controllers that have extended addressing capabilities.) This is accomplished by pairing the BOXCLINK parameter values for the C-Links to form an extended C-Link (e.g., BOXCLINK 1 and BOXCLINK 5 are paired to represent the same physical cable, BOXCLINK 2 and BOXCLINK 6 are paired to represent a second physical cable, etc.).

This extended configuration requires CL/MC to re-address the MCs assigned to BOXCLINK values 5 to 8 to addresses 9 to 16 of C-Links 1 to 4. This calculation is performed only when the -EX (or -EXTEND) compiler switch is specified.

CAUTION

When the Bound Data Point is in a Multifunction Controller that is on a C-Link that has been configured with extended addressing, the -EX compiler switch always should be specified. Otherwise, compilation or runtime errors can result.

The following table shows how the external box parameters relate to the values output by CL/MC.

Table 4-11 — Relation of External Box Parameters to CL/MC Values

External Box Parameters From the HG	Internal Values for Generated Code	
	Without -EX Switch	With -EX Switch
BOXCLINK 1 CLINKNUM 1-8	C-Link 1 Addresses 1-8	C-Link 1 Addresses 1-8
BOXCLINK 2 CLINKNUM 1-8	C-Link 2 Addresses 1-8	C-Link 2 Addresses 1-8
BOXCLINK 3 CLINKNUM 1-8	C-Link 3 Addresses 1-8	C-Link 3 Addresses 1-8
BOXCLINK 4 CLINKNUM 1-8	C-Link 4 Addresses 1-8	C-Link 4 Addresses 1-8
BOXCLINK 5 CLINKNUM 1-8	C-Link 5 Addresses 1-8	C-Link 1 Addresses 9-16
BOXCLINK 6 CLINKNUM 1-8	C-Link 6 Addresses 1-8	C-Link 2 Addresses 9-16
BOXCLINK 7 CLINKNUM 1-8	C-Link 7 Addresses 1-8	C-Link 3 Addresses 9-16
BOXCLINK 8 CLINKNUM 1-8	C-Link 8 Addresses 1-8	C-Link 4 Addresses 9-16

