

# **Control Language Application Module Reference Manual**

**AM27-610**



**Implementation  
Application Module - 3**

***Control Language  
Application Module  
Reference Manual***

**AM27-610  
Release 610  
01/00**

---

# Notices and Trademarks

© Copyright1999 by Honeywell Inc.

Revision 02 – October, 1999

While this information is presented in good faith and believed to be accurate, Honeywell disclaims the implied warranties of merchantability and fitness for a particular purpose and makes no express warranties except as may be stated in its written agreement with and for its customer.

In no event is Honeywell liable to anyone for any indirect, special or consequential damages. The information and specifications in this document are subject to change without notice.

**TotalPlant** and TDC 3000 are U. S. registered trademarks of Honeywell Inc.

Other brand or product names are trademarks of their respective owners.

---

---

## About This Publication

This publication provides reference information about the Honeywell Control Language for the Application Module (CL/AM).

This publication supports **TotalPlant** Solution (TPS) system network Release 610. TPS is the evolution of TDC 3000<sup>X</sup>.

Change bars are used to indicate paragraphs, tables, or illustrations containing changes that have been made to this manual effective with R610. Pages revised only to correct minor typographical errors contain no change bars.



---

# Table of Contents

---

## 1 INTRODUCTION

- 1.1 Purpose/Background
- 1.2 References
  - 1.2.1 General CL/AM Information
  - 1.2.2 Publications with CL/AM-Specific Information
- 1.3 CL/AM Overview

## 2 RULES AND ELEMENTS OF CL/AM

- 2.1 Introduction to CL/AM Rules and Elements
- 2.2 CL/AM Rules and Elements
  - 2.2.1 Character Set Definition
  - 2.2.2 Spacing
  - 2.2.3 Lines
  - 2.2.4 Syntax (Summary is in Appendix A)
  - 2.2.5 CL/AM Restrictions
  - 2.2.6 Comments
  - 2.2.7 Identifiers
  - 2.2.8 Numbers
  - 2.2.9 String
  - 2.2.10 Special Symbols
- 2.3 CL/AM Data Types
  - 2.3.1 Number Data Type
  - 2.3.2 Time Data Type
  - 2.3.3 Discrete Data Types
  - 2.3.4 Data Points Data Type
  - 2.3.5 Array Data Type
  - 2.3.6 String Data Type
  - 2.3.7 Referencing Compound Elements
  - 2.3.8 Indirect Reference to Point Identifiers
- 2.4 Variables and Declarations
  - 2.4.1 Variables and Declarations Syntax
  - 2.4.2 Local Variables
  - 2.4.3 Local Constants
  - 2.4.4 External Data Points
  - 2.4.5 Parameter Declarations
  - 2.4.6 Function Declarations
- 2.5 Expressions and Conditions
  - 2.5.1 Expressions and Conditions Syntax
  - 2.5.2 Arithmetic and Logical Expressions
  - 2.5.3 Time Expressions
  - 2.5.4 Conditions Definition

## 3 CL/AM STATEMENTS

- 3.1 Introduction
- 3.2 Program Statements Definition
  - 3.2.1 Program Statements Syntax
  - 3.2.2 Statement Labels
  - 3.2.3 SET Statement
  - 3.2.4 STATE CHANGE Statement

---

# Table of Contents

---

3.2.5	GOTO Statement
3.2.6	IF, THEN, ELSE Statement
3.2.7	LOOP Statement
3.2.8	REPEAT Statement
3.2.9	CALL Statement
3.2.10	SEND Statement
3.2.11	EXIT Statement
3.2.12	ABORT Statement
3.2.13	END Statement
3.3	Embedded Compiler Directives
3.3.1	Embedded Compiler Directives Syntax
3.3.2	%PAGE Directive
3.3.3	%DEBUG Directive
3.3.4	%RELAX Directive
3.3.5	%INCLUDE_SET Directive
3.3.6	%INCLUDE_SOURCE Directive

## 4 CL/AM STRUCTURES

4.1	CL/AM Structures—General Orientation
4.2	Block Definition
4.2.1	Block Syntax
4.2.2	Block Description
4.2.3	Block-Heading Definition
4.2.4	Block-Heading Syntax
4.2.5	Block-Heading Description
4.2.6	Block-Heading Examples
4.3	Subroutine Definition
4.3.1	Subroutine-Heading Definition
4.3.2	Subroutine-Heading Syntax
4.3.3	Subroutine-Heading Description
4.3.4	Subroutine-Heading Examples
4.3.5	Subroutine Data-Declarations Definition
4.3.6	Subroutine Arguments Definition
4.3.7	Built-in Functions and Subroutines
4.4	Custom Data Segments Definition
4.4.1	Custom Data Segment Syntax
4.4.2	Custom Data Segment Description
4.4.3	Custom Data Segment Heading
4.4.4	Custom Data Segment Parameter Declaration
4.4.5	Custom Data Segment Parameter Attribute Statement
4.5	Parameter List Definition
4.5.1	Parameter List Syntax
4.5.2	Parameter List Description
4.5.3	Parameter List Examples
4.6	Enumeration-Type Definition
4.6.1	Enumeration-Type Definition Syntax
4.6.2	Enumeration-Type Definition Description
4.6.3	Enumeration-Type Definition Examples
4.7	Package Definition
4.7.1	Package Syntax
4.7.2	Package Description
4.7.3	Package Examples



---

# Table of Contents

---

- 4.8 CL Runtime Extensions
- 4.9 Examples of CL/AM Structures
  - 4.9.1 PV Calculation Example
  - 4.9.2 Linearization Example
  - 4.9.3 Custom Data Segment Example
  - 4.9.4 BTU Switch Package Example
- 4.10 File Manager Status Return Values
- 4.11 CDS Request Return Status Values

## **APPENDIX A CL/AM SYNTAX SUMMARY**

- A.1 Syntax (Grammar) Summary
- A.2 Syntax Diagram Summary
- A.3 Notation Used for Syntax Production Rules
- A.4 CL/AM Syntax Production Rules

## **APPENDIX B CL/AM SOFTWARE ENVIRONMENT**

- B.1 References to Control Functions Publications
- B.2 CL/CDS Capacities
- B.3 \$REG\_CTL Parameter List
- B.4 CL Data Access Performance

## **APPENDIX C CL/AM EXTENSION FOR FILE I/O**

- C.1 References
- C.2 Overview of File I/O Extensions
- C.3 Restrictions
- C.4 Packaging
- C.5 Installation and Configuration
- C.6 CL Callable Subroutines
- C.7 Return Status Values
- C.8 Memory Considerations
- C.9 File I/O Performance Example

## **APPENDIX D CL/AM EXTENSION FOR CONTINUOUS HISTORY ACCESS**

- D.1 Overview of the History Access Extension
  - D.1.1 Features of this CL/AM Extension
  - D.1.2 Summary of Subroutines in this CL Extension
  - D.1.3 Memory Use
- D.2 Packaging
- D.3 Installation and Configuration
- D.4 CL Callable Subroutines
  - D.4.1 Introduction
    - D.4.1.1 Including the Set Definition File
    - D.4.1.2 Subroutine Return Status
    - D.4.1.3 CL Aborts
  - D.4.2 AMCL03\$Get\_Collection\_Rate Subroutine
  - D.4.3 AMCL03\$Abs\_History Subroutine
  - D.4.4 AMCL03\$Rel\_History Subroutine
  - D.4.5 AMCL03\$Avg\_History Subroutine

---

# Table of Contents

---

D.5	Special Issues
D.5.1	Bad Values
D.5.2	Time Change
D.5.2.1	Spring Time Change
D.5.2.2	Fall Time Change
D.5.3	Scope of Search

## APPENDIX E CL/AM EXTENSION FOR MATH LIBRARY

E.1	Overview of the Math Library Extension
E.1.1	Features of this CL/AM Extension
E.1.2	Summary of Subroutines in this CL Extension
E.1.3	Hardware Requirements
E.1.3.1	Memory Used
E.1.3.2	Processor Type
E.1.4	Work Areas
E.1.5	Matrix Size Restrictions
E.1.6	Array and Array Size Arguments
E.2	Packaging
E.3	Installation and Configuration
E.4	CL Callable Subroutines
E.4.1	Introduction
E.4.1.1	Including the Set Definition File
E.4.1.2	Subroutine Return Status
E.4.1.3	CL Aborts
E.4.2	AMCL02\$Standard_Deviation Subroutine
E.4.3	AMCL02\$Uniform_Random_Number Subroutine
E.4.4	AMCL02\$Normal_Random_Number Subroutine
E.4.5	AMCL02\$Get_Time Subroutine
E.4.6	AMCL02\$Subtract_Time Subroutine
E.4.7	AMCL02\$CDS_Array_to_Local_Matrix Subroutine
E.4.8	AMCL02\$Local_Matrix_to_CDS_Array Subroutine
E.4.9	AMCL02\$Create_Work_Area Subroutine
E.4.10	AMCL02\$Reserve_Work_Area Subroutine
E.4.11	AMCL02\$Release_Work_Area Subroutine
E.4.12	AMCL02\$Init_Matrix_Work_Area Subroutine
E.4.13	AMCL02\$Init_Matrix_Work_Area2 Subroutine
E.4.14	AMCL02\$Get_Work_Area_Info Subroutine
E.4.15	AMCL02\$Get_Matrix_Work_Area_Info Subroutine
E.4.16	AMCL02\$Get_Matrix_Info Subroutine
E.4.17	AMCL02\$Create_Matrix Subroutine
E.4.18	AMCL02\$Put_Element Subroutine
E.4.19	AMCL02\$Get_Element Subroutine
E.4.20	AMCL02\$Add_Matrices Subroutine
E.4.21	AMCL02\$Subtract_Matrices Subroutine
E.4.22	AMCL02\$Multiply_Matrices Subroutine
E.4.23	AMCL02\$Multiply_Scalar Subroutine
E.4.24	AMCL02\$Transpose_Matrix Subroutine
E.4.25	AMCL02\$Matrix_Inversion Subroutine
E.4.26	AMCL02\$SVD Subroutine
E.5	Matrix Functions Example
E.6	Return Status Values

---

# Table of Contents

---

## APPENDIX F CDS MOVE AND MULTIPLE MOVE PARAMETER EXTENSION

F.1	Overview
F.2	Memory Requirements, Packaging, and AM Configuration
F.2.1	Memory Requirements
F.2.2	Packaging
F.2.3	Distributed Files
F.2.4	Installation
F.2.5	Application Module Configuration
F.3	CDS_Move
F.3.1	Overview
F.3.2	CL Aborts
F.3.3	Performance
F.3.4	CDS_Move Restrictions
F.3.5	AMCL01\$CDS_Move
F.4	Multiple_Move_Parameter
F.4.1	Introduction
F.4.2	Overview
F.4.3	Move Execution Order
F.4.4	Database Consistency
F.4.5	Dynamic Indirection
F.4.6	List Permanence
F.4.7	List Ownership
F.4.8	List Access
F.4.9	Redundancy
F.4.10	Error Handling
F.4.11	Parameter Access Request Limits and Restrictions
F.4.12	Calling the Subroutines
F.4.13	Multiple Move Subroutines
F.4.14	Error Message Summary
F.4.15	AMCL01\$Allocate_List
F.4.16	AMCL01\$Get_List_Id
F.4.17	AMCL01\$Modify_Entry
F.4.18	AMCL01\$Modify_Complex_Entry
F.4.19	AMCL01\$Multiple_Move_Parameter
F.4.20	AMCL01\$Set_Permanent_List
F.4.21	AMCL01\$Deallocate_List
F.4.22	AMCL01\$Get_List_Mem_Stats
F.4.23	AMCL01\$Change_List_Mem_Size
F.4.24	AMCL01\$Set_List_Item_Active
F.4.25	AMCL01\$Get_Exec_Time_Stats

---

# Table of Contents

---

## **APPENDIX G AM EXTENSION FOR FAST EXTERNAL CDS FETCH**

- G.1 Overview
- G.2 AM Performance Impact
- G.3 AM Memory Impact
- G.4 Packaging
- G.5 Distributed Files
- G.6 Installation
- G.7 Application Module Configuration

## **APPENDIX H AM EXTENSION FOR OFF-NODE ACCESS**

- H.1 Overview
- H.2 Functionality of the Extension
- H.3 AM Performance Impact
- H.4 AM Memory Impact
- H.5 Packaging
- H.6 Distributed Files
- H.7 Installation
- H.8 Application Module Configuration

## **APPENDIX I APPLICATION MODULE<sup>X</sup> CL RUNTIME EXTENSIONS**

- I.1 Overview
- I.2 Installation
- I.3 Executing X-Side Applications
- I.4 Hibernating Applications
- I.5 AMCL06\$Execute\_Task\_With\_Wait
- I.6 AMCL06\$Get\_Queue\_Info
- I.7 AMCL06\$Get\_Queue\_Slot\_Info
- I.8 AMCL06\$Store\_XAccess
- I.9 AMCL06\$Initiate\_Task
- I.10 AMCL06\$Activate\_Task
- I.11 AMCL06\$Terminate\_Task
- I.12 AMCL06\$Get\_Hiber\_Task\_Status
- I.13 Stopping X-Side Applications
- I.14 Using the kill\_appls X-Side Tool
- I.15 The X-side display\_appls Tool

## **INDEX**

## 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/AM.*

#### 1.1 PURPOSE/BACKGROUND

This publication provides reference information about Honeywell's Control Language for the TPS Applications Module (CL/AM). CL/AM 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/AM structure (although examples for the AM are included in Section 4); rather, it outlines the rules of CL/AM and describes all the components that can be used to build a CL/AM 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 Application Module (AM) and the data points that reside in it. You should also have an operational knowledge of the Universal Station's TEXT EDITOR, COMMAND PROCESSOR, and DATA ENTITY BUILDER functions that are available through the Engineer's Main Menu.

You do not need to be familiar with earlier Honeywell control languages (such as BPL, TASC, and SOPL), although understanding concepts of those languages may be helpful when applying CL/AM to strategies already built with one of the other languages.

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/AM.

#### 1.2 REFERENCES

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

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

*System Startup Guide, SW11-604*, in the *Implementation/Startup & Reconfiguration - 1* binder – Overview description—Tells how a CL/AM structure is configured into the TPS System.

*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/AM structure.

## 1.2.2 Publications with CL/AM-Specific Information

*Control Language/Application Module Overview, SW27-500*, in the *Implementation/Application Module - 3* binder Presents a tutorial introduction to CL/AM.

*Application Module Control Functions, AM09-502*, in the *Implementation/Application Module - 1* binder—Describe the TPS On-Line Control Software and strategies that may include CL/AM structures. Include discussions of data points that contain CL/AM structures and runtime information, such as possible CL/AM runtime errors.

*Control Language/Application Module Data Entry, AM11-585*, in the *Implementation/Application Module - 3* binder— Describes how to use the COMMAND PROCESSOR facility of the US to compile, link, and load CL/AM structures to an AM.

*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/AM Structure into an AM data point.

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

### 1.3 CL/AM OVERVIEW

CL/AM structures can be used to build a custom-control strategy; they are used to augment or replace the standard TPS algorithms. CL/AM structures can be grouped according to the type of process (continuous or discontinuous) to which the structure applies.

CL/AM applies to continuous processes and executes in the Application Module. CL/AM structures include CL/AM Blocks and Global Data Definitions (GDD). GDDs are Custom Data Segments (CDS), Parameter List Definitions (PLD), and Enumeration-Type Definitions (ETD). CL/AM Blocks manipulate data and GDDs are used to define data. Packages can be built that can contain one or more CL/AM structures that are to be bound (linked-loaded) to a single data point. CL/AM structures are discussed in Section 4.

#### 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/AM 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/Application Module 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/AM structure(s).
2. Use the Universal Station's **TEXT EDITOR** to write/enter the CL/AM structure(s); in other words, create the source file(s).
3. Use **CL** in the Command Processor on the US to compile the CL/AM structure. Compiling your source code turns it into an object file that can be executed in the AM.
4. Use the **DEB** to configure the CL/AM structure into the data point.
5. Use **CL** in the Command Processor on the Universal Station to link/load the CL/AM structure to the fully configured data point.





## RULES AND ELEMENTS OF CL/AM

### Section 2

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

#### 2.1 INTRODUCTION TO CL/AM RULES AND ELEMENTS

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

<u>ENGLISH</u>	<u>CL/AM</u>
grammar	syntax, rules
characters	characters
words, phrases,	data types, variables, declarations,
Clauses	expressions, conditions
sentences	statements (simple command or instruction to manipulate an element)
story, essay	CL/AM Block, Global Data Definition
collection of related stories, anthology	Package with CL/AM Block(s) and/or Global Data Definitions

The part of this section under heading 2.2 describes the basic rules ("grammar") and elements ("words") of CL/AM. The remainder of the section (headings 2.3, 2.4, and 2.5) describes more complex elements of CL/AM (that is, the "phrases" and "clauses" of CL/AM). 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/AM 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/AM 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/AM Statements and Structures.

### 2.2.1 Character Set Definition

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

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

Characters can be combined to generate the following basic elements:

- Comments
- Identifiers (including reserved words; see heading 2.2.7.4—Reserved Words Definition)
- Numbers
- Quotes
- 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; i.e., at least one space must separate adjacent identifiers or numbers. Spaces cannot appear within an element other than in quotes and comments.

### 2.2.3 Lines

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

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

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

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

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

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

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

```
BLOCK good                                (POINT a101;
&          AT Pre_GI;
&          WHEN PV > 17.6)
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:

```
BLOCK bad                                (POINT a101; AT Pre_
&          GI; WHEN PV > 17)              -- identifier can't span lines
LOCAL String_const = "This is a long
& String constant"                      -- String can't span lines
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 called a syntax diagram. The syntax 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 or as many times as required to produce the ID String), until you exit the diagram on the right.

Syntax diagrams have three different symbols with text in them: rectangles, rectangles with curved ends, and circles. Items in rectangles refer to a syntax diagram in another part of the manual. Rectangles with curved ends are reserved words in CL/AM 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/AM Statement.

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

## 2.2.5 CL/AM Restrictions

This section lists the restrictions placed on the language by either the CL 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:

- Data point identifier maximum length is system-dependent. See heading 2.2.5.3 for more information.
- Block identifiers can be no longer than 8 characters.
- Parameter 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.
- Include\_Set filenames can be no longer than 6 characters.

The following restrictions are imposed by the File System:

- The following can be no longer than 8 characters, must begin with an alphabetic character, and cannot contain ISO 646 (refer to heading 2.2.5.2) foreign characters:

Parameter list identifiers  
Enumeration type identifiers

### 2.2.5.2 ISO 646 Compatibility

The CL/AM 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/AM uses only the dollar sign.

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, for example, accented or umlauted characters. Such characters cannot be used as alphabets in CL/AM. (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/AM 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	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

### 2.2.5.3 Data Point Identifier Restrictions

The maximum length of data point identifiers depends on the tagname option selected from the SYSTEM WIDE VALUES menu for the local LCN. If the "short" TAG NAME SIZE is selected, then the data point identifier can be no longer than eight characters plus a 3-character PIN identifier. If the "long" TAG NAME SIZE is selected, then the data point identifier can be no longer than 16 characters plus a 3-character PIN identifier.

The Plant Information Network (PIN) identifier is used to identify off-LCN data points that are accessed through a Network Gateway (NG) node, and is only valid on a system which has an NG. A PIN identifier is composed of a 1- or 2-character remote LCN identifier and a 1-character delimiter (\).

Data point identifier examples:

<u>SHORT IDENTIFIER</u>	<u>LONG IDENTIFIER</u>	
mtr_sw	motor_switch	(local data points)
xy\valve1	x\valve_point1	(off-LCN data points)
xyz\valve1	valve_control_point	(invalid identifiers)

### 2.2.5.4 HPM Fieldbus Parameter Restrictions

There are CL/AM restrictions when accessing parameters of SECM, PECM, and FBCM fieldbus points. The parameters for the fieldbus points are arranged into three classes that are designated as class A, class B, and class C, and the CL/AM restrictions are based on these parameter classes.

- Class A parameters are fieldbus IOP parameters that are contained in the fieldbus IOP. Read and write requests for these parameters are directed to the fieldbus IOP database.
- Class B parameters are fieldbus-device parameters that are cached (and possibly mapped) in the fieldbus IOP. Read requests for these parameter values are directed to the IOP database. Write requests are directed to the fieldbus device.
- Class C parameters are fieldbus-device parameters that are not cached in the fieldbus IOP. Read and write requests for these parameter values are directed to the fieldbus device.

CL/AM programs can control the timing of fieldbus parameter access through the BACKGRND insertion point. Access to fieldbus parameters through a foreground CL/AM program could delay the program and cause processor overruns or use fieldbus data that is old. When creating CL/AM programs that use HPM fieldbus parameters, note that all CL/AM write operations to class B parameter values and all CL/AM read and write operations to class C parameter values must be delayed. This delay should be approximately twice the execution period of the fieldbus function block as specified by the link active schedule of the fieldbus segment.

Read operations to class B parameter values are immediate. However, the value that is read back may not be the current value. The parameter values are not synchronously updated with the CL/AM execution period.

The fieldbus parameters that are supported by CL/AM are contained in a listing in Section 1 (Introduction) of the *HPM Parameter Reference Dictionary*. For ease of use, the same parameter listing in Section 1 also contains the respective parameter class (A, B, or C) for each parameter.

#### 2.2.5.5 Off-LCN Reference Rules

If the Network Gateway is used to connect the local LCN with a remote LCN that uses a different tagname length, the following rules apply.

- A point data request from an LCN using the short name form sent to an LCN using the long name form will be honored. The target LCN will use the short name (padded with trailing spaces to long form length) to attempt a match. However, any such request that uses a tagname greater than 8 characters long will be rejected.
- A point data request from an LCN using the long name form sent to an LCN using the short name form will result in a "Point Not Found" response message if the tagname has more than eight non-blank characters (for example, ABCDEFGH is an acceptable tagname for this purpose, but ABCDEFGHI is not).

## 2.2.6 Comments

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

### 2.2.6.1 Correct Examples of Comments

```
-- A long introductory comment should be written like
-- this, with a separate "--" on each line.
LOCAL hi,          -- Individual comments
&    low,          -- may be placed on each
&    mid: Number-- continuation line
```

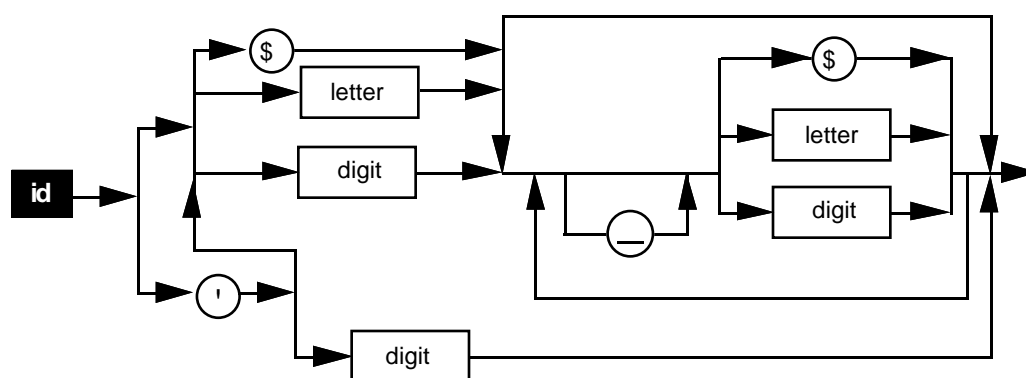
### 2.2.6.2 Incorrect Examples of Comments

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

## 2.2.7 Identifiers

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

### 2.2.7.1 Identifiers Syntax



### 2.2.7.2 Identifiers Description

Identifiers are composed of the dollar sign (\$), alphabetic characters (A to Z, and a to z), numeric characters (0 to 9), the backslash (\), and the break character (underscore or underbar (\_)). Note that the backslash is used only for the referencing of off-LCN data points and when used must be either the second or third character of the identifier.

Special identifiers (see heading 2.2.7.5—Special Identifiers Definition) are prefixed with an apostrophe ('), but the apostrophe is not part of the identifier. An identifier (except for special identifiers) cannot be a single digit numeric; a single letter or \$ is acceptable but not recommended.

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.

Upper-case and lower-case characters can be used in identifiers, but the distinction between cases is not significant; that is, each lower-case character is considered the same as its upper-case 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/AM program, there is no restriction on using identifiers that begin with dollar signs; however, such identifiers cannot be used to define any new object that is visible outside the program.



Use of the dollar sign to name system-visible objects is reserved to Honeywell. The CL/AM compiler does not let you create any of the following objects if its name begins with a dollar sign:

- Custom Data Segment parameters
- Parameter Lists
- Enumeration types
- Enumeration states

There are other system-visible identifiers (for example, data point names), but the objects listed above are the only ones that can be created by CL/AM.

The dollar sign can appear in any position for the following types of identifiers:

LOCAL id	GOTO id
BLOCK id	SUBROUTINE id
label id : LOOP FOR ...	DEFINE id
REPEAT id	

### 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      -- also valid
$abc       -- valid identifier, restricted use
$4995      -- also valid, restricted use
FILE$EXISTS -- name of a Runtime Extension subroutine
fe\al00    -- an off-LCN point identifier
x\b100     -- another off-LCN point identifier on a different LCN

Enumeration $Colors = Red/Blue/Green      -- NOT VALID
Enumeration Signal = $Red/$Yellow/$Green  -- NOT VALID

Param_List Fred$Sam -- NOT VALID
Parameter $Pt       -- valid only if the parameter name exists on
                     the point. Cannot be used to define a Custom
                     Data Segment parameter.

BLOCK a$bc (Generic; at General) -- valid
  Local $I                      -- valid
  $P1 : Loop FOR $I IN 1..10     -- valid
  Send : $I                      -- valid
  Repeat $P1                    -- valid
End a$bc                        -- valid

```

#### 2.2.7.4 Reserved Words Definition

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

**Table 2-2 — CL Reserved Words**

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

There are also a number of predefined identifiers in CL/AM. These identifiers are not reserved and can be redefined in a program; however, 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 states is predefined:

Logical = Off/On

**Insertion Point Names**—Insertion point names and their processing order vary according to the build-type of the data point to which they belong, as shown in Table 2-3.

For a regulatory data point, if a PV algorithm is configured as NULL, insertion points Pre\_PVAg through Pre\_PVa are not executed.

If a Control algorithm is configured as NULL, insertion points Pre\_SP through Pst\_CtAg are not executed. Refer to the *Application Module Control Functions* manual, Sections 3 and 4 for more information on the processing order for AM Regulatory Data Points.

PV\_Alg is executed only if the PV Algorithm is configured as CL/AM.

Ctl\_Alg is executed only if the Control Algorithm is configured as CL/AM.

Table 2-3 — Insertion Points by Build Type

Build Type	Insertion-Point Name
Regulatory Control	1. Pre_GI 2. Pre_PVPr 3. Pre_PVAg 4. PV_Alg 5. Pst_PVAg 6. Pst_PVFL 7. Pre_PVa 8. Pst_PVPr 9. Pre_CTPr 10. Pre_SP 11. Pre_CtAg 12. Ctl_Alg 13. Pst_CtAg 14. Pst_CtPr 15. Pst_GO 16. Backgrnd
Custom	1. General 2. Backgrnd
Switch	1. General 2. Backgrnd

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

Abs	— Built-in Function
Atan	— Built-in Function
Avg	— Built-in Function
Backgrnd	— Regulatory, Custom or Switch insertion-point name
Badval	— Built-in Function
BKG_Change_Priority	— Built-in Subroutine
BKG_Delay	— Built-in Subroutine
BKG_Switchover_Restart	— Built-in Function
CDS_Read	— Built-in Subroutine
CDS_Write	— Built-in Subroutine
Cos	— Built-in Function
Ctl_Alg	— Regulatory, Logic insertion point name, Seg_Class state name
Date_Time	— Built-in Function
Day_Time	— Built-in Function (MC only)
Delete_File	— Built-in Subroutine
Engineer	— Access_Lock state name
Entity_Bldr	— Access_Lock state name
Enum_Value_Store	— Built-in Subroutine
Equal_Null_Point_ID	— Built-in Function
Equal_Point_ID	— Built-in Function
Exists	— Built-in Function
Exp	— Built-in Function

General	— Custom, Switch insertion point name, Seg_Class state name
Get_CL_Slot	— Built-in Subroutine
Int	— Built-in Function
Len	— Built-in Function
Ln	— Built-in Function
Log10	— Built-in Function
Logical	— Type name
Max	— Built-in Function
MC	— Sequence program type
Min	— Built-in Function
Modify_String	— Built-in Subroutine
Move_Parameter	— Built-in Subroutine
Now	— Built-in Function
Number	— Type name
Number_to_String	— Built-in Subroutine
Off	— Logical state name
On	— Logical state name
Operator	— Access_Lock state name
ORD	— Built-in Function
PM	— Sequence program type
Pre_CtAg	— Regulatory insertion-point name
Pre_CtPr	— Regulatory insertion-point name
Pre_GI	— Regulatory insertion-point name
Pre_PVa	— Regulatory insertion-point name
Pre_PVAg	— Regulatory insertion-point name
Pre_PVpr	— Regulatory insertion-point name
Pre_SP	— Regulatory insertion-point name
Program	— Access_Lock state name
Pst_CtAg	— Regulatory insertion-point name
Pst_CtPr	— Regulatory insertion-point name
Pst_GO	— Regulatory insertion-point name
Pst_PVAg	— Regulatory insertion-point name
Pst_PVFL	— Regulatory insertion-point name
Pst_PVPr	— Regulatory insertion-point name
PV_Alg	— Regulatory insertion-point name, Seg_Class state name
Round	— Built-in Function
Set_Bad	— Built-in Subroutine
Set_Null_Point_ID	— Built-in Subroutine
Sin	— Built-in Function
Sqrt	— Built-in Function
String	— Type name
Sum	— Built-in Function
Supervisor	— Access_Lock state name
Tan	— Built-in Function
Time	— Type name

### 2.2.7.5 Special Identifiers Definition

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

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

### 2.2.7.6 Special Identifiers Examples

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

### 2.2.7.7 Conflicts Between Identifiers

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

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

1. There are 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.

The exceptions to this rule are 1) custom parameters cannot have the same name as other objects and 2) custom parameters can have the same name as custom data segments.

#### Custom Data are

- Custom Data Segments
- Custom Parameters

#### Program Units are

- Blocks
- Labels
- Subroutines

#### Objects are

- Local variables
- Local constants
- Functions
- Arguments
- Data Points
- Parameters
- Enumeration states

#### Data Types are

- Number
- Time
- String
- Logical
- Enumerations
- Parameter Lists

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.

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

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

**Scopes** are

Blocks	Subroutines
Parameter Lists	Functions
Custom Data Segments	

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

Scopes can sometimes be nested. Local Subroutines of a Block 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. 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.

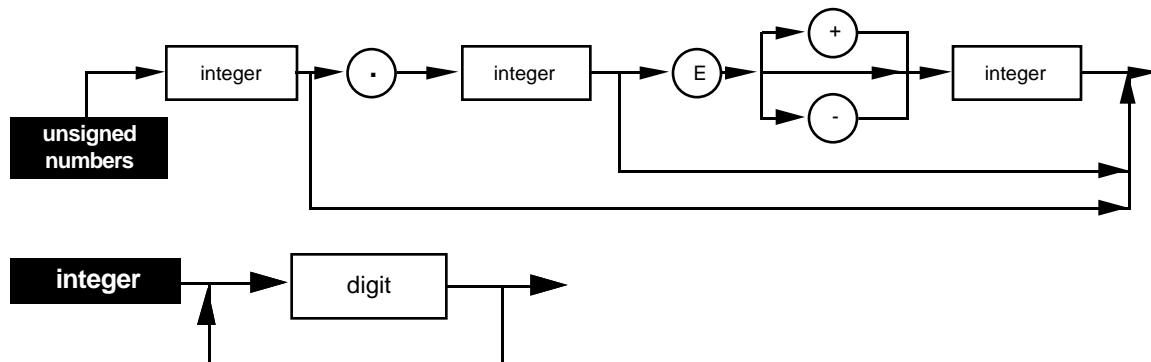
Within packages, Custom Data Segment parameters behave as parameters of the bound data point; therefore, they are visible in all other scopes (except Parameter Lists) in that package, and no other such scope can declare any object in conflict with a Custom Data Segment parameter's name.

4. There are two special situations where the name of an enumeration state of a parameter conflicts with the name of a built-in arithmetic function. The enumeration state INT of the AM \$REG\_CTL parameter INITTYPE conflicts with the arithmetic function Int(x), and in the HG, the enumeration state SQRT of the MC regulatory control point parameter ALGIDDAC conflicts with the arithmetic function Sqrt(x). Because of this conflict, a CL block cannot reference the INT state of the parameter INITTYPE or the SQRT state of the parameter ALGIDDAC. The only way to determine if INITTYPE is in the INT state or if ALGIDDAC is in the SQRT state is to test that the parameter is **not** in any of its **other** possible enumeration states. (See 4.3.7.1 for a list of the built-in arithmetic functions.)

## 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 uppercase or lowercase), 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

```

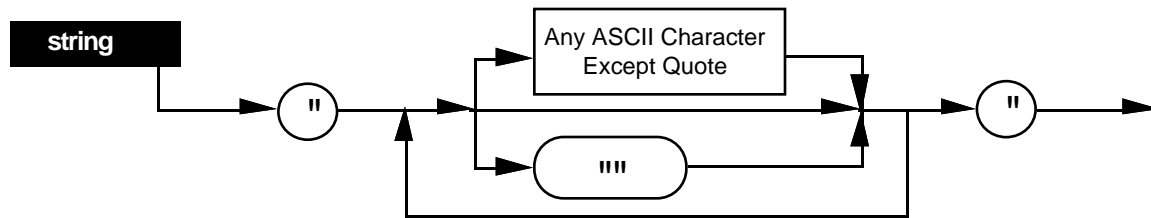
1000 -- valid
1000.    -- NOT VALID; no trailing digit
1000.0   -- valid; same as 1000
0.5      -- valid
.5       -- NOT VALID; no leading digit
10.02E1  -- valid
10.02e1  -- valid; same as 10.02E1
10.02 e1 -- NOT VALID; embedded space
1234.0E-2 -- valid
1234E-2  -- NOT VALID; "1234E" is an identifier
1234.0E -2 -- NOT VALID; embedded space
1234.0E+2 -- valid

```

## 2.2.9 String

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

### 2.2.9.1 String Syntax



### 2.2.9.2 String Description

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

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.

### 2.2.9.3 String Examples

```

"This is a String"      -- a String
"&@$?! system"         -- can contain any printable characters
""                      -- the empty String
"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 2-4 are special symbols:

**Table 2-4 — Special Symbols**

Symbols	Meaning
+ - * / **	Arithmetic operators
< <= > >= <>	Relational operators
=	Equality, assignment operator
..	Range separator
()	Parentheses
::: . ,	Punctuation
"	String separator
--	Comment separator
&	Line continuation
'	Special identifier prefix
\	Separator for off-LCN point identifier



## 2.3 CL/AM DATA TYPES

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

In CL/AM, named Enumeration types and Parameter-List types can be defined in Global Data Definitions. All other data types are built into the language.

There are two kinds of data types: scalar and composite.

Scalar types have no components. They are the built-in types: Number, Time, and Discrete types (Logical and Enumeration types).

Composite types are data points (because a data point can have lots of parameters/components), arrays (again, lots of components), and the built-in type String.

### 2.3.1 Number Data Type

All numeric values in CL/AM 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/AM; numbers may, of course, have integer values, and CL/AM 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.

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

#### 2.3.1.1 Bad Values

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

In the AM, any numeric variable that is incorrectly read is received as a bad value, which can be checked using the built-in predicate Badval, described in Section 2.5.4.5.

A non-numeric variable, however, has no bad value. If an error occurs reading such a variable, it is inaccessible. An attempt to access it is a runtime error and causes the program to abort. The variable can be tested without aborting the program using the built-in function Comm\_Error, described in heading 4.3.7.3.

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

A bad value can arise

- when a variable was never initialized
- as the result of an invalid arithmetic operation; for example,
  - magnitude subtraction of infinities
  - zero multiplied by infinity
  - zero divided by zero
  - infinity divided by infinity
  - infinity MOD any number
  - any number MOD zero
  - square root of a negative number
- because a value read from an external data point was inaccessible (perhaps temporarily)
- or, because an operator or program explicitly chose to store a bad value.

CL/AM supports bad values through the use of

- the predicate `Badval(x)`, which tests the value `x` to determine whether it is bad
- the procedure `Set_Bad(x)`, which stores a bad value into the variable `x`
- the procedure `Allow_Bad(x,y)`, which stores the value `y` into the variable `x`, whether `y` is Normal or Bad
- the built-in subroutine `Move_Parameter` which allows bad value storage
- the automatic propagation of bad values that take part in any arithmetic operation, such that if any operand is bad, the result is Bad
- the abort of a CL/AM block on any attempt to store a bad value as a parameter of any data point, except by using `Set_Bad` or `Allow_Bad`
- the abort of a CL/AM block on any attempted comparison that involves a bad value. Note that this means that the only way to test a number to see if it is Bad is by using `Badval`, because comparing a bad value with anything, even another bad value, aborts any CL/AM program.

Neither `Allow_Bad` nor `Move_Parameter` will allow a bad value to be stored into a parameter of type integer (a runtime "BadValSt" error is reported and the value is not stored).

### 2.3.1.2 Infinite Values

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

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

New infinities arise

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

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

The built-in predicate, Finite, is provided to test for infinities; it is defined under heading 2.5.4.5.

### 2.3.1.3 Uncertain Values

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

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

CL/AM does not automatically propagate uncertain values. You can propagate an uncertain value by storing the state UNCERTN into the PVAUTOST parameter, which updates PVSTS (you cannot write directly to PVSTS).

## 2.3.2 Time Data Type

The data-type Time represents an interval of time in the TPS format. Time values can be expressed in seconds, minutes, hours, or days, or any combination of these.

The minimum resolution of a Time value is one second. The range of a Time value is from  $-(2^{*}31)$  seconds to  $+(2^{*}31 - 1)$  seconds; that is, plus or minus approximately 68 years.

Time values are always exact to the second. They are computed by Time expressions, which are like arithmetic expressions. Values of data-type Number are converted to values of data-type Time by designating the Number as an operand in a time literal. Values of type Time are converted to values of type Number using the "Number" built-in function.

The CDS attribute "value" does not apply to Time. See heading 4.4.5.2.

### 2.3.3 Discrete Data Types

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

CL/AM 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**.

Enumeration data types recognized by CL include the following:

- Honeywell-defined standard enumerations. Examples include MODE, PTEXECST, and ALENBST.
- Customer-defined standard enumerations. These are defined by the CL ENUMERATION statement; see heading 4.6 in this manual.
- Self-defined enumerations (where the customer defines the state names). Examples are the PVs of digital points and flag points.
- Enumerations defined in the CL context only. An example is

```
LOCAL flavors : chocolate/vanilla/strawberry
```

Variables of discrete types can be compared or assigned to only variables or values of the same type; however, the compiler directive %RELAX Linker\_SDE\_Checks allows linking of a CL block that generically references certain self-defined enumeration parameters regardless of their true state names. See heading 3.3.4 in this manual.

#### 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

All discrete types, except Logical, are called Enumeration types. These types are defined by enumerating (specifying) all the possible states that the type can assume. Enumeration types can be globally or locally defined.

Compiling a named global Enumeration-type definition causes that definition to be permanently placed in the system's global data-definition files; thereafter, it can be used by any program simply by naming it. Once an Enumeration-type definition is placed in the system's global data-definition files, it cannot be changed. For example

```
ENUMERATION traflite = red/amber/green
...
BLOCK foo (GENERIC)
LOCAL lights: traflite ARRAY (1..10)
```

A variable or parameter can be locally declared to have a certain Enumeration type; this is done by directly naming the states in the variable declaration, rather than naming the type:

```
LOCAL fred: red/amber/green
```

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

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

A few Enumeration types are known to the CL/AM 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 a system-defined type with states Off/On.

Variables of Enumeration Types can be declared in CL/AM 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.

Also see heading 4.4.3.

### 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, truth values are found in only conditional tests and cannot be stored in variables.

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

Language: Pascal		CL
	VAR flag: Boolean; x: real;	LOCAL flag: Logical 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/AM 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/AM method A is just as efficient as Pascal method A and is easier to read.

Pascal method B and CL/AM 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**. (A data point is like a Pascal RECORD.) Data points are defined through the Engineering Personality (EP) Data Entity Builder (DEB), rather than by CL. Individual parameters generally are identified by dot notation; that is, the point name is followed by a dot (period), then by the parameter name, thus: `point_name.parameter_name`.

#### Example: Data point parameter names

```
A100.PV      -- parameter PV of data point A100
B100.SP      -- parameter SP of data point B100
fe\C100.MODE -- parameter mode of data point named C100 on a
              -- remote LCN named fe
```

#### 2.3.4.1 Bound Data Point Definition

Every CL/AM program must be bound to a particular Application Module data point (of type Regulatory, Switch, or Custom) in order to execute. This is known as its Bound Data Point. The program heading identifies whether the program is **specific** (see heading 2.3.4.3) or **generic** (see heading 2.3.4.4). A specific program can be bound only to the named data point. A generic program can be bound to any number of similar points. Blocks may not be bound to off-LCN points.

The point name is not used when referencing parameters of the point to which the CL is attached (its Bound Data Point) or any Custom Data Segment (CDS) parameters of the Bound Data Point. Use only the parameter name when referencing parameters of the Bound Data Point (or its CDS).

#### Example: Bound Data point parameter name

```
PV      -- parameter PV of the Bound Data Point
```

#### 2.3.4.2 External Data Point Identifiers

CL/AM statements can read/write/compare the parameters of other LCN data points (those external to the Bound Data Point), both on and off-LCN, through both direct and indirect references to the point and parameter names, using dot notation (and the PIN identifier for off-LCN points) as previously described.

A **direct reference** to a data point parameter external to the Bound Data Point requires an EXTERNAL declaration (see heading 2.4.4). The point and parameter names of direct references must exist as valid references and data types at compile time. If an off-LCN point is referenced directly in a CL Block using the EXTERNAL statement, the Block must either be linked to a point on the Internetwork Point Processor (IPP), or be at the BACKGRND insertion point. This is checked by the linker.

**Example: Direct data point reference**

```

BLOCK dir (GENERIC: AT PRE_GI)
  EXTERNAL A100          -- A100 is the name assigned to a
                        -- previously built on-LCN data point
  EXTERNAL ab\C100       -- C100 is the name assigned to a
                        -- previously built off-LCN data point
  IF A100.SP > ab\C100.SP THEN SEND: "A100's SP is larger"
                        -- reads the sp of point A100 and point
                        -- ab\C100 and sends a message if
                        -- A100's SP is larger than ab\C100's SP
  ELSE SEND : "ab\C100's SP is larger"
                        -- otherwise send a message that
                        -- ab\C100's SP is larger
END dir

```

**Indirect references** to data point parameters external to the Bound Data Point require the external point names to be available at CL link time (not earlier at compile time). Indirect reference uses a Custom Data Segment to define a custom parameter of type “data point identifier” (see heading 4.4.4) and permits CL references to parameters of the point to compile so long as the parameters are named in a parameter list which describes the point (see heading 2.3.4.4).

The CL is written to reference external points and parameters indirectly through the names assigned to the CDS parameters of type point id. The compiler uses the CDS parameter name instead of the actual external point name.

At link time, the program’s indirect point id references are bound to the points declared in the CDS parameters. The CL/AM linker checks that the referenced point ids and their parameters exist and that data types are as specified at compile time. The link will not complete if the parameter references cannot be resolved.

Once the program is linked, the CDS point id can be changed to reference another point identifier. A runtime error will be produced if the new point does not contain the same parameters of the same data types as the old point. Also, if you unlink the program following a change of point identifier with this type of error, and then attempt to link it again, the link will fail.

The sequence for using an indirect reference involves:

- a) Declare a Custom Data Segment (CDS) parameter to be of type Parameter List. (You either create your own parameter list or reference the predefined parameter list named \$REG\_CTL. See heading B.3 for a listing of the names and data types of the parameters in \$REG\_CTL.)
- b) Write CL source statements which indirectly reference the parameters of the point using the CDS parameter name. (The source can be entered and the CL compiled without defining the external point name.)
- c) During DEB point build and load operations, attach the CDS to an AM point through its PKGNAME(x) parameter, provide space for the CL to be linked onto the same AM point through the parameter CLSLOTS, and enter the external point name into the custom parameters of type parameter list. The external point name may be either on-LCN or off-LCN.



- d) Link the compiled CL object file to the AM point using the CL commands LK or CLK. If an off-LCN point is referenced indirectly in a CL Block, the Block must either be linked to a point on the Internetwork Point Processor (IPP), or be at the BACKGRND insertion point. This is checked at link time and at run time.

**Example: Indirect data point reference using a custom parameter list**

```

PACKAGE

PARAM_LIST mytyp
  PARAMETER mypv: NUMBER          -- This will be a valid compile
                                   -- parameter. It must exist and
                                   -- be of type NUMBER at link time.
END mytyp

CUSTOM
PARAMETER pnt1 : mytyp           -- Parameter of type point id.
VALUE A100                       -- Indirect ref via CDS parameter
EU "IND_ID"                      -- of type parameter list (A100 is
                                   -- established as the DEB default
                                   -- value).
END CUSTOM

BLOCK ind (GENERIC; AT GENERAL)
  IF pnt1.mypv > 10.0 THEN SEND : "REF_PT IS OVER 10"
-- Reads the 'mypv' parameter of the point named in the parameter pnt1,
-- and sends a message if the value is greater than 10. This
-- indirection starts "on-point." 'mypv' must be a valid parameter on
-- the point referenced (with the list defined above, the parameter
-- 'mypv' is the only parameter this CL can reference)
  END ind
END PACKAGE

```

**Example: Indirect data point reference using the standard parameter list \$REG\_CTL, that references an off-LCN data point**

```

PACKAGE

CUSTOM
PARAMETER pnt1 : $REG_CTL        -- Parameter of type point id.
VALUE fe\B100                   -- Indirect ref via CDS parameter
EU "IND_ID"                     -- of type parameter list
                                   -- (fe\B100 is established as the
                                   -- DEB default value).
END CUSTOM

BLOCK ind (GENERIC; AT GENERAL)
  IF pnt1.PV > 10.0 THEN SEND: "REF_PT IS OVER 10"
-- Reads the 'pv' of the point named in the parameter pnt1 and
-- sends a message if its value is over 10.
-- PV must be a valid parameter on the point referenced.
-- This indirection starts "on-point."
  END ind
END PACKAGE

```

Additional features and restrictions of the indirect reference to point identifiers are described at heading 2.3.8, Indirect Reference to Point Identifiers.

### 2.3.4.3 Specific Programs

A CL/AM program written so that it can only be bound to one specific point is called **specific** (see **Exception** below). Any parameter of the data point identified in the program header can be referenced in the program without need to specify the parameter name and data type. The point named in the header must exist at compile time, must be an AM Regulatory, Switch, or Custom point, and must be on the local LCN.

#### Example: Specific CL block named somename

```
BLOCK somename (POINT reg44; AT PRE_GI)

-- The compiler "understands" the parameters pv and mode,
-- and their data types.

IF pv > 10.0 THEN SEND : "PV of reg44 is in trouble"
IF mode = man THEN SEND : "mode of reg44 is manual"

END somename
```

#### Example: Specific CL block named example

```
BLOCK example (POINT x\A100; AT GENERAL)
    -- invalid, you cannot name an off-LCN point
    -- in a BLOCK heading
```

**Exception:** A specific CL block can be linked to more than one point if:

- The block name is the same,
- The points are in the same unit, and
- Only the block heading is different.

For example, the two blocks "TEST," identical except for the headings shown below, can both be linked as long the body of each block is identical and unchanged.

```
BLOCK TEST (POINT A100; AT PV_ALG)

BLOCK TEST (POINT B100; AT PV_ALG)
```

The object code for these two blocks is identical, and the system will keep only one copy. If, however, the code in the block is changed (or recompiled), the block will have to be unlinked from and relinked to A100 and B100 (and any other points in the unit to which it was linked).