SUMMARY OF SYNTAX FOR CL ELEMENTS, STATEMENTS, & STRUCTURES
Appendix A

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

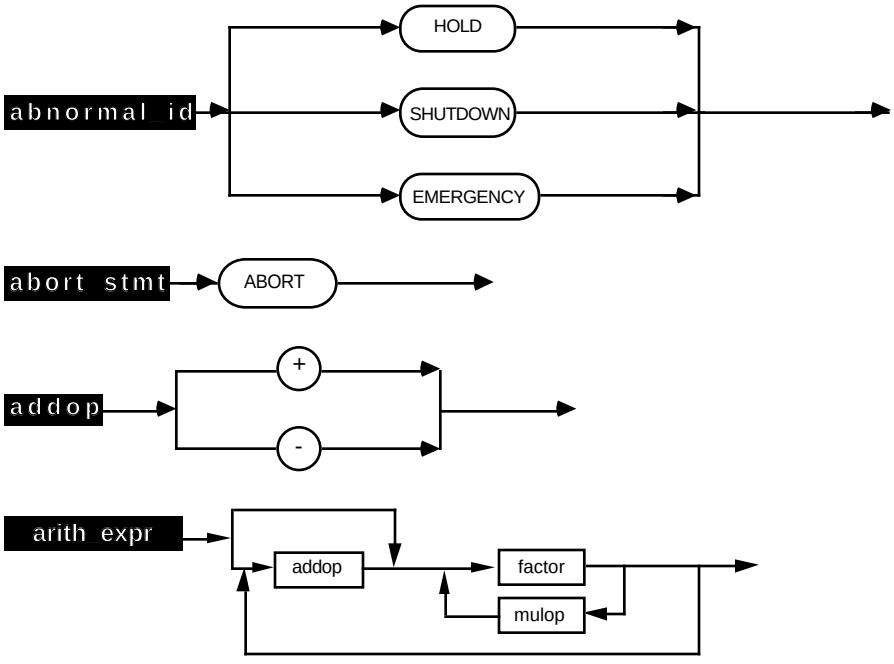
A.1 SYNTAX (GRAMMAR) SUMMARY

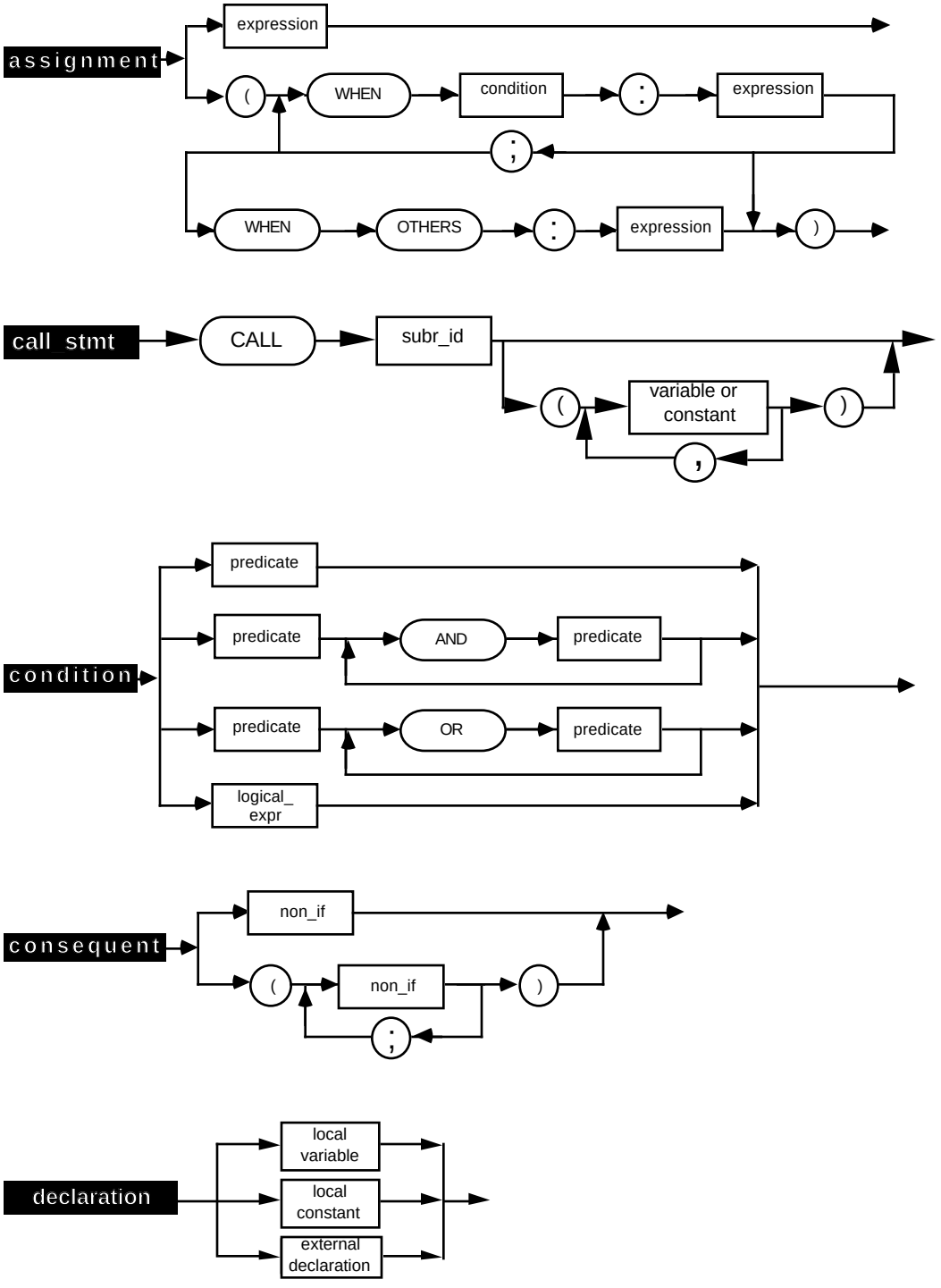
This section is divided into two parts. The first part is a summary of CL syntax in the form of all the syntax diagrams that were presented throughout the manual. Each syntax diagram is labeled (in reverse-video), and they are arranged in alphabetical order.

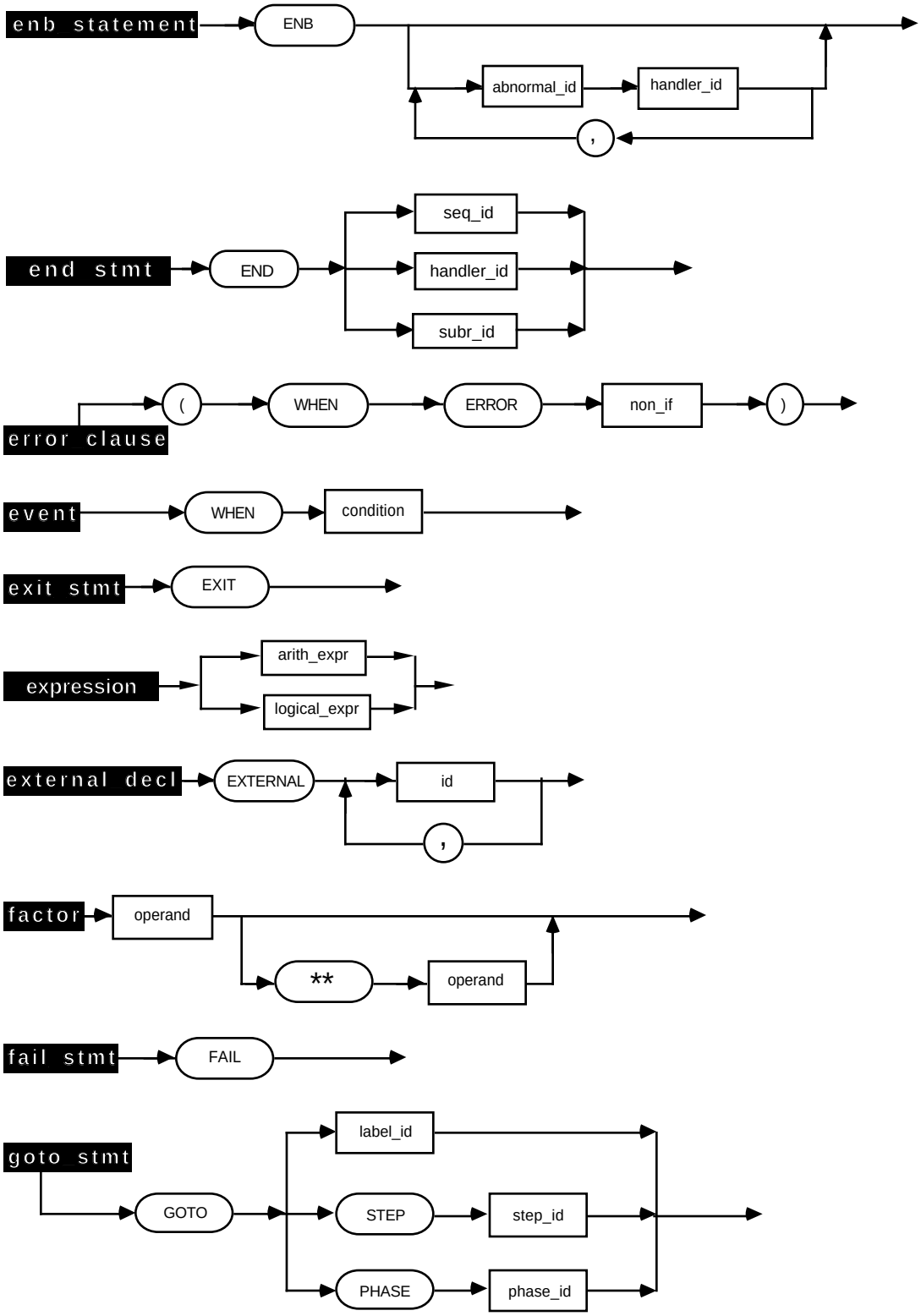
The second part is a summary of CL syntax in the BNF form of production rules, again in alphabetical order.

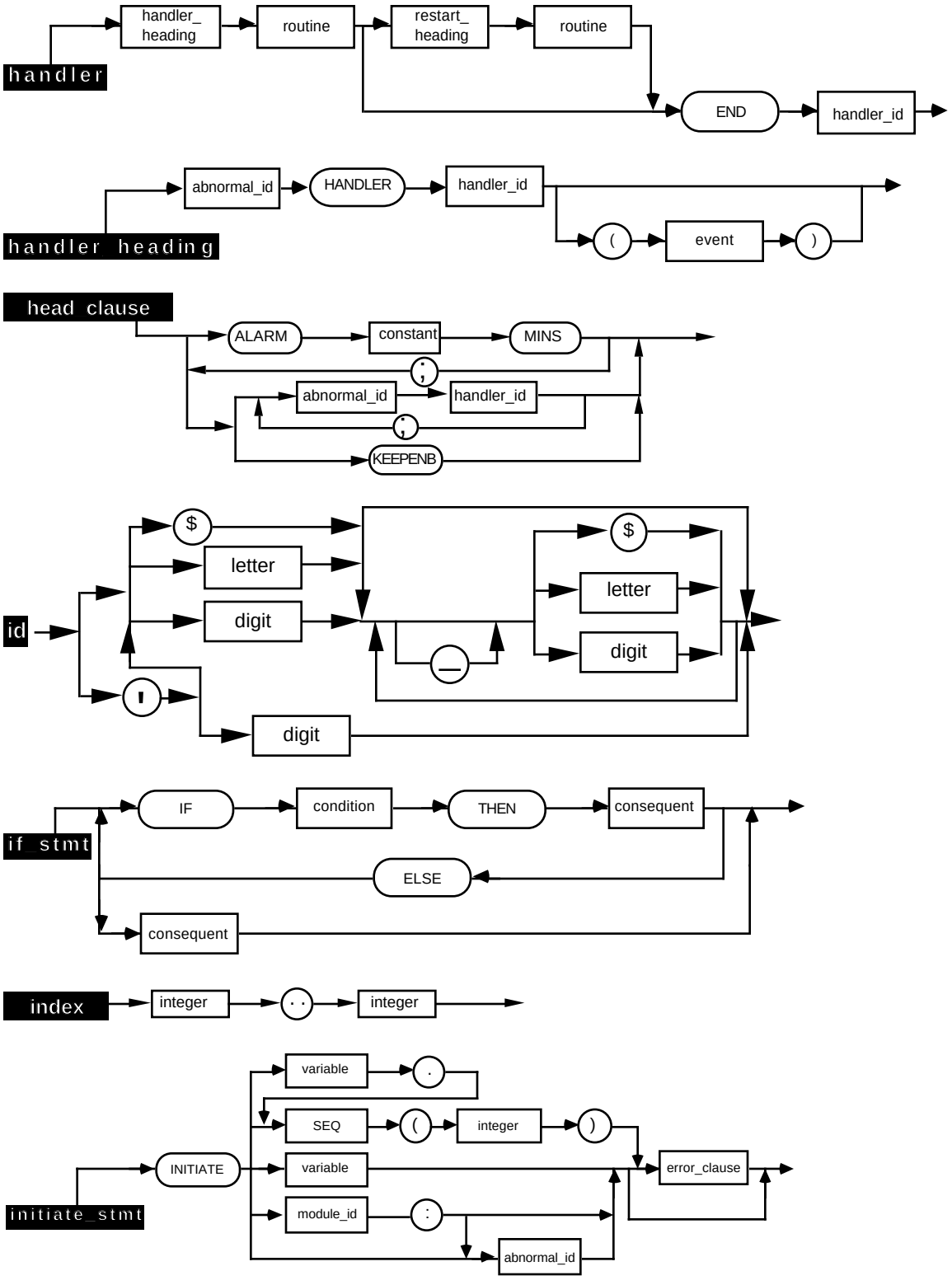
To use either part of this section, decide what item you want to construct and go through either summary (depending on what form you feel most comfortable with) looking for that item on the left-most portion of each page. When you find the item, the way to build it is contained in the diagram or production 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.