### 2.3.4.4 Generic Programs, the PARAMETER Statement, and Parameter Lists

A CL/AM program written so that it can be bound to any of several similar data points is called **generic**. A generic program does not specify a particular point id in its heading, but is bound to a specified point on the local LCN at link time. Any parameters of the Bound Data Point that are referenced in the program must have their names and data types defined to the compiler but they are not required to exist until the CL program is linked to an AM point.

There are two ways to define the Bound Data Point parameters of a generic program:

1. A **PARAMETER** declaration located after the **BLOCK** heading and before the first executable statement. This names the parameter and defines its type.

**Example: Generic CL using PARAMETER declarations**

```
BLOCK cusc2 (GENERIC ; AT GENERAL)
  PARAMETER running      : LOGICAL                -- defines one logical
  PARAMETER numarray     : NUMBER ARRAY (1..10)    -- parameter and an
                                                    -- array of numbers
-- The custom parameters "running" and "numarray" must exist on the
-- AM data point when the CL is linked to it

-- When "running" parameter is true, move 10th number to first number
IF running = ON THEN (SET numarray(1) = numarray(10);
&                      SET running = OFF)
  ELSE SEND: "cusc2 is not running"
END cusc2
```

2. A **Parameter List** reference in the program heading. The named parameter list defines a list of parameters and their data types that will exist on the Bound Data Point. The list must either have been compiled separately before its use in the program, or appear before the **BLOCK** in a **PACKAGE** compilation.

**Example: Parameter List and Custom Data Segment to be used by Generic CL**

```
PACKAGE                                -- CL source file (e.g., parcus.cl)
  PARAM_LIST cuspar
    PARAMETER running : LOGICAL        -- defines one logical
    PARAMETER numarray : NUMBER ARRAY (1..10) -- parameter and an
                                           -- array of numbers
  END cuspar
  CUSTOM
    PARAMETER running : LOGICAL        -- defines one logical
    PARAMETER numarray : NUMBER ARRAY (1..10) -- parameter and an
    VALUE (1,2,3,4,5,6,7,8,9,10)      -- array of numbers
  END CUSTOM
END PACKAGE                            -- PACKAGE must be compiled to
                                       -- update the Custom Name Library
```

**Example: Generic CL using the Parameter List "cuspar"**

```
BLOCK cuscl (GENERIC cuspar; AT GENERAL)
-- The custom parameters "running" and "numarray" must exist on the
-- bound data point when the CL is linked to it (by adding the PACKAGE
-- "parcus" during point build)

-- When "running" parameter is true, move 10th number to first number
IF running = ON THEN (SET numarray(1) = numarray(10);
&                      SET running = OFF)
  ELSE SEND: "cuscl is not running"
END cuscl
```

**Example: Generic CL using standard Parameter List \$REG\_CTL**

```

BLOCK regcl (GENERIC $REG_CTL; AT PRE_GI)
-- $REG_CTL contains standard regulatory parameters used by Generic
-- CL. The parameters PV, SP, OP, and MODE must exist on the Bound
-- Data Point when the CL is linked.

    IF MODE = MAN THEN SEND: "mode is manual"
    IF OP > 50      THEN SEND: "op > 50"
    IF SP < 50      THEN SEND: "sp < 50"
    SEND: "pv is ", PV

END regcl

```

At link time, the program is bound to a specific data point. The CL/AM linker checks that every parameter referenced by the program is actually possessed by the bound data point, and that its data type is as specified at compile time. Parameters that are not used by the program are not checked by the linker, even though they may appear in a **PARAMETER** statement or in a parameter list.

The bound data point can have many parameters that are not described to the program, but the program cannot refer to them. Note that **PARAMETER** declarations within a program can replace or supplement parameter lists.

Generic programs can use the parameter **ENT\_TYPE** (common to all data points) to determine a data point's type and location. See the *HG, NIM, AM, or CG Parameter Reference Dictionary* for the relevant enumerations of this parameter. A simple example of this technique follows.

```

BLOCK entity (GENERIC; AT GENERAL)
EXTERNAL hg05pm01
EXTERNAL pm02
IF hg05pm01.ENT_TYPE = PRCMODHG THEN SEND : "hg ok"
IF      pm02.ENT_TYPE = PRMODNIM THEN SEND : "nim ok"

```

Parameter Lists also are used as the data type definition for parameters of type data point (see heading 4.4.4). This is used for indirect referencing of data points external to the Bound Data Point (see heading 2.3.4.1). The following example shows a package containing a user defined Parameter List and a Custom Data Segment, followed by a Generic block that uses indirect addressing.

**Example: Custom Parameter Lists, CDS, and Indirect Referencing**

```

PACKAGE                                -- CL source file (e.g., entid.cl)
  PARAM_LIST refptr
    PARAMETER running : LOGICAL -- defines a logical parameter and
    PARAMETER numarray : NUMBER ARRAY(1..10) -- an array of numbers
  END refptr

  CUSTOM
    PARAMETER cuspt : refptr          -- Parameter of type point id
                                      -- used to indirectly reference a
                                      -- point that contains the custom
                                      -- parameters running & numarray.

    PARAMETER regpt : $REG_CTL        -- Parameter of type point id used
                                      -- to indirectly reference a point
                                      -- with regulatory point parameters.

  END CUSTOM

BLOCK entcl (GENERIC; AT GENERAL)
  -- This generic block references two external points indirectly
  -- through CDS parameters of type point id instead of direct access
  -- through EXTERNAL statements.

  -- At link time the on-point CDS parameters "cuspt" and "regpt" must
  -- contain point ids that contain the promised parameters.
  -- For example, A100 is the point identifier in "cuspt" and B100 is
  -- the point identifier in "regpt".

  -- Parameters "running" and "numarray" must exist on A100 and mode,
  -- op, pv, and sp must exist on B100 when the CL program is linked

  IF cuspt.running = ON THEN
    (SET cuspt.numarray(1) = cuspt.numarray(10);
    &
    SET cuspt.running = OFF)
  ELSE SEND: "entcl is not running"

  IF regpt.MODE = MAN THEN SEND: "mode is manual"
  IF regpt.OP > 50 THEN SEND: "op > 50"
  IF regpt.SP < 50 THEN SEND: "sp < 50"
  SEND: "pv is ", regpt.PV
  END entcl

END PACKAGE

```

### 2.3.4.5 Aliasing Definition

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

On the AM, aliasing also can arise when a datum is accessed indirectly.

CL/AM 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.

CL/AM cannot refer to the same point both directly and indirectly, or indirectly through more than one path. Attempted aliasing cannot be discovered at compile time. Aliasing errors are fatal if detected when a CL program is linked and cause a runtime error when the indirect identifier of a CL is incorrectly changed to reference the Bound Data Point.

### 2.3.4.6 Aliasing Examples

```
BLOCK xyz (POINT A_100;      -- the bound data point
... )
EXTERNAL A_101              -- another data point
PARAMETER foo : $REG_CTL    -- data point identifier
...
SET PV = 5                  -- The bound data point's PV
SET A_101.PV = 5            -- External point's PV
SET SP = A_101.SP          -- Each point's SP
SET A_100.PV = 5            -- NOT VALID: aliasing
SET foo.PV = 5              -- Indirect through A_100.foo
```

### 2.3.4.7 Box Data Point Identifiers Definition

A box data point is a data point associated with a process-connected box, such as a Multifunction Controller (MC) or a Process Manager (PM). It represents box parameters that are visible to TPS components, including CL/AM programs. 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 or PM 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) 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 but from outside as TMST(nn). In general, parameters visible from outside the box are described in the *Hiway Gateway Parameter Reference Dictionary* and the *Process Manager Parameter Reference Dictionary*. (An exception to this general rule is that only the first 1023 box numerics of a PMM can be accessed from the AM as box data point parameters; the remainder—box numerics 1024 through 2047—can be accessed from the AM only when they have data points built against them.)

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

\$HYnnBmm — for a Multifunction Controller on Hiway nn Box mm  
 \$NMnnBmm — for a Process Manager on UCN nn Device mm

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

#### 2.3.4.8 Box Data Point Identifier Examples

```
$HY01B32.TMST(03) -- external view of timer state
                    -- (e.g, AM store/write)*
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
```

- \* Notice that, in the first example, the box parameter TMST(03) CANNOT be read by a CL/AM program in the AM (in a statement such as SET A = \$HY01B32.TMST(03)). This is because for array-parameter fetches, CL/AM in the AM always fetches the entire array; however, entire-array fetches are not supported by the HG. This example of a parameter reference, when used to fetch/read an array parameter (which include MC Timers, Flags, Logic Blocks, Numerics and Counters) generates a configuration error at runtime; store/writes to these parameters work.

There are two alternative ways for CL/AM in the AM to fetch/read these parameters; these alternatives are explained using an MC Timer's state parameter as an example.

1. Build a Timer (or Flag, Numeric, etc.) data point referencing Timer 03 in the MC. Then CL/AM in the AM could read the timer's state as follows:

```
SET A = TIMER3.STATE.
```

Note that the state parameter is referenced as STATE, the HG name for the state of a Timer (HG-based) data point.

2. Build an AM regulatory point.  
 Use the source of a general input connection (GISRC(n)) to reference the MC box parameter, and then send it to a CDS parameter as the destination of the general input (GIDSTN(n)).

For example, if GISRC(1) = \$HY01B32.TMST(03) and GIDSTN(1) = TIMER3, (TIMER3 is a CDS parameter) then CL/AM in the AM could read the timer's state as follows:

```
SET A = TIMER3
```

Note that the state parameter is referenced as TMST(nn), which is the name of a timer's state (inside the MC view) of the MC box data point.

### 2.3.5 Arrays Data Type

In the AM, arrays are composed of any scalar type, data point type, or the String type and can have one or two dimensions. They can be indexed by any scalar type except Time. String arrays can have only one dimension.

An array whose index type is Number can be indexed by any arithmetic expression. If the result of the index expression is not an integer, it is rounded to the nearest integer. Arrays of data point identifiers cannot be declared as local variables. They must be the parameters of some data point in the system.

A CL/AM background program can fetch an element of an array of strings in an off-unit point, but a foreground program cannot unless the AM Extension for Off-Logical-Node Access (AMCL05) is loaded. If AMCL05 is not loaded, string array elements can only be fetched by a foreground CL/AM program from on-logical-node points (points in the same unit). If AMCL05 is loaded, a foreground CL/AM program can fetch string array elements from any on-physical-node points (points in the same AM), including points in other units. This is checked at link time. Neither foreground nor background programs can move whole arrays of strings. This is detected by the linker for foreground programs and by the runtime software for background programs. The AMCL05 extension is covered in Appendix H.

The logic Block Status Output parameter, SO, of the PM/APM/HPM can be accessed from a CL/AM foreground program only if the LOGMIX parameter is configured for 12\_16\_4 for the PM or 12\_24\_4 for the APM or HPM. Other LOGMIX configurations can be accessed only from background CL programs. See the Control Functions and Algorithms manual for the PM, APM, or HPM, subsection 5.2, for a description of LOGMIX.

There are restrictions on CL/AM program access to array parameters in the MC and PM. Statements that perform store/fetch operations have different restrictions than those for the built-in subroutine `Move_Parameter`. The restrictions also differ between the MC, the PM, and the APM/HPM and between foreground and background CL Blocks. The following tables summarize these differences.

#### NOTE

The box point parameters FATCONN, IOMCARD, IOMFILE, IOMCMD, IOMPER, XIOCHAST, and XIOCHBST are only accessible from Background CL/AM programs, similar to Box Numerics and Flags today.

**Table 2-5 — MC Array Data Access Rules**

	Foreground CL				Background CL			
	Single Element		Array		Single Element		Array	
	Fetch	Store	Fetch	Store	Fetch	Store	Fetch	Store
Statement Access	no	yes	no	no	yes	yes	no	no
Move_Parameter Access	no	yes	no	no	yes	yes	no	no



**Table 2-6 — PM Array Data Access Rules**

	Foreground CL				Background CL			
	Single Element		Array		Single Element		Array	
	Fetch	Store	Fetch	Store	Fetch	Store	Fetch	Store
Statement Access	yes*	yes	no	no	yes	yes	no	no
Move_Parameter Access	yes*	yes	yes*	no	yes	yes	yes*	no

\*Fetches from the following types of PM arrays are not supported: Box Numerics and Box Flags. Also, the following PM parameters are not supported: IOMCARD, IOMFILE, and IOMOPER.

**Table 2-7 — APM/HPM Array Data Access Rules**

	Foreground CL				Background CL			
	Single Element		Array		Single Element		Array	
	Fetch	Store	Fetch	Store	Fetch	Store	Fetch	Store
Statement Access	yes*	yes	no	no	yes	yes	no	no
Move_Parameter Access	yes*	yes	yes*	no	yes	yes	yes*	no

\*Fetches from the following types of APM arrays are not supported: Box Numerics and Box Flags, Box Strings, Box Times, Process Module Strings, and Array Point Strings. Also, the following APM array parameters are not supported: FATCONN, IOMCARD, IOMFILE, IOMOPER, XIOCHAST, and XIOCHBST. Note that CL/AM can access only the first 4095 elements of Box Numerics, Box Flags, Box Strings, or Box Times if configured.

CL/AM access to the APM Array Point's "TIME" and "TIMESECS" parameters follow rules associated with the number of parameters configured, as shown in Table 2-8.

**Table 2-8 — Access to Array Point "Time" and "Timesecs" Parameters**

	Foreground CL				Background CL			
	Single Element		Array		Single Element		Array	
	Fetch	Store	Fetch	Store	Fetch	Store	Fetch	Store
NTIME, with NTIME <= 160								
Statement Access	yes	yes	no	no	yes	yes	no	no
Move_Parameter Access	yes	yes	yes	no	yes	yes	yes	no
NTIME, with NTIME > 160								
Statement Access	no	yes	no	no	yes	yes	no	no
Move_Parameter Access	no	yes	no	no	yes	yes	no	no
TIMESECS, with NTIME <= 240								
Statement Access	yes	yes	no	no	yes	yes	no	no
Move_Parameter Access	yes	yes	yes	no	yes	yes	yes	no

The "TIMESECS" parameter has reduced precision (the value is rounded up and the tenths of milliseconds is zero).

The CL/AM linker will generate an error when a CL/AM foreground program fetches the TIME parameter and the Array Point referenced has an NTIME value greater than 160. A CL runtime configuration error occurs if the Array Point NTIME value is changed to be greater than 160 after the foreground program is linked.

A CL runtime configuration error occurs if the Array Point NTIME value is greater than 160 and a CL/AM background program fetches the entire array with Move\_Parameter.

### 2.3.5.1 Arrays Examples

```
coeff(6)      -- one dimension
matrix(6, 8)  -- two dimensions
status(Casc)  -- indexed by discrete type
pump(x+y*z)   -- indexed by expression
```

## 2.3.6 String Data Type

This type represents variable-length Strings, as in the BASIC language. Their main uses are in the SEND statement and as data point parameters. Strings can be assigned and compared, and their lengths can be tested with the built-in function LEN. CL/AM string variables can be up to 78 characters long.

CL/AM string manipulation features are provided by the built-in subroutines Modify\_String and Number\_to\_String (see heading 4.3.7.5) and by the optional CL Extension for File I/O (see Appendix C).

String comparisons are done as follows: two strings compare equal if the contents of the string match up to the length of the shorter string and the rest of the longer string contains only blanks. CL/AM string compares are upper/lowercase sensitive; for example, "fred" <> "Fred" <> "FRED".

When a String is stored in a data point, it can be truncated to make it fit the space allocated for it at point build time. Do not assume, therefore, that a String read from a data point parameter is equal to the String that was stored in that parameter.

### 2.3.6.1 String Examples

```
"abc" is less than "abd"
"ab"  is less than "abc"
"ab"  is less than "ab "
"abc" is less than "ac"
LOCAL s, t: String
...
SET s = "This is a loooooong String"
SET AX100.DESC = s      -- store in a data point
...                    -- it may be truncated
SET t = AX100.DESC      -- retrieve the String
IF s <> t THEN SEND: "Truncated to", LEN (t)
SEND: "This is a String"
```

#### NOTE

CL/AM does not allow the value of one string variable to be set equal to the contents of another string variable within the context of a conditional.

## 2.3.7 Referencing Compound Elements

Data point parameters can be arrays or the identifiers of other data points. Array components can be data point identifiers.

To reference a component of a component, you use further levels of dot notation or subscripting, as appropriate. This procedure can be carried on to any necessary depth. The CL/AM compiler can restrict the use of indirect referencing and subscripting, when such use conflicts with the prefetch of variables. In CL/AM, at the present time, subscripting is restricted within indirect references in the following two ways:

- 1.) An indirect reference through an array of data point identifiers must stand alone, or can be the last element in a multilevel indirect-reference chain. Such an indirect reference cannot be indirectly referenced any further. The multiple levels of indirection must reference on-unit and on-node points in all but the final reference (which is allowed to be off-unit and off-node).
- 2.) If an off-LCN point is used as an EXTERNAL reference, the referenced parameter of the off-LCN point must not be an indirect reference. Indirection through off-LCN point ids is not supported. See section 2.3.8 for further off-LCN indirection restrictions.

### Examples:

```
A100.PV                -- direct reference
A100.DTA(3)            -- direct reference, indexed
A100.SENSORS(I).PV     -- indirect-indexed reference
A100.SENSORS(I).DTA(J) -- indirect-indexed reference, indexed
A100.PRIMARY.SENSORS(I).PV -- multilevel indirect reference
A100.INPUTS(I).SENSORS(J).PV -- invalid, subscript is too deep in
                                -- indirection chain

EXTERNAL az\motorpnt, pa\motorpnt
SET az\motorpnt.ptexecst = active -- valid off-LCN reference
IF az\motorpnt.sp = pa\motorpnt.sp THEN EXIT -- also valid references
IF az\motorpnt.pt(1).pv > 50.0 THEN GOTO label1
                                -- invalid, attempted off-LCN
                                -- indirection
```

### 2.3.7.1 Compound Elements Examples

```
A_004.coeff(3) -- third element of parameter
               -- "coeff" of data point A_004

tank.mixer.SP -- parameter "SP" of parameter "Mixer"
               -- of data point "tank"

sensors(I).PV -- parameter "PV" of the I'th data
               -- point of array "sensors"
```

## 2.3.8 Indirect Reference to Point Identifiers

Indirect reference to point identifiers (through references to custom parameters of type Data Point Identifier) permits point names used in CL statements to remain undefined at CL compile time. Point name values are assigned to these custom parameters during DEB point building and load. CL statement references to these point names are checked and bound at program link time, and are automatically rebound when a store is made to the custom parameter, based on the restrictions listed in subsection 2.3.8.2.

### 2.3.8.1 Features of Indirect Reference to Point Identifiers

- a) Indirect reference permits the CL source to be entered and compiled before the exact point name being referenced is known.
- b) Indirect reference allows the generic CL source to be linked to different bound data points—each referencing different points—without requiring the entire CL source to be re-edited, recompiled, and relinked.
- c) Indirect reference allows one or multi-level reference to “start on-point” (i.e., a CL statement references a CDS parameter of type Parameter List on the Bound Data Point), or the single or multi-level reference can “start off-point” (i.e., a CL statement references an EXTERNAL point name and is followed by one or more CDS parameters of type Parameter List).
- d) A dynamic indirection capability allows CDS point identifiers of type Parameter List to be changed from the Detail Display (CDS page), a schematic actor, a CG, an AM GI/GO (general input/output), CL Move\_Parameter and Set\_Null\_Point\_id subroutines, or DEB (point load with overwrite). The change can be initiated without inactivating the Bound Data Point (except for DEB change), and does not require the CLs on the Bound Data Point to be relinked in order for them to begin making use of on-point CL indirect reference changes. Indirections get relinked when the point containing the CL block is rebuilt and the off point or multi-level indirection starts on-point.
- e) A “NULL-POINT-IDENTIFIER” (\$NULLPT) is recognized as a “no identifier” specified value in arrays of type Parameter List. These NULL values can be changed to valid point ids and back to NULL. Note that CL runtime requires that the first element of an array of point ids be a valid point that contains all of the parameters indirectly referenced in the CL at link time. See heading 2.3.8.2 (b) for restrictions associated with Indirect Reference for the NULL id.
- f) Dynamic indirection will flag an indirect reference as bad when an indirection change violates restrictions for indirect reference. Bad dynamic indirections are detected when the parameter is referenced at runtime. The Engineering Personality CL Link command will abort when indirection errors are detected.
- g) Off-LCN points may be referenced in indirect references. However, the off-LCN point must be the last element in the indirection. Indirection through off-LCN point ids is not supported.

**Examples:**

```
SET A100.pt(1).pv = 10 -- the value of pt(1) may be an off-LCN point
SET A100.p2.pt3.pv = 10 -- the value of pt2 may not be
                        -- an off-LCN point, but the value of
                        -- pt3 may be an off-LCN point
```

- h) If an off-LCN point is referenced indirectly in a CL Block, the Block must either be linked to a point on the Internetwork Point Processor (IPP), or be at the BACKGRND insertion point. This is checked by the linker.

**2.3.8.2 Restrictions Associated with Indirect Reference**

- a) If an indirect point identifier is set to an invalid indirection:
- 1) An attempt to link will produce a fatal error and the link request is ignored.
  - 2) An attempt to store an invalid point name (from the Detail Display, for example) or store from the wrong access level produces an error response and the change is ignored.
  - 3) When the entity\_id and access level are valid but the indirection is invalid, the change is accepted and a runtime error occurs when the CL statement with the bad reference is executed.
- b) The types of errors for indirect point identifiers are:
- 1) An indirection which points back to the Bound Data Point (aliasing).
  - 2) Multiple-level indirection through an array of identifiers when the array is not the last indirection reference. This is caught at compile time (see example at heading 2.3.8.5).
  - 3) If the AM Extension for Off-Node Access extension (AMCL05) is not loaded, multiple-level indirection reference to an off-unit point when the reference is not the last indirection reference. This is caught at link time, but also is an error at runtime if a reference is changed to this error condition (see example at heading 2.3.8.5). If AMCL05 is loaded, off-unit reference is allowed at any level of multiple-level indirection. Off-physical-node references can occur only in the last level of multiple-level indirection with or without AMCL05. See Appendix H for more information about AMCL05.
  - 4) Runtime reference to a null point identifier.
  - 5) Removing a parameter of type Point ID (Parameter List) from a Custom Data Segment, rebuilding a point with that new definition and then relinking a CL that references the parameter.
  - 6) A CL which attempts to declare a LOCAL variable or subroutine argument of type Entity\_id.

- 7) Changing the value of a parameter of type Point ID (Parameter List) to one which does not contain the same parameters as referenced in the CL (or does not contain the same parameter types as previously defined—for example, a regulatory PV reference changed to digital input PV reference).
- 8) Changing a NULL id in a data point array to a type of point that does not match the first element in the array. Or, if the CL was linked when the array contained all NULL ids, changing the indirection to a parameter that is not non-array Real type. (Before dynamic indirection, a point build and relink of the CL on the point was required to change to or from a NULL id.)

The types of parameter data referenced by CL statements indirecting through a NULL point id are bound at Link time to be the same type and size as that of the first point id in the array of point ids. Thus, when the first array item is a NULL at Link time, the CL statement can—after the NULL identifier is changed to a valid identifier—only access non-array real parameters. If at CL runtime the indirectly referenced parameter is not a non-array real type, the CL will abort with a configuration error (CNFERR).

When the first array item is **not** a NULL at Link time, a CL statement indirecting through another point id in the array with a value of NULL can—after the NULL identifier is changed to a valid identifier—access the same parameters and data types as the first element. If at CL runtime the identifier that is indirectly referenced does not have a parameter that matches the first element in the array, the CL will abort with a configuration error (CNFERR).

- 9) A NULL point id in a non-array parameter (caught at link time or at runtime).
- 10) Changing a point id or a null point id in a data point array to a point on a different LCN than was contained in the parameter when the CL was linked. All points in an array of point ids must be on the same LCN if they are referenced indirectly in a CL Block. This is checked at link time. If a point id from a different LCN is stored into an element of an array of point ids after the CL is linked, a configuration error occurs when that element is referenced at runtime.

Also, changing a point id in a non-array entity id parameter to a point on a different LCN than was contained in the parameter when the CL was linked is an indirection error. This also generates a configuration error at runtime. If you are changing a non-array entity id parameter to a null point id, it is strongly advised that you enter the correct LCN identifier along with the null identifier. This maintains the display of the correct LCN identifier for the parameter. If just a null is entered, and you forget the LCN identifier that was originally entered for the parameter, you must unlink and relink the CL to establish a known LCN id in the parameter.

Note that an indirect reference through any null point identifier generates a runtime configuration error.

**Example:**

If the custom parameter "points" contains the following values:

```
points(1)  value is fe\A100
points(2)  value is fe\C100
points(3)  value is fe\D12345
```

and you wish to enter a null into points(3), the value `fe\null` must be entered.

- c) A change in the point identifier value of an off-point indirection reference starting on-point will be ignored until the on-point (Bound Data Point) CL is relinked, or the on-point parameter is stored into, or the AM is restarted from checkpoint. An example of off-point indirection being ignored when it starts on-point can be found at the end of heading 2.3.8.3.

The AM restart will flag as bad any CL reference that on startup is found to have changed—since it last was last linked and checkpointed—from off-node to on-node (or vice versa). This can only occur when a multi-level indirection “off-point” CDS point-id is changed from on-node to off-node or off-node to on-node without re-entering the “on-point” point id to relink the indirection.

- d) A change of a multi-level indirection starting off-point (the first reference starts with an EXTERNAL point name) is ignored until the CL is relinked. An example of off-point indirection being ignored when it starts off-point can be found at heading 2.3.8.4.
- e) For a CL on a Foreground insertion point, if this point’s processing is between prefetch and the end of point processing, a change in the point identifier value of an indirection does not take effect until the completion of AM processing of this point.
- f) Store requests for multiple Entity\_id parameter changes from a CG, AM GI/GOs, and CL Move\_Parameter or Set\_Null\_Point\_ID subroutines should be phased to prevent sporadic overloads of the AM processing (stores of Point id trigger the re-resolving of all indirections). AM GI/GO has the capability of storing point names into CDS parameters of type Point\_id. This capability results in dynamic indirection resolution for on-point CL programs.

**CAUTION**

Repeated point id stores increase the load on the AM point processing execution cycle and can cause the AM to run behind. Timing of point id stores normally varies from 0.1 millisecond to 10.0 milliseconds per store, and in complex cases, the times can be significantly longer.



- g) The EXISTS function returns OFF (false), the COMM\_ERROR function returns ON (true), or a runtime fetch/store produces an error when the following indirection errors are encountered:
- 1) An indirection which points back to the Bound Data Point (aliasing).
  - 2) Multiple-level indirection reference to off-node or off-unit point when the reference is not the last indirection reference (see the example at heading 2.3.8.5).
  - 3) Reference to a null point identifier.
  - 4) Logic errors such as cannot find the referenced point id, cannot add prefetch request, and startup prefetch on-node/off-node change.
- h) With dynamic indirection, changes to CDS parameters of type Point id will re-resolve indirections on CL programs on the point containing the changed CDS parameter. Exception: When the indirection point id that is changed is off-point (the indirection starts off-point with an external identifier) or is a multi-level indirection that starts on-point (and the first indirection level is not changed), uses of that point id are not re-resolved and an Unlink and Link of the CL programs are required to use the new identifier. The CL compiler lists the warning “Point ID change only valid after unlink and link” for any off-point and multi-level indirections starting on-point (see example at heading 2.3.8.4). The warning message appears as follows:

```
BLOCK off_pt (GENERIC; AT GENERAL)
  EXTERNAL X100      -- data point identifier
  SET pv = X100.pnt1.pv
                    ^
**WARNING** Point ID change only valid after unlink and link
```

A work-around exists if the indirection starts on-point and the same point identifier is stored to the on-point CDS. Each level of a multi-level indirection will be resolved. (For an example of re-establishing an off-point indirection change when it starts on point, see the end of heading 2.3.8.3. If the value A300 is stored to the “pnt2” parameter, any change in the “pnt2.next” parameter will get relinked.)

- i) If a CDS parameter is deleted and the point is rebuilt, the CL programs referencing that parameter will continue to use the “old” point id value. (Find Names can be used to avoid this condition by searching for and resolving the references to the CDS parameter to be deleted before the delete is done.) However, if the CL is unlinked an attempted relink will generate an error.
- j) The CL source that defines the CDS parameters should have parameters of type Entity\_id (Parameter List) build visible and/or contain a value statement. If the parameter is not build visible and without a value statement, on a rebuild of the point the on-point CL programs will continue to work with the “old” value, but the CDS parameter will contain a default value and any subsequent unlink and link may fail.

k) Factors that affect point id store performance:

- Performance is slowed by putting spares in CDS arrays of point identifiers that are indirected through using multilevel indirection.
- More indirections through a parameter cause point identifier stores to that parameter to be slower.
- More parameters indirected through by CLs on a point cause stores to those parameters to be slower.
- Stores to the first level point id become slower as the number of levels of indirections used increases.
- Indirecting through an array of point identifiers when the program only uses one point identifier at a time is inefficient. It is more efficient to move each array element to a scalar point identifier before it is used and then indirect through it.
- Storing point identifiers to points that have been or will be executed during the current processing cycle or that have an outstanding prefetch take longer than storing to inactive points.
- The first execution of stores of off-physical-node identifiers to CDS parameters that were built with on-physical-node identifiers is slower.
- Using the `multiple_move_parameter` routine to store a whole array of point identifiers can take a long time, especially if reference lists are long and a lot of off-physical-node points are referenced. This is because the whole array store is done all at once. This can cause overruns in foreground CLs. In background CLs, it can cause delays in starting the next execution cycle of the point processors.

### 2.3.8.3 Example of Single- and Multi-Level Indirection

Following is a layout of AM point records showing the parameters existing on several points. Points X100 and Y100 are used as Bound Data Points in the examples. The parameters pnt1, pnt2, pnts, and NEXT all contain data point identifiers.

Point names	X100	ab\A100	ab\A200	A300
CDS parameters and [contents]	pnt1 [ab\A100] pnt2 [A300] pnts(1) [C100] pnts(2) [C200] pnts(3) [C300]	PV [1.0]	PV [2.0]	NEXT [A400] PV [2.0]

Point names	A400	C100	C400	Y100
CDS parameters and [contents]	PV [3.0]	PV [4.0]	PV [6.0]	PV [7.0]

Example CL: This program fragment illustrates on-point indirection in CL. It is bound to point X100.

```

PACKAGE
  PARAM_LIST multlvl
    next : $REG_CTL
    pv : NUMBER
  END multlvl

BLOCK dyn_ind (GENERIC; AT GENERAL)
  PARAMETER pnt1 : $REG_CTL
  PARAMETER pnt2 : multlvl
  PARAMETER pnts : $REG_CTL ARRAY (1..3)

```

This CL/CDS is linked to point X100. The data type of parameter pnt1 is data point identifier. Indirection occurs when a parameter of the data point pointed to (identified) by pnt1 is referenced by using pnt1 as the point name in a CL statement. Let's say that pnt1 contains the data point identifier ab\A100, and that pnts(1) contains the data point identifier C100. The following then are indirect references to ab\A100 and C100.

```

IF pnt1.pv > 100 THEN SEND : "over 100"      -- gets value 1.0
If pnts(1).pv > 100 THEN SEND : "over 100" -- gets value 4.0

```

At link time, pnt1.pv is resolved to access ab\A100.pv. Dynamic indirection means that the “re-resolving” of indirect references will take place automatically—without the need for relinking—in the event of a change to the value of the parameters pnt1, pnt2, or pnts. If the pnt1 CDS value is changed to contain ab\A200, the above statement will get the ab\A200.pv value of 2.0 (instead of the ab\A100.pv value of 1.0). Note that the new indirect point identifier must be on the same LCN as the point identifier being replaced.

Now, let's look at an example of multi-level indirection starting on-point. In the code that follows, fetching `pnt2.next.pv`, gets a value of 3.0 from `A400.pv` and the SET statement changes the value of `A300.pv` from 2.0 to 7.0.

```

    IF pnt2.next.pv > 100 THEN SEND : "over 100"
    ELSE SET pnt2.pv = 7.0
    END dyn_ind
END PACKAGE

```

A restriction of on-point multi-level indirection is that if an off-point reference is changed, it is ignored until the on-point CL is relinked or a store is made to the on-point reference (for example, the data point identifier stored in `pnt2`), or the AM is restarted from checkpoint. Thus, if an operator, for example, changes the value of `A300.next` from `A400` to `C400`, a reference to `pnt2.next` will still get a value of 3.0 until one of the above changes occurs (it then will get the 6.0 value).

#### 2.3.8.4 Example of CL with Off-Point Indirection

This program illustrates CL indirection that starts off-point. This CL is bound to point Y100.

```

BLOCK off_pt (GENERIC; AT GENERAL)
  PARAMETER pv
  EXTERNAL X100                      -- data point identifier that has a
                                     -- parameter called pnt1
  SET pv = X100.pnt1.pv
END off_pt

```

If `X100.pnt1` contains the value `ab\A100` when this CL is linked, when this CL runs, the PV of the bound data point (`Y100`) is set equal to the PV of point `ab\A100` (1.0). If the contents of `X100.pnt1` is changed to `ab\A200` after this CL is linked, `Y100.pv` continues to get the PV of point `ab\A100` (1.0). This will continue until this CL is unlinked and linked. It then will get the PV of point `ab\A200` (2.0).

#### 2.3.8.5 Example of CL Multi-Level Indirection

This program illustrates multi-level indirection.

```

PACKAGE
PARAM_LIST n3
  PARAMETER pt4 : $REG_CTL ARRAY (1..3)
  PARAMETER n   : NUMBER
END n3
PARAM_LIST n2
  PARAMETER pt3 : n3
  PARAMETER n   : NUMBER
END n2
PARAM_LIST n1
  PARAMETER pt2 : n2
  PARAMETER n   : NUMBER
END n1

```

```

BLOCK dil (GENERIC; AT GENERAL)
  PARAMETER pt1 : n1
    SET pt1.n = 1.0 -- statement 1, 1-level indirection
    SET pt1.pt2.n = 2.0 -- statement 2, 2-level indirection
    SET pt1.pt2.pt3.n = 3.0 -- statement 3, 3-level indirection
    SET pt1.pt2.pt3.pt4(2).pv = 4.0 -- statement 4, 4-level indirection
END dil
END PACKAGE

```

In this example, if the AM Extension for Off-Node Access (AMCL05) is not loaded, only pt4 can have an off-node or off-unit reference (statement 4). If AMCL05 is loaded, points pt1, pt2, pt3, and pt4 can have off-unit references, but only pt4 can have an off-physical-node reference. If AMCL05 is not loaded and pt2 indirects off-node, statements 1 and 2 will work and statements 3 and 4 will give runtime errors. This will not link with these errors. AMCL05 is covered in Appendix H.

Only the last element can be an array (statement 4). If pt2 is an ARRAY (1..n) then statement 2 could use pt1.pt2(2).n, but pt2(2) would be illegal in statements 3 and 4.

### 2.3.8.6 Example of Indirection Referencing Off-LCN Points

The following program illustrates indirection referencing off-LCN points.

```

PACKAGE
--
CUSTOM
  PARAMETER fepoints : $REG_CTL ARRAY(1..5)
  VALUE (fe\point1,fe\point2,fe\point3,fe\point4,fe\point5)
  -- points on the remote LCN named fe
  PARAMETER azpoints : $REG_CTL ARRAY(1..5)
  VALUE (az\pt1,az\pt2,az\pt3,az\pt4,az\pt5)
  -- points on the remote LCN named az
END CUSTOM
--
BLOCK ex1 (GENERIC; AT GENERAL)
  LOCAL i
  EXTERNAL ct\contrlpt -- direct reference to a point on LCN ct
  --
  label: LOOP FOR i IN 1..5
  IF (fepoints(i).pv<>fepoints(i).sp) OR (fepoints(i).sp<>ct\contrlpt.sp)
  & THEN SEND : "mismatch on LCN fe on element number", i
    -- example of referencing off-LCN points directly and indirectly
  IF (azpoints(i).pv<>azpoints(i).sp) OR (azpoints(i).sp<>ct\contrlpt.sp)
  & THEN SEND : "mismatch on LCN az on element number", i
  REPEAT label
  END ex1
--
END PACKAGE

```

In this example, the parameter `fepoints` contains point ids from an LCN named `fe`. The parameter `azpoints` contains point ids from an LCN named `az`. Also, an EXTERNAL point on LCN `ct` is referenced. Multiple LCNs may be referenced in one CL program as long as an array of point ids referenced indirectly all refer to the same LCN, whether on-LCN or off-LCN.

### 2.3.8.7 CL Subroutine/Function Reference to Point Identifiers

The capability to modify or compare parameters of type `entity_id` does not exist with the CL SET or IF statements. However these capabilities are provided by built-in subroutines and functions.

The subroutines `Move_Parameter` and `Set_Null_Point_id` (see heading 4.3.7.5) allow CDS parameters of type `entity_id` to be changed by CL programs. Indirections through changed point identifiers are automatically re-linked for CL blocks on the same point by CL dynamic indirection.

The CL function `Equal_Point_id` (see heading 4.3.7.3) allows comparison of two CDS parameters of type `entity_id`. The CL function `Equal_Null_Point_id` (also see heading 4.3.7.3) allows comparison of a CDS parameter of type `entity_id` to a Null id (point name value does not have a corresponding LCN point —i.e., a “no-point” value).

`Set_Null_Point_id`, `Equal_Point_id`, and `Equal_Null_Point_id` each can have arguments of single parameter or array element of type `entity_id`. `Move_Parameter` can have arguments of LOCAL variables of any CL type, and the arguments can be single parameters, array elements or entire arrays. `Move_Parameter` arguments must have equal data types and entire array arguments must have equal array sizes.

### 2.3.8.8 Example of CL Point Identifier Subroutines and Functions

Following is a layout of AM point records showing the parameters existing on several points. Point X100 is used as the Bound Data Point in the next example. The parameters `pnt1`, `pnt2`, `pnts`, `NEXT`, `NULL`, and `PREV` all contain data point identifiers.

Point names	X100	A100	A200	A300
CDS parameters and [contents]	pnt1 [A100] pnt2 [A200] pnts(1) [C100] pnts(2) [C200] pnts(3) [C300]	NULL [--] PV [1.0]	NEXT [A300] PREV [C300] PV [2.0]	PV [3.0]

Point names	C100	C200	C300
CDS parameters and [contents]	PV [4.0]	PV [5.0]	PV [6.0]

The following program (linked to point x100) illustrates use of point identifier routines.

```

PACKAGE
  PARAM_LIST multlvl
    next : $REG_CTL
    pv : NUMBER
  END multlvl
  PARAM_LIST nullst
    null : $REG_CTL
  END nullst
  BLOCK sbr_ind (GENERIC; AT GENERAL)
    PARAMETER pnt1 : nullst
    PARAMETER pnt2 : multlvl
    PARAMETER pnts : $REG_CTL ARRAY (1..3)
    LOCAL      stat : CLERRSTS

```

The following code increments the PV of the indirect reference pnt2.next when it points to C300, otherwise it decrements the PV. (Note that pnts(3) is equivalent to a CL constant of C300.)

```

  IF EQUAL_POINT_ID (pnt2.next, pnts(3)) THEN
&    SET pnt2.next.pv = pnt2.next.pv + 1.0
  ELSE SET pnt2.next.pv = pnt2.next.pv - 1.0

```

Next the move routines are used to swap the A200 point (in pnt2) “NEXT” and “PREV” values, then re-store this point parameter (pnt2) to re-link the indirection (the parameter pnts(2) is used for temporary storage.

```

  CALL MOVE_PARAMETER (pnts(2), pnt2.next, stat)
  IF stat = noerror THEN
&    CALL MOVE_PARAMETER (pnt2.next, pnt2.prev, stat)
  IF stat = noerror THEN CALL MOVE_PARAMETER (pnt2.prev, pnts(2), stat)
  IF stat = noerror THEN SEND: "PNT2 NEXT & PREV IDS SWAPPED"

  IF stat = noerror THEN CALL MOVE_PARAMETER (pnt2, pnt2, stat)
  IF stat = noerror THEN
&    SEND: "PNT2 INDIRECTION RELINKED AT PTEXEC END"
  ELSE SEND: "PNT2 INDIRECTION NOT RELINKED"

```

Note that without the relink the earlier section of code would see the point id change (the data point identifier in A200.next changed from A300 to C300), but the decrement and store would still be made to the PV parameter in A300. The pnt2.next and pnt2.prev stores are “off-point” and thus do not relink the pnt2.next “on-point” reference. Storing pnt2 from itself causes the “on-point” pnt2.next indirections to be re-linked.

The following code checks to see the pnt1.NULL is set \$nullpt. If not, it stores \$nullpt to pnt1.null.

```

  IF (NOT EQUAL_NULL_POINT_ID (pnt1.null)) THEN
&    CALL SET_NULL_POINT_ID (pnt1.null, stat)
  END sbr_ind
END PACKAGE

```

### 2.3.9 Referencing Enumeration Parameters on Remote LCNs by Indirection

An indirect access to a parameter of type Standard Enumeration in CL requires a parameter list that defines the parameter type. For example:

```
PACKAGE
  PARAM_LIST one
    PARAMETER m : ptxecst
  END one
  CUSTOM
    PARAMETER point1 : one
  END CUSTOM
  BLOCK test1 (GENERIC; AT GENERAL)
    IF point1.m = active THEN SEND:"m is active"
  END test1
END PACKAGE
```

The above example declares that the parameter `m` is of type `ptxecst`, which is a standard enumeration.

An indirect access to a parameter of type Custom Enumeration in CL also requires a parameter list defining the parameter type. For example:

```
PACKAGE
  ENUMERATION enum1 = red/blue
  PARAM_LIST two
    PARAMETER cus1 : enum1
  END two
  CUSTOM
    PARAMETER point2 : two
  END CUSTOM
  BLOCK test2 (GENERIC; AT GENERAL)
    IF point2.cus1 = red THEN SEND:"cus1 is red"
  END test2
END PACKAGE
```

The above example declares that the parameter `cus1` is of type `enum1`, which is a custom enumeration.

When compiling the above examples, the CL compiler obtains the state names of the enumerations `ptxecst` or `enum1` from the **local** LCN to do the type checking of the statements in the Block. This does not create any problems if the point entered into the parameters `point1` or `point2` are on the local LCN.

However if an off-LCN point (for example, `fe\A100`) will be entered as the value of parameters `point1` or `point2`, the compiler still will obtain the state names of the enumeration from the local LCN, not from the remote LCN. Thus, in order to reference a custom enumeration parameter on a remote LCN, a custom enumeration must be built on the local LCN that has the same state names as the parameter's custom enumeration on the remote LCN. Also, the parameter declaration in the Custom Data Segment must name the new local custom enumeration as its type.



In summary, the following rules apply for obtaining enumeration members of an indirectly referenced off-LCN point:

If a parameter of type enumeration, or an array indexed by enumeration is referenced indirectly in a CL Block, the state names of an enumeration named in a parameter list always are obtained on the **local** LCN.

This is automatic when the parameter is of a Standard Enumeration type since, by definition, TPS Standard Enumerations are identical on the local and remote LCNs.

However, if the parameter is of type Custom Enumeration, a Custom Enumeration must be built on the local LCN with the same state names as the parameter's Custom Enumeration on the remote LCN. The name given to the Custom Enumeration set on the local and remote LCNs need not be the same, but the state names must be identical.

For the above example, assume the off-LCN point `fe\A100` will be entered as the value of parameter `point2`. Point `fe\A100` was built on the LCN `fe` with a custom parameter `cus1` which was defined as type Custom Enumeration `colors`. The states of `colors` are red/blue.

Thus at compile time, the parameter `cus1` on point `fe\A100` is of type Custom Enumeration `enum1`, with state names red/blue. The states red/blue were obtained from the enumeration `enum1` on the local LCN, not from the enumeration `colors` on the remote LCN. Essentially, the Custom Enumeration `enum1` contains the same state names as the Custom Enumeration `colors` on the LCN `fe`.

Note that the actual state names of the parameter are checked at link time (the state names are obtained off-LCN). Also note that the compiler obtains the actual state names of enumeration parameters off-LCN when a point is referenced directly with the EXTERNAL statement.

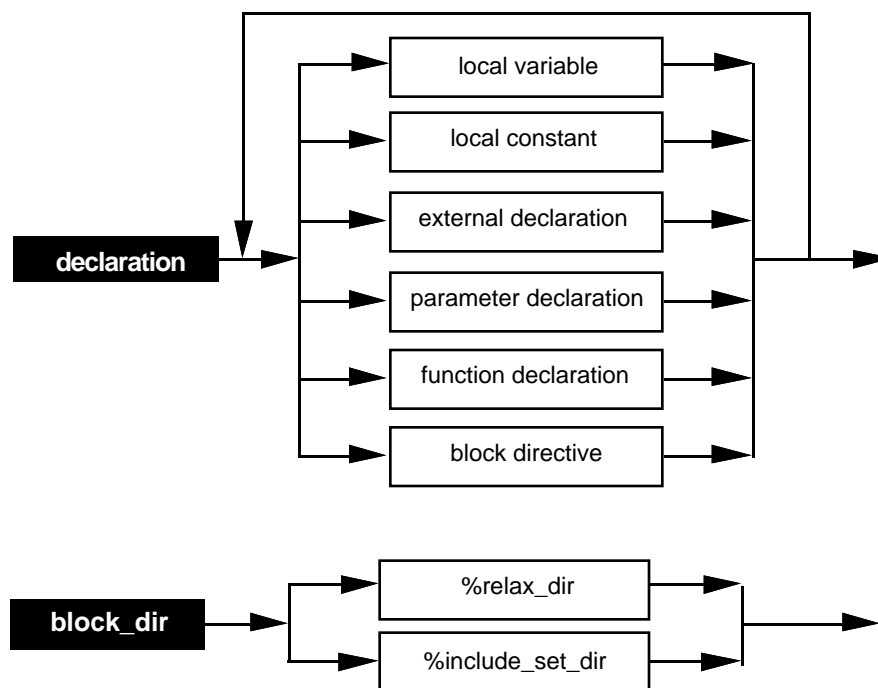
## 2.4 VARIABLES AND DECLARATIONS

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/AM program or CL/AM subroutine. External variables are data points that are defined outside CL. 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



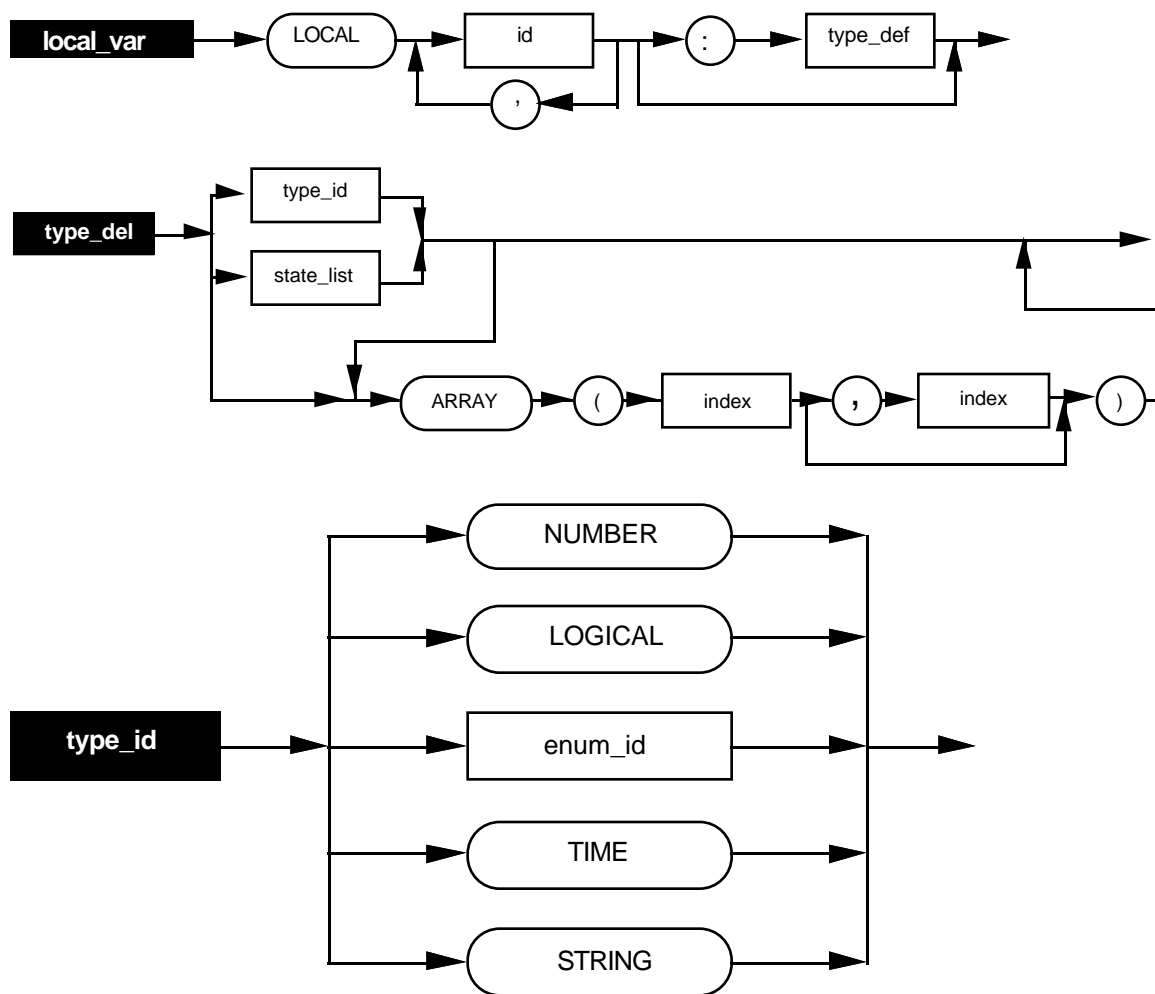
## 2.4.2 Local Variables

Local variables are declared in the heading of a sequence program, block, or CL/AM subroutine. In the AM, their lifetime is that of the program; when the program or subroutine exits, its local variables cease to exist.