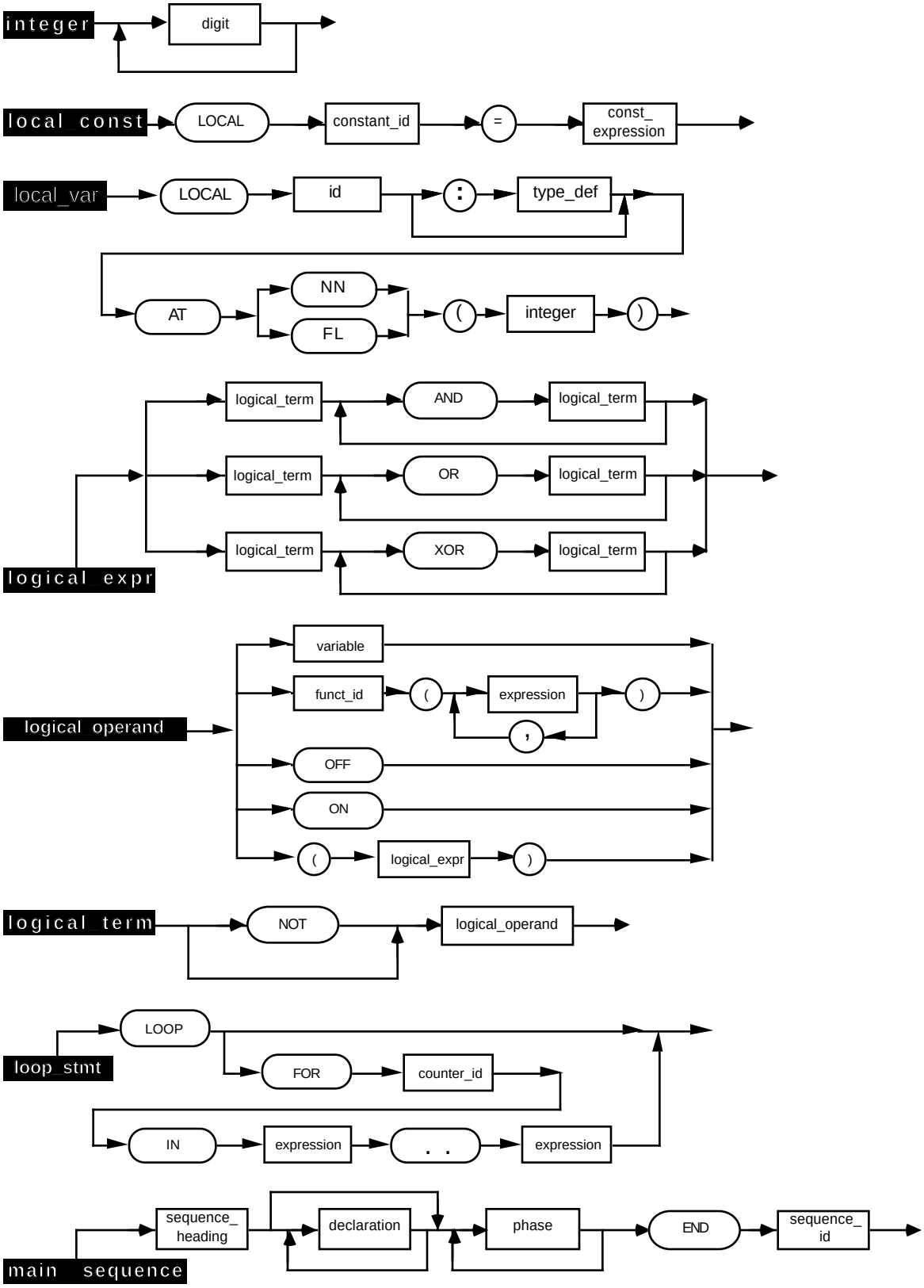
A.2 SYNTAX DIAGRAM SUMMARY

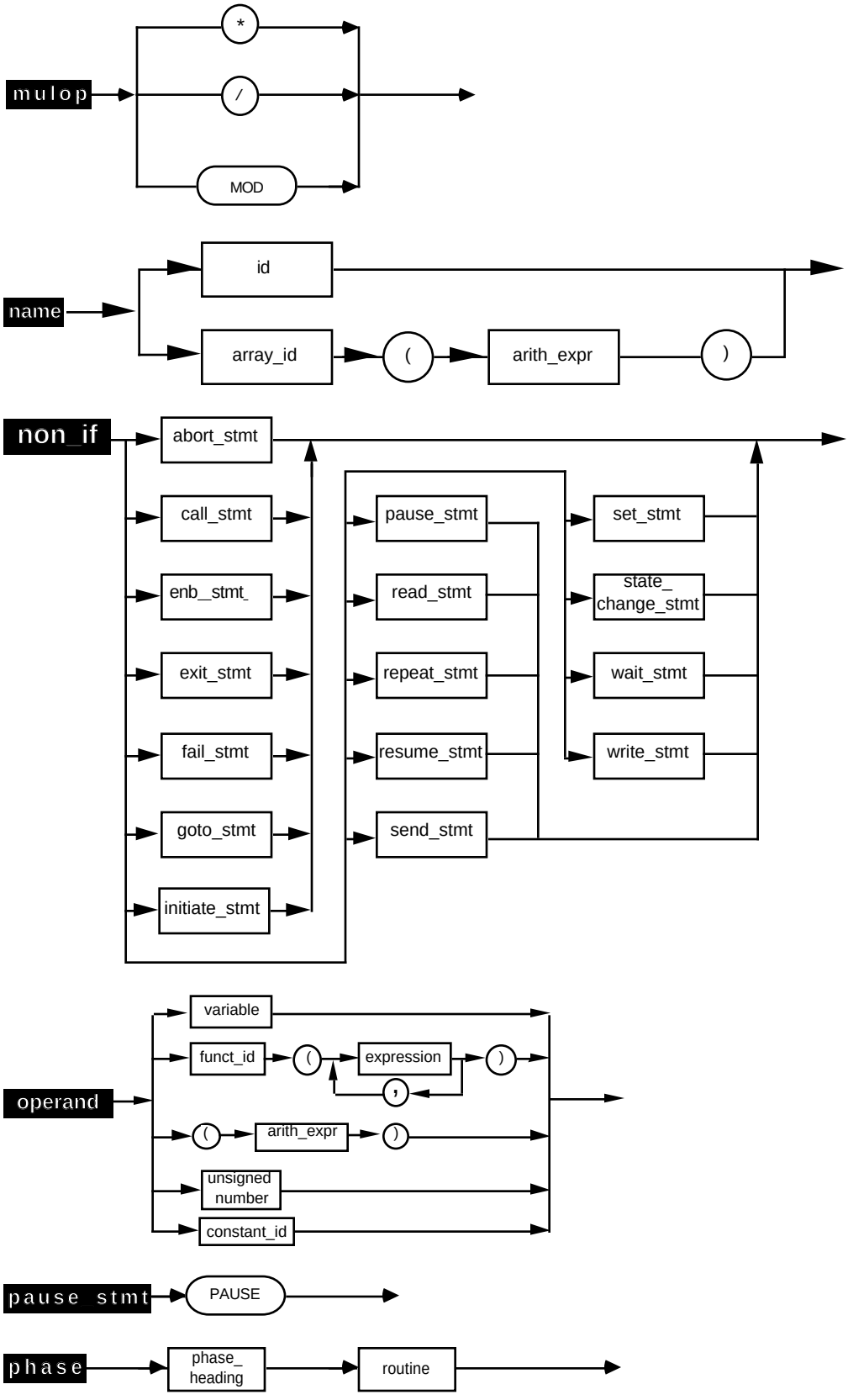


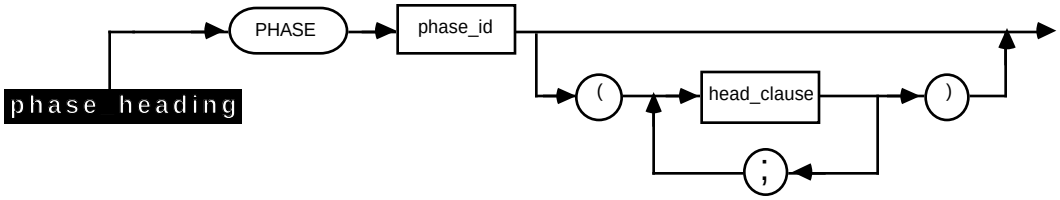

assignment
expression
(
WHEN
condition
:
expression
;
WHEN
OTHERS
:
expression
)
call stmt
CALL
subr_id
(
variable or constant
)
,
condition
predicate
predicate
AND
predicate
predicate
OR
predicate
logical_ expr
consequent
non_if
(
non_if
)
;
declaration
local variable
local constant
external declaration

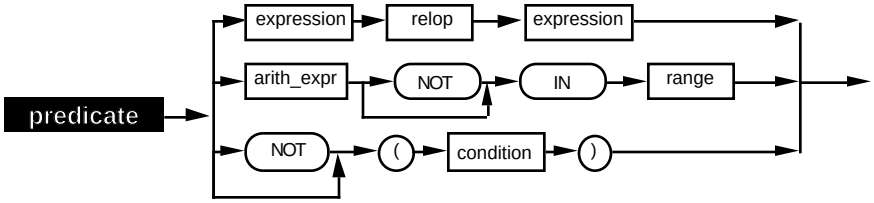

enb statement
ENB
abnormal_id
handler_id
,
end stmt
END
seq_id
handler_id
subr_id
error clause
(
WHEN
ERROR
non_if
)
event
WHEN
condition
exit stmt
EXIT
expression
arith_expr
logical_expr
external decl
EXTERNAL
id
,
factor
operand
**
operand
fail stmt
FAIL
goto stmt
GOTO
label_id
STEP
step_id
PHASE
phase_id

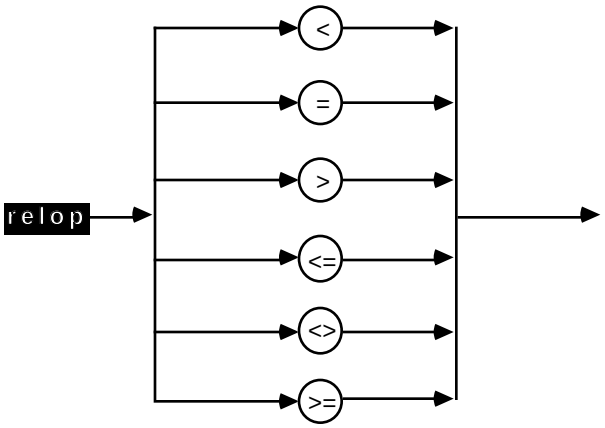

handler
handler_ heading
routine
restart_ heading
routine
END
handler_id
handler heading
abnormal_id
HANDLER
handler_id
(
event
)
head clause
ALARM
constant
MINS
;
abnormal_id
handler_id
;
KEEPENB
id
$
letter
digit
_
$
letter
digit
'
digit
if stmt
IF
condition
THEN
consequent
ELSE
consequent
index
integer
. .
integer
initiate_stmt
INITIATE
variable
.
SEQ
(
integer
)
variable
module_id
:
abnormal_id
error_clause


integer
digit
local_const
LOCAL
constant_id
=
const_
expression
local_var
LOCAL
id
:
type_def
AT
NN
FL
(
integer
)
logical_expr
logical_term
AND
logical_term
logical_term
OR
logical_term
logical_term
XOR
logical_term
logical_operand
variable
funct_id
(
expression
,
)
OFF
ON
(
logical_expr
)
logical_term
NOT
logical_operand
loop_stmt
LOOP
FOR
counter_id
IN
expression
. .
expression
main_sequence
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

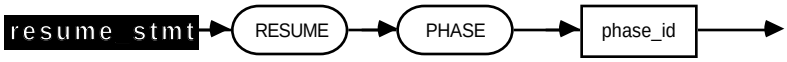

phase heading
PHASE
phase_id
(
head_clause
)
;

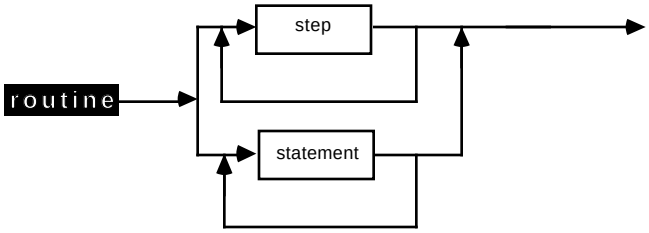

predicate
expression
relop
expression
arith_expr
NOT
IN
range
NOT
(
condition
)