Local variables declared in a program are visible only inside the program they are declared in, any local subroutines belonging to that program, and any functions defined within such a program or subroutine.

Local variables declared in a local subroutine are visible only inside that subroutine and any functions declared within it.

### 2.4.2.1 Local Variables Syntax



### 2.4.2.2 Local Variables Description

Type identifiers are optional; the default is Number.

A type\_ID can be any of the following: Number, Logical, String, Time, Standard Enumeration (for example, Mode), a state name list (for example, red/blue/green), or one- or two-dimensional arrays of any of these. The type\_ID cannot name a parameter list.

Local variables defined with a state list are enumeration variables. The order in which the state identifiers are given defines the order of their internal representation.

Initial values are assigned to Local variables by data type as follows:

Number — Bad Value (NaN)  
 Logical — Off  
 String — Null string  
 Time — 0 Secs  
 Enumeration — first defined state

An array index must be a discrete type (declared by type name) or a Number declared by naming lower and upper bounds. The lower bound is the left-most const\_expression in the index. The upper bound is the right-most const\_expression in the index. In the latter case, the lower and upper bounds must be integers. The value of the lower bound must be less than the value of the upper bound.

### 2.4.2.3 Local Variables Examples

```

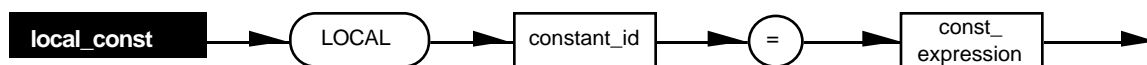
LOCAL x, y, z: NUMBER           -- three local numbers
LOCAL i, j, k                   -- three more (default)
LOCAL pop: coke/pepsi/7up      -- enumeration
LOCAL color: red/green/amber   -- another
LOCAL hue: amber/green/red     -- incompatible with "color"
LOCAL foo: LOGICAL ARRAY (1..5) -- an array
LOCAL bar: ARRAY (1..16, 1..16) -- a 2-d number array
LOCAL sam: NUMBER ARRAY (Mode) -- indexed by enumeration
LOCAL p: up/down ARRAY (1..5)  -- enumeration array

```

## 2.4.3 Local Constants

Local constants of the data types Number and Time can be declared.

### 2.4.3.1 Local Constants Syntax



### 2.4.3.2 Local Constants Description

Local constants cannot be modified.

For CL/AM, constant expressions must be composed of arithmetic or time operators, the built-in function ABS, numeric literals, time 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

```

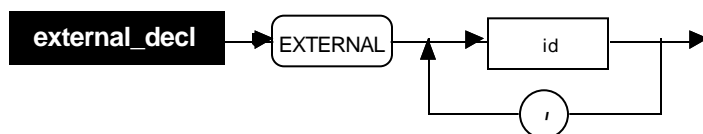
LOCAL pi = 3.14159265           -- a numeric constant
LOCAL ten_K = 1.0e4             -- another
LOCAL 2_pi = pi * 2             -- a constant expression
LOCAL pi_2 = pi/2               -- illegal: divide operator
LOCAL mix_time = 2 HOURS 5 MINS -- a Time constant

```

## 2.4.4 External Data Points

Data points other than the bound data point can be accessed by a CL/AM 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 other than the bound data point. These data points can be on the local LCN (on-LCN) or on a remote LCN connected by a Network Gateway (off-LCN). Each data point named in an EXTERNAL declaration must already exist in the system at the time the program is compiled; otherwise, the compiler reports an error. External data points cannot be declared in subroutines.

CL/AM cannot access parameters of the Processor Status Data Point using the EXTERNAL statement. Use a general input from the Processor Status Data Point parameter into a custom parameter or into some other parameter that CL/AM can access, or use indirection.

### 2.4.4.3 External Data Points Examples

```

EXTERNAL anp049hx, AX_001
EXTERNAL x4\z123, fe\B100  -- off-LCN point identifiers
EXTERNAL $HY03B22          -- Box Data Point*

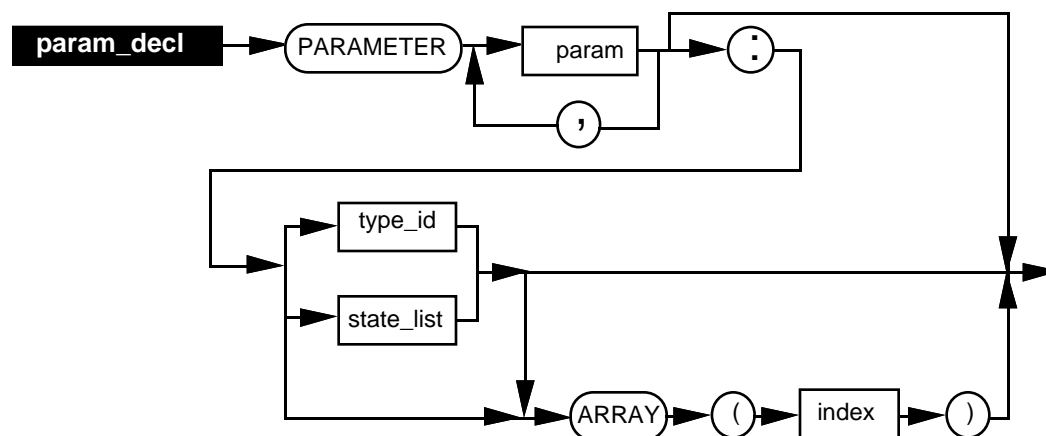
```

\* Refer to headings 2.3.4.7 and 2.3.4.8 for details on accessing box-variables.

## 2.4.5 Parameter Declarations

PARAMETER declarations declare the existence of parameters of the bound data point.

### 2.4.5.1 Parameter Declarations Syntax



### 2.4.5.2 Parameter Declarations Description

The variables named in the PARAMETER declaration introduce the names of one or more parameters of the bound data point. When you declare a data point parameter to the compiler, you also make its value enumerations known.

When the program is linked, all references to parameters are type-checked. These include parameters declared with PARAMETER declarations, as well as those named in parameter lists. Every parameter referenced by the program must be present at link time, and its data type must be as declared at compile time.

The PARAMETER declaration does not create parameters; it merely asserts that they exist.

### 2.4.5.3 Parameter Declaration Examples

PARAMETER PV, SP	-- Bound data point parameters:
PARAMETER Q: red/green	-- Number by default
PARAMETER X: ARRAY (1..8)	-- enumeration
PARAMETER VALVE: \$REG_CTL	-- Number array by default
	-- Data point identifier type

### 2.4.5.4 Parameter Data Types Definition

Data types allowed for parameters are the same as those allowed for local variables except that

- parameter lists are allowed
- 2-dimensional array declarations are not allowed

#### 2.4.5.5 Redundant Parameter Declarations Definition

A parameter can be declared more than once. This usually occurs because it is named in a parameter list (see heading 4.5), as well as in a PARAMETER declaration. In such a case, there is no error as long as all the declarations specify the same data type. If the data types differ, however, there is an error.

#### 2.4.5.6 Parameter References in Packages

Any programs in a package can assume that all Custom Data Segments declared in the same package are attached to the bound data point; therefore, they can use the names of all parameters in those Custom Data Segments without further declaration.

PARAMETER declarations are not required for these parameters and they need not be named in the parameter list named in the BLOCK heading, if a list exists.

#### 2.4.5.7 Parameter References in Packages Examples

```

PACKAGE
  CUSTOM (BLD_VISIBLE)
    PARAMETER start, stop: TIME
    PARAMETER stir_lev: NUMBER
    PARAMETER stir_mot: Anlogout -- data point type
  END CUSTOM
  ...
  BLOCK stirprog (GENERIC;
&                                AT General)
    IF Day_Time = start
&      THEN SET stir_mot.SP = stir_lev
    ELSE IF Day_Time = stop
&      THEN SET stir_mot.SP = 0

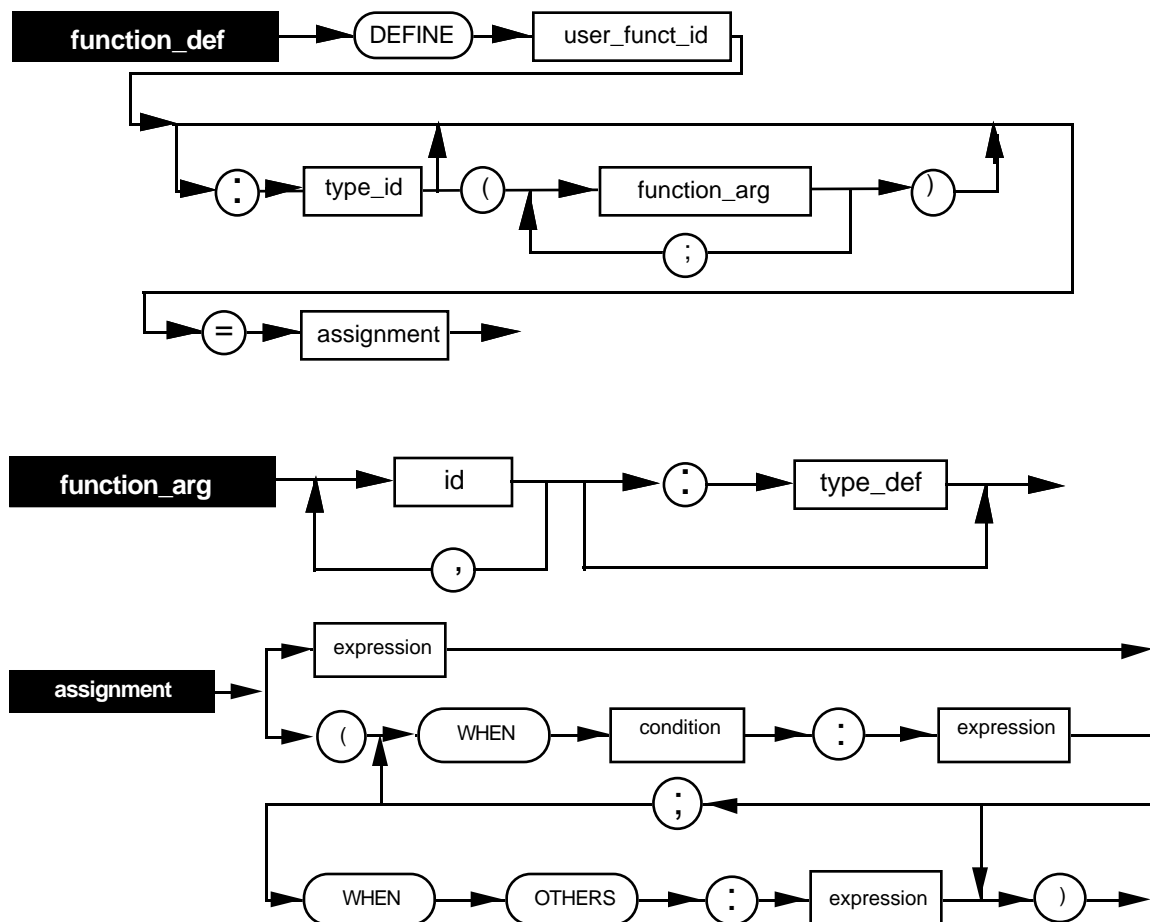
  END stirprog
END PACKAGE

```

## 2.4.6 Function Declarations

A CL/AM program can define single-statement functions. These functions are conceptually like **statement functions** in Fortran. They are valid for only the program in which they are defined, and any subroutines compiled with that program. Function declarations can be intermixed with data declarations.

### 2.4.6.1 Function Declarations Syntax



### 2.4.6.2 Function Declarations Description

Each argument acts as a local constant within the function definition. Functions can reference local or external variables as well as their arguments. Functions can call other functions as required.

The type ID given in the function heading denotes the result type of the function. This must be a scalar type or String. Arguments can be of any type, except that the use of data point identifiers passed as arguments is prohibited.

The default for all optional type identifiers is Number. The type of the expressions and of assignment must match the type of the function.

The conditional form of the function declaration is like the conditional SET statement: see Section 3.



### 2.4.6.3 Function Declarations Examples

```

DEFINE distance(x, y) = SQRT(x*x + y*y)

DEFINE imp: LOGICAL(a, b: LOGICAL) = b OR NOT a

DEFINE sign(x) = (WHEN x > 0: 1;
&           WHEN x = 0: 0;
&           WHEN x < 0: -1)

```

The function declarations example assumes that x is not a bad value. If x were a bad value, the first comparison would cause a runtime error that would abort the program. It may be safer to rewrite the function as follows:

```

DEFINE sign(x) = (WHEN BADVAL(x): 0;
&           WHEN x > 0: 1;
&           WHEN x = 0: 0;
&           WHEN x < 0: -1)

DEFINE alarm_state: LOGICAL = p.alm OR q.alm

```

In the preceding example, p and q are not arguments of the function, but are external data points, because the function was not declared as **DEFINE alarm\_state (p, q)....**

```

LOCAL a : array (1..100)
LOCAL i, j

DEFINE vecprod (i, j: NUMBER) =
&           (WHEN i > j : 0;
&           WHEN OTHERS: a(i) + vecprod(i+1, j))

```

In the last example, **vecprod** recursively calls itself. This is permitted; however, recursion is an advanced and powerful programming technique that should always be used with care.

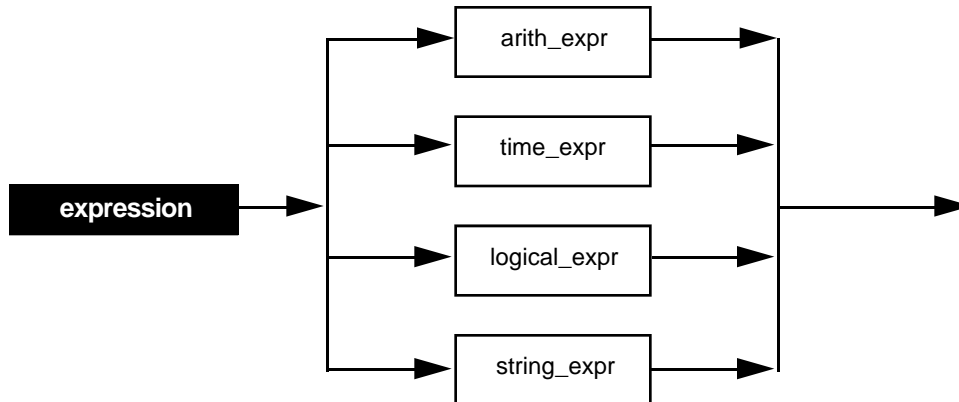
### 2.4.7 Block Directives

See headings 3.3.4, %RELAX Directive, and 3.3.5, %INCLUDE\_SET Directive.

## 2.5 EXPRESSIONS AND CONDITIONS

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

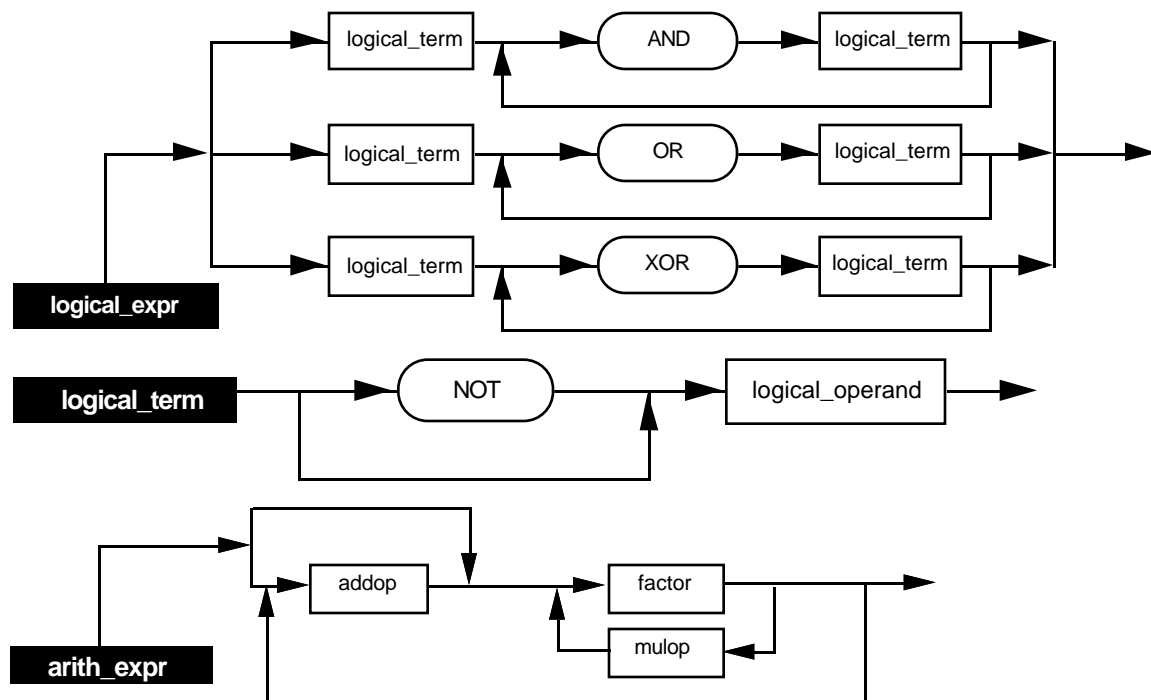
### 2.5.1 Expressions and Conditions Syntax

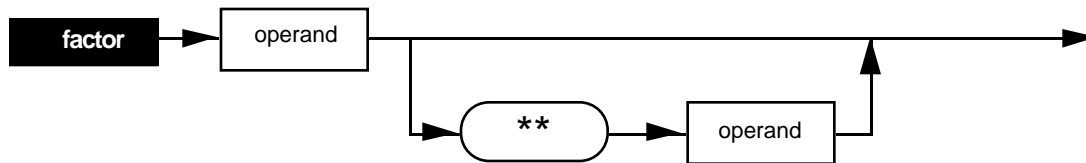


### 2.5.2 Arithmetic and Logical Expressions

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

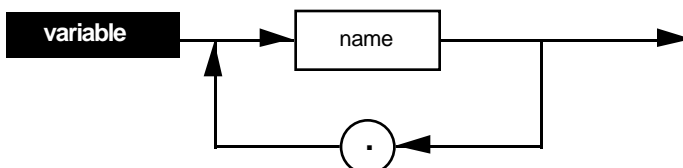
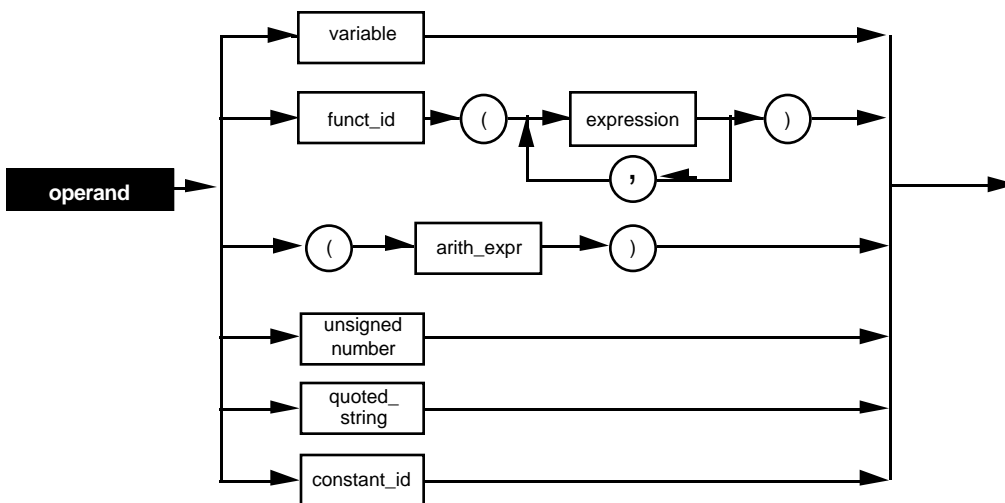
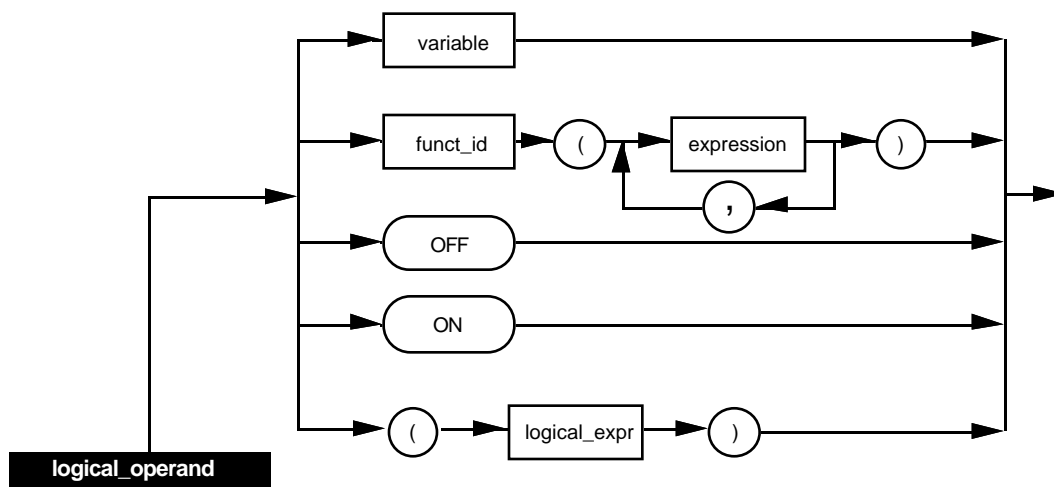


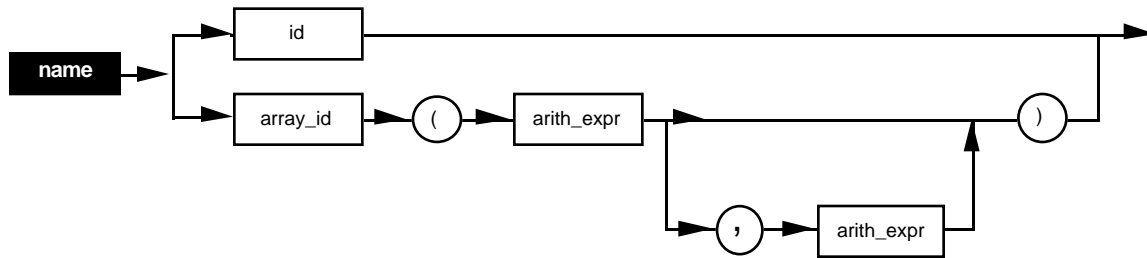


### 2.5.2.2 Operand Definition

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 2-9. Higher priority means closer binding.

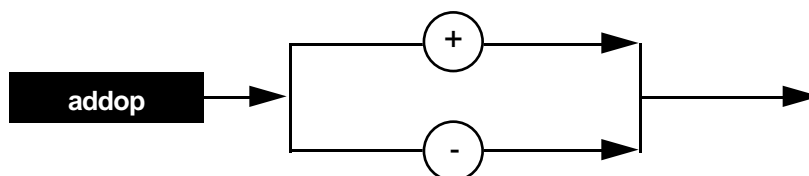
For example, in the expression  $a + b * c$ ,  $b * c$  is evaluated first because the  $*$  operator has a higher priority. Operators with the same priority are evaluated left to right.

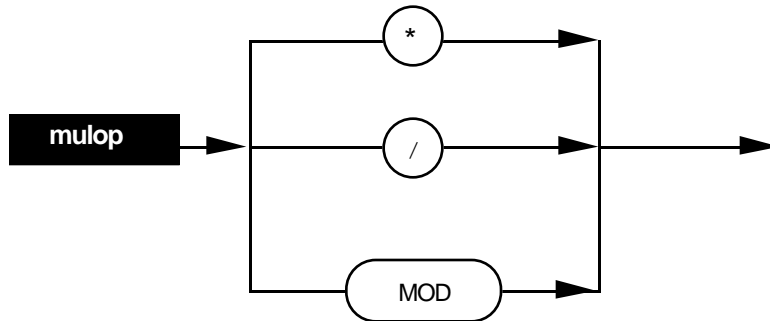
**Table 2-9 — Operator Priorities**

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

**Arithmetic Operators Definition**—The arithmetic operators are +, -, \*, /, MOD and \*\*; their usual meanings are as listed in Table 2-9. 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 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 2-10. Logical operators can take only operands of type LOGICAL (also see Section 2.5.4.6).

**Table 2-10 — 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 Time Expressions

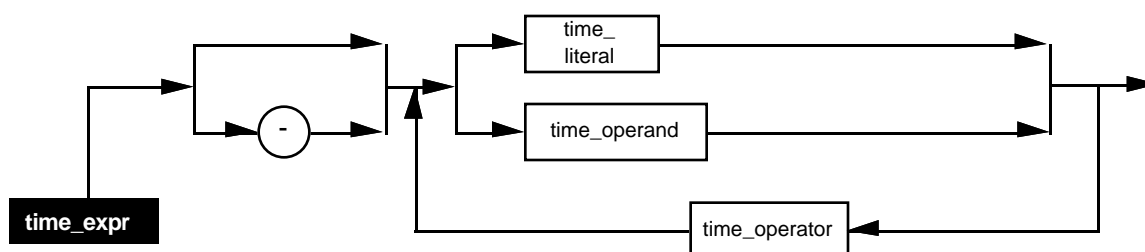
A time expression is a special form of expression whose result is of the data-type Time. It is used to express a duration.

Time expressions can be used in a SET statement that stores into a Time variable. They can also be used in the argument list of a function or subroutine that requires an argument of type Time.

A time expression is composed of time operands and time operators.

The result of a time calculation that is negative will not be displayed correctly; use the NUMBER function on the time result and send that result to the Universal Station.

#### 2.5.3.1 Time Expressions Syntax



#### 2.5.3.2 Time Expressions Examples

```

LOCAL t1, sked: TIME
LOCAL n,m: NUMBER
...
SET t1 = DATE_TIME      -- current date and time
SET sked = 20 SECS      -- set a time variable
SET t1 = n SECS
  
```

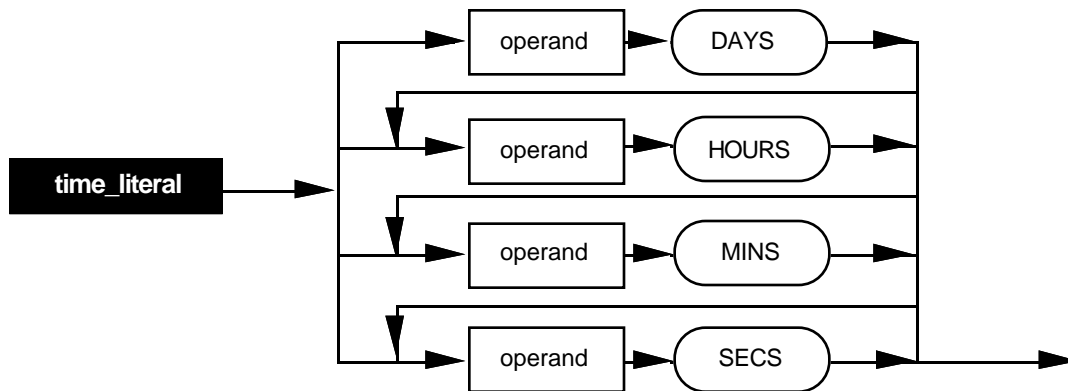
#### 2.5.3.3 Time Operands Definition

A time operand can be a variable, parameter, constant, array element of type Time, a time literal, or a function call with a result of type Time.

#### 2.5.3.4 Time Literals Definition

A time literal represents a defined amount of time.

#### 2.5.3.5 Time Literals Syntax



#### 2.5.3.6 Time Literals Description

The operands in a time literal must be of data-type Number but do not need to have integer values. The value of any operand can be negative or zero. The whole time literal can have a negative or zero value; however, most contexts that require time expressions treat a negative duration as zero.

Using the multiplier appropriate to its suffix, each operand in the time literal is computed as required and converted into an integer number of seconds (see Table 2-11). Rounding takes place after multiplication. The results of these conversions are then added to obtain the final time literal.

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

**Table 2-11 — Time Literal Multipliers**

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

### 2.5.3.7 Time Literals Examples

```

5 MINS                -- five minutes
300 SECS              -- five minutes
(1/12) HOURS          -- five minutes
4 MINS 60 SECS        -- five minutes
6 MINS (-60) SECS     -- five minutes
2 MINS 3 MINS         -- NOT VALID (MINS twice)
10 SECS 5 MINS        -- NOT VALID (out of order)
1 DAYS 6.5 HOURS      -- valid
(x * y + z) SECS      -- valid
h HOURS m MINS s SECS -- valid
Max (x,y) DAYS        -- valid
0.5 MINS              -- 30 seconds
0.125 DAYS            -- three hours
0.01667 HOURS         -- one minute

```

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

### 2.5.3.8 Time Operators Definition

The Time operators are a subset of the arithmetic operators, and they have the same priority. The types of operands required for these operators are shown in Table 2-12.

**Table 2-12 — Time Operators**

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

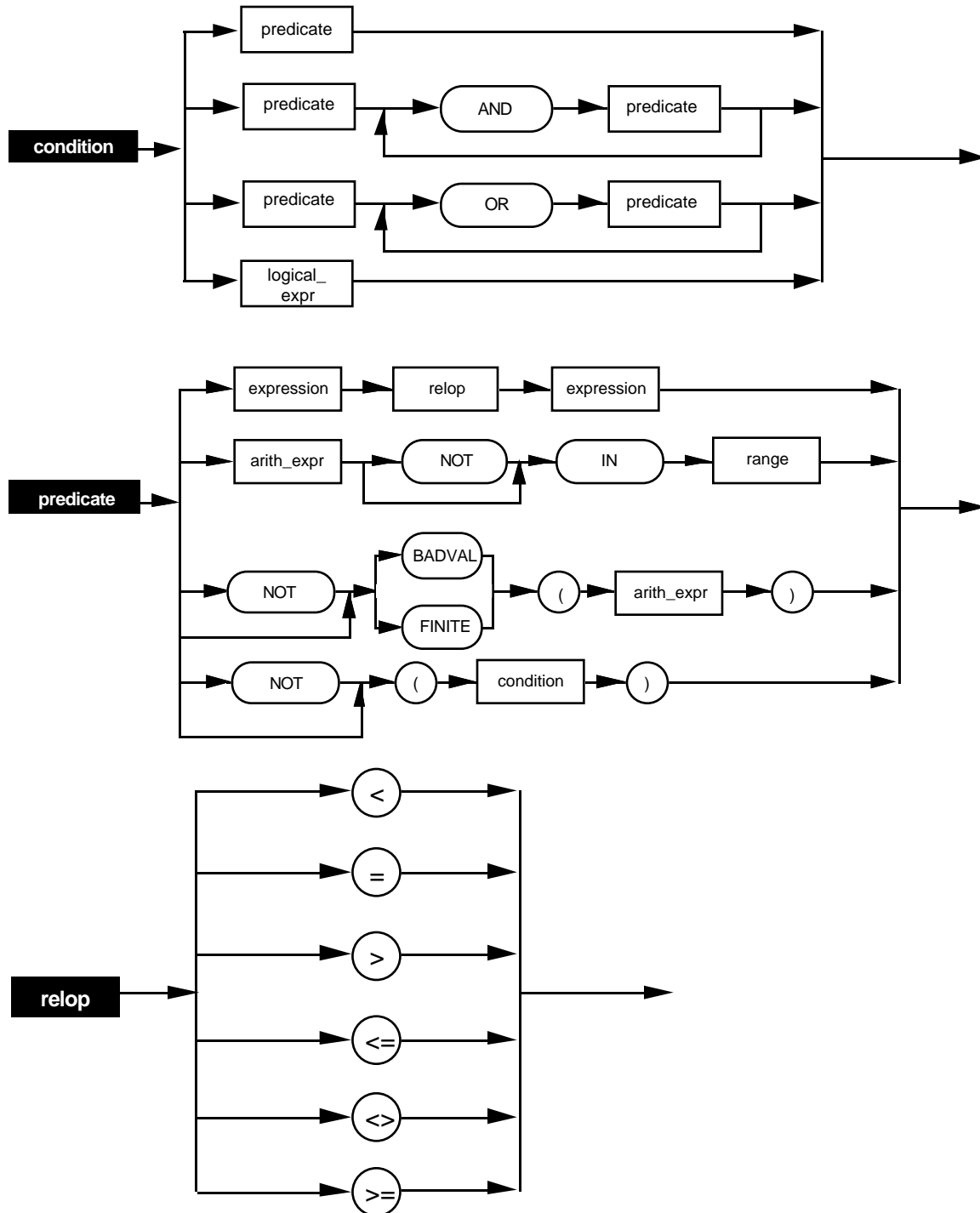
In Table 2-12, the permitted operations are shown according to the left-hand and right-hand sides of the operators. The final entry under the minus sign shows the use as a monadic (operating on one operand) operator.



## 2.5.4 Conditions Definition

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

### 2.5.4.1 Conditions Syntax



### 2.5.4.2 Conditions Description

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

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

```
IF x AND y AND NOT z THEN ...
```

is equivalent to

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

### 2.5.4.3 Relations Definition

The relational operators are shown in Table 2-13.

**Table 2-13 — Relational Operators**

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

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

### 2.5.4.4 Range Tests Definition

Range tests ( $x \text{ IN } y..z$ , a  $\text{NOT IN } b..c$ ) are defined on only Number data type, not on Time, String or Enumeration types. 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.

### 2.5.4.5 Testing for Bad Values and Finite Values

A predicate is a property of a subject that can be affirmed or denied. The built-in predicates Badval and Finite can be applied to expressions of type Number (the subject). Badval(X) returns ON if, and only if, X is a bad value. As an example of the possible use of Badval, suppose you are calculating a pv (heat density) as follows (the parameter declarations, block heading and preliminary calculations are not shown):

```

SET delta_t    =    temp_in.pv - temp_out.pv
SET x          =    flow.pv * specheat * density * delta_t / convert
- -
- -    If any of the inputs has a bad value, the value of x will be
- -    bad; in such cases set the calculated pv bad.
- -
IF BADVAL(x) THEN GOTO badpv
ELSE (SET pvcalc = x;
&      SET pvautost = normal;
&      EXIT)

- -
- -    One or more of the inputs required to calculate heat density is
- -    bad; therefore, set the pv bad and exit.
- -
badpv  :  CALL SET_BAD (pvcalc)
          SET pvautost = bad
EXIT

```

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

For more information on Badval and Finite, see the NOTE under heading 4.3.7.2.

### 2.5.4.6 Connecting Conditions with OR and AND

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

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

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

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

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

When used to connect conditions rather than as Logical operators, AND and OR are evaluated through the **short-circuit** method. This means that a condition is evaluated only so far as necessary to determine the truth or falsity of the entire condition. Evaluation proceeds from left to right.

Each connective evaluates its left-hand operand first; then, if the condition's truth or falsity can be decided from that information, the right-hand operand is not evaluated.

You can use this fact to guard against evaluation of an undefined or invalid expression.

For example, the following conditions guard against

```
BADVAL(x) OR x < 0      -- Comparison with Bad value
x <> 0 AND y/x < 1      -- division by zero
EXISTS(p) AND p.PV = 2  -- use of undefined point ID
```

## CL/AM STATEMENTS Section 3

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

### 3.1 INTRODUCTION

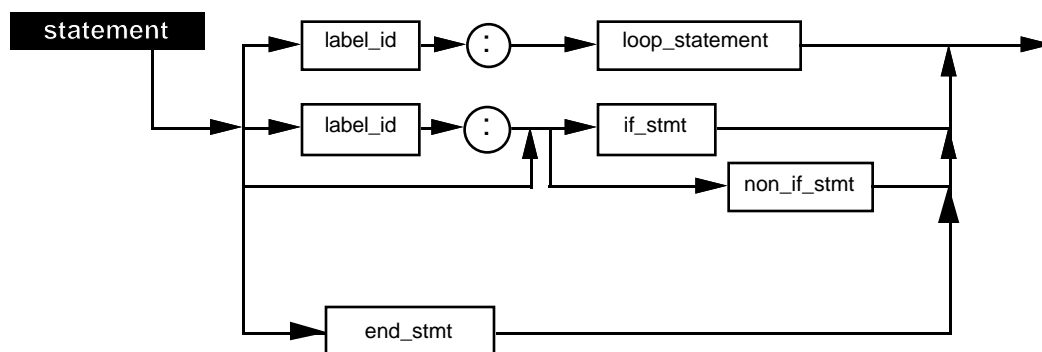
This section describes CL/AM statements. A statement defines a single, simple action to be performed within a CL/AM structure. CL/AM statements can be grouped into three major categories, according to function: **Attribute Statements**, **Program Statements**, and **Embedded Compiler Directives**. Attribute statements apply to Custom Data Segments, which are described in Section 4.

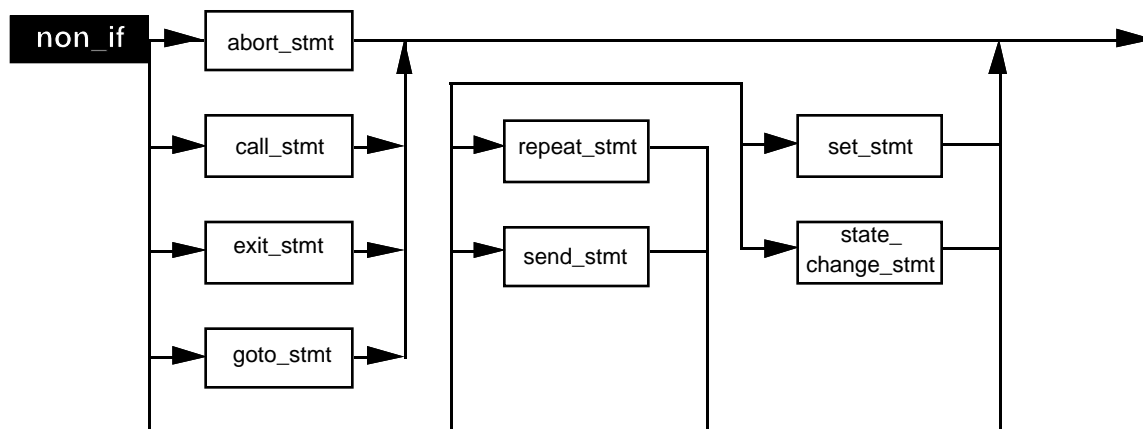
### 3.2 PROGRAM STATEMENTS DEFINITION

CL/AM Program statements can be categorized as follows:

- Assignment statements, whose purpose is to change the value of one or more variables: SET and the STATE-CHANGE statement.
- Control statements, which direct the flow of control within a program: GOTO, IF/THEN/ELSE, LOOP/REPEAT, and CALL.
- Communication statements, which communicate with an operator: SEND.
- Termination statements, which signify the termination of the program: EXIT, ABORT, and END.

#### 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

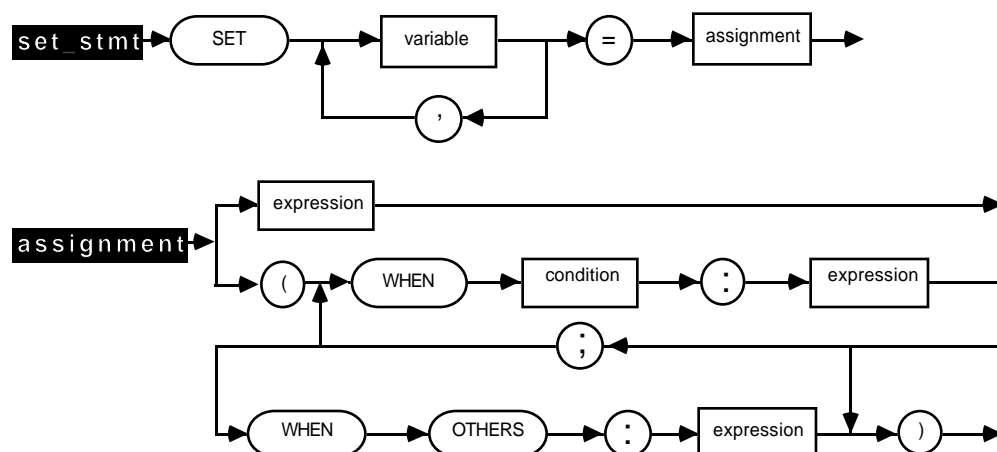
```

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)
SET ab\A100.sp, x,m4\A122.t2 = 44 -- SET referencing off-LCN points

```

#### NOTE

SET statements containing multiple point parameters, which happen to reside in different data owner types such as PM and APM, may cause a compiler Type Mismatch error.

Example: SET PMDC.OP, APMD.C.OP = 'ALARM

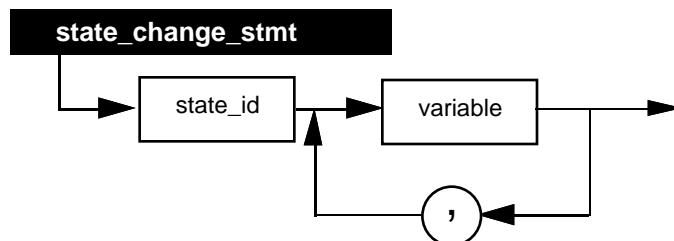
This can be corrected by simply separating the SET statements as shown in the following example:

Example: SET PMDC.OP = 'ALARM  
SET APMD.C.OP = 'ALARM

### 3.2.4 STATE CHANGE Statement

This statement sets the state of the Output (OP) parameter of one or more Digital Output data points that have two or more discrete output states.

#### 3.2.4.1 STATE CHANGE Syntax





### 3.2.4.2 STATE CHANGE Description

The variables must identify data points that have an OP parameter of a discrete type. The data points can be on-LCN or off-LCN. 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.

**close A100** is the same as **SET A100.OP = close**, and

**close A100, B100** is the same as **SET A100.OP, B100.OP = close**.

If the point identifier is omitted, the bound data point is implied. A maximum of 16 variables can be referenced with one state-change statement.

#### CAUTION

When state change statements are executed at runtime, they are subject to and checked for data access restrictions. These include access rights checks, range and limit checks, and data-transport checks. See heading 4.1.4 in *AM Control Functions*.

### 3.2.4.3 STATE CHANGE Examples

Suppose:

Motor1's output states are start/stop;

Valve2 and Valve3's output states are open/close;

A100.Sensor is a data point identifier naming a point having a Logical output;

37SW's output states are low/high;

A device control point, DEV\_CTL, has output states of up/down.

fe\A100's output states are run/stop

Then,

```
stop Motor1
open Valve2, Valve3
Off A100.Sensor
high 37SW
down DEV_CTL
run fe\A100
```

are valid STATE CHANGE statements.

## 3.2.5 GOTO Statement

This statement unconditionally branches to another place in the program.

### 3.2.5.1 GOTO Syntax



### 3.2.5.2 GOTO Description

A GOTO in a block can go to only some label within that block.

A GOTO statement can branch either forward or backward in the program. On a backward branch in the AM, a test is made to see whether the program has executed too long (in other words, a backward branch time-out); if so, the block is aborted and control processing is resumed. (Refer to *Application Module Control Functions*, Section 4, for detailed information on runtime error reporting and the parameters that affect the execution time-out.)

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

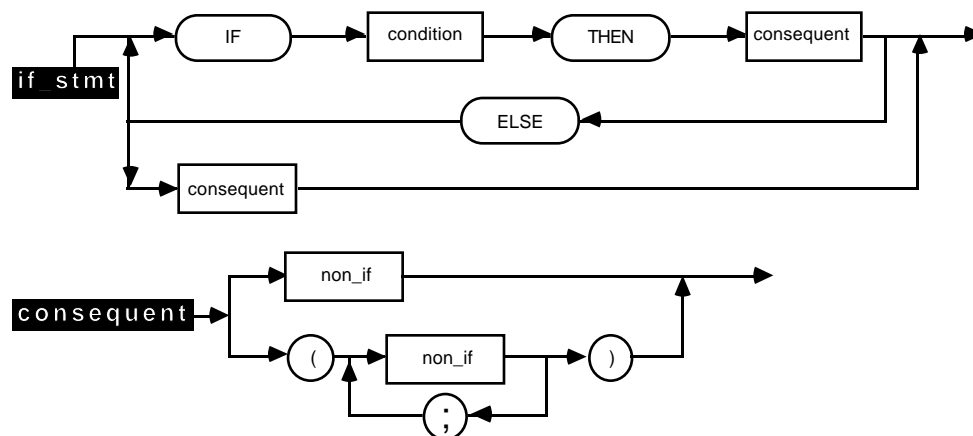
### 3.2.5.3 GOTO Examples

```
GOTO L1
GOTO Process_Error
```

## 3.2.6 IF, THEN, ELSE Statement

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

### 3.2.6.1 IF, THEN, ELSE Syntax



### 3.2.6.2 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 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.

## 3.2.6.3 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)

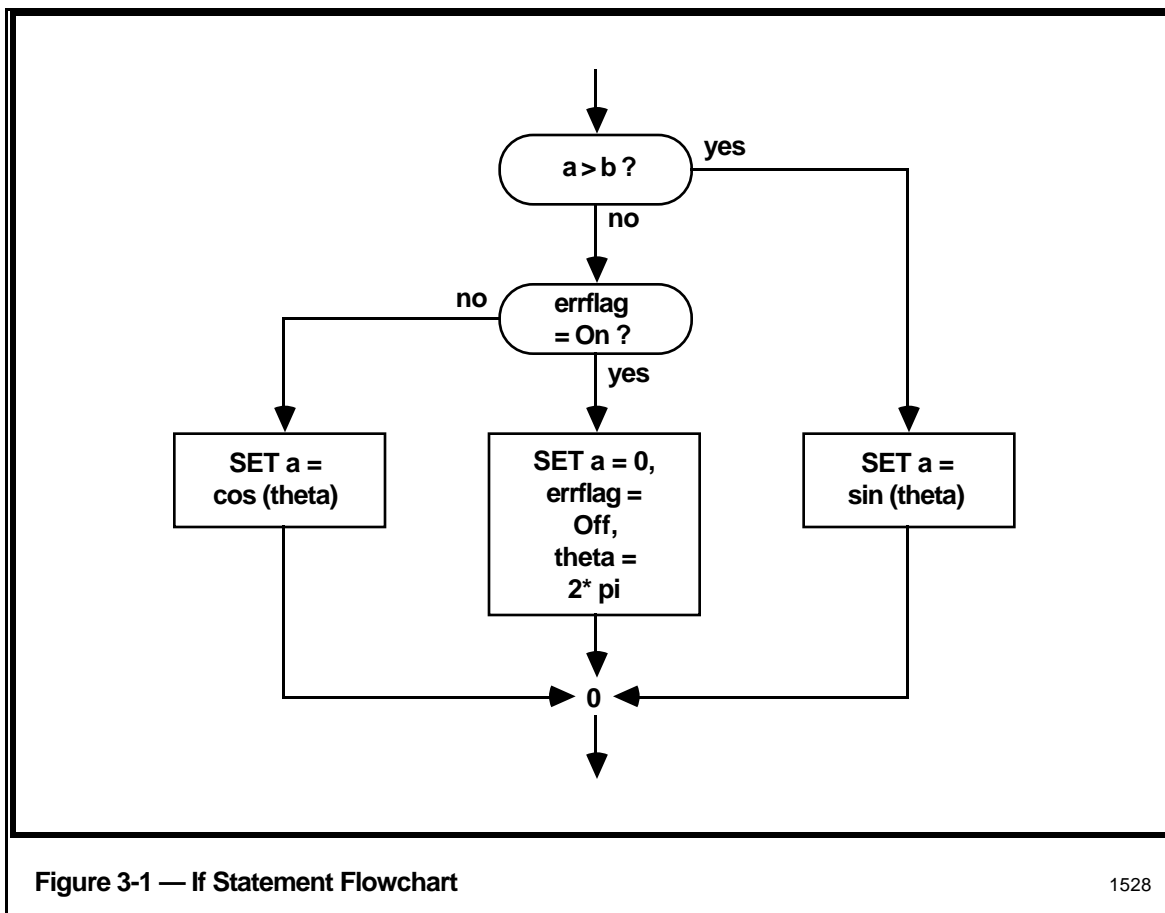
```

```

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.

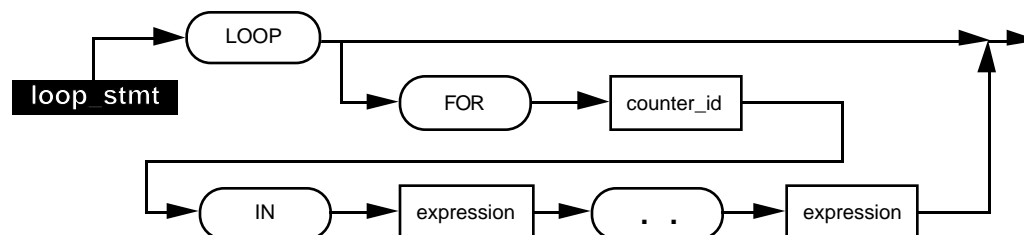


1528

### 3.2.7 LOOP Statement

This statement provides loop control in a block or subroutine.

#### 3.2.7.1 LOOP Syntax



#### 3.2.7.2 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 data 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. In CL/AM, the upper bound of the FOR range is evaluated only when beginning the loop. The lower bound must be stated first.

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.

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

A LOOP statement must have a label.

#### 3.2.7.3 LOOP Examples

```

label: LOOP
label: LOOP FOR count IN 10 .. 20
label: LOOP FOR count IN 20 .. 10  --Invalid Construct

```

### 3.2.8 REPEAT Statement

This statement causes a loop to be repeated.

#### 3.2.8.1 REPEAT Syntax



### 3.2.8.2 REPEAT Description

The target of a REPEAT statement must be a label in the current block, or subroutine.

The label\_ID must define a loop; i.e., 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.

It is a runtime error to attempt to execute a REPEAT statement without having executed its corresponding LOOP statement.

On a backward branch in the AM caused by a REPEAT statement, a test is made to see whether the program has executed too long (in other words, a backward branch time-out); if so, the block is aborted and control processing is resumed. (Refer to *Application Module Control Functions*, Section 4, for detailed information on the runtime error reporting and parameters that affect the execution time-out.)

### 3.2.8.3 REPEAT Examples

The following example demonstrates conditional execution of a REPEAT statement.

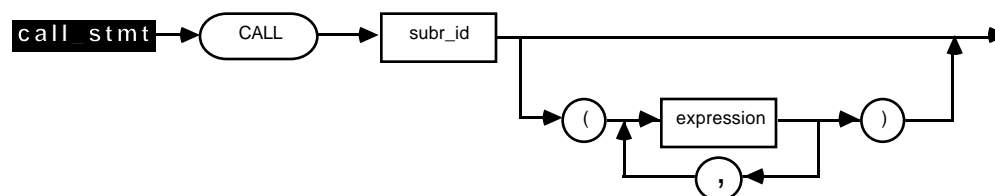
```
setx:          LOOP FOR i IN 1 .. 5
              SET x.SP = 2
              IF x.PV <> 2 THEN (REPEAT setx;
&                SEND: "x.PV bad";
&                FAIL)
```

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 reinvoke 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
```

### 3.2.9.1 CALL Syntax



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

Note that the following CALL examples match the subroutine header definitions at heading 4.3.6.1, Subroutine Arguments Examples.

```

PARAMETER PV, SP
EXTERNAL AM2DATA1      -- AM regulatory point
EXTERNAL RP002J03      -- PM regulatory PV point
EXTERNAL HG050601      -- HG digital input point
EXTERNAL D1713J03      -- PM digital input point
EXTERNAL $NM10B03      -- PM box point
LOCAL num, i, j
LOCAL flagarr : Logical ARRAY (1..3)
LOCAL numarr : Number ARRAY (1..10, 1..5)
...
CALL sub1 (HG050601.PV, D171J03.PV) -- digital input PVs
CALL sub2 (num, $NM10B03.FL(14))     -- scalar local variable, array
                                     -- parameter indexed by a const
CALL sub3 (flagarr(2), 5.5, numarr(i,j)) -- element of local array,
                                     -- numeric literal, two dimension
                                     -- array with calculated index
CALL sub4 (RP002J03.C)               -- scalar parameter
CALL sub5 (flagarr)                  -- ILLEGAL a subroutine argument
                                     -- of a whole array cannot be
                                     -- used in CL/AM
CALL sub6 (AM2DATA1.mode, PV, SP)    -- enumeration on an external
                                     -- point, two parameters on the
                                     -- linked point
CALL sub7 (open)                     -- enumeration state identifier

```

When calling a CL Runtime Extension subroutine (see heading 4.8), the subroutine name must be preceded by the CL Runtime Extension file name and a "\$". For example:

```
BLOCK sample (GENERIC; AT GENERAL)
  EXTERNAL A100
  %INCLUDE_SET modify      -- Name of a file containing definitions of the
                          -- CL Runtime Extension subroutines and
                          -- functions to be called

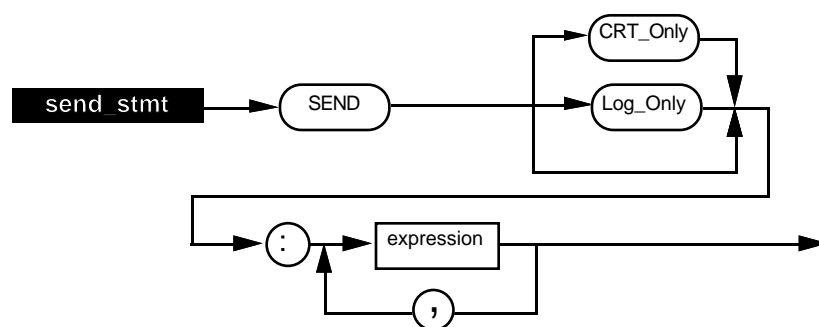
  LOCAL str : STRING
  CALL modify$INTEGER_TO_ASCII(A100.PV, str) -- File name and $ precede
                                          -- the called routine name

  SEND : str
END sample
```

### 3.2.10 SEND Statement

This statement sends a message to the operator of the Bound Data Point's Process Unit. It also can be used to request the issuing of Event Initiated Reports. These messages go to both the operator's CRT and Log device, unless the destination is specified as CRT\_Only or Log\_Only.

#### 3.2.10.1 SEND Syntax



#### 3.2.10.2 SEND Description

A special destination option **CRT\_Only** or **Log\_Only** may be specified to the left of the colon. All operator messages also go to the Operator Message Event History regardless of specified or implied destination.

The "expressions" in the SEND statement are the items to be sent. Each item can have a value of any of the types Number, Logical, Time or String (no enumerations). Integer numbers are sent as Reals. There is a 64-character limit.

The normal format for the display of time values in CL/AM messages is HH:MM:SS. However, when the -ETD compiler switch is used, the display format varies with time duration. See heading 5.3.1.2 in the *Control Language/Application Module Data Entry* manual for details. Be aware that when the -ETD switch is used, the number of characters used to display each time value could increase from 8 to 16, thus limiting the space for additional message information.

An array cannot be sent as a whole, but its elements can be individually sent.

Tag names cannot be sent in the SEND statement.

### 3.2.10.3 SEND Examples

```
SEND: "Begin cleanup"
SEND CRT_Only: "This is a CRT message"
```

### 3.2.10.4 Event Initiated Reports from CL

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

### 3.2.10.5 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.

### 3.2.10.6 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.

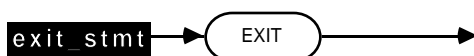
#### CAUTION

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.

## 3.2.11 EXIT Statement

When this statement is executed in a subroutine, it returns control to the subroutine's caller. When executed in a main program, it terminates the program.

### 3.2.11.1 EXIT Syntax





### 3.2.11.2 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.

### 3.2.12 ABORT Statement

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

The effect of an ABORT statement on point processing depends on the build type of the bound data point.

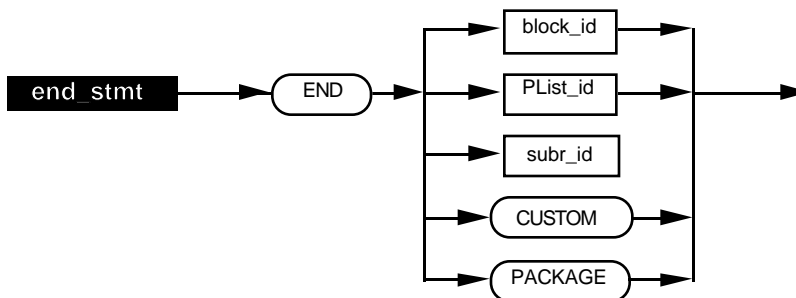
#### 3.2.12.1 ABORT Syntax



### 3.2.13 END Statement

This must be the last statement of a block, subroutine, custom data segment, parameter list, or package. Execution of this statement is identical to the EXIT statement.

#### 3.2.13.1 END Syntax



#### 3.2.13.2 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 block, subroutine, or parameter list; however, the reserved words CUSTOM or PACKAGE are used in place of an ID with Custom Data Segments and packages, respectively.

### 3.3 EMBEDDED COMPILER DIRECTIVES

This section describes compiler directives that can be directly embedded in a CL/AM 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/AM 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/AM 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 %RELAX Directive

This compiler directive allows CL/AM to relax certain Type checks performed by the linker. The specific check to be relaxed is specified in an option field. Currently, the only legal option for this directive is Linker\_SDE\_Checks. The location of the %RELAX directive within a CL/AM package determines its scope of applicability.

The %RELAX directive can occur anywhere in a CL/AM package before the first CL block header (however, it cannot be inside other definition constructs—e.g., Custom Data Segment definition, Parameter List definition). When it appears before the first block header, the %RELAX directive applies to all blocks in the package.

The %RELAX directive also can occur within a CL block between the block header and the first executable statement of the block. In this case the %RELAX directive applies to that block only.

#### 3.3.4.1 %RELAX Linker\_SDE\_Checks Option

Normally, compiled assertions regarding the state description of self-defined enumerations must match the state found in the database exactly. That is, all state descriptions listed in the source file for a self-defined enumeration parameter must match those in the database and must be in the same order.

The %RELAX Linker\_SDE\_Checks option allows CL/AM code to operate on self-defined enumerations (for example, PV of Digital points, PV of Flag points, S1CURSTS of CL Switch points, etc.) independent of the state descriptions of the individual points. This option applies to only self-defined enumeration parameters referenced indirectly (the parameter is defined in a parameter list). When this option is used, the state descriptions are not checked at Link Time; however, the linker does require that the number of states is less than or equal to the number defined at compile time.

CL execution is unchanged. That is, execution is based on the internal state values of the enumerations, determined by the order of the states. When Linker SDE checks are relaxed, it is possible for stores to be attempted that exceed the range of the destination parameter. This causes a RANGE runtime error.

The Linker\_SDE\_Checks option applies to self-defined enumeration data types only. It has no effect on logicals, customer-defined standard enumerations (defined with the ENUMERATION statement), or Honeywell-defined standard enumerations (e.g., MODE, PTEXECST, etc.).

Note that %RELAX Linker\_SDE\_Checks only relaxes Link Time checking (checking against actual parameter state names in the database). All other compiler type checking is still in effect. See example below.

Four new standard enumerations have been added to the system to be used as generic enumerations. These can be used or you can pick any other legal enumeration to define the parameter in the parameter list. The four new standard enumerations and their states are shown in Figure 3-2.

<u>Enumeration Set Name</u>	<u>States</u>
\$2ORDS	\$ORD0, \$ORD1
\$3ORDS	\$ORD0, \$ORD1, \$ORD2
\$4ORDS	\$ORD0, \$ORD1, \$ORD2, \$ORD3
\$5ORDS	\$ORD0, \$ORD1, \$ORD2, \$ORD3, \$ORD4

**Figure 3-2 — Standard Enumerations with Generic States**

**%RELAX Linker\_SDE\_Checks Example**

```

PACKAGE

    ENUMERATION colors = red/green

    PARAM_LIST plist1
        PARAMETER op   : $2ORDS
        PARAMETER pv   : logical
        PARAMETER pve  : colors
    END plist1

    CUSTOM
        PARAMETER pts : plist1 array (1..10)
    END CUSTOM

    BLOCK fred (generic; at general)

        EXTERNAL hgpt          -- hgpt.op is open/close
        EXTERNAL ampt          -- ampt has parameter pts on it
        LOCAL i
        %RELAX Linker_SDE_Checks -- must come before first
                                -- executable statement

        LOCAL a: red/green
            SET a = red
            SET hgpt.op = a          -- ERROR; not indirect reference,
                                    -- compile typechecking fails
            SET pts(1).op = a        -- ERROR; data types do not match,
                                    -- compile typechecking fails

            L1: LOOP FOR i in 1..10
                SET pts(i).op = $ord1 -- compiles & links correctly
                REPEAT L1
                SET pts(1).pv = on    -- compiles & links correctly
                SET ampt.pts(2).op = $ord0 -- compiles & links correctly
                SET ampt.pts(3).pve = a -- compiles & links correctly

            END fred

    END PACKAGE

```

In the above example, each of the points in the "pts" array can have different state names for their OP, PV, and PVE parameters. Relaxing the linker type checking does not force all of these parameters to have the same state names as listed in the source file.

If the %RELAX Linker\_SDE\_Checks directive is **not** used, all of the following have to be true at link time or an error is generated:

- All of the points in the parameter "pts" on the bound data point have to have state names of \$ord0/\$ord1 for their OP and state names of off/on for their PV.
- All of the points in the parameter "pts" on the point "ampt" have to have state names of \$ord0/\$ord1 for their OP and state names of red/green for their PVE.

### 3.3.5 %INCLUDE\_SET Directive

The %INCLUDE\_SET compiler directive allows CL/AM to call CL Runtime Extension subroutines and/or functions (see heading 4.8). Each Include Set contains one or more subroutines and/or functions.

The %INCLUDE\_SET directive can occur anywhere in a CL/AM package before the first CL BLOCK header, but cannot be within a Custom Data Segment definition or a Parameter List definition. In this case, the %INCLUDE\_SET directive applies to all blocks in the package.

The %INCLUDE\_SET directive also can occur within a CL block, between the BLOCK header and the first executable statement of the block. In this case, the %INCLUDE\_SET directive applies to that block only.

Your use of include sets is checked at compile time to confirm proper syntax for the call and at link time to confirm that the requested sets are loaded in the AM.

Naming conventions for CL Runtime Extension routines are:

- a) The File Set name can be no longer than six characters.
- b) When calling a routine from the CL Runtime Extension set, the File Set name and a "\$" must precede the name of the routine (e.g., `filenm$subroutine_name`).
- c) The routine name combined with the File Set name can be no longer than 80 characters.

Syntax of the %INCLUDE SET directive allows one or more Include Set file names (separated by commas).

#### 3.3.5.1 %INCLUDE\_SET Example

```
PACKAGE
%INCLUDE_SET init                -- 6 character maximum filename

ENUMERATION colors = red/green  -- "$" not allowed in enumeration set
                                -- name or in enumeration state names

PARAM_LIST plist1               -- "$" not allowed in PList name
  PARAMETER op : $2ords         -- "$" okay in parameter name
  PARAMETER pv : Logical
  PARAMETER pve : colors
END plist1

CUSTOM
  PARAMETER pts : plist1 ARRAY (1..10) -- "$" not allowed in CDS
END CUSTOM                      -- parameter names

%INCLUDE_SET pkggrtn
```

```

BLOCK fred (GENERIC; AT GENERAL)
  LOCAL i
  %INCLUDE_SET elmset      --must precede the first executable statement
  LOCAL a$ : red/green    -- "$" allowed in local variable names
  CALL init$init_values (pts(1).op,
&      pts(2).pv,        -- this routine is visible to
&      pts(3).pve)      -- all CL blocks in the package
  SET a$ = red
  CALL elmset$subr_nol(a$,i)  -- routine visible only to block fred
  SET a$ = pkgrtn$funct_nol(pts) -- routine visible to all blocks
END fred

BLOCK sam (GENERIC; AT GENERAL)
  LOCAL i
  LOCAL a$b$c : colors
  CALL init$init_values2 (i, a$b$c)  -- routine visible to all blocks
  SET a$b$c =pkgrtn$routine_nol(pts) -- routine visible to all blocks
END sam
END PACKAGE

```

### 3.3.6 %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 compiler expands the source file by replacing each %INCLUDE\_SOURCE directive with the material from the included file. The compiler treats the result as one CL source file, and all the normal structure and syntax rules that apply to any CL source file also apply to the source file with expanded include files. 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.

### 3.3.6.1 %INCLUDE\_SOURCE CL/AM Source Example

```

PACKAGE
%INCLUDE_SOURCE Enumer.CL
PARAM_LIST plist1
%INCLUDE_SOURCE Params
END plist1
CUSTOM
%INCLUDE_SOURCE Params
END CUSTOM
%INCLUDE_SOURCE NET>TEST>BlockA
%INCLUDE_SOURCE $F8>CL>BlockB.CL
END PACKAGE

```

### 3.3.6.2 %INCLUDE\_SOURCE CL/AM Listings Example

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

File Line Loc Text

```

      1        PACKAGE
      2        %INCLUDE_SOURCE Enumer.CL
1       3        ENUMERATION TDC = R230/R300/R400
      4        PARAM_LIST plist1
      5        %INCLUDE_SOURCE Params
2       6        PARAMETER Release    : TDC
2       7        PARAMETER Version    : NUMBER
2       8        PARAMETER Revision   : NUMBER
      9        END plist1
     10        CUSTOM
     11        %INCLUDE_SOURCE Params
3       12        PARAMETER Release    : TDC
3       13        PARAMETER Version    : NUMBER
3       14        PARAMETER Revision   : NUMBER
     15        END CUSTOM
     16        %INCLUDE_SOURCE NET>TEST>BlockA
4       17        BLOCK old(GENERIC; at General)
4       18        %INCLUDE_SOURCE Params
5       19        PARAMETER Release    : TDC
5       20        PARAMETER Version    : NUMBER
5       21        PARAMETER Revision   : NUMBER
4       22       71 SET Release    = R300
4       23       86 SET Version    = 30
4       24       102 SET Revision = 64
4       25       118 CALL VerRev(Release, Version, Revision)
4       26        END old
4       27        %INCLUDE_SOURCE VerRev
6       28        SUBROUTINE VerRev(Rel:TDC; Ver, Rev:NUMBER)
6       29       185 IF Rel=R230 THEN
6       30       191 &SEND:"R230 Release has the following Revision:",ver,rev
6       31       255 ELSE IF Rel=R300 THEN
6       32       262 &SEND:"R300 Release has the following Revision:",ver,rev
6       33       326 ELSE IF Rel=R400 THEN
6       34       333 &SEND:"R400 Release has the following Revision:",ver,rev
6       35        END VerRev
      36        %INCLUDE_SOURCE $F8>CL>BlockB.CL
7       37        BLOCK new(GENERIC; at General)

```

```

7  38      %INCLUDE_SOURCE Params
8  39      PARAMETER Release  : TDC
8  40      PARAMETER Version  : NUMBER
8  41      PARAMETER Revision : NUMBER
7  42      71 SET Release  = R400
7  43      86 SET Version  = 40
7  44     102 SET Revision = 24
7  45     118 CALL VerRev(Release, Version, Revision)
7  46      END new
7  47      %INCLUDE_SOURCE VerRev
9  48      SUBROUTINE VerRev(Rel:TDC; Ver, Rev:NUMBER)
9  49     185 IF Rel=R230 THEN
9  50     191 &SEND:"R230 Release has the following Revision:",ver,rev
9  51     255 ELSE IF Rel=R300 THEN
9  52     262 &SEND:"R300 Release has the following Revision:",ver,rev
9  53     326 ELSE IF Rel=R400 THEN
9  54     333 &SEND:"R400 Release has the following Revision:",ver,rev
9  55      END VerRev
7  56      END PACKAGE

*****No errors detected

-----COMPILATION RESULTS-----

BLOCK OLD   :Code=  399 words out of a maximum of  16000 words
      Reference List=  49 words out of a maximum of  16001 words
BLOCK NEW   :Code=  399 words out of a maximum of  16000 words
      Reference List=  49 words out of a maximum of  16001 words

New AM object file:
      File TECHPUBS.AO created
New PARAMETER List file:
      File PLIST1.PL created

Compilation was on a UP.  Memory limit in words    =  320,000
Words of heap memory used:  12,034

Options Selected:  -NoXRef -UpdateLib

The following source file(s) were referenced:

File          File Path
-----
1      NET>CL>TECHPUBS.CL (Main Source File)
2      NET>CL>ENUMER.CL
3      NET>CL>PARAMS.CL
4      NET>CL>PARAMS.CL
5      NET>TEST>BLOCKA.CL
6      NET>CL>PARAMS.CL
7      NET>CL>VERREV.CL
8      $F8>CL>BLOCKB.CL
9      NET>CL>PARAMS.CL
10     NET>CL>VERREV.CL

```



## CL/AM STRUCTURES

### Section 4

*This section describes the CL structures that you can build to control continuous processes. This section describes those CL structures that execute in the Application Module.*

#### 4.1 CL/AM STRUCTURES—GENERAL ORIENTATION

CL/AM 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/AM structures in this section are grouped into **blocks**, **subroutines**, **global data definitions**, and **packages**.

A CL/AM **block** (see heading 4.2) is a structure (program) that can be bound (linked) to a variety of data points in the Application Module; it can have **subroutines** (heading 4.3), and can be part of a **package** (heading 4.7).

**Global data definitions** consist of Custom Data Segment definitions (heading 4.4), Parameter List definitions (heading 4.5), or Enumeration-type definitions (heading 4.6). They can also be part of a **package**.

**CL runtime extensions** (heading 4.8) consist of optional subroutines and functions that can be added to your system.

The smallest executable instruction in a CL/AM structure is the statement. A program statement is a command to perform a simple action, and is used in CL/AM blocks, and subroutines. (By contrast, Attribute statements are used in Custom Data Segments and are used to describe or define rather than do.) All executable program statements are described in Section 3.

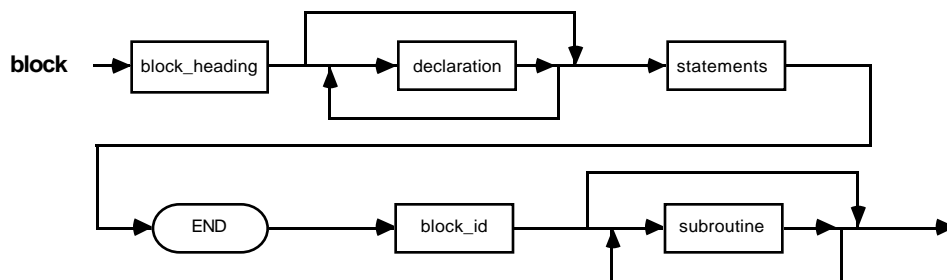
#### 4.2 BLOCK DEFINITION

A CL/AM block is a program (usually small) that is used to augment the standard features of the processing cycle of continuous control. It can be inserted at any of several defined insertion points in the point processing cycle (see Table 2-3).

A CL/AM block is the unit of CL/AM execution in continuous control. A block can be bound to a Regulatory Control, Switch, or Custom data point. It can substitute for that point's PV Algorithm or Control Algorithm, or can be inserted into its point-processing cycle at a predefined insertion point. The insertion point is specified in the header of the CL/AM block.

A block consists of a single statement group. Any subroutines to be compiled with the block must follow it.

### 4.2.1 Block Syntax



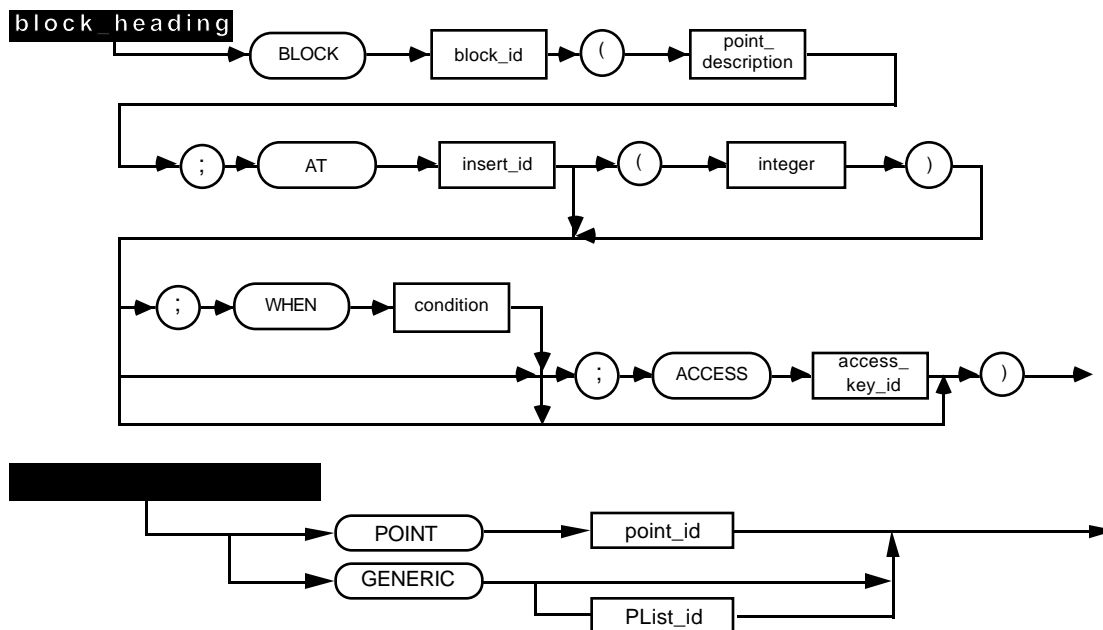
### 4.2.2 Block Description

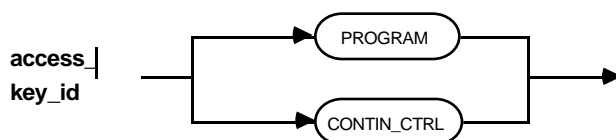
The block identifier in the END statement must match that in the heading.

### 4.2.3 Block-Heading Definition

The block heading identifies the block and, optionally, its bound data point (or parameter list). It names the PV or Control algorithm for which the block is to be substituted, or the insertion point where the block is to be executed. It can optionally name a triggering condition for block execution.

### 4.2.4 Block-Heading Syntax





### 4.2.5 Block-Heading Description

The point description identifies the data point or if the block is generic, the point description can name a parameter list. Note that the data point must be on-LCN. The parameter list can be omitted if it is unnecessary; this occurs when the program accesses no parameter of the bound data point except those named in PARAMETER declarations. The insertion-point ID names the insertion point at which the Block is to execute. A list of insertion-point identifiers is found in Table 2-3.

Multiple CL/AM blocks can be bound to a single point. Of these, some may share the same insertion point. In this case, you can specify the desired order of execution with a number in parentheses following the insertion-point ID. A block that specifies an order is executed after all blocks with lower numbers and before all blocks with higher numbers. All blocks that specify an execution order are executed before those that do not specify an order. Two blocks that specify the same number cannot be bound to the same insertion point of the same data point at the same time. The number given must be a positive, nonzero integer. The WHEN clause specifies a condition that must be true for the block to execute.

The valid insertion-point identifiers and their processing order are a function of the Build type of the bound data point. The applicability of the insertion-point identifier is checked when you link the block to the data point. If you attempt to bind a CL/AM block to a data point that does not have the specified insertion point, you get a linkage error.

The access key specifies the access privileges of the block. The access key of CONTIN\_CTRL is the default and permits full privileges. The other access key is PROGRAM, which inhibits stores to certain parameters depending on the value of the point's MODATTR parameter. Refer to heading 4.1.4.3 of *Application Module Control Functions*.

## 4.2.6 Block-Heading Examples

```

BLOCK pva200b (POINT A_101; -- specific point
&                AT PV_Alg) -- algorithm substitution

```

This example can be linked to only the specific point A\_101. It is executed at the PV\_Alg insertion point. The insertion points PV\_Alg and Ctl\_Alg are executed only if the data point is configured for the CL/AM PV or Control algorithm, respectively.

```

BLOCK badlim (GENERIC $REG_CTL;

&                AT Pst_PVPr (3);
&                WHEN PV NOT IN lolim .. hilim)

```

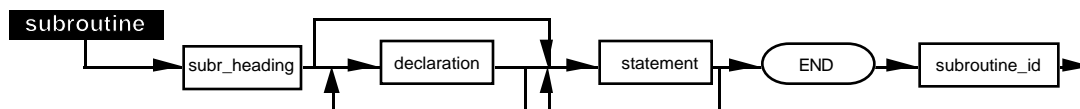
This example can be applied to any data point that is compatible with the parameter list called \$REG\_CTL. It is to be executed at the Pst\_PVPr insertion point, after Blocks at the same insertion point that are numbered 1 and 2 (if any such exist), but before Blocks at the same insertion point that are numbered greater than 3 or that do not specify an execution order. It is to be executed only if the condition **PV NOT IN lolim..hilim** is true.

## 4.3 SUBROUTINE DEFINITION

This section describes user-written CL/AM 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.

A subroutine is compiled together with a CL/AM block.

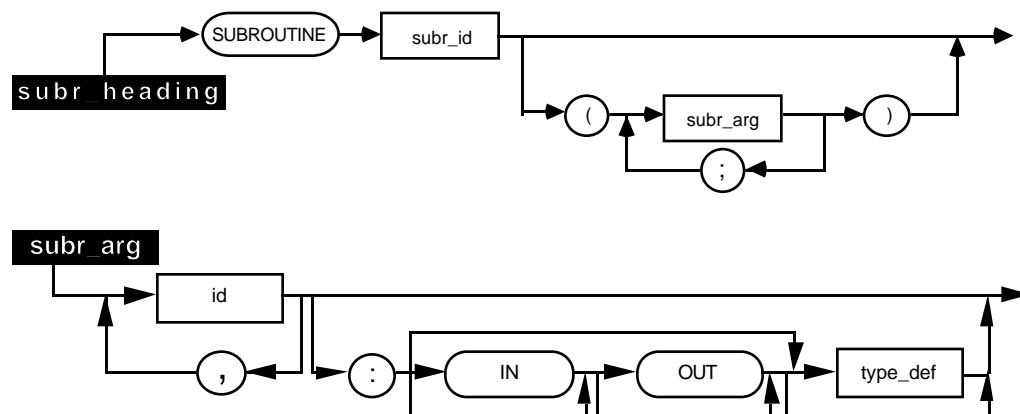
Subroutines have the following syntax:



### 4.3.1 Subroutine-Heading Definition

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

### 4.3.2 Subroutine-Heading Syntax



### 4.3.3 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 are both accessed and set. Note that OUT arguments are always stored to on exit from the subroutine, whether or not they are specifically referenced by the subroutine.

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

Before you make indiscriminate use of IN OUT for IN-only arguments, or if you are tempted to reuse a point.parameter name in the arguments list—e.g., `CALL SUB1 (A100.PV, A100.PV)`—please consider the following information on how CL/AM subroutines work:

1. When a subroutine argument is declared to be of mode IN (or IN OUT), and the subroutine is called with a point.parameter name, the current parameter value is fetched in order to pass it to the subroutine.
2. When a subroutine exits, a store is performed for each OUT (or IN OUT) argument. Thus, if a value is not assigned to an argument in the subroutine, the fetched value as sent to the subroutine is stored back into the argument.
3. The stores for the OUT arguments are done in backwards order of their appearance in the arguments list.

If any local variable that the subroutine directly accesses (that is, by name) is passed as an argument to a subroutine, it constitutes aliasing (refer to heading 2.3.4.5—Aliasing Definition). This error may be detected at compile time.

When creating your own CL/AM subroutines, you need to be aware of the following:

When a point.parameter is passed to a CL/AM subroutine as an OUT argument, the subroutine does not store directly to that parameter. Instead, the subroutine's store is made to a stack location and only the final value is stored to the parameter location. The actual store to the parameter location occurs only after completion of the subroutine. In addition, the parameter stores are made in an inverse sequence, starting with the LAST argument in the header. Thus, you should avoid subroutines that require that each store in a series be made to the same parameter or that imply an ordering of data changes.

In the two examples shown below, the intent is to store the value 44 into point.parameter pointx.amfl(3).pv. The subroutine named sub2 accomplishes this, but sub3 does not. After exiting the subroutine named sub3, value1 is stored first and thus, since index has not yet been modified, 44 is stored to pointx.amfl(1).pv instead.

#### Example 1:

```
BLOCK store2 (GENERIC; AT GENERAL)
  EXTERNAL pointx
  LOCAL index
  SET index = 1
  CALL sub2 (pointx.amfl(index).pv, index)
END store2

SUBROUTINE sub2 (value1 : OUT NUMBER; value2 : OUT NUMBER)
  SET value1 = 44
  SET value2 = 3
END sub2
```

#### Example 2:

```
BLOCK store3 (GENERIC; AT GENERAL)
  EXTERNAL point3
  LOCAL index
  SET index = 1
  CALL sub3 (index, pointx.amfl(index).pv)
END store3

SUBROUTINE sub3 (value2 : OUT NUMBER; value1 : OUT NUMBER)
  SET value1 = 44
  SET value2 = 3
END sub3
```

### 4.3.4 Subroutine-Heading Examples

```
SUBROUTINE test (x, y: NUMBER; b: OUT LOGICAL)
  LOCAL i: NUMBER
  LOCAL q: LOGICAL
  SET i = x          -- valid
  SET y = i          -- NOT VALID
  SET q = b          -- NOT VALID
  SET b = q          -- valid
END test
```

### 4.3.5 Subroutine Data-Declarations Definition

A subroutine can contain a declaration section, having the same format as the declaration section of a CL/AM block or sequence program. Local variables and functions can be defined in all subroutines, but PARAMETER and EXTERNAL declarations are not permitted.

A subroutine's local variables are not accessible from outside the subroutine. Each time the subroutine is called, its local variables are newly created; they are deleted when it exits. They do not retain information from one call to the next.

A subroutine can access all functions and variables visible from the main program with which it is compiled; however, the main program cannot access variables declared within the subroutine.

### 4.3.6 Subroutine Arguments Definition

The following data types—and array elements of these types—can be used as arguments with any category of CL/AM subroutine (built-in, user-written, or runtime extension):

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

Whole arrays and parameters of type Data Point ID cannot be passed to user-written subroutines, but can be passed to Runtime Extension subroutines and/or functions (see heading 4.8). In addition, the following built-in routines accept arguments of these types:

<u>Routine Name</u>	<u>Whole Array</u>	<u>Parameter of Type Data Point ID</u>
Move_Parameter	yes	yes
Set_Null_Point_ID	no	yes
Equal_Null_Point_ID	no	yes
Equal_Point_ID	no	yes

#### 4.3.6.1 Subroutine Arguments Example

Note that the following Subroutine Argument examples match the calling sequence at heading 3.2.9.3, CALL Examples.

```

SUBROUTINE sub1 (x : state1/state2; y : OUT off/on)
SUBROUTINE sub2 (index; log1 : IN OUT LOGICAL)
SUBROUTINE sub3 (flag : IN LOGICAL; num1 : IN NUMBER; output :
&                OUT NUMBER)
SUBROUTINE sub4 (a : IN NUMBER)
SUBROUTINE sub5 (arr1 : IN OUT LOGICAL ARRAY (1..3))
                -- sub5 will never be called since the calling
                -- sequence in 3.2.9.3 is ILLEGAL
SUBROUTINE sub6 (status : IN mode; value1, value2 :IN OUT NUMBER)
SUBROUTINE sub7 (state : IN open/close/bad/moving/inbtwn)

```

### 4.3.7 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 which 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/AM program.

#### 4.3.7.1 Built-In Arithmetic Functions

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

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

#### 4.3.7.2 Built-In Functions on Arrays

These functions accept one argument whose type is Number ARRAY. The array must be LOCAL. The number of elements in the array, and its number of dimensions, do not matter. The result of applying one of these functions to all elements in the array is a single number.

Avg(x)	-- average
Max(x)	-- maximum
Min(x)	-- minimum
Sum(x)	-- sum

#### NOTE

Badval and Finite, which can appear in the same contexts as Logical functions, are predicates returning truth values, not functions returning Logical values. Their results can be used in conditions within IF statements and WHEN clauses, but cannot be stored, compared, returned as the results of functions, used as array indices, passed to subroutines or functions, or sent by SEND statement. See heading 2.5.4.5.



### 4.3.7.3 Built-In Logical Functions

These functions return Logical results.

#### **BKG\_Switchover\_Restart**

This function is used to detect switchover and program restart in the backup AM. If CL is running on a point when a failure of the primary AM occurs, the point's CLs are re-queued (restarted) in the backup. BKG\_Switchover\_Restart returns an ON value only if called during the first execution of a CL block following switchover. Background CLs can use this function as part of a failover detection and recovery process. If, under normal running of an application, information on the state of the process has been recorded in CDS parameters, the program can use this information to recover and restart processing appropriately. For more information about background CL in redundant AMs, refer to subsection 7.4 in *Application Module Implementation Guidelines*.

#### **Comm\_Error (x)**

Comm\_Error accepts an argument that is an unsubscripted data point/parameter reference of any data type. It returns ON if there is a communications error when this parameter is prefetched, OFF otherwise. The data point may be on-LCN or off-LCN. If the data point is one that is not subject to prefetch, a communications error cannot happen and this function always returns OFF.

This function is required for safe access to non-numeric parameters.

The requirement for an unsubscripted reference means that you cannot use Comm\_Error to test an element of a point's array parameter. Use Comm\_Error on the entire parameter rather than just one element. For example, Comm\_Error(A100.x(3)) is wrong, but Comm\_Error(A100.x) can be used in its place.

Note: HG boxes placed in TEST control mode return Comm\_Error ON when the parameter requested is HG box resident (for example, MODE). Otherwise, TEST control mode returns Comm\_Error OFF when the parameter is HG resident (for example, PTEXECST).

#### **Equal\_Null\_Point\_id (p)**

This function determines if a parameter of type point\_id is an LCN null point identifier (the value being compared can be either a single CDS parameter or a subscripted element of an array). The null identifier may be an on-LCN or an off-LCN identifier. It returns ON if the point\_id is an LCN null identifier (see heading 2.3.8 for more information on null point identifiers). It will return OFF for any of the following reasons:

- 1) point\_id value is not null.
- 2) cannot access the parameter.
- 3) point\_id specifies an entire array.
- 4) point\_id is not of type entity\_id.

### Equal\_Null Point\_id Example

```

PACKAGE
CUSTOM
  PARAMETER pt1 : $REG_CTL
  EU "PT_IDENT"
  VALUE a100 -- a valid system point identifier
END CUSTOM

BLOCK eqlnul (GENERIC; AT GENERAL)
  EXTERNAL x100 -- another point with the above package attached
  IF EQUAL_NULL_POINT_ID (pt1) THEN SEND: "pt1 equals null"
  IF EQUAL_NULL_POINT_ID (x100.pt1) THEN SEND: "x100.pt1 = null"
-- sends message when the point name is equal to null
END eqlnul
END PACKAGE

```

### Equal\_Point\_id (p1, p2)

This function compares two CDS parameters of type point\_id (either value being compared can be a single CDS parameter or a subscripted element of an array). The points may be on-LCN or off-LCN. It returns ON if the two CDS point\_id values are equal (see heading 2.3.4.2 for more information on data point identifiers). It will return OFF for any of the following reasons:

- 1) point\_id values of the arguments are not the same.
- 2) cannot access either parameter.
- 3) either point\_id specifies an entire array.
- 4) either point\_id is not of type entity\_id.

### Equal\_Point\_id Example

```

PACKAGE
CUSTOM
  PARAMETER pt1 : $REG_CTL
  EU "PT_IDENT"
  VALUE a100 -- a valid system point identifier
END CUSTOM

BLOCK eqlpnt (GENERIC; AT GENERAL)
  EXTERNAL x100 -- another point with the above package attached
  IF NOT EQUAL_POINT_ID (pt1, x100.pt1) THEN
&    SEND: "pt1 does not equal x100.pt1"
-- sends message if the point names are not equal

  END eqlpnt
END PACKAGE

```

**Exists (p)**

This function accepts an argument that is an unsubscripted data point/parameter reference of any data type. It returns ON if the data point and parameter exist. It returns OFF if either the data point does not exist or the point exists but the parameter does not, or if the Exists check is through a null point identifier, or if a communications error occurs on the Exists call, or if a software inconsistency exists between CL and the data owner (ERROR1). The data point may be on-LCN or off-LCN. The argument of Exists is restricted in the same way as that of Comm\_Error.

Note: HG boxes placed in TEST control mode return Exists OFF when the parameter requested is HG box resident (for example, MODE). Otherwise, HG TEST control mode returns Exists ON when the parameter is HG resident (for example, PTEXECST).

**4.3.7.4 Built-In Miscellaneous Functions**

The “Date\_Time” and “Now” functions return Time values; all other functions in this group return Number values.

**Date\_Time**

Date\_Time returns the absolute **TotalPlant** Solution (TPS) System date/time; that is, seconds since midnight of 1 Jan 1979.

**Len (s)**

Len returns the length of the String it is passed. For example,  
`Len ( "abcde" ) = 5; Len ( " " ) = 0.`

**Now**

Now returns wall-clock time (seconds since midnight of the current day).

**Number (t)**

This function takes a Time expression (t) as input and returns the resulting time duration as a number. The returned value is the number of seconds represented by the time expression. Time has greater precision than number so the returned value may be less precise than the time expression (see heading 2.5.3.5).

**Ord (e)**

This function takes the current state of a standard enumeration, custom enumeration, or self-defining enumeration and returns a number value (where zero represents the first enumeration member). The enumeration can be in a CL LOCAL variable, a parameter on the Bound Data Point, or a parameter on another point. The CL will abort on any of these errors:

- 1) Reference is not to an enumeration or self-defining enumeration (“cnferr” error returned at runtime).
- 2) Reference specifies a whole array (“cnferr” error returned at runtime).
- 3) Cannot access enumeration of the point referenced (runtime error type is data owner dependent).

**Ord Example:**

```

BLOCK ordx (GENERIC; AT GENERAL)
  EXTERNAL x100      -- other AM point with valid PTEXECST parameter
  LOCAL i            -- type number
  LOCAL m : PTEXECST

  SET m = ACTIVE
  SET i = ORD (m)
  SEND: "local ptexecst active is ", i

  IF ORD (x100.PTEXECST) = i THEN
&   SEND: "x100 active is ", ORD(x100.PTEXECST)
  ELSE SEND: "x100 inactive is " , ORD (x100.PTEXECST)

END ordx

```

**Self**

This function returns the slot number of the currently executing CL/AM block. For example, suppose a CL/AM block needs to read the parameter CLBLKERR to determine whether any errors occurred the previous time the block executed. A data point can have many CL/AM blocks; therefore, all parameters of the CL/AM segment that deal with individual blocks, like CLBLKERR, are arrays indexed by slot number. A CL/AM block must be able to discover its slot number, because the slot number can change as blocks are linked and unlinked to the point. This function is the mechanism provided to do so; therefore, the statement

```

  IF clblkerr(SELF) <> noerror THEN
&   SEND: "Error last time I ran"

```

could be used to determine if the block had errors the last time it executed.

**4.3.7.5 Built-In Subroutines****Allow\_Bad (x, y)**

This subroutine stores the value of y into the numeric variable x. If x is a parameter of a data point and y is a bad value, the store proceeds without error. Note that if a SET statement had been used, an attempt to store a bad value into a parameter of a data point would constitute a runtime error; the store would not be completed and the program would abort.

Note that if y is a bad value and you attempt to store it into an integer parameter x, a runtime error occurs and the store is not completed even if Allow\_Bad is used.

Note also, that use of an equivalence statement instead of a comma—e.g., Allow\_Bad (x=y)—results in the compiler error message "Internal Error, Save source and call Honeywell." If this occurs, just fix the syntax error and recompile.

**BKG\_Change\_Priority (p)**

This subroutine permits background CL programs to vary their priority (where p is a \$BKGPRTY enumeration value of LOW, MEDIUM, or HIGH). Background CL programs always start at HIGH priority; however, this priority is lower than all other tasks in the AM (i.e., lower than checkpointing, point building, and CL linking).

By changing the priority of a Background CL Block while it is running to HIGH, that block—once initiated—will run from beginning to end without being interrupted by another background task set to MEDIUM or LOW priority. The background executive resets the background priority to HIGH on entry and exit of the CL.

If the CL block is already at the priority being requested, there is no error.

BKG\_Change\_Priority aborts the CL on these errors:

- 1) request to set priority is not successful (error1)
- 2) request value is not LOW, MEDIUM, or HIGH (range)

**BKG\_Change\_Priority Example:**

```
BLOCK abc (GENERIC; AT BACKGRND)
  LOCAL prior : $BKGPRTY

  CALL BKG_CHANGE_PRIORITY (LOW)
  -- starts at high priority and can be changed on entry
  . . .
  -- and at any time during execution
  SET prior = HIGH
  CALL BKG_CHANGE_PRIORITY (prior)
END abc
```

**BKG\_Delay (t)**

This subroutine sets a delay time, in seconds, for a Background CL Block. The minimum delay interval is zero; the maximum is one hour. A zero delay length causes the Background CL Block to stop running for as short a time as the system will allow.

A CL abort check is performed at 5 second intervals during the delay. BKG\_Delay aborts the CL for these reasons:

- 1) the delay time is greater than one hour (range).
- 2) system delay error (error1).
- 3) operator abort (abort).

**BKG\_Delay Example:**