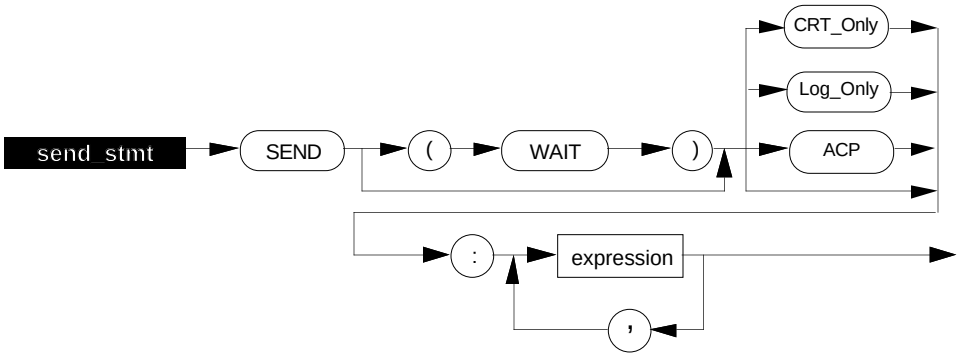

relop
<
=
>
<=
<>
>=

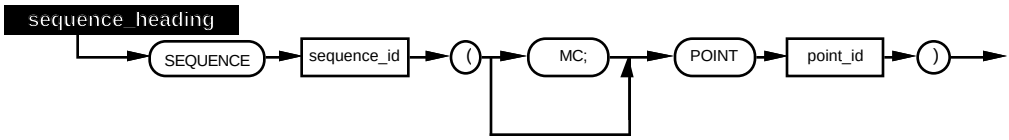

repeat stmt
REPEAT
label_id

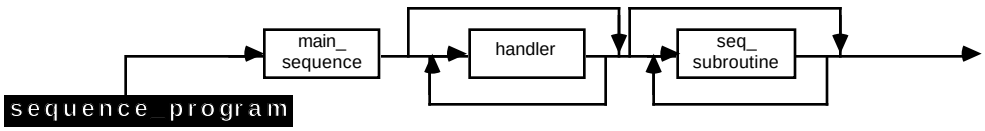

restart heading
RESTART

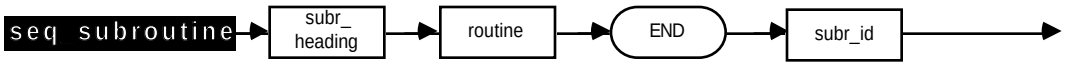

resume stmt
RESUME
PHASE
phase_id

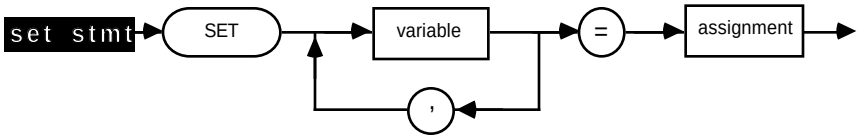

routine
step
statement

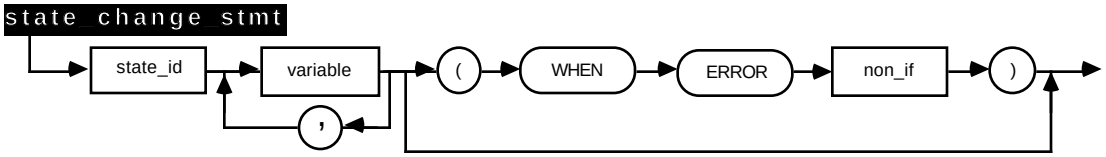

send_stmt
SEND
(
WAIT
)
CRT_Only
Log_Only
ACP
:
expression
,

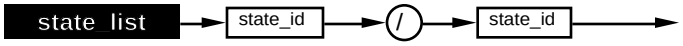

sequence_heading
SEQUENCE
sequence_id
(
MC;
POINT
point_id
)

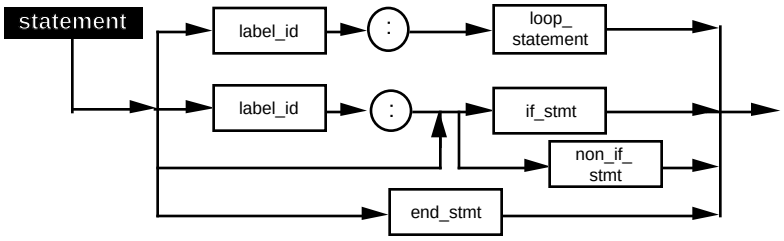

main_
sequence
handler
seq_
subroutine
sequence_program

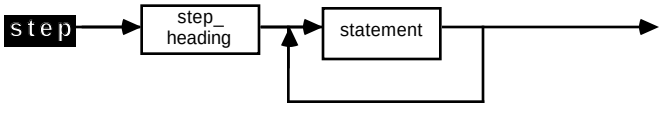

seq_subroutine
subr_
heading
routine
END
subr_id

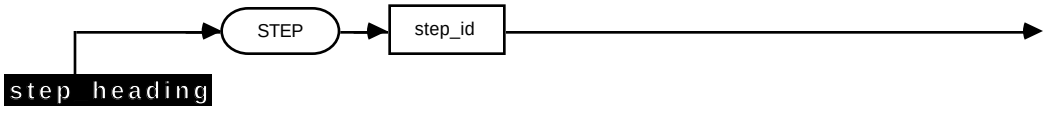

set_stmt
SET
variable
=
assignment
,

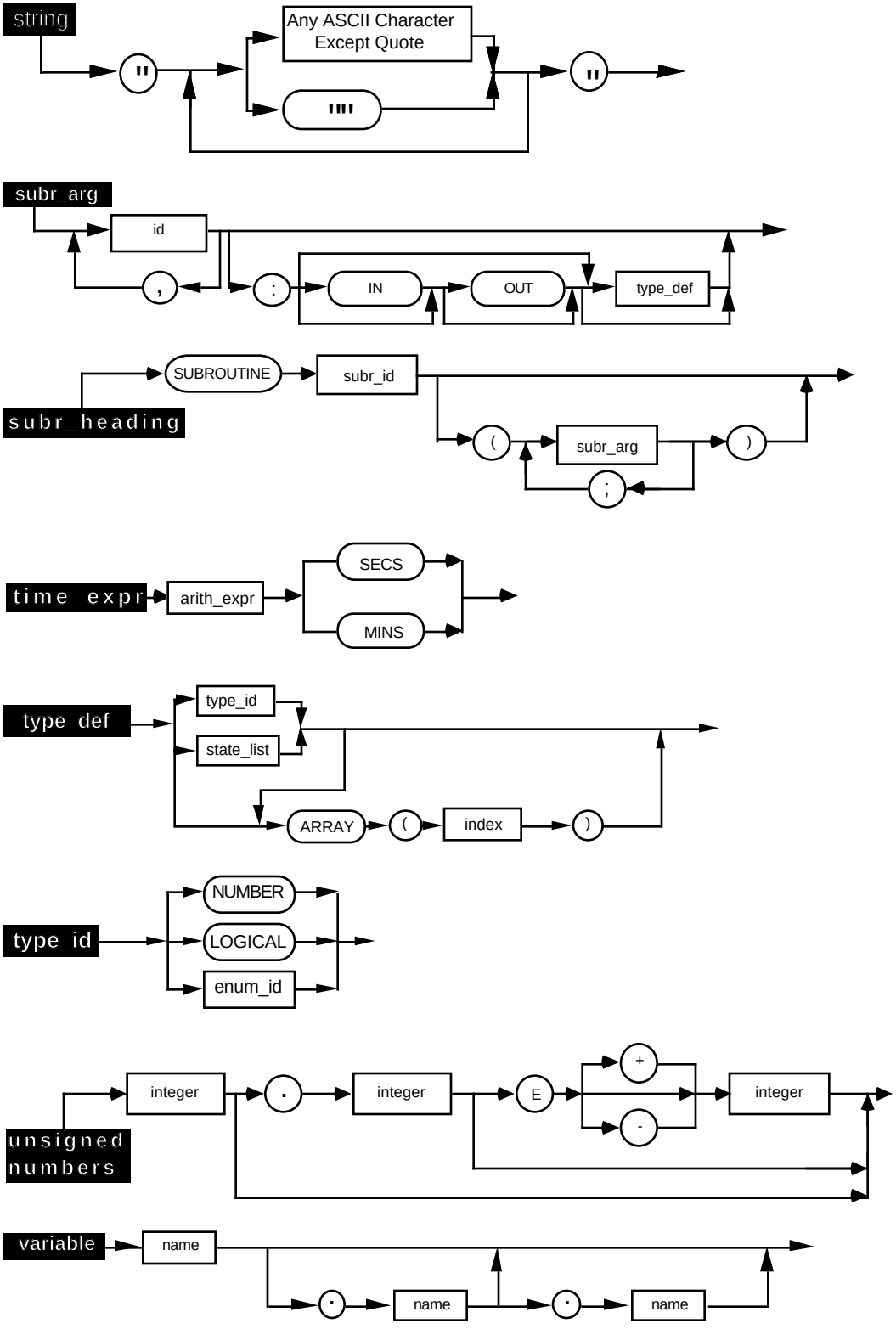

state_change_stmt
state_id
variable
(
WHEN
ERROR
non_if
)
,

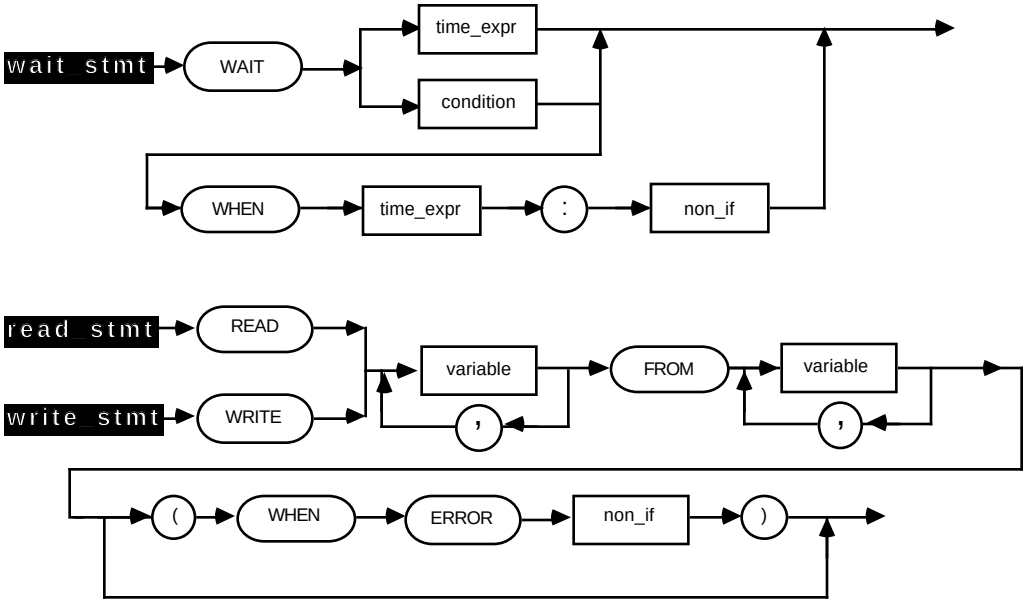

state_list
state_id
/
state_id


statement
label_id
:
loop_
statement
label_id
:
if_stmt
non_if_
stmt
end_stmt


step
step_
heading
statement


STEP
step_id
step_heading


string
"
Any ASCII Character
Except Quote
""
"
subr arg
id
,
:
IN
OUT
type_def
subr heading
SUBROUTINE
subr_id
(
subr_arg
)
;
time expr
arith_expr
SECS
MINS
type def
type_id
state_list
ARRAY
(
index
)
type id
NUMBER
LOGICAL
enum_id
unsigned
numbers
integer
.
integer
E
+
-
integer
variable
name
.
name
.
name



A.3 NOTATION USED FOR SYNTAX PRODUCTION RULES

The following presentation of CL syntax effectively 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; that is, 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, as follows: 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 (that is, **anything_id :: = id**).