```
BLOCK bcd (GENERIC; AT BACKGRND)
  LOCAL tconst : TIME

  SET tcnst = 1 MINS 5 SECS
  SEND: "Start"
  CALL BKG_DELAY (2 MINS)
  SEND: "2 mins"
  CALL BKG_DELAY (20 SECS)
  SEND: "20 secs"
  CALL BKG_DELAY (tconst)
  SEND: tconst
END bcd
```

**CDS\_Read (s, fm, p, n)**

This subroutine reads a Custom Data Segment (CDS) from a History Module or removable mediafile (see CDS\_Write) and overwrites an existing CDS on the Bound Data Point. It can only be called by a Background CL Block.

**NOTE**

Heavy use of CDS reads and/or writes can adversely affect other HM-related activities such as schematic callup and history collection.

Where: “s” will contain the return status of the read request  
 “fm” will contain additional status information for some failure types (see C.2.4 for removable media or remote LCN path identification)  
 “p” is the full pathname of the file from which the CDS values will be read (must be uppercase)  
 “n” is the name of a CDS on the Bound Data Point (must be uppercase)

This subroutine can overlay an existing CDS only when the structure of the source (stored) CDS and destination (existing) CDS match. This means that both CDS must have the same number of parameters and the data type of each parameter must match. Values in the parameters and names of the parameters can differ between the two CDS. If the CDS structures do not match, CDS\_Read will not overlay values in the existing CDS and will return a status value of MISMATCH.

The only status values to be expected (for a complete listing see heading 4.11) are:

OK	= successful read of CDS values
BADPATH	= an invalid file path was specified
BADFILE	= the specified file does not contain a CDS
MISMATCH	= structures of the existing and file CDS do not match
NOCDS	= the specified CDS does not exist
FMNOFIL	= the specified file does not exist
FMHARD	= the device on which the file is located reported a physical error condition
FMNOVOL	= the specified volume does not exist
FMMOUNT	= the specified volume is not mounted
FMOTHER	= another type of File Manager error has occurred; the “fm” status argument contains a supplemental status code

“fm” Status Information—There are several problems that the file manager can encounter that are caused by incorrect system configuration or by failures in system hardware or software. When one of these occurs, the “s” status value is set to FMOTHER and the “fm” status will contain a File Manager status number (see heading 4.10).

**CDS\_Read Example:**

```
LOCAL stat      : $CLFSTAT
LOCAL fmstat    : NUMBER
LOCAL path, cds_name : STRING
```

```
SET path = "NET>BTCH>RECIPE34.XX"
SET cds_name = "RECIPE"
```

```
CALL CDS_READ (stat, fmstat, path, cds_name)
```

If no error is detected, the Bound Data Point’s CDS named RECIPE will contain the CDS values from the file named RECIPE34.X.

**CDS\_Write (s, fm, p, n, o)**

This subroutine writes an existing Custom Data Segment (CDS) to a History Module or removable media file. It can only be called by a Background CL Block.

**NOTE**

Heavy use of CDS writes and/or reads can adversely affect other HM-related activities such as schematic callup and history collection.

Where: “s” will contain the return status of the write request

- “fm” will contain additional status information for some failure types
- “p” is the full pathname of the file to which the CDS will be written (see C.2.4)
- “n” is the name of a CDS on the Bound Data Point (must be upper case)
- “o” indicates whether the file should be overwritten if it already exists

The CDS\_Write subroutine allows the contents of an existing Custom Data Segment (CDS) to be stored in a file. The CDS can subsequently be written over the original CDS or any in the AM of identical structure (see the CDS\_Read routine).

The only status values to be expected (for a complete listing see heading 4.11) are:

- |         |   |
|---------|---|
| OK      | = successful store of CDS values  |
| BADPATH | = an invalid file path was specified  |
| NOCDS   | = the specified CDS does not exist  |
| CORRUPT | = write of the CDS failed after the initial record was written to the file<br>The file exists, but it does not contain a complete CDS. Reading this file to a CDS will cause indeterminate results. |
| FMEXIST | = the file already exists and the Overwrite option was not specified  |
| FMNOFIL | = the specified file does not exist   |
| FMHARD  | = the device on which the file is located reported a physical error condition   |
| FMFILES | = no more files can be created on the specified volume  |
| FMNOVOL | = the specified volume does not exist   |
| FMMOUNT | = the specified volume is not mounted   |
| FMVFULL | = there is no more room on the specified volume   |
| FMOTHER | = another type of File Manager error has occurred; the “fm” status argument contains a supplemental status code   |

“fm” Status Information—There are several problems that the file manager can encounter that are caused by incorrect system configuration or by failures in system hardware or software. When one of these occurs, the “s” status value is set to FMOTHER and the “fm” status will contain a File Manager status number (see heading 4.10).

The file name must include a suffix; however, this can be any valid file suffix (e.g., name.x).

Overwrite—the “o” argument controls whether an existing CDS file should be overwritten. The following table shows the actions taken depending on the value of “o” is ON or OFF, and whether or not the file exists.

	Overwrite value	
	ON	OFF
File Exists	Delete File & Create File	Error Return
File Does NOT Exist	Create File	Create File

#### CDS\_Write Example:

```
LOCAL stat      : $CLFSTAT
LOCAL fmstat    : NUMBER
LOCAL path, cds_name : STRING
```

```
SET path = "NET>BTCH>RECIPE34.XX"
SET cds_name = "RECIPE"
```

```
CALL CDS_WRITE (stat, fmstat, path, cds_name, OFF)
```

#### Delete\_File (s, fm, p)

This subroutine deletes the named History Module file (unless the file is protected). It can only be called by a Background CL Block.

Where: “s” will contain the return status of the read request

“fm” will contain additional status information for some failure types

“p” is the full pathname of the file to be deleted (must be uppercase)

The only status values to be expected (for a complete listing see heading 4.11) are:

OK	= successful file deletion
BADPATH	= an invalid file path was specified
FMNOFIL	= the specified file does not exist
FMHARD	= the device on which the file is located reported a physical error condition
FMNOVOL	= the specified volume does not exist
FMMOUNT	= the specified volume is not mounted
FMOTHER	= another type of File Manager error has occurred; the “fm” status argument contains a supplemental status code

“fm” Status Information—There are several problems that the file manager can encounter that are caused by incorrect system configuration or by failures in system hardware or software. When one of these occurs, the “s” status value is set to FMOTHER and the “fm” status will contain a File Manager status number (see heading 4.10 for interpretations).



**Delete\_File Example:**

```

LOCAL stat      : $CLFSTAT
LOCAL fmstat    : NUMBER
LOCAL path, cds_name : STRING

SET path = "NET>BTCH>RECIPE34.X"
CALL DELETE_FILE (stat, fmstat, path)

```

If no error is detected, the file named `recipe34.x` will have been deleted from the directory named `btch` on the History Module.

**Enum\_Value\_Store (e, o, s)**

This subroutine takes a real number as input, converts it to integer, stores that value as the state of an enumeration or SDE and returns a success/fail status value.

Where: “e” is a reference to a standard enumeration, or custom enumeration, or self-defining enumeration (the enumeration can be a CL LOCAL variable, a parameter on the Bound Data Point, or a parameter on another point) to be stored to.

“o” is a NUMBER value that represents the desired ordinal state of the referenced enumeration (where zero is the first enumeration state).

“s” is a CLERRSTS enumeration value expressing return status.

Off-node store requests initiated from Background CL Blocks take place immediately; Off-node store requests from Foreground Blocks take place at the end of point processing.

The CL will have an abnormal status return for any of these reasons:

- 1) Reference is not to an enumeration or self-defining enumeration (cnferr).
- 2) Reference specifies a whole array (cnferr).
- 3) Cannot store enumeration to the referenced point.parameter (error type is data owner dependent).
- 4) CL LOCAL store value out of range (cnferr)

**Enum\_Value\_Store Example:**

```

BLOCK enmx (GENERIC; AT GENERAL)
  EXTERNAL x100      -- other AM point with valid PTEXECST parameter
  LOCAL status : CLERRSTS
  LOCAL m : PTEXECST

  -- set local variable active through Enum_Value_Store
  CALL ENUM_VALUE_STORE (m, 1.0, status)
  IF status <> NOERROR THEN SEND: "enum val store err", ORD (status)

  -- set external point active through Enum_Value_Store
  CALL ENUM_VALUE_STORE (x100.PTEXECST, ORD(M), status)
  IF status <> NOERROR THEN SEND: "enum val store err", ORD (status)
  -- the following produces similar results, but Enum_Value_Store
  -- has the advantage of returning a success/fail status
  SET x100.PTEXECST = ACTIVE
END enmx

```

**Get\_CL\_Slot (s, i)**

This subroutine takes a string input (s), compares it with the names of all CL blocks on the point and returns real number index value (i) if a match is found (returns -1 if no match is found). If the input string is longer than eight characters, only the first eight characters are used for the comparison. The character string must be all in upper case.

Reasons for Get\_CL\_Slot index to be invalid are:

- 1) slot name cannot be found (-1.0).
- 2) the character string contains lower case alphabetic characters (-1.0)

**Get\_CL\_Slot Example:**

```
BLOCK gtslt (GENERIC; AT GENERAL)
  LOCAL i

  CALL GET_CL_SLOT ("GTSLT", i) -- note upper case string
  IF i <> -1.0 THEN SEND: "gtslt is cl slot", i
  ELSE SEND: "gtslt not found"
  -- is similar to the following (however, get_cl_slot can find
  -- other on-point block names
  SET i = SELF
  SEND: "gtslt is cl slot ", i

  CALL GET_CL_SLOT ("ABCDEFGHijkl", i)
  -- note that only a-h are used for the comparison
  IF i <> -1.0 THEN SEND: "abcdefgh is cl slot", i
  ELSE SEND: "gtslt not found"

END gtslt
```

**Modify\_String (s, ts, tp, n, ss, sp)**

This subroutine changes the value of a substring in a target string to the value of a substring copied from a source string. It can be called by either Background or Foreground CL Blocks.

Where: "s" is a \$MODSTR enumeration value expressing return status.

"ts" is the target string to be modified (IN OUT)

"tp" is the index to the first character of the target string to modified

"n" is the number of characters to move from the source string to the target string

"ss" is the string from which characters will be fetched

"sp" is the index to the first character to be fetched from the source string

If the number of characters (n) is greater than the number of characters between the source position index (sp) and the end of the source string, only the number of characters that exist in the source substring are moved to the target substring. A status of "OK" is returned.

If the number of characters (n) is greater than the number of characters between the target position index (tp) and the end of the target string, the number of characters that will be moved is limited by the maximum target string length of 78 characters. A status of "OK" is returned.

If the target position index (tp) is greater than 78, or the source position index (sp) is greater than the length of the source string, a status of FAILED is returned.

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

#### Modify\_String Example 1:

```
LOCAL stat : $MODSTR
LOCAL target, source : STRING
LOCAL tindex, sindex, chars

SET target = "net>cust>recipenn.x"
SET tindex = 16.0
SET chars = 2.0
SET source = "ab"
SET sindex = 1.0

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

(Results of this example are target = "net>cust>recipeab.x" and stat = OK)

#### Modify\_String Example 2:

```
SET target = "net>cust>recipenn.x"
SET tindex = 16.0
SET chars = 2.0
SET source = "ab"
SET sindex = 2.0

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

(Results of this example are target = "net>cust>recipebn.x" and stat = OK)

### Move\_Parameter (x, y, s)

This subroutine fetches a parameter value, copies it to a specified destination, and returns a success/fail status value.

Where: "x" identifies the parameter that is to receive the value (destination data).

"y" identifies the parameter value to be fetched (source data).

"s" is a CLERRSTS enumeration value that defines the success or failure of the store request.

The parameters being moved can be scalar (single elements), array elements, or entire arrays. The parameters may be on on-LCN or off-LCN points. The source and destination data types must match (see exceptions following) and, when an entire array is referenced, the array sizes also must match. Attempting to move entire arrays of strings generates a runtime configuration error (CONFERR). The requirement for data type matching allows these exceptions:

- Transfers between LCN Real and Integer data types are allowed.
- Transfers between LCN Enumeration and Self-Defined Enumeration data types are allowed as long as the value does not exceed the target enumeration range.

**NOTE**

For a foreground CL with off-node destination, the status indicates the status of a store to a post-store buffer. For a background CL, the status indicates the store completion status.

Off-node Move\_Parameter requests initiated from Background CL Blocks take place immediately; Off-node Move\_Parameter requests from Foreground Blocks take place at the end of point processing.

The values referenced in a Move\_Parameter call can be CL LOCAL variables, Bound Data Point parameters, and parameters on other points (on-node or off-node). Transfers between CL LOCAL variables and LCN parameters of comparable type and size are allowed.

Move\_Parameter can transfer point id type data from one CDS parameter to another. CL indirections through changed point identifiers are automatically relinked. Indirection changes initiated from Background CL Blocks take effect immediately, while indirection changes initiated from foreground blocks take effect at the end of point processing. See heading 2.3.4.2, Data Point Identifiers Definition for additional information.

The Move\_Parameter subroutine is similar to Allow\_Bad in its handling of real values. A communication error encountered on a real (NUMBER) parameter fetch results in a bad value being stored to the destination parameter and a COMERR status being returned. Also, if a range error is encountered on a real parameter fetch, a bad value is stored in the destination parameter and a NOERROR status is returned.

Off-node Move\_Parameter requests initiated from Background CL Blocks take place immediately; off-node Move\_Parameter requests initiated from Foreground CL Blocks take place at the end of point processing.

Move\_Parameter can fetch/store AM Interface (AMIF)/AM Gateway (AMG) parameters. The AMIF/AMG parameters cannot do entire array access.

Stores greater than 78 characters long to LOCAL strings are truncated at 78 with no error indication.

Move\_Parameter performs validity checks on the fetch and store in one CL operation. Any errors encountered are returned to the CL as “status” and the CL does not abort. (Note that Foreground post-store errors are not available.) The CL will have an abnormal status return for any of these reasons:

- 1) NaN real value was stored (comerr/badvalst)
- 2) local enum store with value greater than max (cnferr).
- 3) local self-defining enumeration store with value greater than max (cnferr)
- 4) source and destination data type not the same (cnferr)
- 5) source and destination array type or number of elements not the same (cnferr)
- 6) value fetch or store failed (data owner dependent)
- 7) entire array moves that exceed the maximum limit (arraylim), which is 1000 for integer, real, boolean, and enumeration types; 666 for time type; 500 for entity type; 285 for self-defining enumeration type; and 50 for local string type.

## Move\_Parameter Example:

```

BLOCK locarr (GENERIC; AT GENERAL)
-- local array and array element examples
  LOCAL l, ll : LOGICAL ARRAY (1..2)
  LOCAL t, tt : TIME      ARRAY (1..2)
  LOCAL r, rr : NUMBER    ARRAY (1..2)
  LOCAL e, ee : MODE      ARRAY (1..2)
  LOCAL s, ss : STRING    ARRAY (1..2)
  LOCAL status : CLERRSTS

  SET l(1) = ON
  SET l(2) = ON
  SET ll(1) = OFF
  SET ll(2) = OFF

  CALL MOVE_PARAMETER (ll, l, status) -- overwrite "ll" parameter
                                     -- array "OFF" values with "l"
                                     -- parameter array "ON" values
  IF status <> NOERROR THEN SEND: "logical array move failed"

  SET l(1) = OFF -- move l element 1 to overwrite ll element 2
  CALL MOVE_PARAMETER (ll(2), l(1), status)
  IF status <> NOERROR THEN SEND: "element move failed"

  SET t(1) = 0 days 1 hours 2 mins 3 secs
  SET t(2) = 0 days 4 hours 5 mins 6 secs
  SET tt(1) = 0 days 3 hours 2 mins 1 secs
  SET tt(2) = 0 days 6 hours 5 mins 4 secs
  -- move t array to overwrite tt array
  CALL MOVE_PARAMETER (tt, t, status)
  IF status <> NOERROR THEN SEND: "time array move failed"

  SET r(1) = 0.0
  SET r(2) = 1.0
  SET rr(1) = 2.0
  SET rr(2) = 3.0
  -- move r array to overwrite rr array
  CALL MOVE_PARAMETER (rr, r, status)
  IF status <> NOERROR THEN SEND: "real array move failed"

  SET e(1) = MAN
  SET e(2) = MAN
  SET ee(1) = CAS
  SET ee(2) = CAS
  -- move e array to overwrite ee array
  CALL MOVE_PARAMETER (ee, e, status)
  IF status <> NOERROR THEN SEND: "enum array move failed"

  SET s(1) = "abcd"
  SET s(2) = "efgh"
  SET ss(1) = "ijkl"
  SET ss(2) = "mnop"
  -- move s array to overwrite ss array
  CALL MOVE_PARAMETER (ss, s, status)
  IF status <> NOERROR THEN SEND: "string array move failed"

END locarr

```

```

PACKAGE
  CUSTOM
-- illustrates parameters on bound point and another point
  PARAMETER areal    : NUMBER
  PARAMETER astring  : STRING
  PARAMETER alogic   : LOGICAL
  PARAMETER atime    : TIME
  PARAMETER amode     : MODE
  PARAMETER pt       : $REG_CTL
  VALUE al00 -- a valid system point identifier
END CUSTOM
-- single parameter and local/bound point/other point example
BLOCK single (GENERIC; AT GENERAL)
  EXTERNAL x100    -- another AM point with above package attached
  LOCAL r : NUMBER
  LOCAL s : STRING
  LOCAL l : LOGICAL
  LOCAL t : TIME
  LOCAL e : MODE    -- enumeration
  LOCAL status : CLERRSTS

-- overwrite alogic value on bound point and off-point
  SET l = ON
  CALL MOVE_PARAMETER (alogic, l, status)
  IF status <> NOERROR THEN SEND: "BDP logical move failed"
  CALL MOVE_PARAMETER (x100.alogic, l, status)
  IF status <> NOERROR THEN SEND: "offpt logical move failed"

-- overwrite atime value on bound point and off-point
  SET t = 0 days 1 hours 2 mins 3 secs
  CALL MOVE_PARAMETER (atime, t, status)
  IF status <> NOERROR THEN SEND: "BDP time move failed"
  CALL MOVE_PARAMETER (x100.atime, t, status)
  IF status <> NOERROR THEN SEND: "offpt time move failed"

-- overwrite areal value on bound point and off-point
  SET r = 12.3
  CALL MOVE_PARAMETER (areal, r, status)
  IF status <> NOERROR THEN SEND: "BDP real move failed"
  CALL MOVE_PARAMETER (x100.areal, r, status)
  IF status <> NOERROR THEN SEND: "offpt real move failed"

-- overwrite amode value on bound point and off-point
  SET e = MAN
  CALL MOVE_PARAMETER (amode, e, status)
  IF status <> NOERROR THEN SEND: "BDP enum move failed"
  CALL MOVE_PARAMETER (x100.amode, e, status)
  IF status <> NOERROR THEN SEND: "offpt enum move failed"

-- overwrite astring value on bound point and off-point
  SET s = "string"
  CALL MOVE_PARAMETER (astring, s, status)
  IF status <> NOERROR THEN SEND: "BDP string move failed"
  CALL MOVE_PARAMETER (x100.astring, s, status)
  IF status <> NOERROR THEN SEND: "offpt string move failed"

```

```
-- move point_id parameter value
  CALL MOVE_PARAMETER (x100.pt, pt, status)
  IF status <> NOERROR THEN SEND: "offpt indirection move failed"
END single
END PACKAGE
```

### **Number\_to\_String (s, str, i, f)**

This subroutine creates a human readable characterization of a real number. It can be called by either Background or Foreground CL Blocks.

Where: “s” will contain the return status of the store request.

OK = successful conversion

BADFORM = the format string was incorrect

BADVALU = the number provided was bad

“str” will contain the character representation of the specified real value

“i” identifies the number to convert

“f” is a format specification describing the desired character representation of the converted number. Format specifications are as described in Appendix A of the *Picture Editor Reference Manual*. `Number_to_String` accepts only format specifications “Real” and “Unknown”. There is no default, thus a value **must** be specified for this argument.

### **Number\_to\_String Example:**

```
LOCAL i
LOCAL str      : STRING
LOCAL fmt      : STRING
LOCAL stat:    : $CONVRS

SET fmt = "G99999"
SET i = 23.44
CALL NUMBER_TO_STRING (stat, str, i, fmt)
SEND: "The number is =", str
```

This example will produce the following message in the operator message display:  
The number is = 23.44

### **Set\_Bad (x)**

This subroutine stores the bad value bit pattern into the numeric variable x.

**Set\_Null\_Point\_id (p, s)**

This subroutine will store a null point identifier to a CDS parameter (p) of type data point identifier (this can be either a single CDS parameter or a subscripted element of an array) and returns a success/fail status (s). The status is a CLERRSTS enumeration value that will be abnormal for any of the following reasons:

- 1) point\_id specifies an entire array (cnferr).
- 2) destination is not a CDS of type point\_id (cnferr).
- 3) store failed (data owner dependent)

For CL Blocks compiled on Release 400 and subsequent releases, a call to the Set\_Null\_Point\_id subroutine first fetches the value of the parameter to determine the LCN identifier of the point, and then will store the null point identifier with the correct LCN identifier.

For CL Blocks compiled prior to Release 400, a call to the Set\_Null\_Point\_id subroutine stores an on-LCN null point id into the referenced parameter. If the parameter contains an off-LCN point id, the LCN indication is lost, and a reference to that parameter generates a configuration error at runtime. If you need to be able to retain an off-LCN indication in the parameter for these blocks, unlink, recompile the CL Block on Release 400 and relink the block.

**NOTE**

For a foreground CL with off-node destination, the status indicates the status of a store to a post-store buffer. For a background CL, the status indicates the store completion status.

Off-node store requests initiated from Background CL Blocks take place immediately, while off-node store requests initiated from foreground blocks take place at the end of point processing.

**Set\_Null\_Point\_id Example**

```

PACKAGE
CUSTOM
  PARAMETER pt1 : $REG_CTL
  EU "PT_IDENT"
  VALUE a100 -- a valid system point identifier
END CUSTOM

BLOCK setnul (GENERIC; AT GENERAL)
  EXTERNAL x100    -- another point with the above package attached
  LOCAL status : CLERRSTS

  -- change a100 on BDP to null value (-----)
  CALL SET_NULL_POINT_ID (pt1, status)
  IF status <> NOERROR THEN SEND: "pt1 error", ORD(status)

  -- change a100 on external point to null value (-----)
  CALL SET_NULL_POINT_ID (x100.pt1, status)
  IF status <> NOERROR THEN SEND: "x100.pt1 error", ORD(status)

END setnul
END PACKAGE

```



## 4.4 CUSTOM DATA SEGMENTS DEFINITION

A Custom Data Segment (CDS) is a set of parameters that can be added to a data point. The CL/AM compiler defines a CDS to the system; the Data Entity Builder (DEB) is then used to add instances of that CDS to one or more data points.

The main use of Custom Data Segments is to support CL/AM programs, but a data point can have Custom Data Segments that are not used by CL; in fact, Custom Data Segments can be attached to data points to which no CL/AM block is bound (for example, a CDS in a Computer Gateway).

With R530, the number of Custom Data Segment names has increased from 2000 to 4000 to allow for a greater capacity. The load time for loading the CDS names has also been optimized with R530; the load time is significantly less than the load time required before adding the additional 2000 CDS names.

Custom Data Segments can have parameters of only the following data types, but can have 1-dimensional arrays of any of these types:

- (single precision) Number
- Time
- discrete types
- data point identifiers
- String

For each parameter, you can specify attributes that help to define how it is to be built, accessed, and displayed. These attributes include

- access level
- visibility to the DEB
- initial value
- Engineering Units description
- initial default value

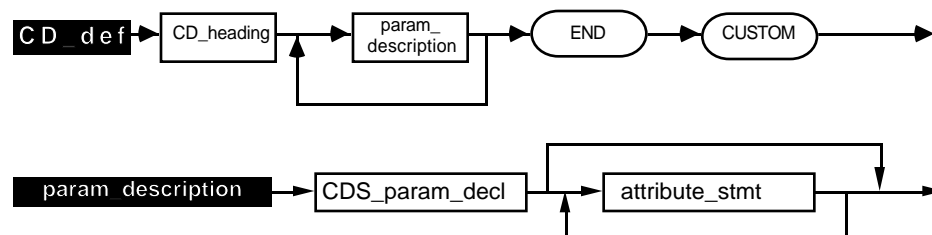
Additional information on attributes is in the *Application Module Control Functions* manual.

### WARNING

For performance, use local CL variables rather than parameters in custom data segments or prefetched data. Access times for custom parameters and prefetched data is about an order of magnitude slower than local CL access.

### 4.4.1 Custom Data Segment Syntax

A Custom Data Segment (CDS) Definition consists of a **segment heading** followed by one or more **parameter descriptions**, and ends with END CUSTOM. Each parameter description consists of a **parameter declaration** followed by zero or more **attribute statements**.



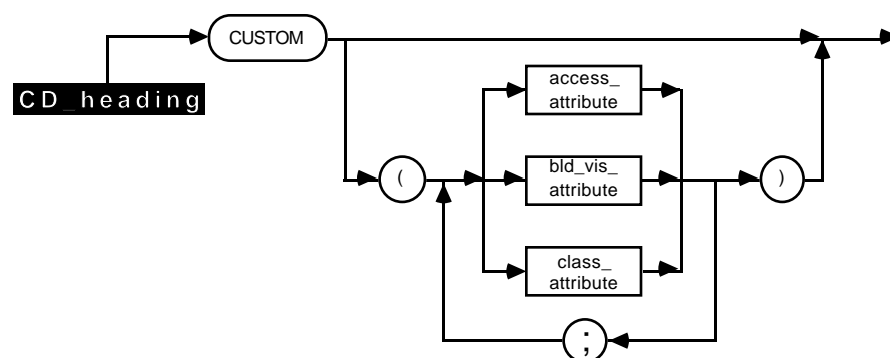
### 4.4.2 Custom Data Segment Description

The Custom Data definition describes a set of custom parameters to be applied to a data point. These parameters are identified to the system as a Custom Data Segment (CDS), whose name is the same as that of the CL/AM source file in which the Custom Data definition was compiled.

### 4.4.3 Custom Data Segment Heading

The Custom Data Segment heading establishes default values for display class, access level, and visibility to the Data Entity Builder for all parameters defined in the CDS. These values can be overridden by individual parameter attribute assignments (see heading 4.4.5).

#### 4.4.3.1 Custom Data Segment Heading Syntax



#### 4.4.3.2 Custom Data Segment Heading Description

The CLASS attribute defines the display class (General, PV\_Al, or Ctl\_Al) of the parameters in the Custom data definition. If no display class is named, the default is General. An access attribute specified for an individual parameter (see heading 4.4.5.3) will override the CDS heading's class attribute definition or default value.

The General, PV\_Al, and Ctl\_Al classes indicate the page of the Detail Display (of the point the CDS is attached to) on which the parameters are displayed, as follows:

<u>Class</u>	<u>Page of Display</u>
General	Custom Data Segment
PV_Al	PV Algorithm (PV Coefficients parameter group)
Ctl_Al	Control Algorithm (with Tuning parameters)

Note that no more than 28 items (PV coefficients and custom parameters) can appear in the PV coefficients-parameter group of the PV Algorithm page, even if more exist. Also, no more than 26 items (tuning parameters and custom parameters) can appear in the Tuning Parameters area of the Control Algorithm page, even if more exist; and, if the point's Detail Display has no PV Algorithm or Control Algorithm pages, the PV\_Al or Ctl\_Al parameters, respectively, are not displayed.

In addition, the following restrictions apply to the display of Custom parameters.

- For parameters in the General class that appear on the CDS page of the Detail Display, all parameter types are fully supported for display.
- For parameters in the PV\_Al class (PV Coefficients parameter group) or Ctl\_Al Class (Tuning Parameters parameter group), the degree of support varies according to the data type of the parameter(s) that you want to appear on the display, as follows:

<u>Data Type</u>	<u>How Supported</u>
Number	Fully
Enumeration	Fully
Logical	Fully
Time	Eight characters, time only—no date(duration expressed in Hours, Minutes, and Seconds)
String	Eight characters
Data point (Entity ID)	Fully
Arrays	Not supported

The ACCESS attribute restricts write-access to the parameters as follows: Operator, Supervisor, Engineer, and Entity\_Bldr access levels restrict access through the Universal Station to users with the appropriate keylock level. Program access grants write-permission to only user-written programs. If no access is named in the heading, the default is Engineer. An access attribute specified for an individual parameter (see heading 4.4.5.3) will override the CDS heading's access attribute definition or default value.

The BLD\_VISIBLE attribute displays the parameter to the engineer at CDS build-time. The engineer can enter a value for the parameter or change the value if one is entered through the VALUE statement. NOT BLD\_VISIBLE prevents display of the parameter at build-time. If not specified in the CDS heading, the default is BLD\_VISIBLE. Either BLD\_VISIBLE or NOT BLD\_VISIBLE specified for an individual parameter (see heading 4.4.5.4) will override the CDS heading's attribute definition or its default value.

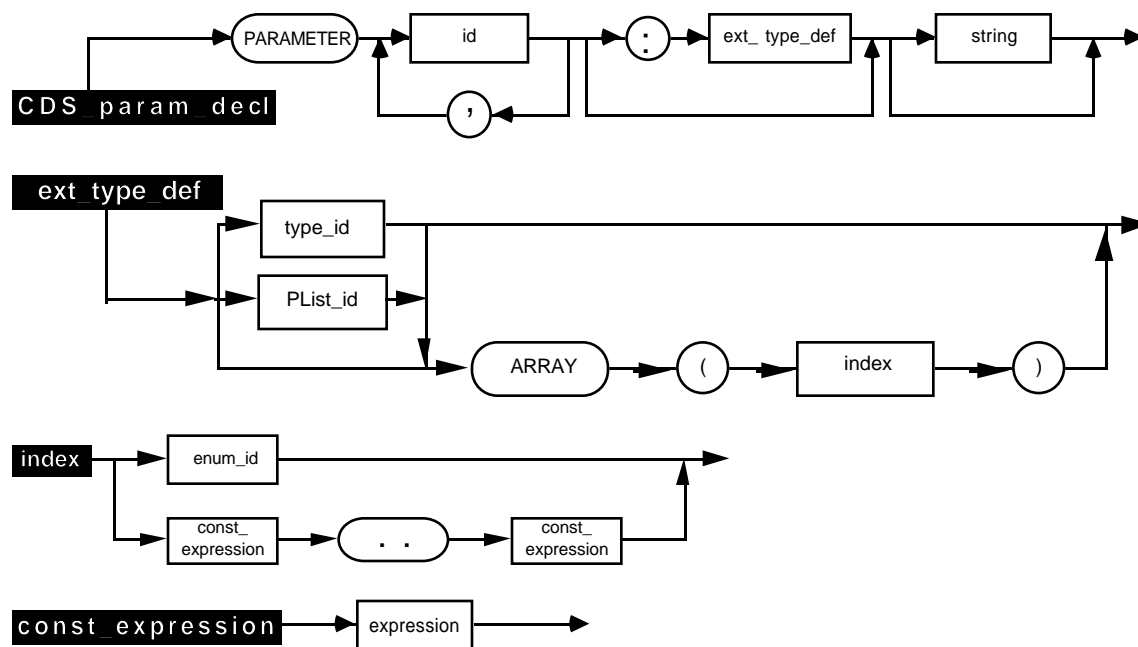
#### **4.4.3.3 Custom Data Segment Heading Examples**

```
CUSTOM (CLASS PV_Alg; ACCESS Operator)
CUSTOM (NOT BLD_VISIBLE)
CUSTOM (CLASS Ctl_Alg)
```

#### 4.4.4 Custom Data Segment Parameter Declaration

The parameter declaration identifies the parameter and its type, and optionally provides a description to be displayed by the Data Entity Builder.

##### 4.4.4.1 Custom Data Segment Parameter Declaration Syntax



##### 4.4.4.2 Custom Data Segment Parameter Declaration Description

The parameter ID must be unique within the defined Custom Data Segment. Type identifiers are optional; the default is Number. Enumeration parameters must have types that are built in, externally defined, or named in an ENUMERATION declaration. An enumeration-parameter statement in a block can be defined by specifying all possible states.

Arrays in Custom Data Segments are limited to one dimension.

When the parameter type is a Parameter List, it defines the parameter to be of “data point identifier” type. Thus, the value of the parameter can be the identifier of any data point. This data point is presumed by the compiler to possess parameters named in the Parameter List. This presumption is used to generate code for indirect reference through that parameter. At link time, the linker checks these indirect references to ensure that the actual parameter types are consistent with the assumptions made at compile time. No other checks are made at link time.

The optional String in the PARAMETER heading is displayed by the Data Entity Builder when the CDS is built. This provides a brief description of the parameter, which is useful at Build time.

#### 4.4.4.3 Custom Data Segment Parameter Declaration Examples

```

PARAMETER swdbd          "switch deadband value"
PARAMETER spec_hi: NUMBER
PARAMETER reverse: LOGICAL "ON = reverse contact closure"
PARAMETER message: String "message for batch 03"
PARAMETER batchtim: Time
PARAMETER switch1: dig_out "emergency switch" --a point ID
PARAMETER mode: Mode      --enumeration
PARAMETER pvvalues: ARRAY(1..3) --three numbers
PARAMETER joe, jim: NUMBER ARRAY(Mode) --array indexed by Mode

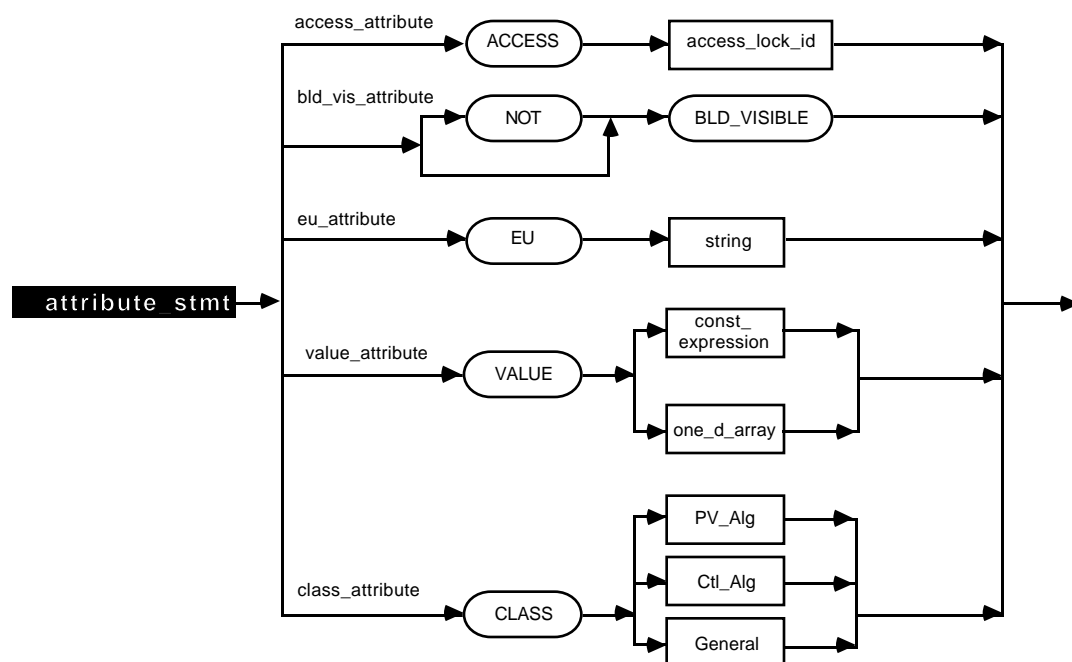
PARAMETER sam: ARRAY(1..10,1..10) -- INVALID, two dimensions
PARAMETER jill: red/blue          -- INVALID, no type ID

```

#### 4.4.5 Custom Data Segment Parameter Attribute Statement

Attribute statements are used in Custom Data Segment definitions to define the attributes of individual parameters. Attribute statements are ACCESS, BLD\_VISIBLE, EU, VALUE, and CLASS. For the attributes CLASS, BLD\_VISIBLE, and ACCESS, an individual parameter's attribute overrides the defined attribute or default in the CDS heading.

##### 4.4.5.1 Attribute Statement Syntax



#### 4.4.5.2 ATTRIBUTE Statement Description

The applicability of attribute statements to a parameter depends on the parameter's data type. This is shown in Table 4-1; OPT = Optional.

**Table 4-1 — Applicability of Attribute Statements**

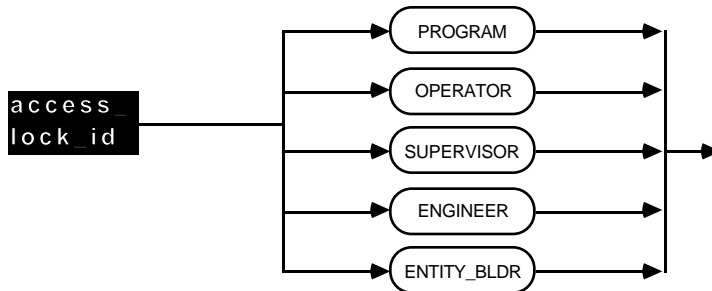
Attribute Statement					
Data Type	ACCESS	EU	BLD VISIBLE	VALUE	CLASS
Number	Opt	Opt	Opt	Opt	Opt
Time	Opt	Opt	Opt	N/A	Opt
Logical	Opt	Opt	Opt	Opt	Opt
Enumeration	Opt	Opt	Opt	Opt	Opt
String	Opt	Opt	Opt	Opt	Opt
Data Point Identifier	Opt	Opt	Opt	Opt	Opt

### 4.4.5.3 ACCESS Attribute

#### 4.4.5.3.1 ACCESS Attribute Definition

This attribute lists who can change the value of the parameter, and is valid for all parameter types.

#### 4.4.5.3.2 ACCESS Attribute Syntax



(See the *Application Module Parameter Reference Dictionary*, in the *Implementation/Application Module - 1* binder, and the *Computer Gateway Parameter Reference Dictionary*, in the *Implementation/CM60* binder for information on using the access\_lock.)

#### 4.4.5.3.3 ACCESS Attribute Description

This statement is optional; if omitted the default is to the header value.

See also heading 4.4.3.2.

#### 4.4.5.3.4 ACCESS Attribute Example

ACCESS Program



#### **4.4.5.4 BLD\_VISIBLE Attribute**

##### **4.4.5.4.1 BLD\_VISIBLE Attribute Definition**

This attribute specifies whether the parameter is visible at the Data Entity Builder (on the CRT screen form) when the Custom Data Segment is attached to a data point.

##### **4.4.5.4.2 BLD\_VISIBLE Attribute Syntax**

See heading 4.4.5.1.

##### **4.4.5.4.3 BLD\_VISIBLE Attribute Description**

BLD\_VISIBLE displays the parameter to the engineer at Custom Data Segment build-time. The engineer can enter a value for the parameter or change the value if one is entered through the VALUE statement.

NOT BLD\_VISIBLE does not display the parameter at build time; the parameter is completely defined by the attributes and defaults specified in its description.

This statement is optional; if omitted, the default is to the header value.

See also heading 4.4.3.2

##### **4.4.5.4.4 BLD\_VISIBLE Attribute Example**

```
NOT BLD_VISIBLE
```

#### **4.4.5.5 EU Attribute**

##### **4.4.5.5.1 EU Attribute Definition**

This statement names the parameter's engineering units.

##### **4.4.5.5.2 EU Attribute Syntax**

See heading 4.4.5.1.

##### **4.4.5.5.3 EU Attribute Description**

The EU String can be a maximum of eight characters long.

This statement is optional, and if omitted, defaults to blanks.

##### **4.4.5.5.4 EU Attribute Examples**

```
EU "pounds"
EU "gallons"
EU "days"
```

#### 4.4.5.6 VALUE Attribute

##### 4.4.5.6.1 VALUE Attribute Definition

This attribute specifies a default initial value for the parameter. When the Custom Data Segment is built by the Data Entity Builder and attached to a specific point, this value becomes the value of the parameter, unless the engineer selects a different value.

##### 4.4.5.6.2 VALUE Attribute Syntax

See heading 4.4.5.1.

##### 4.4.5.6.3 VALUE Attribute Description

The data type of the constant expression must match the parameter type; for example, Logical parameters must have Logical values (OFF or ON). Elements of an array of data point name can be assigned the value "null data point name" using \$NULLPT.

A 1-dimensional data array is used to set the initial values for an array parameter. You also can use a single constant expression to initialize all the elements of the array to the same value.

The VALUE statement is optional and, if omitted, one of two actions occurs at Custom Data Segment Build-time:

1. If the BLD\_VISIBLE attribute is specified, the parameter is displayed for data entry, with the default value as shown in Table 4-2.
2. If the parameter is NOT BLD\_VISIBLE, it is set to a default value, depending on its data type, as shown in Table 4-2.

**Table 4-2 — Default Parameter Values**

Data Type	Default Value
Number	Bad value
Logical	Off
data point name	null data point name
String	value = 1 (space or blank) (length 1)
Time	0 SECS
enumeration	the first named state

#### 4.4.5.6.4 VALUE Attribute Examples

```

VALUE 10.1
VALUE 2*3          -- a constant expression
VALUE OFF
VALUE calculus     -- a discrete state name
VALUE "this is a String"
VALUE AX100         -- data point identifier
VALUE (22.1, 33.2, 44.3) -- a 3-element array
VALUE 50           -- initializes all elements of the array to 50
VALUE (AX100,$NULLPT) -- Second element of the 2-element array is
                     -- the null datapoint name
VALUE ab\Z444      -- off-LCN data point identifier
VALUE (fe\A100,fe\B100,fe\ $nullpt) -- 3-element array, where the
                                     -- third element is an off-LCN
                                     -- null point identifier

```

#### 4.4.5.7 CLASS Attribute

##### 4.4.5.7.1 CLASS Attribute Definition

This attribute specifies the display class of the parameter.

##### 4.4.5.7.2 CLASS Attribute Syntax

See heading 4.4.5.1.

##### 4.4.5.7.3 CLASS Attribute Description

This statement is optional. If omitted, the parameter's display class defaults to the header value.

See also heading 4.4.3.2.

##### 4.4.5.7.4 CLASS Attribute Example

```

CLASS PV_Alg

```

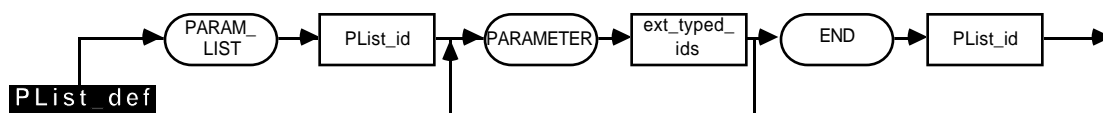
## 4.5 PARAMETER LIST DEFINITION

A parameter list is a list of the names and types that the parameters of a data point can have. At compile time, the compiler can refer to a parameter list to get a description of the referenced data points; as such, the parameter list serves as a surrogate for the on-process database (because the database may not yet be fully described).

With R530, the number of parameter names has increased from 2000 to 4000 to allow for a greater capacity. The load time for loading the parameter names has also been optimized with R530; the load time will be significantly less than the load time required before adding the additional 2000 CDS parameter names.

A Parameter List definition consists of a Parameter List heading, followed by one or more parameter definitions, and closed by an END statement.

### 4.5.1 Parameter List Syntax



### 4.5.2 Parameter List Description

The PList\_ID is an identifier by which the Parameter List is to be externally known.

The parameter declaration is exactly as in a Custom Data Segment, but without the descriptive String. Note that the type must be named; it cannot be a state list as in parameter declarations in a block. The PList\_ID in the END statement must match that in the heading.

See heading B.3 for contents of the prebuilt parameter list \$REG\_CTL.

### 4.5.3 Parameter List Examples

```

PARAM_LIST my_PList
  PARAMETER PV, SP: Number
  PARAMETER OP                      -- Number by default
  PARAMETER Specvals: ARRAY(1..12)
  PARAMETER Backup: my_PList
END my_PList
  
```

This parameter list defines a point that has numeric PV, SP, and OP parameters, a numeric array-parameter Specvals, and a parameter backup that is a data point identifier. Any actual point whose identifier is stored in the parameter backup must possess all the parameters declared in the parameter list my\_PList, that is, this parameter list itself.

If the bound data point has this parameter list, the following references are valid:

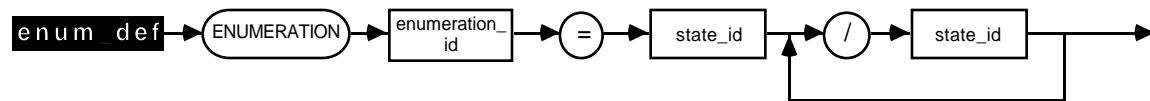
```

BLOCK xyz (GENERIC My_PList; AT Pre_GI)
  SET PV = SP
  SET OP = Backup.PV
  SET Backup.Backup.SP = MIN(Specvals)
  
```

## 4.6 ENUMERATION-TYPE DEFINITION

An Enumeration-type definition lists all the states that a discrete variable can take on; e.g., On/Off/In\_Between, Red/Blue/Green/Amber, or Open/Shut. An Enumeration description defines an Enumeration type, listing each state that can be assumed by a variable of that type.

### 4.6.1 Enumeration-Type Definition Syntax



### 4.6.2 Enumeration-Type Definition Description

The Enumeration identifier defines the name by which the Enumeration is to be externally known. The identifiers on the right-hand side of the equal sign are the state names. The order in which the state names are given defines the order of their internal representation. The Enumeration definition is entered into the Universal Station's Custom Name Library under the name given.

The database can contain multiple Enumeration definitions that have the same state names. Such types are compatible if the order of their states is the same, but incompatible if their state order differs. Enumeration variables that are compatible can be assigned to each other and compared to each other.

### 4.6.3 Enumeration-Type Definition Examples

```

ENUMERATION course = calculus/biology/data_str
ENUMERATION lampcolr = red/amber/green
ENUMERATION crtcolor = red/yellow/blue/cyan/green/magenta
ENUMERATION swstate = red/yellow/off
  
```

Note that the identifier **red** is used in three of the examples, and **green** and **yellow** are each in two. **Off** appears in one example, although it is also a member of the built-in discrete-type Logical. There is no conflict in these examples: Any nonreserved identifier, predefined or user-defined, can be used in as many Enumeration types as desired.

```

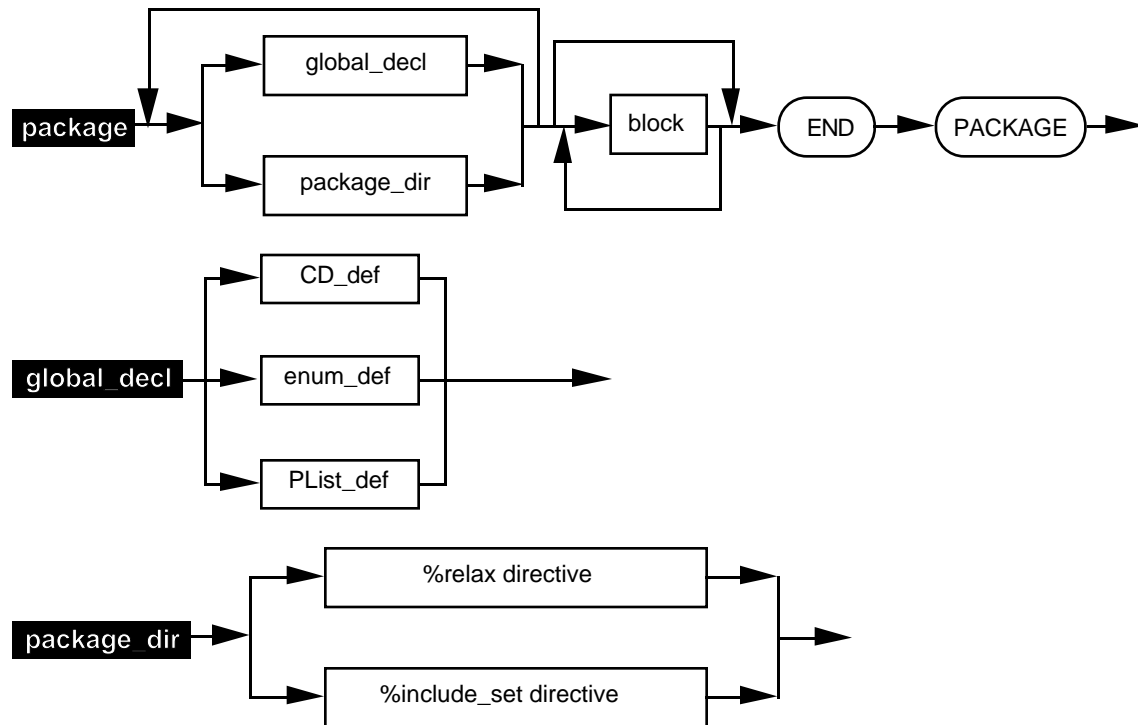
ENUMERATION quark = up/down/sideways
ENUMERATION directn = up/down/sideways
ENUMERATION posn = sideways/up/down
  
```

In these examples, **quark** and **directn** are compatible types, but **posn** is not compatible with either one.

## 4.7 PACKAGE DEFINITION

A package is a related set of CL programs (blocks) and Global Data definitions, intended to be applied to a single data point as part of a single control strategy.

### 4.7.1 Package Syntax



### 4.7.2 Package Description

Packages can be composed of zero or more Global Data definitions (however, only one Custom Data definition can appear in a package), followed by zero or more programs. A package cannot be empty; it must contain at least one Global Data definition or program.

All types of Global Data definitions (Parameter List definitions, Enumeration-type definitions, and one Custom-data definition), CL blocks, and subroutines are allowed in packages. If more than one Global Data definition and/or block is to be compiled in one file, a package is required.

Programs in a package have enhanced visibility to all Custom Data Segment parameters declared in the same package; see heading 2.4.5.6, Parameter References in Packages, for further details.

All programs in a given package form a single object file and can be bound to a data point in a single operation. Individual Blocks cannot be extracted from the compiled package and independently linked.

### 4.7.3 Package Examples

The following example shows two closely related Blocks. They are grouped in a package to ensure that they are always applied together to a data point.

```

PACKAGE

    BLOCK one (GENERIC; AT Pre_PVAg;
&          WHEN pv > thresh)
        ...
    END one

    BLOCK two (GENERIC; AT Pre_PVAg;
&          WHEN pv <= thresh)
        ...
    END two

END PACKAGE

```

The following example shows a set of related data declarations. They are grouped in a package strictly for convenience.

```

PACKAGE

    ENUMERATION traflite = green/amber/red

    PARAM_LIST lite_pt
        PARAMETER PV, OP: traflite
        PARAMETER          l_back,
&          l_ahed: lite_pt
        PARAMETER cycles: Logical ARRAY (traflite)
    END lite_pt

    CUSTOM
        PARAMETER          l_back,
&          l_ahed: lite_pt
        ACCESS PROGRAM
        PARAMETER cycles: Logical ARRAY (traflite)
        VALUE (ON, ON, OFF)
        PARAMETER all_red: Logical
    END CUSTOM

END PACKAGE

```

The following example consists of a set of related CL blocks and a Custom Data Segment used by each. They are grouped as a package for the reasons given in the first two examples and to reduce the number of PARAMETER declarations in the CL blocks.

```

PACKAGE

  CUSTOM
    PARAMETER counter: Number
    ...
  END CUSTOM

  BLOCK myPValg (GENERIC; AT PV_alg)
    SET counter = counter + 1
    ...
  END myPValg
  BLOCK myctlalg (GENERIC; AT Ctl_Alg)
    SET counter = counter - 1
    ...
  END myctlalg

END PACKAGE

```

## 4.8 CL RUNTIME EXTENSIONS

Applications oriented subroutines and functions that are callable by CL/AM programs can be purchased from Honeywell and added to your system. These subroutines and functions are known as CL Runtime Extensions and are packaged in files called Include Sets. These optional files are loaded into the AM during system startup and configuration.

Each CL/AM program that will use runtime extensions must contain one or more %INCLUDE\_SET directives (see heading 3.3.5) to identify each Include Set file that contains subroutines and/or functions to be used by that program.

Information on available Include Sets and the subroutines and functions that they contain is available with the optional product or package.

When calling a CL Runtime Extension function or subroutine, the function or subroutine name must be preceded by the CL Runtime Extension file name and a "\$". For example:

```

BLOCK example (GENERIC; AT GENERAL)
  EXTERNAL A100
  %INCLUDE_SET modify -- file name containing definitions of CL
                    -- Runtime Extension subroutines/functions

  LOCAL a, b
  SET a = 1
  SET b = modify$INCREMENT_REAL(a) -- call of extension function named
  SEND: a,b                       -- INCREMENT_REAL in file "modify"
END example

```



## 4.9 EXAMPLES OF CL/AM STRUCTURES

Each of the sample CL/AM compilations in this section contains comments that describe what the program is doing (comments start with --).

### 4.9.1 PV Calculation Example

```

BLOCK calc_pv (GENERIC;
&                AT pv_alg)
--
-- This routine is translated from a BPL example.
--
LOCAL y
PARAMETER pv
PARAMETER dvalsrc      -- the delayed value source
PARAMETER inmove       -- the incremental move value
PARAMETER pvlolim      -- the low limit
PARAMETER pvhilim      -- the high limit
PARAMETER pveulo, pveuhi -- E.U. limits
--
EXTERNAL ANC000HDW
--
IF dvalsrc NOT IN pvlolim .. pvhilim THEN SET inmove = -inmove
SET dvalsrc = dvalsrc + inmove
SET y = (pv - pveulo) * (ANC000HDW.pveuhi - ANC000HDW.pveulo)
&                / (pveuhi - pveulo)
SET ANC000HDW.pv = y + ANC000HDW.pveulo
--
END calc_pv

```

## 4.9.2 Linearization Example

```

BLOCK linear (GENERIC;
&                AT pv_alg)
--
--Linearization program for an oxygen sensor,
--translated from a BPL example.
--
  PARAMETER PVAUTOST: PVVALST
  PARAMETER xsair, k1, k2, pvcalc
  LOCAL j1 = 0.25      -- combustion proportion constant
  LOCAL j2, c1, m1, m2, m3
  EXTERNAL al3ar03
--
--  linearization equation
--
  SET pvcalc = 10 ** ((k1 - al3ar03.pv) / k2)
  SET pvautost = normal
--
--  calculate excess air for burner
--
  SET j2 = 990 * j1 / (1 + j1)
  SET c1 = 990 / (1 + j1)
  SET m1 = j2 / 4.216
  SET m2 = c1 / 11.012
  SET m3 = m1 * 2.11 + m2 * 6.44
  SET xsair = 100 * pv * (m2 / m3) / (21.3 - pv)
--
END linear

```

### 4.9.3 Custom Data Segment Example

The following example of a Custom Data Segment definition (descriptor) includes one parameter of each data type.

```

CUSTOM

PARAMETER SWDBD    "switch deadband value"
--number by default
ACCESS Engineer
EU    "psi"
VALUE 0.5

NOT BLD_VISIBLE

PARAMETER Reverse: logical
-- use the default access
VALUE OFF
NOT BLD_VISIBLE

PARAMETER CtlValve: logic_pt

-- Logic_pt is a Parameter List
ACCESS Engineer
-- no value specified

PARAMETER Err_Mesg: STRING
ACCESS engineer
-- no value specified

PARAMETER snapshot: TIME
ACCESS Engineer
-- no value specified

PARAMETER Fuel: gas_oil
-- an Enumeration parameter
VALUE Oil
ACCESS Engineer

PARAMETER PVO_vals:  NUMBER ARRAY (1..3)
ACCESS Engineer
EU "V/Meter"
VALUE (21, 23, 27)

END CUSTOM

```

### 4.9.4 BTU Switch Package Example

```

PACKAGE
--
--   The objective of this package is to duplicate several existing
--   TDC 2000 algorithms using TDC 3000X facilities.
--
--   ENUMERATION fuels = gas/oil
--   ENUMERATION selsppv = s_p/p_v
-----
CUSTOM (CLASS general ; ACCESS Engineer)
--
PARAMETER swch_pos: fuels "gas or oil"
    ACCESS engineer
    VALUE gas
PARAMETER swch_req: fuels "requested switch position"
    ACCESS operator
    VALUE gas

PARAMETER Gaspnt : $Reg_ctl "Gas fuel flow data point"
PARAMETER Oilpnt : $Reg_ctl "Oil fuel flow data point"
PARAMETER COT : $Reg_ctl "Coil outlet temperature data point"
PARAMETER Bad_swch : Logical
    NOT BLD_VISIBLE
    VALUE Off
PARAMETER Bad_ctl : Logical
    NOT BLD_VISIBLE

    VALUE Off
-----
PARAMETER HG "Heat value of fuel gas"
    EU "KCAL/KG"
    VALUE 12593
PARAMETER HO "Heat value of fuel oil"
    EU "KCAL/KG"
    VALUE 9723
PARAMETER CG "Scaling constant for gas"
    VALUE 0.001
PARAMETER CO "Scaling constant for oil"
    VALUE 0.001
PARAMETER PVGAS "Fuel gas flow"
    NOT BLD_VISIBLE
    ACCESS program
    EU "KG/HR"
PARAMETER PVOIL "Fuel oil flow"
    NOT BLD_VISIBLE
    ACCESS program
    EU "KG/HR"
PARAMETER PAR_SEL : selsppv "selects sp or pv"
--
END CUSTOM

```

```

BLOCK Btu_swch (GENERIC $REG_CTL; AT Pre_PVAg;
& WHEN swch_pos <> swch_req)
--
--This routine performs the functions of PMX Switch algo No. 31 (BTU
--2/3). This block runs when a switch position change is requested.
--The appropriate control path is selected (gas or oil). Status and
--mode of the secondary are checked, as well as the mode of the
--alternate secondary.
--
--If these are not proper, the bad switch exit is taken. If OK, the
--new control configuration is set up and the temperature control
--primary is set up for initialization. Finally, the current switch
--position is set to the requested switch position.
--
--
--      If swch_req = oil THEN GOTO gas_to_oil
--
--Change from oil to gas. Check current modes and status.
--
--      IF oilpnt.ptexecst      <> active
&      OR oilpnt.mode          <> cas
&      OR gaspnt.mode          <> Auto THEN GOTO badsw
--
--If OK, disable Oil SP control.
--      First, set oil output status to inactive.
--
--      SET coactsts(2)                = inactive
--      SET oilpnt.mode                = auto
--
--Set up Gas SP control.
--      First, set gas output status to active.
--
--      SET coactsts(1)                = active
--      SET gaspnt.mode                = cas
--
--Set up primary initialization & close switch.
--
--      SET gaspnt.ctrlinit            = on
--      SET COT.ctrlinit              = on
--      SET swch_pos                  = swch_req
--      EXIT
--
--      Change from gas to oil. Check current modes and status.
--
gas_to_oil : IF gaspnt.ptexecst      <> active
&      OR gaspnt.mode                <> cas
&      OR oilpnt.mode                <> Auto THEN GOTO badsw
--
--If OK, disable gas SP control.
--      First, set gas output status to inactive.
--

```

```

        SET coactsts(1)                = inactive
        SET gaspnt.mode                 = auto
--
-- Set up Gas SP control.
-- First, set oil output status to active.
--
        SET coactsts(2)                = active
        SET oilpnt.mode                 = cas
        SET oilpnt.ctrlinit             = on
--
-- Set up primary initialization & open switch.
--
        SET COT.ctrlinit               = on
--
--
        SET swch_pos                   = swch_req
--
        EXIT
--
-- Bad switch exit.
--
--
badsw:      SET bad_swch = on
            END Btu_swch
--
-----
--
        BLOCK BTUswPV (GENERIC $REG_CTL; AT pv_alg)
--
-- This routine performs the functions of PMX PV algorithm No. 102.
-- (HTCALC)
-- Calculate the total heat input to the furnace.
-- First, test input variables.
--
        IF (oilpnt.pvautost<>normal)or(gaspnt.pvautost<>normal)
&          THEN EXIT
--
-- Now solve for the total heat input.
--
--
        SET PVCALC = HG*CG*GASPNT.PV+ HO*CO*OILPNT.PV
        SET PVAUTOST=NORMAL
        END BtuswPV

```

```

BLOCK BtuSwCtl (GENERIC $REG_CTL; AT ctl_alg)
  LOCAL PVGASC,PVOILC -- Only need for intermediate calcs.
--
--This routine performs the functions of PMX control algo No. 57.
--(BTU 2/3)
--
--Note: Control has separate inputs for gas and oil flow PVs. This
--is done so that the control inputs can optionally be connected to
--the respective setpoints, thus avoiding flow PV noise effects.
--The selection is made at point build.
--
--Selects the fuels setpoints or PVs for control input. Switch is
--set at configuration.
--
  SET Bad_ctl = off
  IF par_sel = s_p THEN GOTO sp_sel
  ELSE IF par_sel = p_v THEN GOTO pv_sel
  ELSE SET bad_ctl = on
  EXIT
--
--Selects control from the fuel flow SPs.
sp_sel:   SET PVGASC = gaspnt.sp
         SET PVOILC = oilpnt.sp
         GOTO swtest
--
--Selects control from the fuel flow PVs.
pv_sel:   SET PVGASC = gaspnt.pv
         SET PVOILC = oilpnt.pv
--
--Test which fuel is selected for control.
--
swtest:   IF swch_pos = oil THEN GOTO ctloil -- Switch open
         ELSE GOTO ctlgas
--
--
--Oil selected for control. Test for initialization.
--
ctloil:   IF pathind = fwd THEN GOTO oilcal
--
--Initialization required.
--
  SET SP = PVOILC*HO*CO + PVGASC*HG*CG
--
oilcal:   SET CV = (SP - HG*CG*PVGASC)/(HO*CO)
         EXIT
--
--Gas selected for control. Test for initialization.
--
ctlgas:   IF pathind = fwd THEN GOTO gascal
--

```

```
-- Initialization required.  
--  
--      SET SP = PVOILC*HO*CO + PVGASC*HG*CG  
--  
gascal:      SET CV = (SP - HO*CO*PVOILC)/(HG*CG)  
--  
      END BtuSwCtl  
END PACKAGE
```



## 4.10 FILE MANAGER STATUS RETURN VALUES

The following is a list of all possible File Manager Status values returned from file access calls.

### NOTE

You can make file access calls by using either built-in subroutines (see heading 4.3.7) or the File I/O Extensions (see Appendix C).

- 0 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 1 — The user program attempted to read or write beyond the end of the file. This error can occur when using calls in the File I/O extension; however, it should never be returned with the built-in CDS\_Read or CDS\_Write subroutines. If it does, report the error as a problem to Honeywell TAC.
- 2 — A response from a remote node to a file manager request has not been received in the allotted time. This can happen when an HM goes down. One retry has been performed at a lower level before this return.
- 3 — The HM reported a physical error; there probably is a bad spot on the HM.
- 4 — The file may be corrupted; delete the file.
- 5 — The volume may be corrupted.
- 6 — Device type not compatible with the command (for example, read from printer).
- 7 — The HM driver reported a hardware time-out error.
- 8 — An illegal or unused command was specified. Report this problem to Honeywell TAC.
- 9 — The logical resource number used in the file manager call was invalid. Report this problem to Honeywell TAC.
- 10 — The number of concurrent open files is at its maximum. Ten retries have been performed at a lower level before this return.
- 11 — The number of concurrent open files is at its maximum. Ten retries have been performed at a lower level before this return.
- 12 — The logical resource number used in the file manager call is currently marked as unused (for example, closing a logical resource number that was not in use). Report this problem to Honeywell TAC.
- 13 — The file is currently reserved for some other operation (for example, attempting to open a file on a device which is being initialized).

- 14 — The file is currently reserved for some other operation (for example, attempting to open a file on a device which is being initialized).
- 15 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 16 — Conflicting options have been selected in the File Manager request (for example, contiguous file opened by the File I/O extension subroutine call File\$Open).
- 17 — The file name or file extension used in a request is invalid.
- 18 — An effort was made to install a file in a directory which already contains a file with the same name.
- 19 — The specified file name was not found in the directory.
- 20 — The physical device associated with the virtual volume or virtual device could not be found. This can occur when a device is not included in the system generation.
- 21 — The request is prohibited, due either to a file protection attributes conflict or to a file access privilege violation. Possible on CDS read/write or File\$Open\_File.
- 22 — Invalid buffer length. The buffer length specified is negative, zero, or greater than the maximum message size allowed. Report this problem to Honeywell TAC.
- 23 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 24 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 25 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 26 — Insufficient space available on the medium to perform the request.
- 27 — There is insufficient HM memory available to perform the request. This can occasionally happen; however, 10 retries have been performed at a lower level before this return. Thus, if this problem occurs frequently, it should be reported to Honeywell TAC. If the status resulted from a CL file I/O extension call, see the Note in subsection C.2.3 concerning AM memory allocation.
- 28 — An attempt to create a new file failed because it would have exceeded the maximum number of files per volume.
- 29 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 30 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 31 — The record being read is larger than the maximum record size.

- 32 — The volume (or device) requested does not contain initialized media.
- 33 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 34 — An operation requiring that the file be closed was attempted on a file which was open (for example, deletion of an open file).
- 35 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 36 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 37 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 38 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 39 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 40 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 41 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 42 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 43 — Size conflict during write. Either the record could not fit into the data block or the number of units specified to write is zero. Report this problem to Honeywell TAC.
- 44 — Variable length record on a write with a length of zero (0) or an attempt to rewrite a variable length record with the incorrect size.
- 45 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 46 — An operation requiring the file to be open was attempted on a closed file.
- 47 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 48 — HM has failed.
- 49 — Invalid buffer address. Report this problem to Honeywell TAC.

- 50 — The files specified in the Safe\_Rename subroutine call are not on the same volume.
- 51 — Not an error; the request completed successfully.
- 52 — The volume name specified in the File Manager call is syntactically incorrect.
- 53 — The volume specified in the pathname was not found.
- 54 — A request was made to access a volume that existed on two different nodes.
- 55 — An attempt was made to define a volume alias on a specified volume and that alias name was already defined.
- 56 — An attempt was made to remove a volume alias on a specified volume and that alias name was not found.
- 57 — The logical device identifier used in a request could not be found.
- 58 — The request attempted to define a logical device identifier which was already defined.
- 59 — The local physical node has insufficient memory available to satisfy the request. If this happens frequently, Honeywell TAC should be notified.
- 60 — The local physical node has insufficient memory available to satisfy the request. This request is retried 10 times at a lower level before the return. If this happens frequently, Honeywell TAC should be notified.
- 61 — Invalid number of words (greater than 32757) or invalid address specified in a call.
- 62 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 63 — The attributes of the source and destination file are not compatible for a given copy operation (for example, linked vs contiguous).
- 64 — The logical resource number specified was previously cancelled. The caller must close or release this logical resource number.
- 65 — The logical resource number specified was marked failed due to a remote node failure. The caller must close or release this logical resource number.
- 66 — The logical resource number (LRN) to be deallocated was NOT obtained with its reciprocative call. An LRN obtained by an Open call must be deallocated using the Close call. An LRN obtained by an Assign LRN call must be deallocated using the Release LRN call. An LRN obtained by using a Read Directory Entry call must be deallocated using a Read Directory Entry Complete call.
- 67 — The space reserved for defining Logical Device IDs has been exhausted.
- 68 — The logical or virtual device ID specified is not syntactically correct.
- 69 — The CRB Identifier is not valid.

- 70 — The call or option on the call is not supported on the personality for which the request was made. For example, trying to create a volume on the Winchester disk from an operator station.
- 71 — The directory attributes of the source file of a copy operation have been corrupted.
- 72 — An attempt was made to assign a virtual volume alias to a volume that has already been assigned the maximum number (63) of alias volume names.
- 73 — The total number of tracks specified in creating virtual devices has exceeded the maximum number of tracks on the physical device.
- 74 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 75 — The call accessed a failed device.
- 76 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 77 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 78 — The volume currently is being synchronized. This request is retried 10 times before the return. If it occurs frequently, it should be reported to Honeywell TAC.
- 79 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 80 — This error value should never be returned. If it occurs, report the error as a problem to Honeywell TAC.
- 81 — The subdirectory associated with this alias name is not empty. To delete an alias name there must be no files left in the directory. Delete all files and then retry the Delete Alias.
- 82 — A redundant operation was attempted on a device that is not redundant.
- 83 — The redundant device is not available.
- 84 — The Set Device State transition was not valid. For example, is not valid to set a disk off-line if it is already off-line. Check the current device state.
- 85 — A File Manager operation requiring operational software was attempted, but the optional software was not purchased.
- 86 — A device address specified is out of range.

## 4.11 CDS Request Return Status Values

The status returned by each of the built-in subroutines CDS\_Read, CDS\_Write, and Delete\_File is a subset of the standard enumeration \$CLFSTAT. Following is the complete set of values in this enumeration set organized in order of their ordinal values:

<u>Ordinal Value</u>	<u>ASCII Value</u>
0	OK
1	BADPATH
2	BADFILE
3	MISMATCH
4	NOCDS
5	CORRUPT
6	FMPVIOL
7	FMEXIST
8	FMNOFIL
9	FMHARD
10	FMFILES
11	FMSPACE
12	FMNOVOL
13	FMMOUNT
14	FMVFULL
15	FMOTHER
16	RELINK
17	SPARE01
18	SPARE02

This ordering information is useful if you need to send an operator message that contains this status information. Unless the optional File I/O Extension is present, CL/AM does not have the ability to send the actual state name; however, you can convert the state name to its ordinal value—using the built-in function ORD( )—and include this value in an operator message.

## CL/AM SYNTAX SUMMARY

### Appendix A

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

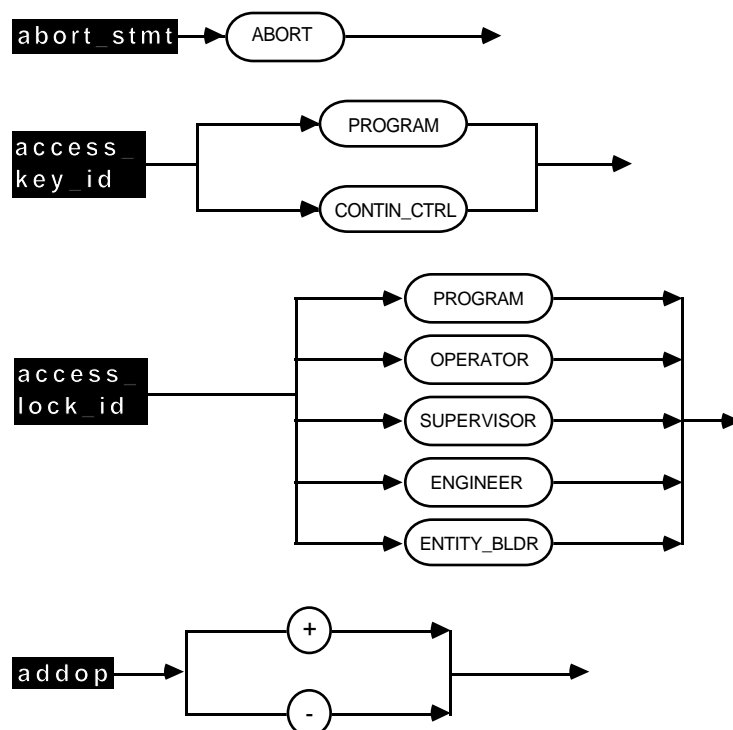
#### A.1 SYNTAX (GRAMMAR) SUMMARY

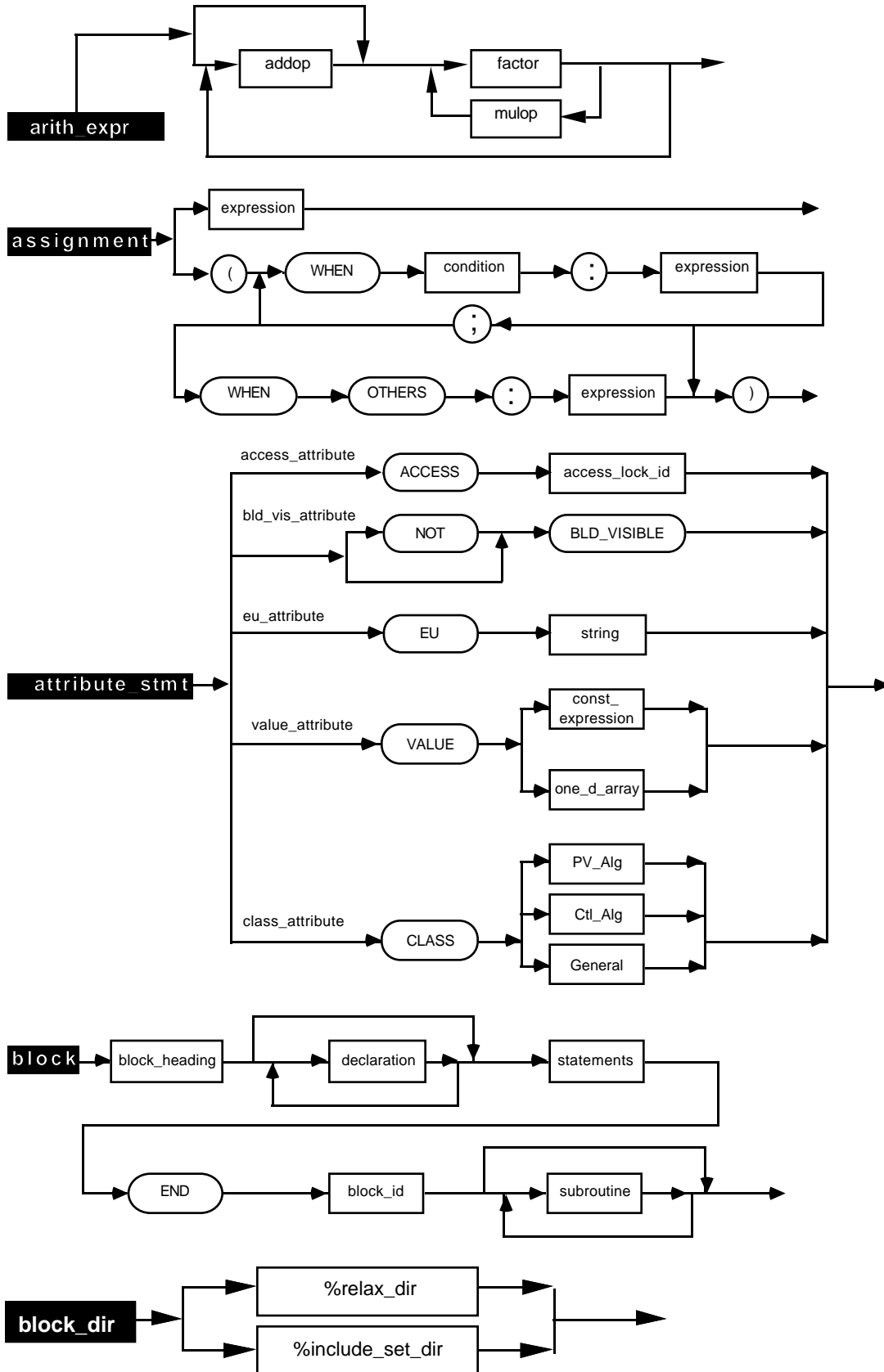
This section is divided into two parts. The first part is a summary of CL/AM syntax in the form of all the syntax diagrams that were presented throughout the manual. Each syntax diagram is labeled (in reverse-video), and they are arranged in alphabetical order.

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

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

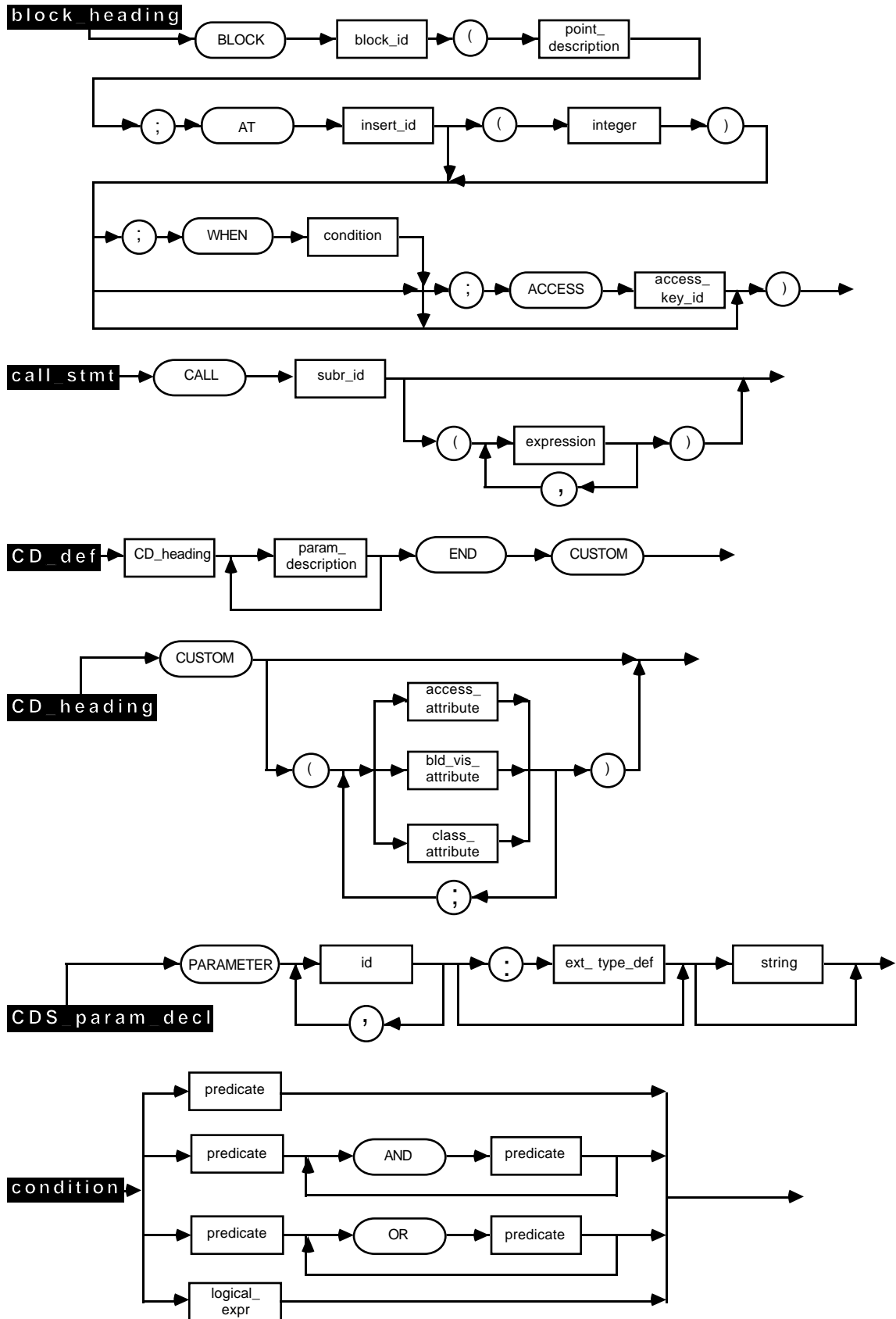
#### A.2 SYNTAX DIAGRAM SUMMARY

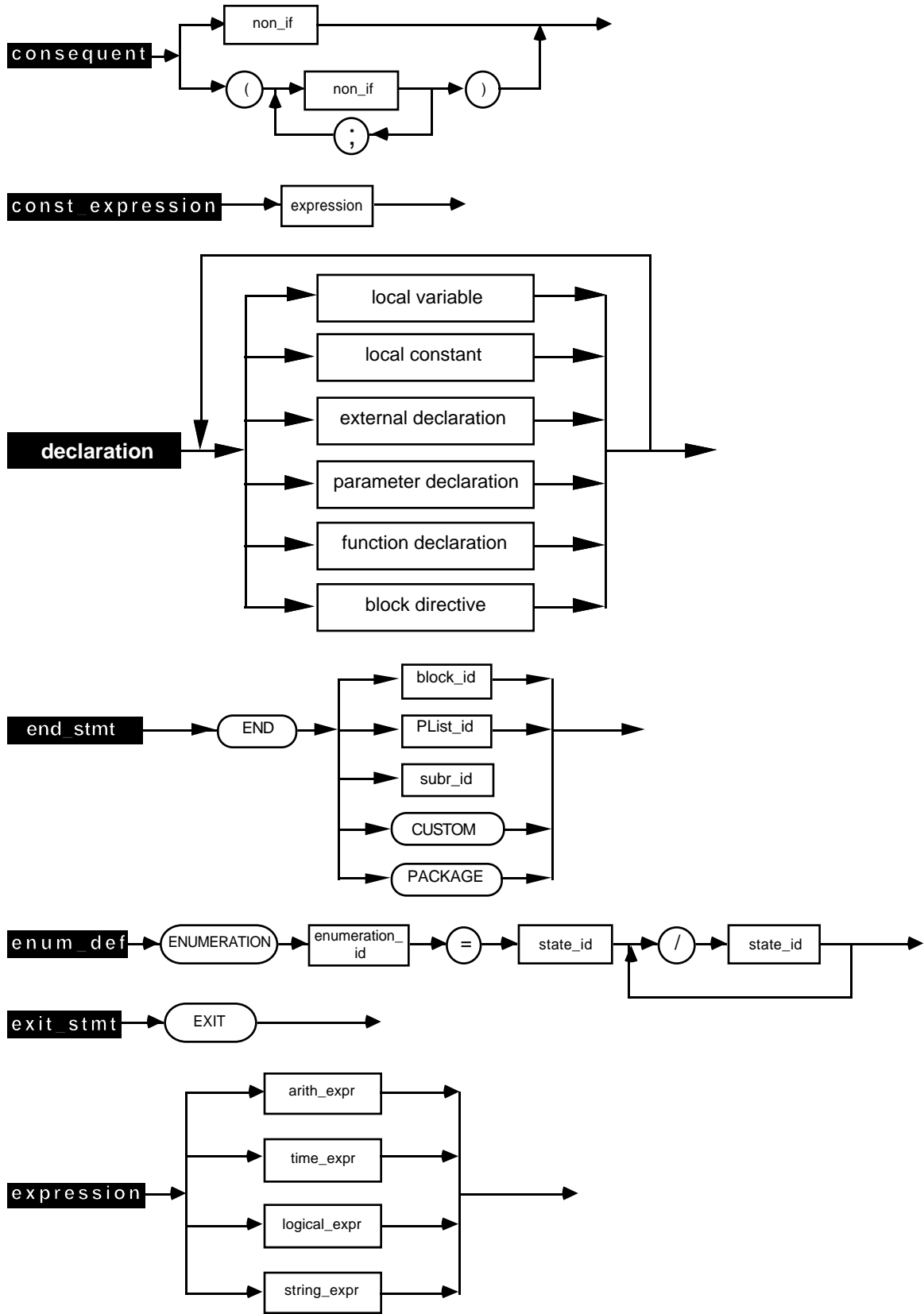


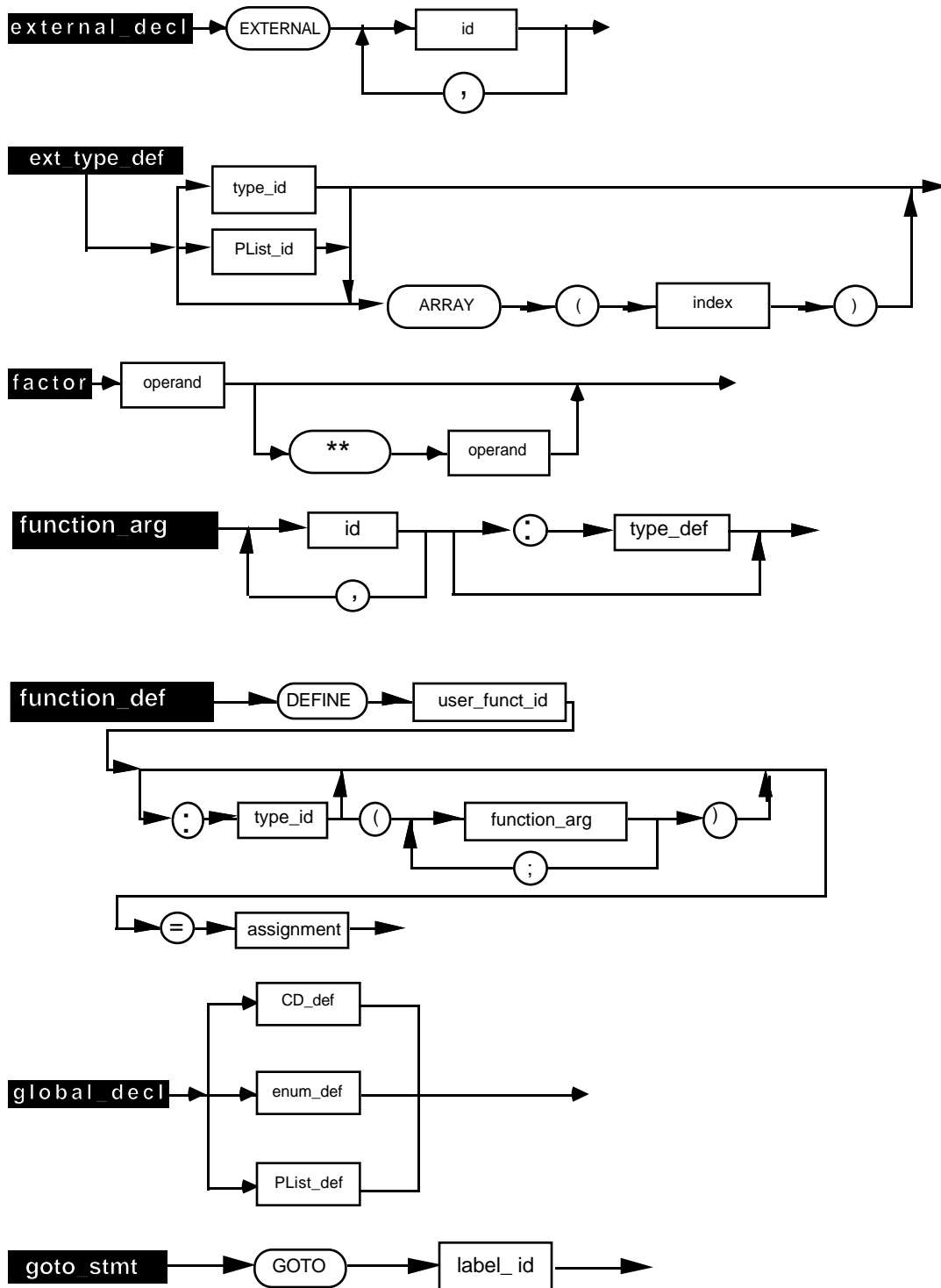


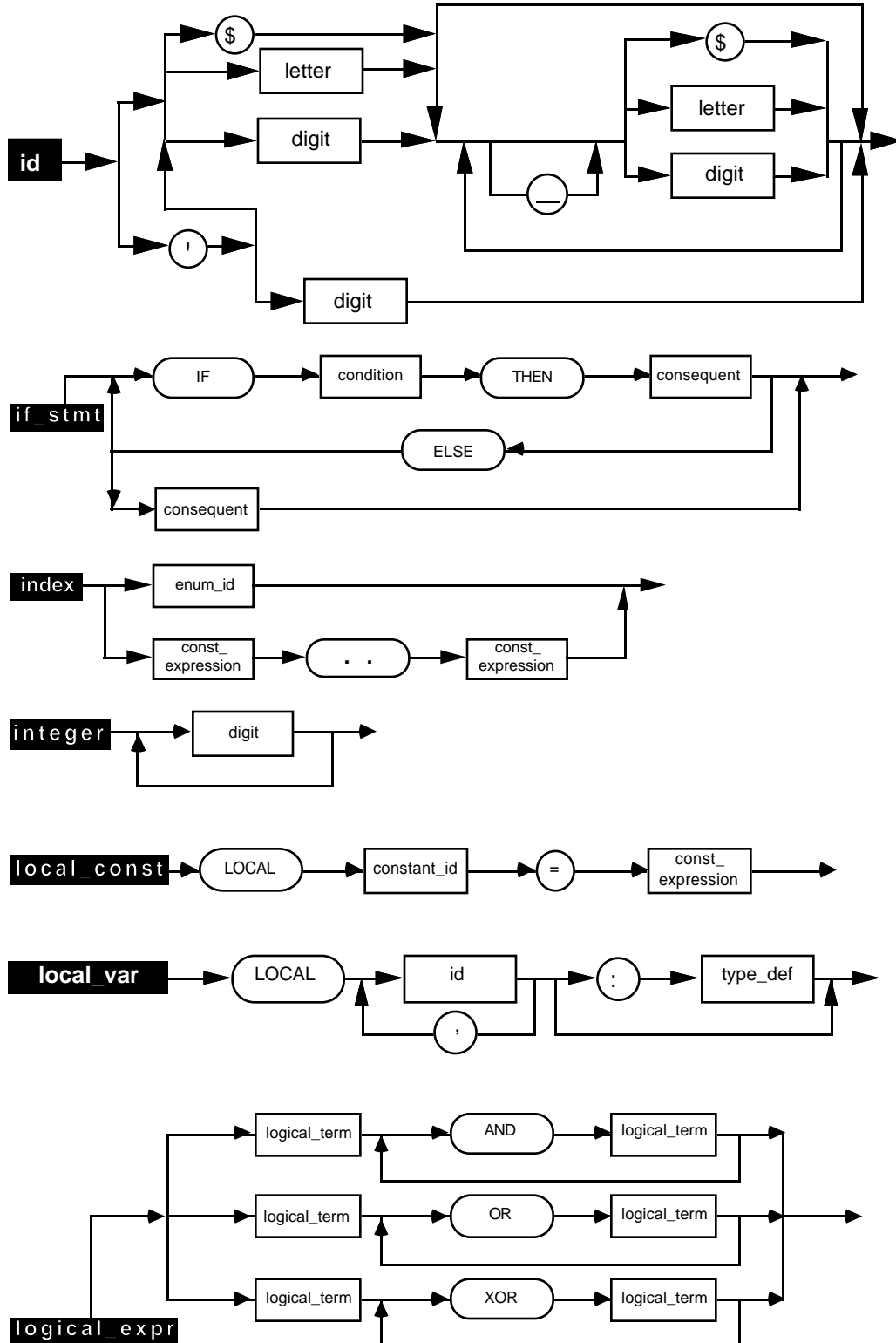


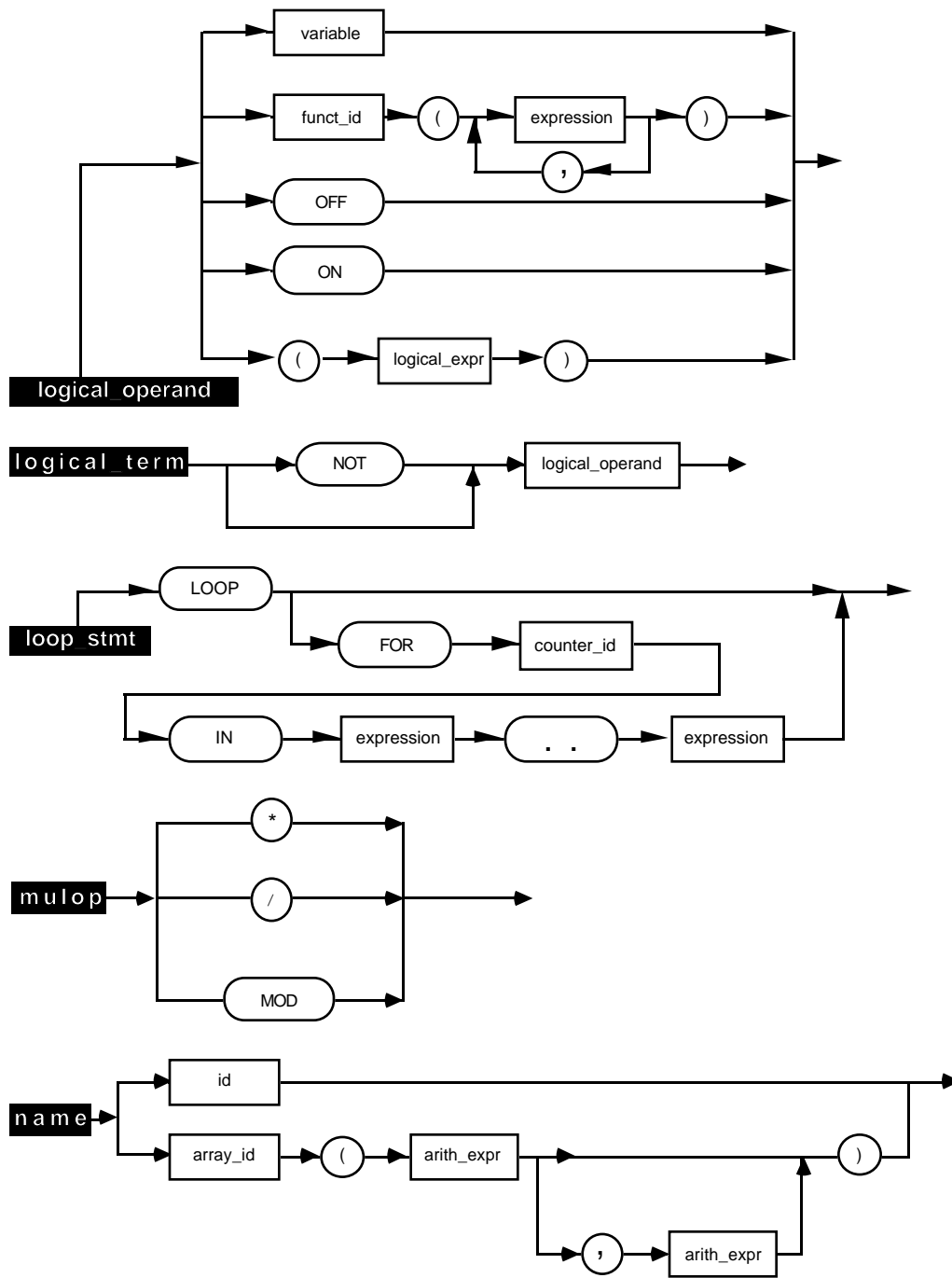
## A.2

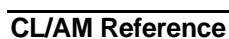


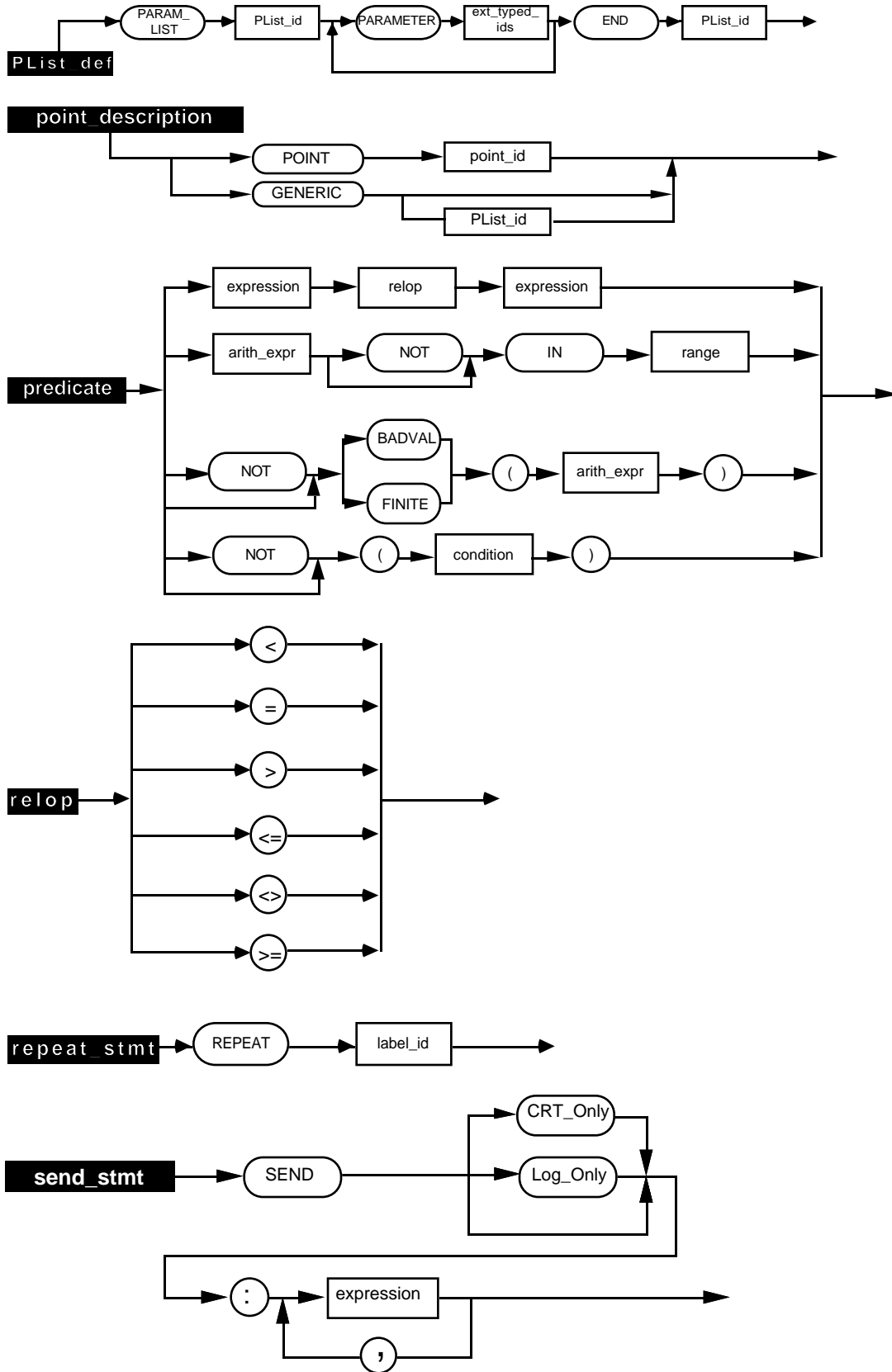


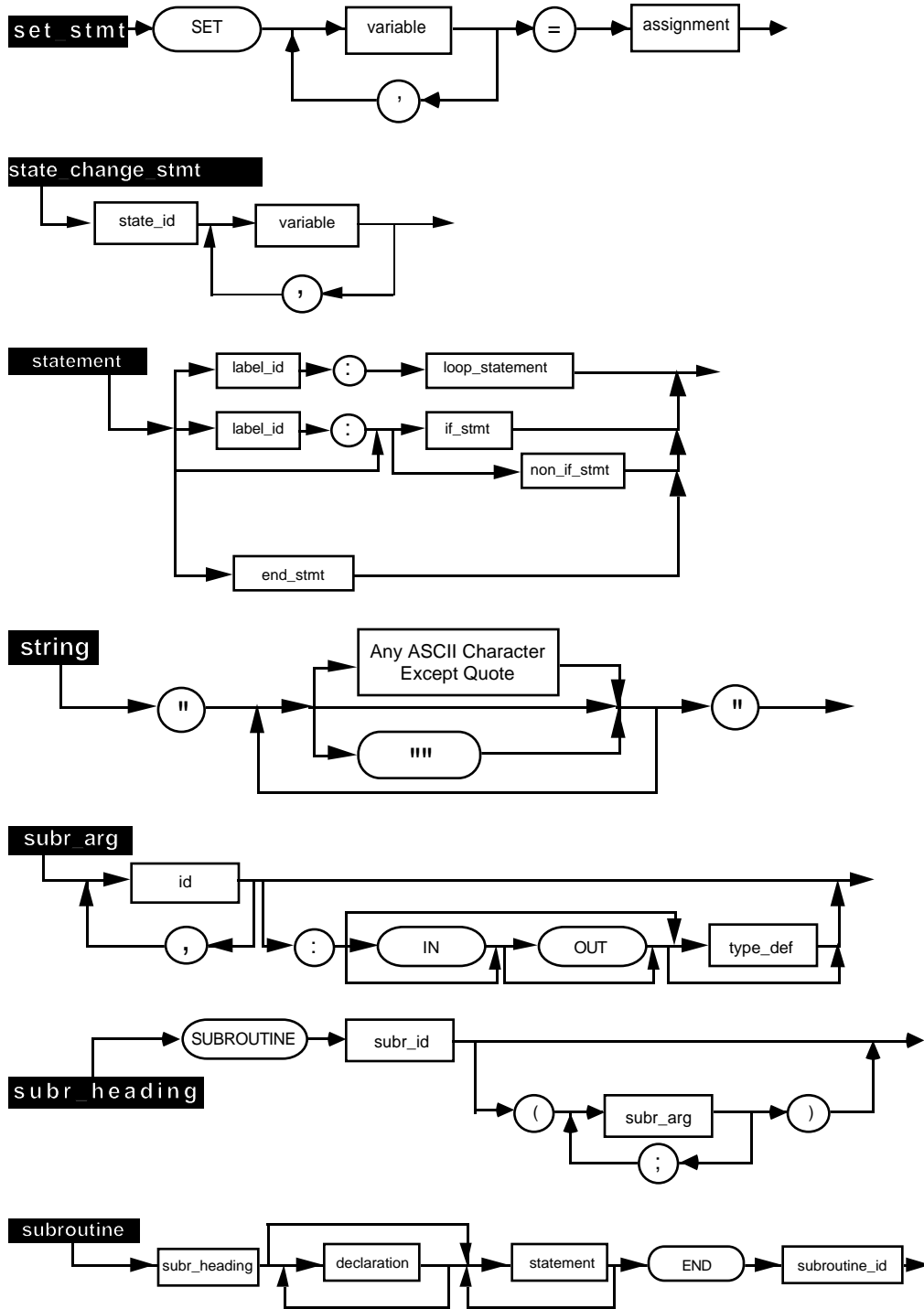




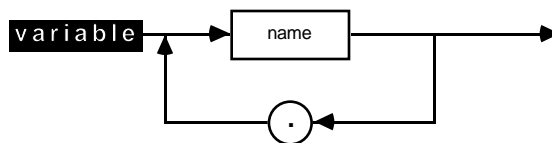
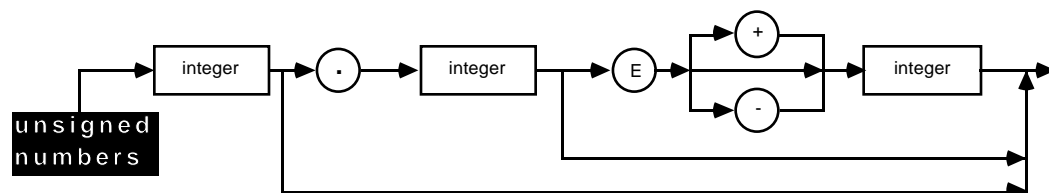
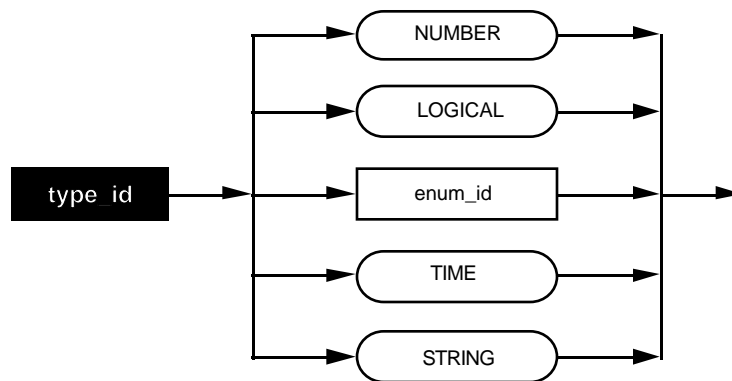
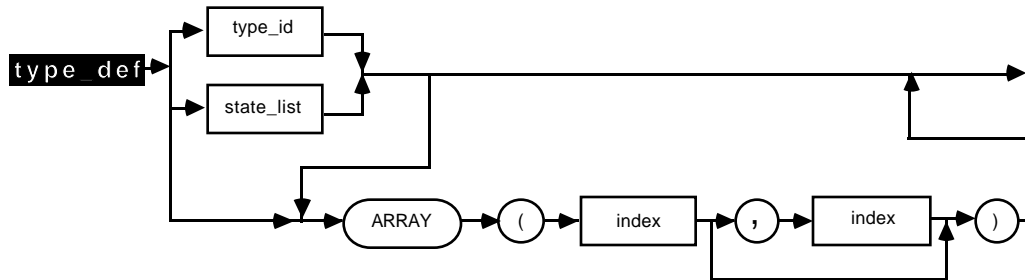
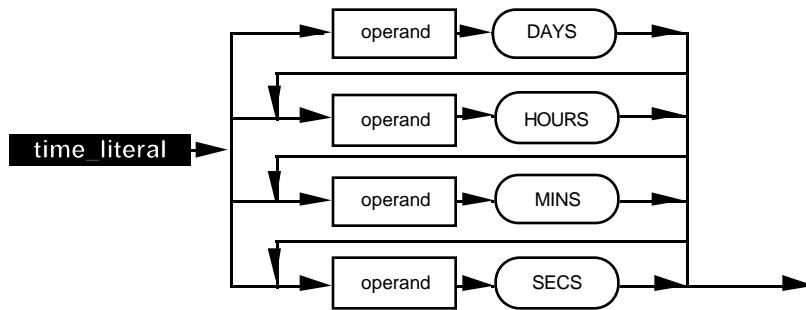
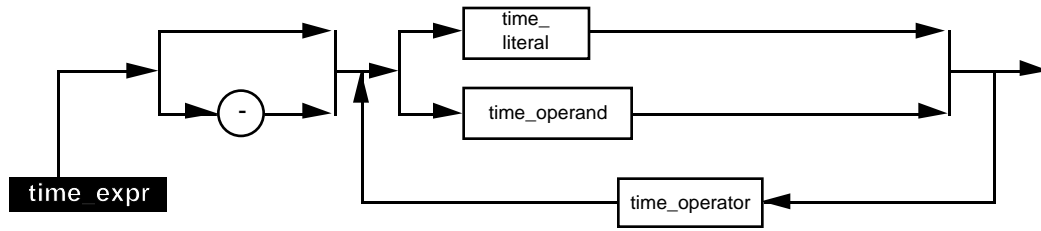












### A.3 NOTATION USED FOR SYNTAX PRODUCTION RULES

The following presentation of CL/AM syntax effectively follows BNF notation conventions as follows:

- Sequences of lowercase characters and embedded underscores mean things are to be combined according to the exact form expressed under this Syntax heading.
- Uppercase characters and special characters appear as written, except for the symbols `::=`, `{`, `}`, `[`, `]`, and `|`, which are explained as follows.
  - An item enclosed in braces (`{`, `}`) stands for the occurrence of that item zero or more times.
  - An item enclosed in square brackets (`[`, `]`) stands for the occurrence of that item zero or one times; i.e., the item is optional.
  - The symbols `::=` and `|` stand for production (how to build, or put together) and alternation, respectively. For example, `x ::= y | z` is 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 (i.e., **anything\_id ::= id**).

### A.4 CL/AM SYNTAX PRODUCTION RULES