A.4 CL/MC PRODUCTION RULES

abnormal_id :: = HOLD | SHUTDOWN | EMERGENCY

abort_stmt :: = ABORT

access_attribute :: = ACCESS access_lock_id

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})]

compilation_unit :: = sequence_program
| lib_subroutine

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

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 | logical_expr

expressions :: = expression {, expression}

external_decl :: = EXTERNAL id {, id}

factor :: = operand [** operand]

fail_stmt :: = FAIL

first_char :: = $ | letter

first_param :: =   point_description
                 |   subr_arg

goto_stmt :: =  GOTO label_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 constant MINS]
                  |  [ALARM constant MINS ;] KEEPENB
                  |  [ALARM constant MINS ;] abnormal_id handler_id
                                {; abnormal_id handler_id}

help_attribute :: = HELP String

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 SEQ (integer) [error_clause]
                  |  INITIATE [variable] [error_clause]
                  |  INITIATE [module_id :] abnormal_id [error_clause]
                  |  INITIATE variable.SEQ (integer) [error_clause]

integer :: = digit {digit}

letter :: =  upper_case_alphabetic
           |  lower_case_alphabetic
           |  Katakana

letter_or_digit :: = letter | digit | $

local_const :: = LOCAL constant_id = constant_decl

local_var :: = LOCAL id [ : type_def] AT 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
            |  constant_id

param_id :: = NN | FL

pause_stmt :: = PAUSE

phase :: =  phase_heading
            routine

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

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

program :: =  sequence_program

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

schedule :: =  event
             |  time_expr
             |  WAIT

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

sequence_heading :: = SEQUENCE sequence_id ([MC ;] POINT point_id)

sequence_program :: =  main_sequence
                       {handler}
                       {seq_subroutine}  (see subroutine syntax)

seq_subroutine :: = subr_heading
                    routine
                    END subr_id

set_stmt :: = SET variables = assignment
```

```
sign :: = + | -

state_change_stmt :: = state_id [variables] [error_clause]

state_list :: = 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} "

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 :: = (first_param {; subr_arg})

subr_heading :: = SUBROUTINE subr_id [subr_args]

term :: =    factor {mulop factor}

time_expr ::= arith_expr SECS
               |          arith_expr MINS

type_def :: = type_id [array_def]
               |  state_list [array_def]
               |  array_def

type_id :: = NUMBER | LOGICAL | enum_id

unlabeled_stmt :: =   if_stmt
                    |   non_if

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

variable :: = name
               | name . name
               | name . 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/MC SOFTWARE ENVIRONMENT
Appendix B

This appendix refers you to the various control functions reference manuals for information on how **TotalPlant** *Solution (TPS) System control functions support CL/MC (in other words, the CL/MC Runtime Environment). This appendix also details some of the limitations the TPS CL/MC Runtime Environment places on various aspects of building CL/MC structures, such as memory usage.*

B.1 REFERENCES TO CONTROL FUNCTIONS PUBLICATIONS

The relationships between the TPS Software environment and CL/MC is found in the *System Control Functions*, and *Hiway Gateway Control Functions* publications.

Although you should read all of those publications to understand TPS Data Acquisition & Control, the following subsections contain information specific to CL/MC; 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/MC as it relates to the standard TPS Control Software.

HEADING	TOPIC
SYSTEM CONTROL FUNCTIONS	
3.3.1.1.2	CL Access (to parameters — general)
3.3.7	Advanced Functions
HG CONTROL FUNCTIONS	
3.8 (all)	Process Module Data Point

B.2 CL/MC CAPACITIES

Max. Lines of Code/Sequence Step (depends on statement complexity)	100-300

B.2.1 CL/MC Limits

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	254
Max. blocks of code for each sequence	512
Block size	16 words
Max. blocks of code for all 16 sequence slots in MC; the MC must be configured: 'SOPLMEM' = SOPLMEM and 'BOXTREND' = NOTREND	776
Max. phases and steps in a sequence is only limited by the number of slots available for phases and step identifiers in the HG Library.	
Max. size for expression or condition, where both operators and operands count as 1 item each; example: (x + y) < 6 contains 5 items	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.	256
Words for each statement (depends heavily on complexity of statement)	10W to 400W; avg. ~ 20W

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

Index

READER COMMENTS

Honeywell IAC's 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 or mail using this form, or by placing a toll-free telephone call. We would like to acknowledge your comments; please include your complete name, address, and telephone number.

BY FAX: 1-602-313-4842

BY MAIL: Honeywell Inc.
Industrial Automation and Control
Automation College
2820 W. Kelton Lane
Phoenix, AZ 85023-3028

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

Title of Publication: **CL/MC Reference Manual** Issue Date: **10/96**

Publication Number: **PC27-510**

Writer: **Maria Nelson**

COMMENTS: _____

_____

_____

_____

_____

_____

_____

_____

RECOMMENDATIONS: _____

_____

_____

_____

_____

_____

_____

NAME _____ DATE _____

TITLE _____

COMPANY _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

TELEPHONE _____ FAX _____

Honeywell

Industrial Automation and Control
Honeywell Inc.
16404 North Black Canyon Highway
Phoenix, Arizona 85023-3033

Helping You Control Your World