```
abort_stmt ::= ABORT

access_attribute ::= ACCESS access_lock_id
access_key_id ::= PROGRAM
                | CONTIN_CTRL

access_lock_id = PROGRAM
                OPERATOR
                SUPERVISOR
                ENGINEER
                ENTITY_BLDR

addop ::= + | -
```

```

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

array_def ::= ARRAY ( index [, index] )

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

attribute_stmt ::= access_attribute
                  | bld_vis_attribute
                  | eu_attribute
                  | value_attribute
                  | class_attribute

bld_vis_attribute ::= [NOT] BLD_VISIBLE

Block ::= Block_heading
         { declaration }
         statements
         END Block_id
         { subroutine }

block_dir ::= %relax_dir | %include_set_dir

Block_heading ::= BLOCK Block_id ( point_description ;
                                AT insert_id [ (integer) ]
                                [; WHEN condition ]
                                [; ACCESS access_key_id]))

call_stmt ::= CALL subr_id [ (expressions) ]

CD_def ::= CD_heading
          param_description
          { param_description }
          END CUSTOM

CD_heading ::= CUSTOM [( CD_head_clause
                       {; CD_head_clause} )]

CD_head_clause ::= access_attribute
                  | bld_vis_attribute
                  | class_attribute

CDS_param_decl ::= PARAMETER ext_typed_ids [String]

class_attribute ::= CLASS class_id

class_id ::= PV_Alg
            | Ctl_Alg
            | General

```

```

compilation_unit :: = package
                    | sequence_program
                    | Block
                    | CDS_def
                    | enum_def
                    | PList_def
                    | lib_subroutine

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

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

const_expression :: = expression

constant_decl :: = const_expression

declaration :: = local_var
               | local_const
               | external_decl
               | param_decl
               | function_def
               | block_dir

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

else_if_stmt :: = ELSE IF condition THEN consequent

else_stmt :: = ELSE consequent

end_stmt :: = END block_id
            | END PList_id
            | END subr_id
            | END CUSTOM
            | END PACKAGE

enum_def :: = ENUMERATION enumeration_id = state_list

error_clause :: = (WHEN ERROR non_if)

eu_attribute :: = EU String

exit_stmt :: = EXIT

expression :: = arith_expr | time_expr | logical_expr

expressions :: = expression {, expression}

external_decl :: = EXTERNAL ids

ext_typed_ids :: = ids [: ext_type_def]

```

```

ext_type_def :: = type_id
                | [type_id] ARRAY ( index )

factor :: = operand [** operand]

first_char :: = $ | letter

first_param :: = point_description
                | subr_arg

funct_args :: = (typed_ids {; typed_ids} )

function_def :: = DEFINE function_id [: type_id] [funct_args] = assignment

function_id :: = user_funct_id
                | built_in_funct_id

global_decl :: = CDS_def
                | enum_def
                | PList_def

goto_stmt :: = GOTO label_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 :: = enum_id | const_expression .. const_expression

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 typed_ids

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 ]

mulop ::= * | / | MOD

name ::= id
        | array_id (arith_expr [, arith_expr] )

non_if ::= set_stmt      | state_change_stmt | goto_stmt
         | repeat_stmt   | call_stmt         | send_stmt
         | exit_stmt     | abort_stmt

one_d_array ::= (const_expression {, const_expression} )

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

package ::= PACKAGE { global_decl } { package_dir } { blocks } END PACKAGE

package_dir ::= %relax_dir | %include_set_dir

par_type_def ::= type_id [ARRAY (index)]
               | state_list [ARRAY (index)]
               | ARRAY (index)

param ::= [point_id .] param_id

param_decl ::= PARAMETER params [: par_type_def]

param_description ::= CDS_param_decl { attribute_stmt }

params ::= param {, param}

PList_def ::= PARAM_LIST PList_id
             PList_param_decl
             { PList_param_decl }
             END PList_id

PList_param_decl ::= PARAMETER ext_typed_ids

```

```

point_description ::= POINT point_id
                  |   GENERIC [ PList_id ]

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

program ::= sequence_program
         |   Block

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

repeat_stmt ::= REPEAT label_id

send_stmt ::= SEND [variable] : expressions

set_stmt ::= SET variables = assignment

sign ::= + | -

simple_type ::= type_id
            |   state_list

state_change_stmt ::= state_id [variables]

state_list ::= state_id / state_id { / state_id }

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

statements ::= statement { statement }

String ::= " {String_chr} "

String_chr ::= any ASCII character_except_quote
            |   ""

```

```

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

subr_args :: = (first_param {; subr_arg} )

subr_heading :: = SUBROUTINE subr_id [subr_args]

subroutine :: = subr_heading {declaration} statements END subr_id

term :: = factor {mulop factor}

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

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

time_operand :: = operand

time_operator :: = + | -

time_term :: = time_literal | time_operand

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

type_id :: = NUMBER | LOGICAL | enum_id | TIME | STRING

typed_ids :: = ids [: type_def]

unlabeled_stmt :: = if_stmt
                  | non_if

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

value_attribute :: = VALUE const_expression
                  | VALUE one_d_array

variable :: = name {. name}

variables :: = variable {, variable}

```



## CL/AM 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/AM (in other words, the CL Run-time Environment). This appendix also details some of the limitations the TPS CL Runtime Environment places on various aspects of building CL/AM structures, such as memory usage. Finally, this appendix contains a table of all the \$REG\_CTL parameters available to CL/AM.*

#### B.1 REFERENCES TO CONTROL FUNCTIONS PUBLICATIONS

The relationships between the TPS Software environment and CL/AM is found in the *System Control Functions and Application Module 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; 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/AM 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
4.7.4 & 4.7.5	Value Stores (CL corrective action on errors)

#### AM CONTROL FUNCTIONS

2.3.8 & 2.3.9	CL Runtime Errors (general)
3.1.3	Processing Order (Insertion points)
3.1.6	Interface of SP-handling Functions (of Regulatory Data Points) to User-written Programs
3.1.10	Windup Status in CL Programs
3.6 (all)	AM Custom Data Points
3.7 (all)	CL Switches on AM Data Points
Section 4 (all)	CL Support

## B.2 CL/CDS CAPACITIES

### B.2.1 Custom Names—System Limits

Custom Parameter Names	4000
Custom Data Segment Names	4000
Parameter List Names	250
Enumeration Set Names	200
Enumeration Set Members	1000
Members in a single enumeration	250

### B.2.2 CL/AM Limits

Max. Blocks for each point	255
Max. Blocks for each Insertion Point	255
Max. Lines of Code/Block (depends on statement complexity)	300-900
Max. Local Variable Space for CL Block or Subroutine	16 k words
Words for each Variable:	
Number	2
Time	2
Logical	1
Enumeration	1
String	40

### B.2.3 CL/AM Sizes

#### NOTE

Generic blocks are counted once for each unit they are used in, regardless of the number of points they are linked to. If a block makes any references, there is a Reference List for each point a CL block is linked to.

CL Block Overhead for each block	17 W
Words for each statement (depends heavily on complexity of statement)	10 W to 400 W; avg. ~ 20 W
Ref. List Entries, each block Where n is number of parameter accesses (Indirect reference counts as two accesses; indirect through array of point names of size K = 1 + K accesses)	1 + 16 n
CL Ref. List Overhead for each reference list	4 W

## B.2.4

Maximum blocks per unit	1000
Maximum reference lists per unit	4000
Maximum references per block	1000
Maximum words of code for each block	16000
Prefetch Overhead	13 W for each parameter fetched off node
CL Overhead per AM	1034 W

### B.2.4 CDS Limits

Max. Packages for each point	10
Max. Parameter Names for each Segment	225
Max. Parameter Names for each Class on a point (Classes are: PV_Alg, Ctl_Alg, General)	225
Max. Items in an Array	1000
Array Boundary (Subscript) Range	-4096 to +4095
Max. Size for each Segment (see size of an element below)	8000
Max. Build Visible Parameters for each Package	63 scalar parameters if no arrays. (For arrays, each element past the first array element counts as 1/2 of a normal parameter.)
Max. number of characters for all Value statements in a CDS	3000
Max. CDS size Data Entity Builder can handle.	Approximately 972 loaded parameters (where a loaded parameter element is a scalar or an element of an array, either of which is Build Visible or Not Build Visible and has a VALUE statement specified).

### B.2.5 CDS Sizes

Segment Size in the Point Record	
Overhead for each CDS	3 W
Size of an element: Number	2 W
Logical	1 W
Enumeration	2 W
Time	3 W
Point Name	4 W
String	21 W
CDS Description Size	24 + 25 * n
(In AM, once for each CDS Description used);	
(In checkpoint, once for each unit on which	
it is used) where n is number of parameter names	

#### NOTE

The CL Compiler does not error a CDS that is too big for the DEB to handle. The compile is allowed to take place, but the DEB issues an error when you try to add the CDS to a point. The maximum size the DEB can handle is approximately 972 loaded parameter elements. A loaded parameter element is a scalar parameter or an element of an array that is either Build Visible or not Build Visible and has a Value statement specified.

### B.3 \$REG\_CTL PARAMETER LIST

Table B-1 is a list of the parameter names and their types in the \$REG\_CTL parameter list.

**Table B-1 — \$REG\_CTL Parameters and Types**

Parameter Name	Parameter Type	Parameter Name	Parameter Type
ARWDI	ENM: POLARITY	INITMAN	LOGICAL
ARWNET	ENM: WINDUP	INITREQ [ 1.. 4]	LOGICAL
ARWOP	ENM: WINDUP	INITTYPE	ENM: INITTYPE
ADVSP	NUMBER	INITVAL	NUMBER
ADVSP	NUMBER	K	NUMBER
AV	NUMBER	K1	NUMBER
AVCOUNTS	NUMBER	K2	NUMBER
AVDEV1FL	LOGICAL	K3	NUMBER
AVDEV2FL	LOGICAL	K4	NUMBER
AVP	NUMBER	KEXT	NUMBER
AVTV	NUMBER	KFF	NUMBER
AVTVFL	LOGICAL	KGAP	NUMBER
B	NUMBER	KLIN	NUMBER
B1	NUMBER	KNL	NUMBER
B2	NUMBER	LASTPV	NUMBER
B3	NUMBER	MODE	ENM: MODE
B4	NUMBER	NAME	STRING
BADCTFL	LOGICAL	NMODE	ENM: MODE
BADPVFL	LOGICAL	OFFNDIAK	LOGICAL
BFF	NUMBER	OFFNDIRQ	LOGICAL
BIAS	NUMBER	OP	NUMBER
BYPASS	LOGICAL	OPEU	NUMBER
CASREQ	ENM: CASREQ	OPHIFL	LOGICAL
CIACCSTS	ENM: PASTATUS	OPHILM	NUMBER
CIACSTS	ENM: IOACTSTS	OPLOFL	LOGICAL
COACCSTS	ENM: PASTATUS	OPLOLM	NUMBER
COACTSTS [ 1.. 8]	ENM: IOACTSTS	OPMCHLM	NUMBER
COMMAND	ENM: COMMAND	OPROCLM	NUMBER
CONTCUT	LOGICAL	ORFBSEC	NUMBER
CTRLINIT	LOGICAL	PATHIND	ENM: PATHIND
CV	NUMBER	PIACCSTS	ENM: PASTATUS
CVEUHI	NUMBER	PIACTSTS	ENM: IOACTSTS
CVEULO	NUMBER	PPS	LOGICAL
CVTYPE	ENM: CVTYPE	PPSREQ	ENM: PPSTYPE
		PTDESC	STRING
DEVLOFL	LOGICAL	PTEXECST	ENM: PTEXECST
ESW AUTO	LOGICAL	PTINAL	LOGICAL
ESWCAS	LOGICAL	PTORST	ENM: ORSTATUS
ESWMAN	LOGICAL	PV	NUMBER
FSELIN	ENM: PINP	PV AUTO	NUMBER
GAPHI	NUMBER	PV AUTOST	ENM: PVVALST
GAPLO	NUMBER	PV CALC	NUMBER
GIACCSTS	ENM: PASTATUS	PV COUNTS	NUMBER
GIACSTS	ENM: IOACTSTS	PVEUHI	NUMBER
GOACCSTS	ENM: PASTATUS	PVEULO	NUMBER
GOACTSTS	ENM: IOACTSTS	PVEXEUHI	NUMBER
HOLDCMD	LOGICAL	PVEXEULO	NUMBER

(Continued)

Table B-1 — \$REG\_CTL Parameters and Types (Continued)

Parameter Name	Parameter Type	Parameter Name	Parameter Type
PVEXHIFL	LOGICAL		
PVEXLOFL	LOGICAL		
PVHHFL	LOGICAL		
PVHIFL	LOGICAL		
PVINIT	LOGICAL		
PVLLFL	LOGICAL		
PVLOFL	LOGICAL		
PVP	NUMBER		
PVROCNFL	LOGICAL		
PVROCPFL	LOGICAL		
PVSGCHFL	LOGICAL		
PVSOURCE	ENM: PVSOURCE		
PVSTS	ENM: PVVALST		
PVTV	NUMBER		
PVTVP	NUMBER		
RATIO	NUMBER		
RESETCMD	ENM: RESET		
RESETVAL	NUMBER		
RESTART	ENM: RESTART		
S1	LOGICAL		
S2	LOGICAL		
S3	LOGICAL		
S4	LOGICAL		
SECARW [ 1.. 8]	ENM: WINDUP		
SELINP	ENM: PINP		
SELXINP	ENM: XINP		
SHEDMODE	ENM: MODE		
SP	NUMBER		
SPEUHI	NUMBER		
SPEULO	NUMBER		
SPEXEUHI	NUMBER		
SPEXEULO	NUMBER		
SPHIFL	LOGICAL		
SPHILM	NUMBER		
SPLOFL	LOGICAL		
SPLOLM	NUMBER		
SPP	NUMBER		
SPSTS	ENM: PVVALST		
SPTV	NUMBER		
SPTVP	NUMBER		
STATE	ENM: STATE		
STRTSTOP	ENM: STRTSTOP		
T1	NUMBER		
T2	NUMBER		
T3	NUMBER		
TIMELEFT	NUMBER		
TS	NUMBER		

## B.4 CL DATA ACCESS PERFORMANCE

The performance figures given here are intended to give you an idea of the relative performance of various types of access so that you can write CL programs in an optimum manner. Except where otherwise stated, the times given are CPU time and are for the 68020 only. They are approximate times and should be used only for comparative purposes.

CDS access times apply only to releases 321.12 and greater in the 300 release series, and 401 and greater in the 400 series. Older releases have significantly slower access times for off-point and/or background CDS access. For 68000 nodes, the times should be approximately doubled except for background CL, which requires the doubling plus approximately one millisecond additional CPU time for each access.

**Table B-2 — CL Approximate Data Access Times**

Access Type	Access Time
Foreground/background CL SET statement accessing local number scalar	2 microseconds
Foreground/background CL SET statement accessing local variable with computed local variable index	40 microseconds
Foreground CL SET statement accessing on-physical-node CDS parameter	100 microseconds
Foreground CL move parameter call accessing on-physical-node CDS parameter	180 microseconds
Foreground CL SET statement fetching an off-physical-node CDS scalar parameter from CVB to a local variable	225 microseconds
Foreground move parameter call fetching off-physical-node CDS parameter from CVB to a local variable	300 microseconds
Background CL SET statement accessing on-physical-node CDS parameter	150 microseconds
Background CL move parameter call accessing on-physical-node CDS parameter	300 microseconds
Foreground move parameter call moving 40-character string off-physical-node from CVB to local variable	325 microseconds
Background CL move parameter off-unit, on-physical-node 50-element number array fetch to local	600 microseconds
Foreground move parameter 40-character string on-physical-node fetch to local	200 microseconds
Foreground SET statement 40-character string on-physical-node fetch to local	120 microseconds
Background move parameter 40-character string on-physical-node fetch to local	260 microseconds
Background move parameter 40-character string on-physical-node fetch to local	260 microseconds

(continued)

**Table B-2 — CL Approximate Data Access Times** (continued)

Foreground CL “fast” standard parameter on-physical-node fetch	220 microseconds (Background is somewhat slower)
Foreground CL “fast” standard parameter on-physical-node store	400 microseconds (Background is somewhat slower)
Foreground CL off-physical-node fetch of a standard parameter from CVB	350 microseconds
Foreground CL off-physical-node standard parameter store (to CVB)	450 microseconds
Foreground CL standard parameter on-physical-node fetch	800 microseconds (Background is somewhat slower)
Foreground CL standard parameter on-physical-node store	1 millisecond (Background is somewhat slower)
Background CL single parameter off-physical-node fetch/store using set statement or move parameter	100 milliseconds (approx.) (Elapsed time, not CPU time)
Background CL parameter off-physical-node fetch/store using multiple move parameter	190 milliseconds plus 11 milliseconds per item moved (Elapsed time, including list build time)
Background CL parameter off-physical-node fetch/store using multiple move parameter	180 milliseconds plus 3 milliseconds per item moved (Elapsed time, not including list build time)
Background entity id move parameter to an on-physical-node entity which is indirected through once	2 milliseconds
Background entity id move parameter to a single entity in a large array that is indirected through once	2.8 milliseconds
Background entity id move parameter to a single entity in a large array on a point with many reference lists with a number of multilevel indirections through each entity	20 milliseconds

Foreground entity id stores take about as much time as background entity id stores with the exception that foreground CL on-point stores are about 1 millisecond faster than background on-point stores. The actual indirection resolution time is a little slower for foreground stores, but foreground on-point entity stores are about an order of magnitude faster than background on-point entity stores. This means that for very simple cases, foreground on-point stores are significantly faster than background on-point stores. For complex cases, the relative differences are not so great.

Entity id stores to points that are currently being processed or that have an outstanding prefetch are slower because the actual indirection resolution is queued and done only after point processing is completed. Thus, if an entity id store is timed by measuring the elapsed time taken by the actual store, most of the relink activity may not be measured.

All CL off-point, on-physical node accesses are slower the first time they are executed after an indirection resolution than they are for subsequent accesses. For fetches, this first time execution is considerably faster if the CL extension AMCL04 is loaded.



## CL EXTENSION FOR FILE I/O

### Appendix C

*This appendix explains an optionally available set of subroutines that enable you to move data between CL/AM programs and files on either the History Module or removable media.*

#### C.1 REFERENCES

The following publications contain information required to fully understand the capabilities provided in the optional CL Extension for File I/O.

- The *Picture Editor Reference Manual* contains a complete description of the format specifications used by this package. Please note that these format specifications are case sensitive and thus they must be entered using upper case (capital) letters as shown in the examples in this appendix.
- Section 2 of the *Engineer's Reference Manual* explains **TotalPlant** Solution (TPS) system file system concepts.
- The *Command Processor Operation* manual contains additional information on the TPS file system.
- The *Messages Directory* contains a list defining the TPS Data Access error codes.

#### C.2 OVERVIEW OF FILE I/O EXTENSION

The CL Extension for File I/O consists of a set of subroutines that can be loaded in an AM. These subroutines enable a CL/AM program to both write data to and read data from ASCII text files on the History Module (HM), and, starting at R320, to create and access volumes, directories, and ASCII text files on removable media (floppies or cartridge disks). Data conversion between external and internal forms is performed as appropriate. Additional data manipulation subroutines further support the file processing capabilities.

Most of the subroutines in this package can be used only by CL blocks linked to Background insertion points. The exceptions are noted in the individual subroutine explanations starting at heading C.6.1. Table C-1 provides a summary of the subroutines in this package.

##### C.2.1 Data Conversions

The data conversion subroutines in this package convert values held in CL programs to human readable (ASCII character) form or values in human readable form to CL internal form. For the remainder of this document, human readable data is referred to as "external" data and data that resides in parameters and CL variables is referred to as "internal" data.

Table C-1 — Subroutines in File I/O Extension

Subroutine Name	Subroutine Function	Section
File\$Create_File	Create a new file	C.6.1
File\$Open_File	Open existing file	C.6.2
File\$Put_Field	Write values to fields in the file's I/O buffer	C.6.3
File\$Put_Field_S	Write values to fields in a string variable*	C.6.4
File\$Write_Record	Write record in the file's I/O buffer to the file	C.6.5
File\$Read_Record	Read record from HM to the file's I/O buffer	C.6.6
File\$Get_Field	Read values from fields in the file's I/O buffer	C.6.7
File\$Get_Field_S	Read values from a string variable*	C.6.8
File\$Close_File	Close a previously opened file	C.6.9
File\$Copy_Record	Copy a record from one I/O buffer to another	C.6.10
File\$Copy_File	Make copy of an existing file	C.6.11
File\$Exists	Determine if a given file exists	C.6.12
File\$Find_Field_Pointer	Find a field within the file's I/O buffer	C.6.13
File\$Find_Field_Pointer_S	Find a field within a string variable*	C.6.14
File\$Set_Null	Set length of the file's I/O buffer to zero	C.6.15
File\$Blank_Fill	Place blank characters into the file's I/O buffer	C.6.16
File\$Blank_Fill_S	Place blank characters into a string variable*	C.6.17
File\$Rename	Change the name of an existing file	C.6.18
File\$Safe_Rename	Give the name of one file to another file	C.6.19
File\$Protect_File	Protect/unprotect file from deletion	C.6.20
File\$Convert_SDE_to_Internal	Convert SDE to internal form	C.6.21
File\$Convert_SDE_to_ASCII	Convert SDE to external form	C.6.22
File\$Get_Volume_Statistics	Get space use information for volume	C.6.23
File\$Create_Character	Convert number to ASCII character*	C.6.24
File\$Convert_FM_Status	Convert FM status value to string	C.6.25
File\$Strip_Blanks	Enable/disable stripping by File\$Get_Field	C.6.26
File\$Upper_Case	Convert string characters to upper case*	C.6.27
File\$Delete_File	Delete file from fixed or removable media	C.6.28
File\$Create_Volume	Create a volume on a floppy/cartridge disk	C.6.29
File\$Create_Directory	Create a directory on a floppy/cartridge disk	C.6.30
File\$Delete_Directory	Delete a directory from a floppy/cartridge disk	C.6.31
File\$Read_Directory	List all files that match specified search criteria	C.6.32
File\$Get_File_Attributes	Get file data from files in Read_Directory list	C.6.33
File\$Read_Directory_Complete	Deallocate space used by Read_Directory list	C.6.34
File\$Get_Volume_Directories	List all directories on the specified volume	C.6.35
File\$Create_Volume_With_Descriptors	Create a volume with descriptors on a floppy/cartridge disk	C.6.36
File\$Modify_Volume_ID_String	Modify a volume ID string	C.6.37
File\$Modify_File_Descriptor	Add, modify, or delete a file's descriptor	C.6.38
File\$Modify_Directory_Descriptor	Add, modify, or delete a directory descriptor	C.6.39
File\$Get_File_Descriptor	Get the descriptor of a file previously created	C.6.40
File\$Get_Volume_Directories_D	Get list of directories and descriptors	C.6.41

\* File I/O Extension subroutines that can be called by non-Background CL programs.

External values can be extracted from records in a file or from string variables. Each record or string is composed of one or more fields. Fields are composed of one or more characters. Fields are read from the record or string, converted to internal form and stored to a CL local variable or to a parameter.

Internal values are taken from a CL local variable or a parameter, converted to external form and stored into a field in a record or in a string variable.

Strings are supported so that a CL program can build complicated messages or interpret external information entered by an operator.

All the CL supported data types (that is, number, logical, enumeration, date/time, string) plus entity names (point names) and self-defined enumerations are converted from internal to external and external to internal form.

### **C.2.2 File I/O**

The file input/output operations provided in these subroutines include:

- Create volumes and directories on a floppy or cartridge; Delete directories from a floppy or cartridge.
- Create, Open, Close, and Delete ASCII text files on the HM or on a floppy/cartridge.
- Convert CL internal data types to external values, store them into fields, combine the fields into records, and write the records into files.
- Read records from files and convert fields in the record to any appropriate CL internal data type
- Convert string characters to upper case (for use in comparisons)
- Rename files
- Protect and Unprotect files
- Copy files
- Copy individual records from one file to another
- Determine volume size and free space
- Determine if a file exists
- Search for files with certain name characteristics; get file attributes for each of those files

Note that files on the HM also can be deleted via the standard CL function Delete\_File.

### C.2.2.1 File Operations Through the Network Gateway

The only CL/AM file operations that can be performed through the Network Gateway to a remote LCN are:

- File\$Copy\_File
- File\$Delete\_File
- File\$Exists
- File\$Get\_Volume\_Directories
- File\$Read\_Directory
- File\$Read\_Directory\_Complete

### C.2.2.2 File Access Types

File access type is specified when a file is opened and remains in effect as long as it is open. This package supports the following access types:

- Shared-Read-No-Write (SRNW)—multiple readers but no writes allowed.
- Exclusive (E)—essentially private access, both update and append allowed.

The following matrix indicates whether or not an access request will be granted, based on the access requested and what (if any) access has been granted to other users.

		Access Currently Granted to Other Users of the Resource		
		not in use	SRNW	E
Access Requested	SRNW	yes	yes	no
	E	yes	no	no

### C.2.3 Return Status

Because this set of subroutines is not part of the standard AM Personality, status information is returned from the File I/O Extension subroutines as numbers rather than as standard enumerations. Most of these subroutines have two return status values.

The first of the two status values indicates the more common errors, but does not cover obscure File Manager or Data Access errors. A list of these status argument values with a brief description can be found at heading C.7, "Return Status Values."

The second status value, if required, contains either a File Manager or a Data Access Error code that further defines the problem. The Data Access Error codes are documented in the *Messages Directory*, while File Manager status values are interpreted at heading 4.10 of this publication. You also can use the File\$Convert\_FM\_Status routine to obtain a terse ASCII string explanation of the File Manager status code. File Manager errors for which retries are appropriate, are retried automatically. Note that a Data Access Error code of 0 (zero) indicates no errors, whereas a File Manager code of 51 indicates no errors.

## NOTE

A return status code of 12.0 indicating Other Fiile Manager Error with a File Manager status code of 27.0 may also indicate that the file request attempted to allocate more AM memory than is allowed for a background task.

In order to focus on the functionality of the File I/O subroutines, the examples shown do not include code to test status and process indicated errors. A complete program will, of course, include such code. If the first status (Return\_Status) is zero indicating no errors, it is not necessary to test the second status (File Manager Status or Data Access Status).

### C.2.4 Media Path Identification

Two forms of file access path identification are supported, depending on where the volume and file reside.

On History Module—NET>voldir>file\_id.extension  
or—rl\NET>voldir>file\_id.extension  
On floppy/cartridge—PN:nn>DEV:xxmm>voldir>file\_id.extension

where

nn = the physical node number  
xx = FD for floppy drive, or RM for cartridge drive  
mm = device number on the node (00 or 01)  
voldir = volume or directory identifier  
rl = the PIN identifier for a remote LCN

Examples: NET>gene>amhist1.yy  
PN:30>DEV:RM00>gene>amhist1.yy

Heading C.2.9 states rules that must be followed when naming any volumes, directories and files that you create.

**NOTE**

Pathforms of the type \$F7>volume\_id>file\_id.extension (where \$F7 is the logical device id) are **not** allowed.

### C.2.5 CL Aborts

If there is insufficient stack memory available for the set to perform the requested operation, the CL Block is aborted with a CL Error Status (CLERRSTS) of Program Error (PROGERR). The File I/O Extension uses 500 words of stack to perform its operations. An additional 4000 words is used while reading or writing parameters, whether this is done in one of the subroutines in this set or directly in the CL Block.

If invalid arguments are passed to one of these subroutines, the CL Block is aborted with a CL Error Status (CLERRSTS) of Program Error (PROGERR). For example, passing an argument of type self-defined enumeration to a subroutine in the Set that cannot handle self-defined enumerations will cause a CL Abort. For standard CL runtime functions, this type of error is caught by the compiler. For these File I/O subroutines, however, some type checking is deferred until runtime; thus, CL Aborts can result.

## **C.2.6 Similarities to FORTRAN**

### **C.2.6.1 Write to Record**

The output routines in this File I/O Extension have arguments for data values and format specifications similar to FORTRAN, except that only one value (field) is added to a record on each function call. Consecutive calls result in appending values to the same record.

### **C.2.6.2 Read from Record**

Reading values is similar to FORTRAN except that:

- A format specification is not supplied on a read. The value is read from a field terminated by a delimiter and is converted to internal form based on the type of the variable that receives it.
- Fields in a record do not start in a specific column and are not fixed length. They typically start right after the previous field and are terminated by a special delimiter character.

## **C.2.7 Records and Fields**

Records are the basic units of storage in a file. Each read from a file causes a record to be moved from the file into memory. Each record in a file can be a different length, although there is a maximum record size established for each file when the file is created. There is an absolute maximum record length of 15666 characters. Note that if a record is to be read by the text editor, it can be no more than 1024 characters long and only the first 80 characters are displayed.

The File I/O Extension assumes that records are made up of one or more fields. Fields contain one or more characters and—except for string types—are terminated by a delimiter character (fields of type string are of a known length). The delimiter character can be any character that never appears in the main part of the field; a space or a comma is the most typical delimiter.

**NOTE**

The delimiter can be one or many characters. If the delimiter is defined as a CDS parameter of type string, the apparent length of the string depends on its source.

If the CDS parameter value is loaded from the Data Entity Builder, or stored by an Advanced Control Program (ACP) through a CG or PLNM, the delimiter length is defined as 40 characters (trailing blanks are included in this count).

If the CDS parameters is stored from the Operator Personality through the entity's detail display or a schematic, then the delimiter length is defined as the number of characters entered, excluding trailing blanks. If zero or more blanks are the only characters entered from the operator station, then the string is a null string.

If the CDS parameter is stored from a CL program, then the delimiter length is defined as the number of characters entered, including trailing blanks.

Fields contain data in external form that can be converted to internal form and stored into CL Local Variables or Parameters. Conversely, fields are created by converting data in internal form to external form and storing into them.

A file must be opened before it can be read from or written to. Each time a file is opened, I/O buffer space for it is allocated in memory. The amount of space to be allocated is specified as an argument of the open call. Records to be written to a file are created in the file's I/O buffer space (see File\$Put\_Field). Records read from a file are read into the file's I/O buffer space. The record stored in a file's I/O buffer space at a particular time is referred to as its **current record**.

Each current record has an associated character pointer which is used to reference fields in that record. The character pointer can be adjusted to point at a specific absolute character position, or set to point at the beginning of a specific field.

## C.2.8 Strings

This package supports inserting (and extracting) fields into (from) strings identical to the way record fields are processed. CL string variables may contain up to 78 characters. CDS string parameters may contain up to 40 characters.

## C.2.9 Volume, Directory and File Names

Volume and Directory names are from one to four characters long and can be composed of any combination of alphabetic characters, numerics, exclamation points (!), ampersands (&), and underscore (\_) characters.

File names are composed of a name and an extension. The name normally identifies the information that the file contains while the extension is used to convey the type of the file. File names must be unique within a directory.

File names and extensions must start with an alphabetic character. File names can be from one to eight characters long. The extension must be one or two characters long.

Katakana characters must be translated to alphanumeric characters before calling the file access routines, since the eighth bit of all characters in the file name is ignored by the file system .

### **C.2.10 Unique File Name Generation**

Requests for a unique file name within a volume are accepted when a file is created via the Create\_File and Copy\_File subroutines by embedding one or more question marks ("?",) in the file name or file extension. Each question mark is replaced non-deterministically with a numeric character (i.e., 0-9) when the file is created. The generated name is guaranteed to be different from any existing file name unless there are no numeric characters that will provide a unique name.

Examples of user specified unique file names to be generated are H????????.BH, DUMP?.DU, and LOAD?.L?.

#### **NOTE**

Do not use the wild card character when attempting to open an existing file. Instead of opening the existing file, a new file is created.

### **C.2.11 Temporary Files**

By convention, files with a file name extension of "\_\_\_" (two underscore characters) are considered temporary files. Temporary files that exist on the History Module are deleted when the History Module is started and on failure of a remote node that has such a file open.

Temporary files are used in conjunction with the Safe\_Rename routine.

### **C.2.12 Operator Interaction**

Any CL/AM program that makes use of these removable media functions must provide any necessary operator prompts (such as mount or remove a specified floppy/cartridge) through CL SEND statements. Note that these CL messages go to all consoles assigned to the process unit of the CL's bound data point.



### C.3 RESTRICTIONS

The following restrictions apply to this package:

- External time values read using `Get_Field` or `Get_Field_S` must appear in the form "HH:MM:SS" (e.g., 15:30:45). They can be written using `Put_Field` or `Put_Field_S` with the text format "TIMEHH:MM:SSENDTIME".
- External date values read using the `Get_Field` or `Get_Field_S` must appear in the form "DD MMM YY" (e.g. 25 DEC 88). They can be written using `Put_Field` or `Put_Field_S` with the text format "DATEDD AM:3 YYENDDATE".
- The subroutines available in this package that write to or read from a file or that modify the current record associated with a file can only be invoked from CL Blocks linked to background insertion points.
- A maximum of ten files may be open simultaneously from one Application Module.
- A maximum of 40,000 characters of buffer space may be allocated at a time by `Open_File` requests from one Application Module.
- Enumeration values are written out (and read in) as state names.
- Point IDs are written and read as ASCII point names.

### C.4 PACKAGING

The CL Extension for File I/O is packaged as two sets, `Conv` and `File`, and a set definition file used to define the `File` set subroutines to the CL compiler.

`Conv` contains data access conversion subroutines that are used by the `File` set. These subroutines are not directly available to CL/AM blocks. They convert enumeration values, self defined enumeration values and point ids from internal to external form and from external to internal form.

The `File` set contains subroutines that convert the data types not supported by the `Conv` set and performs operations on files. Segregating the data access subroutines from other subroutines allows other sets to use the data access conversion subroutines without having to package them internally.

## C.5 INSTALLATION AND CONFIGURATION

The two set image files CONV.LO and FILE.LO must be installed in a directory named &CUS on the type of media from which the AM will be loaded. The set definition file FILE.SF must be installed in the directory &CLX on a History Module.

Any AM which is loaded with the sets must be correctly configured via the Network Configurator. To do this, perform the following steps

1. Set the default Volume Path for the Network Configurator Backup to point at a device where the current NCF will be saved.
2. Invoke the Network Configurator from the Engineers Main Menu by selecting "LCN Nodes."
3. Select the node number of the AM to be configured.
4. Select the "Modify Node" target on page 1.
5. Page forward to page 2.
6. Enter the following information on Page 2
  - "Background CL Tasks" = nn; where nn is the number of background CL blocks that may run concurrently. This can vary from 1 to 10, but setting it too large may cause the AM to run out of memory and crash while it is loading.
  - "Number of Concurrent Data Accesses from Background CLs" = nn; where nn is the number of data access requestor tasks that may be run concurrently. This can vary from 1 to 4.
  - "Background Task Stack Size" should be set to the default which is 15000.
7. On page 3 enter the name of the File Set (i.e., FILE) in the list of externally loaded modules and press the Enter key. This should cause the name of the conversion set (i.e., CONV) to also be placed in the list of externally loaded modules.
8. Press the Check Key (F1) and the configuration should check OK.
9. Press the Install Key(F2) and the configuration should install without error.

The AM can now be loaded.

### NOTE

Honeywell recommends that you load AMs individually to avoid problems. For example, if you have the CONV external load module configured for your AMs, you may get a CNAMREV error when trying to load more than one AM at the same time. To recover from that specific error, reload the nodes which have that warning.

## C.6 CL CALLABLE SUBROUTINES

The operations provided in this extension are accomplished by CL callable subroutines. The subroutines are declared in the set definition file which is read by the CL compiler when an "include\_set" directive is encountered in a CL source file within the scope of a CL Block. The directive would appear as follows in a source file:

```
%INCLUDE_SET FILE
```

Each routine may be invoked by calling it in a CL Block or Subroutine via the Call statement; for instance:

```
Call File$Create_File(status, fm_status, record_length,  
file_name)
```

invokes the subroutine to create a file.

### NOTE

Several of the subroutines in this extension set have arguments designated to be of the stand-in data type "cl\_type." The actual data types of these arguments, as specified in your calls, can be any valid CL/AM data type within the rules given below. For these arguments only, normal CL compiler data-type checking is inhibited, and any data type mismatches you may accidentally create are not detected until the CL block runs.

An IN argument of "cl\_type" accepts a local variable, parameter, or expression of any valid CL/AM data type.

An OUT argument of "cl\_type" accepts a local variable or parameter of any valid CL/AM data type.

Following is a description of each subroutine included in this extension.

### C.6.1 File\$Create\_File Subroutine

This routine creates a new file.

```
SUBROUTINE File$Create_File  
  (Return_Status      : OUT NUMBER; -- File R/W package error  
  status  
   File_Manager_Status : OUT NUMBER; -- File Manager error  
  enumeration  
   Max_Record_Length   : IN  NUMBER; -- Number of Characters  
  per record  
   Path                : IN  STRING) -- Full path of file to  
  be created
```

This subroutine can only be called from a CL block linked to a background insertion point.

It creates a file of variable length records with the maximum number of characters allowed in a record specified by Max\_Record\_Length. The Max\_Record\_Length may be from 1 to 15,666.

"Path" must specify the pathname of a file on the history module or a floppy/cartridge (see heading C.2.4).

If the file already exists, an error is returned and the file is not opened.

**Possible Return\_Status Values for this function:**

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 4.0 — File already exists
- 7.0 — File table for the specified volume is full
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 10.0 — No more space on the volume
- 12.0 — Other File Manager error
- 29.0 — File reserved for another operation
- 38.0 — Invalid record length in Create\_File or Open\_File call

**Examples:**

The following CL code segment creates a file named "file.x" with a maximum record length of 80 characters.

```

LOCAL record_len = 80                -- Max record length of new file

LOCAL status, fm_status : NUMBER    -- Return status values
LOCAL path              : STRING    -- Pathname of a file

SET path = "net>test>file.x"

CALL File$Create_File(status, fm_status, record_len, path)

```

**C.6.2 File\$Open\_File Subroutine**

Before a file can be read from or written to, it must be opened. You should close files after all reading/writing is complete even though they are automatically closed when a CL Block terminates or is aborted.

```

SUBROUTINE File$Open_File
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   File_Manager_Status : OUT NUMBER; -- File Manager error status
   File_Number        : OUT NUMBER; -- File id used on subsequent calls
   Number_Of_Records  : OUT NUMBER; -- Number of records now in file.
   Buffer_Size         : IN  NUMBER; -- Maximum characters per record.
   Path               : IN  STRING;  -- Full name of file to be opened.
   Access_privilege   : IN  NUMBER) -- File access type

```

This subroutine can only be called from a CL block linked to a background insertion point. All operations performed on a file (e.g., read, write, close) must be done from the CL Block that opens the file.

This subroutine opens an existing file containing variable length records. "Path" must specify the pathname of a file on the history module or a floppy/cartridge (see heading C.2.4). If the file does not exist, an error is returned.

Buffer\_Size specifies the number of characters allocated for the I/O buffer used to hold records read from or written to the file (i.e., the current record).

Access\_privilege specifies the access level that will be enforced when the file is accessed:

0.0 = Exclusive,  
1.0 = Shared\_Read\_No\_Write

Number\_of\_Records returns with the existing number of records in the file.

A maximum of 10 files may be opened via this routine on an AM at one time. The memory allocated for the I/O buffer for each opened file is acquired from a 40,000 character pool reserved for this package. If Buffer\_Size on this request is large enough to cause the memory pool to be exhausted, return\_status will be 23.0 (Memory limit exceeded).

**Possible Return\_Status Values for this function:**

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 2.0 — File exists; but is of wrong type
- 3.0 — File privilege violation
- 5.0 — File does not exist
- 6.0 — Device error while reading file
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 22.0 — Maximum number of files already open
- 23.0 — Memory limit exceeded
- 29.0 — File reserved for another operation
- 37.0 — Invalid file access code in Open\_File call
- 38.0 — Invalid record length in Create\_File or Open\_File call

**Examples:**

The following CL code segment opens a file named "file.x" with a maximum record length of 132 characters and an access privilege of "Exclusive."

```

LOCAL status,           -- Return status
&    fm_status,         -- File Manager Status
&    file_id,           -- File Identifier
&    num_records,       -- Number of Records in File
&    buffer_size,       -- Number of characters in I/O buffer
&    access_priv : NUMBER -- Access Privilege
LOCAL path              : STRING -- Pathname of a file

SET path          = "net>test>file.x"
SET buffer_size = 132

CALL File$Open_File (status, fm_status, file_id, num_records,
&                   buffer_size, path, 0)
```

### C.6.3 File\$Put\_Field Subroutine

This routine writes values to fields in the specified file's current record.

```
SUBROUTINE File$Put_Field
  (Return_Status : OUT NUMBER;      -- File R/W package error status
   DA_Status     : OUT NUMBER;      -- Data Access error status
   Character_Ptr  : IN OUT NUMBER;   -- Offset to beginning of field
   File_Number   : IN NUMBER;        -- File id from open_file call
   Input_Value    : IN cl_type;      -- Value to be converted (cannot be
                                     -- self-defined enumeration type)
   Format         : IN STRING;        -- Specified format of field
   Suffix         : IN STRING)       -- Suffix added to field
```

This subroutine can only be called from a CL block linked to a background insertion point.

Put\_Field takes an Input\_Value in internal form, converts it to a string and places it at the Character\_Ptr position in the current record.

If Input\_Value is an array, the CL Block is aborted. To convert an array each element must be converted separately.

Character\_Ptr is the character-offset from the beginning of the record to the beginning of the field to be written. When Put\_Field returns it will contain the position of the character after the written field. Note that the character pointer must be initialized (normally to 1) the first time a field is placed in a record. Normally, when sequential calls are used to create a single record image, the Character\_Ptr value returned on previous calls is passed without change on subsequent Put\_Field calls, but Character\_Ptr can be set to any position.

Character\_Ptr may be set to point beyond the last character written to the current record and the record is blank filled from the character position after the last character to the one preceding that pointed to by Character\_Ptr. Character\_ptr may point at a previously written field in the current record and characters in that field will be overwritten.

The string Suffix is an optional value appended to the field. Normally, Suffix is a single delimiter such as a comma or a space, but any string is allowed.

File\_Number must be the value returned on a previous Open\_File call.

Format is a specification of how the external value will appear after conversion. If Format is set to the empty string (i.e., "") the default format specification is used. The default format specification for the various data types are:

number	R-ZZZZ9.99
logical	TEXTL1:5
string	TEXTL1:10
time	DATE MM-DD-YY ENDATE
enumeration	TEXTL1:10
entity (point id)	TEXTL1:10

Use the "File\$Convert\_SDE\_To\_ASCII" subroutine to convert a self-defined enumeration to a string. The string then can be transferred by this subroutine.

**Possible Return\_Status Values for this function:**

0.0 — Good status  
 13.0 — File number is invalid  
 14.0 — Attempt to read/write field beyond end of buffer  
 15.0 — Value has invalid data type  
 16.0 — Format String is invalid  
 17.0 — Conversion of a value failed  
 18.0 — Error returned on Get/Store parameter; specific error in Data Access status  
 20.0 — Can't read file control table pointer  
 25.0 — Reference list number invalid  
 34.0 — Number not assigned to this CL Block—access denied  
 40.0 — Invalid character pointer

**Examples:**

The following CL code segment puts 3 fields in the current record. The first field is the name of the entity referred to from the CDS parameter "PSI." It will start at character position 1 of the record and is right justified in a field 20 characters long with a blank following it in character position 21.

The second field will be a number. It will start at character position 22, be right justified, and will be 6 characters long with a blank following it in character position 28. The value will be read from the PV on the point referred to by the parameter PSI.

The third field will be a time value. It will start at character position 29 and will be 8 characters long with a blank following it in character position 37. The record will look as follows after this code segment is executed:

PSI\_003    85.33 14:34:36

CUSTOM

  PARAMETER psi : \$REG\_CTL  
END CUSTOM

```

LOCAL status,           -- Return status
&    da_status,         -- Data Access Status
&    file_id,           -- File Identifier
&    offset              : NUMBER    -- Character offset of field
LOCAL format,           -- Field format specifier
&    delimiter          : STRING     -- Field delimiter character
LOCAL cur_time          : TIME       -- Current time

SET format1      = "TEXTR1:20"
SET format2      = "TIMEHH:MM:SSSENDTIME"
SET delimiter    = " "
SET offset       = 1
SET cur_time     = Now

CALL File$Put_Field (status, da_status, offset, file_id,
&                   psi, format1, delimiter)
CALL File$Put_Field (status, da_status, offset, file_id,
&                   psi.pv, "RZZ9.99", delimiter)
CALL File$Put_Field (status, da_status, offset, file_id,
&                   cur_time, format2, delimiter)

```

### C.6.4 File\$Put\_Field\_S Subroutine

This routine writes values to fields in a CL or CDS string.

```
SUBROUTINE File$Put_Field_S
  (Return_Status : OUT NUMBER;      -- File R/W package error status
   DA_Status     : OUT NUMBER;      -- Data Access error status
   Character_Ptr  : IN OUT NUMBER;  -- Pointer to field
   Output_String : IN OUT STRING;   -- String to place field in
   Input_Value   : IN cl_type;      -- Value to be converted (cannot be
                                     -- self-defined enumeration type)
   Format        : IN STRING;       -- Specified field format
   Suffix        : IN STRING)      -- Suffix added to field
```

This subroutine can be called from CL blocks linked to any insertion point.

The Put\_Field\_S function takes Input\_Value in internal form, converts it to external form and places it, beginning at the Character\_Ptr position, in Output\_String.

Output\_String can be either a local CL string or a CDS string parameter.

If Input\_Value is an array, the CL is aborted. To convert an array each element must be converted separately.

Suffix is an optional value appended to the field. Normally, suffix is a single delimiter such as a comma or a space, but any string is allowed.

Character\_Ptr is the character-offset from the beginning of the string to the beginning of the field to be written. It is returned with the position of the character after the written field. Normally, the Character\_Ptr value returned on previous calls is passed on subsequent calls, but Character\_Ptr can be set to any position in the string.

The length of Output\_String is updated each time a Put\_Field\_S causes Output\_String to be extended. Note that it is recommended that Output\_String be initialized with a null string (i.e., "") to set its length to zero before putting the first field into it.

Character\_Ptr can be set to point beyond the current length of the string and the string will be blank filled from the current end of the string to the character preceding the one pointed to by Character\_Ptr. Character\_Ptr may point at any character in the string and that character, and subsequent characters, will be overwritten with the new converted value.

Format is a specification of how the external value will appear after conversion. The default format specification for the various data types are:

number	R-ZZZZ9.99
logical	TEXTL1:5
string	TEXTL1:10
enumeration	TEXTL1:10
entity (point id)	TEXTL1:10
time	DATE MM-DD-YY ENDATE

Note: Do not use the default time format for date if you plan to read the field with the function File\$Get\_Field\_S. Use DATEDD AM:3 YYENDDATE (see subsection C.3 and C.6.8).



Use the "File\$Convert\_SDE\_To\_ASCII" subroutine to convert a self-defined enumeration to a string. The string then can be transferred by this subroutine.

**Possible Return\_Status Values for this function:**

- 0.0 — Good status
- 15.0 — Value has invalid data type
- 16.0 — Format String is invalid (for example, "TEXTL1:133" exceeds the maximum 132)
- 17.0 — Conversion of a value failed
- 18.0 — Error returned on Get/Store parameter; specific error in Data Access status
- 24.0 — Attempt to write beyond end of Output\_String (for example, writing beyond the 78th position of a CL local string)
- 25.0 — Reference list number invalid
- 40.0 — Invalid character pointer (calling the function with Character\_Ptr outside the allowable range (for example, outside 1-78 for CL local strings))

**Examples:**

The following CL code segment demonstrates the functions File\$Put\_Field\_S and File\$Get\_Field\_S. It puts 3 fields into a CDS string and then reads them back. The first field will be the name of the Entity referred to by the parameter "psi." It will start at character position 1 of the string and will be right justified in a field 20 characters long with a blank following it in character position 21.

The second field will be a number. It will start at character position 22, be right justified, and will be 6 characters long with a blank following it in character position 28. The value will be read from the PV on the point referred to by the parameter PSI.

The third field will be a time value. It will start at character position 29 and will be 8 characters long with a blank following it in character position 37. After reading back the three fields, the program outputs the values in a message. The message will look similar to the following after this code segment is executed:

```
AMTEST1  1.00000  15:11:40
```

Also refer to subsection C.6.8 for another example of the use of the File\$Put\_Field\_S and File\$Get\_Field\_S functions.

```

PACKAGE

CUSTOM
  PARAMETER psi          : $REG_CTL
  PARAMETER psi_data    : STRING
END CUSTOM

BLOCK testgps (POINT jims_pt; at BACKGRND)

  %INCLUDE_SET file

  LOCAL status,                -- Return status
&    da_status,                -- Data Access Status
&    offset,                   -- Character offset of field
&    psi_pv      : NUMBER      -- Variable to get psi.pv into
  LOCAL format,                -- Field format specifier
&    delimiter   : STRING      -- Field delimiter character
  LOCAL cur_time,              -- Current time
&    ct          : TIME        -- Variable to get time into

  SET format      = "TIMEHH:MM:SSSENDTIME"
  SET delimiter   = " "
  SET offset      = 1
  SET cur_time    = date_time

  CALL File$Put_Field_S (status, da_status, offset, psi_data,
&    psi, "TEXT1:20", delimiter)
  CALL File$Put_Field_S (status, da_status, offset, psi_data,
&    psi.pv, "RZZ9.99", delimiter)
  CALL File$Put_Field_S (status, da_status, offset, psi_data,
&    cur_time, format, delimiter)

  SET offset      = 1

  CALL File$Get_Field_S (status, da_status, offset, psi_data, psi, 0)
  CALL File$Get_Field_S (status, da_status, offset, psi_data, psi_pv, 0)
  CALL File$Get_Field_S (status, da_status, offset, psi_data, ct, 0)

  SEND: psi.name, psi_pv, ct

END testgps

END PACKAGE

```

## C.6.5 File\$Write\_Record Subroutine

This routine writes a file's current record to that file.

```
SUBROUTINE File$Write_Record
  (Return_Status      : OUT NUMBER;      -- File R/W package error status
   File_Manager_Status : OUT NUMBER;      -- File Manager error status
   File_Number        : IN  NUMBER;      -- File id from Open_File call
   Record_Number      : IN OUT NUMBER) -- Record number to write
```

This subroutine can only be called from a CL block linked to a background insertion point.

Record\_Number specifies the record to be written or overwritten. If Record\_Number is zero, the record is appended to the end of the file. On appends, the number of the record written is returned in Record\_Number. If the record pointed to by Record\_Number already exists, the new record must have the same record length as the existing record. The record is not altered in any way if an error occurs during a write (the data in the record is not erased).

File\_Number must be the value returned on a previous Open\_File call.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 6.0 — Device error while reading file
- 10.0 — No more space on the volume
- 11.0 — Read or write beyond end of file
- 12.0 — Other File Manager error
- 13.0 — File number is invalid
- 20.0 — Can't read file control table pointer
- 31.0 — Invalid record number
- 34.0 — Number not assigned to this CL Block—access denied

### Examples:

The following CL code segment appends the current record to the file associated with file\_id.

```
LOCAL status,          -- Return status
&      fm_status,      -- File Manager Status
&      file_id,        -- File Identifier
&      rec_number : NUMBER -- Record number

SET rec_number = 0

CALL File$Write_Record (status, fm_status, file_id, rec_number)
```

## C.6.6 File\$Read\_Record Subroutine

This routine reads a record from a file to the file's current record.

```
SUBROUTINE File$Read_Record
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   File_Manager_Status : OUT NUMBER; -- File Manager error status
   File_Number        : IN  NUMBER; -- File id from Open_File call
   Record_Size        : OUT NUMBER; -- Characters in read record
   Record_Number      : IN  NUMBER) -- Number of record to read
```

This subroutine can only be called from a CL block linked to a background insertion point.

Read\_Record reads the record specified by Record\_Number from the file indicated by File\_Number. File\_Number must be the value returned on a previous Open\_File call.

Record\_Size is returned with the number of characters in the record just read.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 6.0 — Device error while reading file
- 11.0 — Read or write beyond end of file
- 12.0 — Other File Manager error
- 13.0 — File number is invalid
- 20.0 — Can't read file control table pointer
- 31.0 — Invalid record number
- 34.0 — Number not assigned to this CL Block—access denied
- 35.0 — Maximum record length exceeded
- 36.0 — Can't access file (due to HM failure, etc.)

### Examples:

The following CL code segment reads the third record from the file associated with file\_id into the current record.

```
LOCAL status,          -- Return status
&      fm_status,      -- File Manager Status
&      file_id,        -- File Identifier
&      rec_size  : NUMBER -- Number of characters in record

CALL File$Read_Record (status, fm_status, file_id, rec_size, 3)
```

### C.6.7 File\$Get\_Field Subroutine

This routine reads values from fields in the specified file's current record.

```
SUBROUTINE File$Get_Field
  (Return_Status : OUT NUMBER;      -- File R/W package error status
   DA_Status     : OUT NUMBER;      -- Data Access error status
   Character_Ptr  : IN OUT NUMBER;   -- Character offset to the field
   File_Number    : IN  NUMBER;      -- File number from Open_File call
   Output_Value   : OUT cl_type;     -- Variable of any valid CL data type
                                     -- excepting self-defined enumeration
   Field_Width    : IN  NUMBER)      -- Width of field (for output
                                     -- strings)
```

This subroutine can only be called from a CL block linked to a background insertion point.

Get\_Field extracts an external value from a field in the current record of the specified file, converts it to internal form and stores it into Output\_Value. Character\_Ptr must point to the beginning of the field. Anticipated field delimiters are: Space, Comma, Left Parenthesis, Right Parenthesis, Colon, Semicolon, and Carriage Return.

The following applies only to Output\_Values not of type "string":

Get\_Field scans the current record from left to right skipping any leading blank characters and extracts characters until it finds a delimiter.

After the extracted external value is converted to internal form and stored in Output\_Value, Character\_Ptr is adjusted to point to the character after the delimiter .

For Output\_Values of data type "string," the following special rules apply:

Field\_Width specifies the number of consecutive characters that will be moved into Output\_Value. Any field delimiters found among these characters are considered to be part of the string. Character\_Ptr is returned pointing at the character immediately following the last character transferred. (Please note that, depending on the record structure, Character\_Ptr could wind up pointing to data, or to a field delimiter, or it could be pointing outside the record).

If File\$Strip\_Blanks (see heading C.6.26) was last called with Strip set to "ON" (or if File\$Strip\_Blanks was never called by this program), any leading blank characters are skipped over and the count of characters to be transferred starts at the first non-blank character. If only blanks remain in the current record, Return\_Status is set to 14.0.

If File\$Strip\_Blanks was last called with Strip set to "OFF," the character pointed to by Character\_Ptr (blank or not) becomes the first of the specified number of characters to be moved to Output\_Value.

If the extracted string is a self-defined enumeration, you can convert it to internal form by using the "File\$Convert\_SDE\_To\_Internal" subroutine.

File\_Number must be the value returned on a previous Open\_File call.

For values of type TIME, the fourth through sixth characters of the input field are used to determine if the value is a date or a time. If these characters are alphabetic (abbreviated name of a month), the input is assumed to be of type date, otherwise the input is assumed to be of type time. For input fields of type "Date," the field must contain a value in the form DD MMM YY, where DD is 01-31, MMM is the first 3 letters of the month in upper case letters, and YY is 00-99. For input fields of type "Time," the input field must contain a value in the form HH:MM:SS, where HH is 01-24, MM is 00-59, and SS is 00-59.

In order to create a Time value that includes the date and a time on that date, two separate fields must be extracted from the record. The first field contains the date which is read into one time variable, the second field contains a time which is read into another time variable; the two variables then can be added to get a combined date and time. See C.6.8 for an example of combining the date and time variables.

**Possible Return\_Status Values for this function:**

- 0.0 — Good status
- 13.0 — File number is invalid
- 14.0 — Attempt to read/write field beyond end of buffer
- 15.0 — Value has invalid data type
- 17.0 — Conversion of a value failed
- 18.0 — Error returned on Get/Store parameter; specific error in Data Access status
- 20.0 — Can't read file control table pointer
- 25.0 — Reference list number invalid
- 34.0 — Number not assigned to this CL Block—access denied
- 39.0 — Invalid string width in Get\_Field or Get\_Field\_S
- 40.0 — Invalid character pointer

**Examples:**

The following CL code segment gets 3 fields from the current record—an entity name (for CDS parameter psi), a parameter value (psi.pv), and a time value, as shown below:

```
PSI_003  85.33 14:34:36
```

See C.6.3 for an example of how these fields were written. The following code uses a local variable (psi\_pv) to receive the value of psi.pv from the record. Using psi.pv to receive the value would result in a write to the on-line data value psi.pv.

```
CUSTOM
  PARAMETER psi : $REG_CTL
END CUSTOM

  LOCAL status,           -- Return status
&      da_status,        -- Data Access Status
&      file_id,          -- File Identifier
&      offset,           -- Character offset of field
&      psi_pv : NUMBER   -- Variable to read psi.pv into
  LOCAL cur_time : TIME   -- Current time

  SET offset      = 1

  CALL File$Get_Field (status, da_status, offset, file_id, psi, 0)
  CALL File$Get_Field (status, da_status, offset, file_id, psi_pv, 0)
  CALL File$Get_Field (status, da_status, offset, file_id, cur_time, 0)
```

### C.6.8 File\$Get\_Field\_S Subroutine

This routine reads values from a string variable.

```
SUBROUTINE File$Get_Field_S
  (Return_Status : OUT NUMBER;      -- File R/W package error status
   DA_Status     : OUT NUMBER;      -- Data Access error status
   Character_Ptr  : IN OUT NUMBER;   -- Character offset to the field
   Input_String   : IN  STRING;      -- String to read field from
   Output_Value   : OUT cl_type;     -- Variable of any valid CL data type
                                     -- excepting self-defined enumeration
   Field_Width    : IN  NUMBER)     -- Field width (for output strings)
```

This subroutine can be called from CL blocks linked to any insertion point.

Input\_String can be either a local CL string or a CDS string parameter.

Get\_Field\_S extracts an external value from a field within an Input\_String, converts it to internal form and stores it into Output\_Value. Character\_Ptr must point to the beginning of the field. Get\_Field\_S scans Input\_String from left to right skipping any leading blank characters and extracts characters until it finds a delimiter.

After the extracted string is converted to internal form and stored in Output\_Value, Character\_Ptr is adjusted to point to the character after the delimiter.

If the extracted string is a self-defined enumeration, you can convert it to internal form by using the "File\$Convert\_SDE\_To\_Internal" subroutine.

For output\_values of data type "string," the width of the field is specified by Field\_Width. The string starts with the first non-blank character in the field and Character\_Ptr is adjusted to point to the character after the number of characters specified by Field\_Width. Note that leading blanks on a string can cause problems since blanks are used both as delimiters and for fill characters; therefore, correct field length is important.

Anticipated field delimiters are:

- Space
- Comma
- Left Parenthesis
- Right Parenthesis
- Colon
- Semicolon
- Carriage Return

For values of type TIME, the fourth through sixth characters of the input field are used to determine if the value is a date or a time. If these characters are alphabetic (abbreviated name of a month), the input is assumed to be of type date, otherwise, the input is assumed to be of type time. For input fields of type "Date," the field must contain a value in the form DD MMM YY, where DD is 01-31, MMM is the first 3 letters of the month in upper case letters, and YY is 00-99. For input fields of type "Time," the input field must contain a value in the form HH:MM:SS, where HH is 01-24, MM is 00-59, and SS is 00-59.

NOTE: To create a Time value that includes the date and a time on that date, two separate fields must be extracted from the record. The first contains the date which is read into one time variable, the second contains a time which is read into another time variable and then these two variables can be added to get a date with a time.

**Possible Return\_Status Values for this function:**

- 0.0 — Good status
- 15.0 — Value has invalid data type
- 17.0 — Conversion of a value failed
- 18.0 — Error returned on Get/Store parameter; specific error in Data Access status
- 24.0 — Attempt to read beyond end of Input\_String (for example, reading beyond the 78th position of a CL local string)
- 25.0 — Reference list number invalid
- 39.0 — Invalid string width in Get\_Field or Get\_Field\_S
- 40.0 — Invalid character pointer (calling the function with Character\_Ptr outside the allowable range; for example, outside 1-78 for CL local strings)



**Examples:**

The following CL code segment illustrates the creation of a TIME variable that includes the date and time. The program outputs a message similar to the following:

```
Date and Time are: 29Nov92  08:57:11
```

Also see subsection C.6.4 for another example of the use of the File\$Put\_Field\_S and File\$Get\_Field\_S functions.

```
-- Note: Compile with Expanded Time Display (-ETD) Option
```

```
BLOCK timedt (POINT jims_pt; at BACKGRND)
```

```
%INCLUDESET file

LOCAL status,                -- Return status
&    da_status,              -- Data Access Status
&    offset : NUMBER         -- Character offset of field
LOCAL format,                -- Field format specifier
&    delimiter,              -- Field delimiter character
&    temp : STRING           -- String to put and get fields to/from
LOCAL t, t1, t2, t3 : TIME    -- Time variables

SET format      = "DATEDD AM:3 YYENDDATE"
SET delimiter    = ";"
SET offset      = 1          -- Set pointer to start of record
SET t           = date_time  -- Get current date in t

CALL File$Put_Field_S (status, da_status, offset, temp, t,
&    format, delimiter) -- Put date field in record "temp"

SET format      = "TIMEHH:MM:SSENDTIME"
SET t           = now        -- Get current time in t

CALL File$Put_Field_S (status, da_status, offset, temp, t,
&    format, delimiter) -- Put time field in record "temp"

SET offset      = 1          -- Reset pointer to start of record

CALL File$Get_Field_S (status, da_status, offset, temp, t1, 0)
-- Get date field in t1
CALL File$Get_Field_S (status, da_status, offset, temp, t2, 0)
-- Get time field in t2
SET t3 = t1 + t2            -- Combine date and time in t3

SEND: "Date and Time are: ",t3 -- Send date and time message

END timedt
```

### C.6.9 File\$Close\_File Subroutine

This routine closes a previously opened file.

```
SUBROUTINE File$Close_File
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   File_Manager_Status : OUT NUMBER; -- File Manager error status
   File_Number        : IN NUMBER)  -- File id from Open_File
```

This subroutine can only be called from a CL block linked to a background insertion point.

File\_number is the value returned by the Open\_File call.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 6.0 — Device error while reading file
- 12.0 — Other File Manager error
- 13.0 — File number is invalid
- 20.0 — Can't read file control table pointer
- 34.0 — Number not assigned to this CL Block—access denied

#### Examples:

The following CL code segment closes a file.

```
LOCAL status,          -- Return status
&    fm_status,        -- File Manager status
&    file_id : NUMBER -- File Identifier

CALL File$Close_File (status, fm_status, file_id)
```

## C.6.10 File\$Copy\_Record Subroutine

This routine copies the current record associated with one open file to the current record associated with another open file.

```
SUBROUTINE File$Copy_Record
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   Source_File        : IN  NUMBER; -- File id from Open_File
   Destination_File    : IN  NUMBER) -- File id from Open_File
```

This subroutine can only be called from a CL block linked to a background insertion point.

Copy\_Record makes a copy of the current record (see heading C.2.7 for the definition of current record) of Source\_File and puts it in the current record of Destination\_File. Any data in the current record of Destination\_File is overwritten. Note that a File\$Write\_Record of the current record of Destination\_File must be done to get the record out to the file.

If the memory allocated to the current record of Destination\_File is insufficient to hold the contents of the current record of Source\_File, the current\_record of Destination\_File is filled with as much of the current record of source\_file as will fit and Return\_Status will be 32.0 (Data placed in destination file record was truncated).

Source\_File and Destination\_File must be the values returned on previous Open\_File calls.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 13.0 — File number is invalid
- 20.0 — Can't read file control table pointer
- 32.0 — Record was truncated
- 34.0 — Number not assigned to this CL Block—access denied

### Examples:

The following CL code segment copies the current record associated with file\_id1 to the current record associated with file\_id2.

```
LOCAL status,          -- Return status
&    fm_status,        -- File Manager status
&    num_records,      -- Number of records in the file
&    file_id1,         -- Source file identifier
&    file_id2,         -- Destination file identifier
&    rec_size : NUMBER -- Number of characters in record
LOCAL path : STRING

SET path = "net>test>file.x"
CALL File$Open_File (status, fm_status, file_id1, num_records,
&                  132, path, 0)

SET path = "net>test>file2.x"
CALL File$Open_File (status, fm_status, file_id2, num_records,
&                  80, path, 0)

CALL File$Read_Record (status, fm_status, file_id1, rec_size, 3)
CALL File$Copy_Record (status, file_id1, file_id2)
CALL File$Write_Record (status, fm_status, file_id2, 3)
```

## C.6.11 File\$Copy\_File Subroutine

This routine makes a copy of an existing file.

```
SUBROUTINE File$Copy_File
  (Return_Status      : OUT NUMBER; File R/W package error status
   File_Manager_Status : OUT NUMBER; File Manager error status
   Offender           : OUT NUMBER; File to which status pertains
   Source_File_Path    : IN  STRING; Pathname of source file.
   Destination_File_Path : IN  STRING) Pathname of destination file.
```

This subroutine can only be called from a CL block linked to a background insertion point.

Offender specifies the file to which the status fields refer (0=source, 1=destination, 2=temporary, 3=status is not specific to a file).

This call is used to copy one file and its attributes (e.g., protection, max\_record\_length) to another file. If the destination file does not exist, it is created by the file manager. If the destination file exists, it is deleted, then recreated with the attributes of the source file and the source file is copied into it. The source file is opened with "shared read no write." The destination file is opened with "exclusive read/write." If the source file contains no records, the destination file is created with no records.

This subroutine can be used to copy a file from one LCN to another LCN. When either the source or destination is on a remote LCN, that file must be on a History Module.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 3.0 — File privilege violation
- 4.0 — File already exists (new file)
- 5.0 — File does not exist (old file)
- 6.0 — Device error while reading file
- 7.0 — File table for the specified volume is full
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 10.0 — No more space on the volume
- 12.0 — Other File Manager error
- 22.0 — Maximum number of files already open
- 28.0 — File already open
- 29.0 — File reserved for another operation
- 35.0 — Maximum record length exceeded
- 36.0 — Can't access file (due to HM failure, etc.)

**Examples:**

The following CL code segment copies the file associated with file\_id1 to the file associated with file\_id2 on the remote LCN named "am."

```
LOCAL status,          -- Return status
&      fm_status,      -- File Manager return status
&      culprit        : NUMBER -- which file id caused error
LOCAL file_id1,        -- File Identifier
&      file_id2       : STRING -- File Identifier

SET file_id1 = "net>test>origfile.x"
SET file_id2 = "am\net>test>copyfile.x"
CALL File$Copy_file (status, fm_status, culprit, file_id1, file_id2)
```

## C.6.12 File\$Exists Subroutine

This routine determines if a given file exists.

```
SUBROUTINE File$Exists
  (Return_Status      : OUT NUMBER;  -- File R/W package error status
   File_Manager_Status : OUT NUMBER;  -- File Manager error status
   Path               : IN  STRING;   -- Name of file to search for.
   File_Exists        : OUT LOGICAL) -- Indicates if file exists.
```

This subroutine can only be called from a CL block linked to a background insertion point.

The Exists subroutine determines whether the file named in Path exists. If it exists, File\_Exists is ON; if it does not exist File\_Exists is OFF. Question marks in the file name in Path are illegal.

This subroutine can be used to determine if a file exists on another LCN. In this case, the file to be checked for must be on a History Module.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error while reading file
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 28.0 — File already open
- 29.0 — File reserved for another operation
- 36.0 — Can't access file (due to HM failure, etc.)

### Examples:

The following CL code segment determines if the file "net>test>test.x" exists.

```
LOCAL status,          -- Return status
&   fm_status : NUMBER  -- File Manager return status
LOCAL path      : STRING -- Name of the file
LOCAL file_exist : LOGICAL -- Indicates existence of file

SET path = "net>test>test.x"

CALL File$Exists (status, fm_status, path, file_exist)
```

### C.6.13 File\$Find\_Field\_Pointer Subroutine

This routine finds a field in the specified file's current record.

```
SUBROUTINE File$Find_Field_Pointer
  (Return_Status   : OUT NUMBER; -- File R/W package error status
   Character_Ptr   : OUT NUMBER; -- Character offset to field
   Field_Number    : IN  NUMBER; -- Number of the requested field
   File_Number     : IN  NUMBER; -- File id from Open_File call
   Field_Delimiter : IN  STRING;  -- Character that separates fields
   Escape_Marker   : IN  STRING) -- Character to jump over on scan
```

This subroutine can only be called from a CL block linked to a background insertion point.

Character\_Ptr is returned with the character position of the beginning of the field specified by Field\_Number. Note that if the requested field is empty (i.e., two consecutive field delimiters) the pointer will be set to the character position of the first delimiter. A request via File\$Get\_Field to get the contents of an empty field results in a conversion error.

Field\_Delimiter is the character used to separate fields in the current record of the file specified by File\_Number. If Field\_Delimiter does not occur in the current record, Return\_Status will be 14.0.

Escape\_Marker is an optional value that specifies a character which is used to bracket a string of characters which will not be searched for Field\_Delimiter.

Only the first character of Field\_Delimiter or Escape\_Marker is used.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 13.0 — File number is invalid
- 14.0 — Attempt to read/write field beyond end of buffer
- 20.0 — Can't read file control table pointer
- 30.0 — Invalid field number
- 34.0 — Number not assigned to this CL Block—access denied

**Examples:**

Given this record

```
123,5.6," d,d,d","5
```

the following code segment varies the values of Field\_Number, and Escape\_Marker (Field\_Delimiter is unchanged) to produce different returned values for Character\_Ptr as follows:

Field_Number	Escape_Marker	Character_Ptr
1	none	1
2	none	5
2	"	5
4	"	19
4	none	13

```

LOCAL status,          -- Return status
& char_ptr,           -- Character pointer
& file_id : NUMBER    -- File identifier
LOCAL delimiter,      -- Field delimiter character
& escape : STRING     -- Character to mark escape

SET delimiter = ","
SET escape = ""

CALL File$Find_Field_Pointer (status, char_ptr, 1, file_id, delimiter,
& escape)
CALL File$Find_Field_Pointer (status, char_ptr, 2, file_id, delimiter,
& escape)
SET escape = ""
CALL File$Find_Field_Pointer (status, char_ptr, 2, file_id, delimiter,
& escape)
CALL File$Find_Field_Pointer (status, char_ptr, 4, file_id, delimiter,
& escape)
SET escape = ""
CALL File$Find_Field_Pointer (status, char_ptr, 4, file_id, delimiter,
& escape)

```



### C.6.14 File\$Find\_Field\_Pointer\_S Subroutine

This routine finds a field in a string.

```
SUBROUTINE File$Find_Field_Pointer_S
  (Return_Status   : OUT NUMBER; -- File R/W package error status
   Character_Ptr    : OUT NUMBER; -- Character offset to field
   Field_Number     : IN  NUMBER; -- Number of the requested field
   Input_String     : IN  STRING;  -- String to be scanned
   Field_Delimiter  : IN  STRING;  -- Character that separates fields
   Escape_Marker    : IN  STRING) -- Character to jump over on scan
```

This subroutine can be called from CL blocks linked to any insertion point.

Character\_Ptr is returned with the character position of the beginning of the field specified by Field\_Number. Note that if the requested field is empty (i.e., two consecutive field delimiters) the pointer will be set to the character position of the first delimiter. A request via File\$Get\_Field\_S to get the contents of an empty field results in a conversion error.

Field\_Delimiter is the character used to separate fields in the Input\_String.

Escape\_Marker is an optional value that specifies a character which is used to bracket a string of characters which will not be searched for Field\_Delimiter.

Only the first character of Field\_Delimiter or Escape\_Marker is used.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 24.0 — Attempt to read/write field beyond end of string
- 30.0 — Invalid field number

**Examples:**

Given this value in Input\_String

123,5.6," d,d,d," ,5

the following code segment varies the values of Field\_Number, and Escape\_Marker (Field\_Delimiter is unchanged) to produce different returned values for Character\_Ptr as follows:

Field_Number	Escape_Marker	Character_Ptr
1	none	1
2	none	5
2	"	5
4	"	19
4	none	13

```

LOCAL status,                -- Return status
&   char_ptr   : NUMBER      -- Character pointer
LOCAL delimiter,            -- Field delimiter character
&   input,      -- String to locate fields in
&   escape      : STRING     -- Character to mark escape

SET delimiter = ","
SET escape    = ""
SET offset    = 1

CALL File$Find_Field_Pointer_S (status, char_ptr, 1, input, delimiter,
&                               escape)
CALL File$Find_Field_Pointer_S (status, char_ptr, 2, input, delimiter,
&                               escape)
SET escape = ""
CALL File$Find_Field_Pointer_S (status, char_ptr, 2, input, delimiter,
&                               escape)
CALL File$Find_Field_Pointer_S (status, char_ptr, 4, input, delimiter,
&                               escape)
SET escape = ""
CALL File$Find_Field_Pointer_S (status, char_ptr, 4, input, delimiter,
&                               escape)

```

### C.6.15 File\$Set\_Null Subroutine

This routine sets the length of the current record to zero.

```
SUBROUTINE File$Set_Null
  (Return_Status : OUT NUMBER; -- File R/W package error status
   File_Number   : IN  NUMBER) -- File id from Open_File call.
```

This subroutine can only be called from a CL block linked to a background insertion point.

This call sets the current record length to 0. The maximum record length specified on the Open\_File call is not affected.

File\_Number is the value returned from an Open\_File call.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 13.0 — File number is invalid
- 20.0 — Can't read file control table pointer
- 34.0 — Number not assigned to this CL Block—access denied

#### Examples:

The following CL code segment sets the current record for the file associated with file\_id to empty.

```
LOCAL status,           -- Return status
&    file_id   : NUMBER -- File identifier

CALL File$Set_Null (status, file_id)
```

## C.6.16 File\$Blank\_Fill Subroutine

This routine places a string of blank characters into the specified file's current record.

```
SUBROUTINE File$Blank_Fill
  (Return_Status      : OUT NUMBER;      -- File R/W package error status
   Character_Ptr       : IN OUT NUMBER;   -- Offset at which to start fill
   Fill_to_Character   : IN  NUMBER;      -- Offset at which to end fill
   File_Number         : IN  NUMBER)      -- File id from Open_File call.
```

This subroutine can only be called from a CL block linked to a background insertion point.

This call fills the current record of the file specified by File\_Number with blanks from the Character\_Ptr position up to, and including, the Fill\_To\_Character position.

If the Character\_Ptr is equal to, or larger than, the Fill\_to\_Character, nothing is done.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 13.0 — File number is invalid
- 14.0 — Attempt to read/write field beyond end of buffer
- 20.0 — Can't read file control table pointer
- 34.0 — Number not assigned to this CL Block—access denied
- 40.0 — Invalid character pointer

### Examples:

The following CL code segment fills the current record for the file associated with file\_id from character 10 to character 14 with blanks.

```
LOCAL status,          -- Return status
&    offset,           -- Where to start fill
&    stop,             -- Where to stop fill
&    file_id  : NUMBER -- File Identifier

SET offset = 10
SET stop   = 14

CALL File$Blank_Fill (status, offset, stop, file_id)
```

## C.6.17 File\$Blank\_Fill\_S Subroutine

This routine places a string of blank characters into a string.

```
SUBROUTINE File$Blank_Fill_S
  (Return_Status      : OUT NUMBER;      -- File R/W package error status
   Character_Ptr       : IN  OUT NUMBER; -- Offset at which to start fill
   Fill_to_Character   : IN  NUMBER;      -- Offset at which to end fill
   Output_String       : IN  OUT STRING) -- String to place blanks in
```

This subroutine can be called from CL blocks linked to any insertion point.

This call fills the string Output\_String with blanks from the Character\_Ptr position up to, and including, position specified by Fill\_to\_Character.

If the Character\_Ptr is equal to, or larger than, the Fill\_to\_Character, nothing is done.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 24.0 — Attempt to read/write field beyond end of string
- 40.0 — Invalid character pointer

### Examples:

The following CL code segment fills the string "blankety" from character 10 to character 14 with blanks.

```
LOCAL status,          -- Return status
& offset,             -- Where to start fill
& stop                : NUMBER -- Where to stop fill
LOCAL blankety        : STRING -- String to blank fill

SET offset = 10
SET stop   = 14

CALL File$Blank_Fill_S (status, offset, stop, blankety)
```

## C.6.18 File\$Rename Subroutine

This routine changes the name of an existing file.

```
SUBROUTINE File$Rename
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   File_Manager_Status : OUT NUMBER; -- File Manager error status
   Old_File_Path      : IN  STRING; -- Existing file's name
   New_Name            : IN  STRING; -- New file name
   New_Extension       : IN  STRING) -- New file extension
```

This subroutine can only be called from a CL block linked to a background insertion point.

Rename\_File changes the name of the file specified by Old\_File\_Path to the name specified by New\_Name and New\_Extension. The file must be closed and unprotected or an error code is returned.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 3.0 — File privilege violation
- 4.0 — File already exists (new file)
- 5.0 — File does not exist (old file)
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 26.0 — Bad file name
- 27.0 — Bad file extension
- 28.0 — File already open
- 29.0 — File reserved for another operation

### Examples:

The following CL code segment renames the file "net>test>oldfile.ls" to "net>test>rename.x."

```
LOCAL status,          -- Return status
&   fm_status  : NUMBER -- File Manager return status
LOCAL cur_name,        -- Current name of file
   new_ext      : STRING -- New file name extension

SET cur_name = "net>test>oldfile.ls"
SET new_ext  = "x"

CALL File$Rename (status, fm_status, cur_name, "rename", new_ext)
```

## C.6.19 File\$Safe\_Rename Subroutine

This routine takes the name of an existing file and gives it to a new file of the same type.

```
SUBROUTINE File$Safe_Rename
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   File_Manager_Status : OUT NUMBER; -- File Manager error status
   Old_File_Path      : IN  STRING;  -- File whose identity is stripped
   Temporary_Name      : IN  STRING;  -- Name of file to be renamed
   Temporary_Extension : IN  STRING) -- Extension of file to be renamed
```

This subroutine can only be called from a CL block linked to a background insertion point.

Safe\_Rename gives the full pathname of the original file (specified by Old\_File\_Path) to the Temporary File (Temporary\_Name.Temporary\_Extension) and deletes the original file. Both files must be unprotected, on the same volume and closed.

This method of renaming a file ensures that the file is not corrupted if a remote node fails in the middle of a rename operation.

To use this function, first create a Temporary File (see heading C.2.11) which holds the contents of the original file, then modify the temporary file. Once the modifications are complete, invoke the Safe\_Rename routine to complete the operation.

If the history module fails during the safe rename operation, two files with the same name could exist. During startup of the history module the file manager checks for duplicate file names and deletes the one with the oldest time stamp.

If the node which initiated the safe rename function fails before the safe rename operation is complete, the temporary file is renamed and the original file is deleted.

If the Old\_File\_Path file or the Temporary\_Name file is open when this subroutine is called, a File Reserved for Another Operation error code is returned.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 3.0 — File privilege violation
- 5.0 — File does not exist (old file)
- 6.0 — Device error while reading file
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 22.0 — Maximum number of files already open
- 26.0 — Bad file name
- 27.0 — Bad file extension
- 29.0 — File reserved for another operation

**Examples:**

The following CL code segment first makes a copy of the file `net>test>oldfile.ls`, naming it `net>test>tempfile.__`, then safely renames the temporary file to `net>test>oldfile.ls`. Note that the `COPY_FILE` subroutine uses full pathnames to identify both the original and the temporary file, while `SAFE_RENAME` uses the full pathname to identify the file giving up its identity, but uses only the name and extension to identify the temporary file (which must be on the same volume as the file giving up its identity).

```

LOCAL status,           -- Return status
&    fm_status,         -- File Manager return status
&    culprit    : NUMBER -- Which file id caused error on copy
LOCAL cur_name,         -- Pathname of file to give up its identity
&    temp_name,         -- Pathname of temporary file
&    temp_ext  : STRING -- Extension for temporary file

SET cur_name = "net>test>oldfile.ls"
SET temp_name = "net>test>tempfile.__"
SET temp_ext  = "__"

CALL File$Copy_File (status, fm_status, culprit, cur_name, temp_name)

-- Add code here to ensure that call was successful, and to modify
-- the temporary file as required.

CALL File$Safe_Rename (status, fm_status, cur_name, "tempfile",
&                      temp_ext)
```



## C.6.20 File\$Protect\_File Subroutine

Use this routine to protect the named file from deletion or to unprotect it (allow deletion by the built-in subroutine Delete\_File).

```
SUBROUTINE File$Protect_File
  (Return_Status      : OUT NUMBER;  -- File R/W package error status
   File_Manager_Status : OUT NUMBER;  -- File Manager status
   Path               : IN  STRING;   -- Name of file to protect
   Protect             : IN  LOGICAL) -- ON = protect, OFF = unprotect
```

This subroutine can only be called from a CL block linked to a background insertion point.

Protect\_File changes the protection status of the file name in Path. When a file is protected, it cannot be deleted. The file must be closed before calling this subroutine or a "File Already Open" error code is returned.

If Protect contains ON, the file will be protected. If Protect contains OFF, the file will be unprotected. If Protect is ON and the file is already protected, then nothing is done. If Protect is OFF and the file is already unprotected, then nothing is done.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 3.0 — File privilege violation
- 5.0 — File does not exist
- 6.0 — Device error while reading file
- 12.0 — Other File Manager error
- 28.0 — File already open
- 29.0 — File reserved for another operation
- 36.0 — Can't access file (due to HM failure, etc.)

### Examples:

The following CL code segment protects the file "net>test>protect.ls" from being deleted.

```
LOCAL status,          -- Return status
&   fm_status  : NUMBER -- File Manager return status
LOCAL file_name : STRING -- Name of file to protect

SET file_name = "net>test>protect.ls"

CALL File$Protect_File (status, fm_status, file_name, ON)
```

## C.6.21 File\$Convert\_SDE\_to\_Internal Subroutine

This routine converts a self-defined enumeration state name to its internal form.

```
SUBROUTINE File$Convert_SDE_to_internal
  (Status      : OUT NUMBER;          -- File R/W package error status
   State_Name  : IN  STRING;          -- State Name from an SDE
   Param_Name  : IN  cl_type;         -- Name of SDE parameter
   SDE_Value   : OUT cl_type)         -- Ordinal value of state name
```

This subroutine can only be called from a CL block linked to a background insertion point.

This routine converts State\_Name, the name of a state from a self-defined enumeration (SDE), to its ordinal value and stores it in SDE\_Value. Param\_Name must be a parameter of SDE type that contains a state with the same name as the value for State\_Name, or the CL block is aborted with a PROGERR.

SDE\_Value can be a CL local variable or a parameter. Its data type can be Number, Enumeration, or Self-Defined Enumeration. If it is not one of these types, the CL block is aborted with a PROGERR.

If SDE\_Value is of type Number, the ordinal value associated with State\_Name is stored in it. For instance, if Param\_Name specifies a parameter of type Self-Defined Enumeration with states red/green, and State\_Name is green, then SDE\_Value is 1.0 (the ordinal value of the first state in an enumeration is zero).

If SDE\_Value is of data type Enumeration or Self-Defined Enumeration, the value to be stored in it is the state which is in the ordinal position corresponding to the ordinal position of the value State\_Name in the enumeration defined by Param\_Name. For instance, if Param\_Name specifies a parameter of self-defined enumeration with states of red/green, the State\_Name value is green, and SDE\_Value identifies an enumeration with state names of dog/cat, then the value returned in SDE\_Value is cat.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 18.0 — Error returned on Get/Store parameter
- 19.0 — Error on Data Access conversion (bad state name or bad point.parameter)

### Examples:

The following CL code segment converts the state name "badpv" to its ordinal value. If hg050602.PV is defined as having states badpv/state1/state2, then ordinal\_value will contain the value 0.0 after the call.

```
LOCAL status,                -- Return status
      ordinal_value : NUMBER -- Numeric value of the state

EXTERNAL hg050602
CALL File$Convert_SDE_to_Internal (status, "BADPV", hg050602.pv,
&                                ordinal_value)
```

## C.6.22 File\$Convert\_SDE\_To\_ASCII Subroutine

This routine converts an internal value of type Self-Defined Enumeration to external form.

```
SUBROUTINE File$Convert_SDE_to_ASCII
  (Status      : OUT NUMBER;      -- File R/W package error status
   State_Name   : OUT STRING;      -- State name from an SDE
   Param_Name   : IN  cl_type;     -- Parameter of type SDE
   SDE_Value    : IN  cl_type)     -- Ordinal value of state name
```

This subroutine can only be called from a CL block linked to a background insertion point.

This subroutine converts sde\_value to its state name.

Param\_Name must be a parameter of SDE type that contains at least as many states as the ordinal value of SDE\_Value, or the CL block is aborted with a PROGERR.

SDE\_Value can be a CL local variable or a parameter. Its data type can be Number, Enumeration or Self-Defined Enumeration. If it is not one of these types, the CL block is aborted with a PROGERR.

If SDE\_Value is of type Number, the the name returned in State\_Name is the state at the ordinal position specified by SDE\_Value. For instance, if Param\_Name specifies a parameter of self-defined enumeration with states of red/green, and SDE\_Value is 1.0, then the value returned for State\_Name is green (note that the ordinal value of the first state in an enumeration is zero).

If SDE\_Value is of type Enumeration or Self-Defined Enumeration, the name returned in State\_Name corresponds to the state name which is in the ordinal position specified by SDE\_Value in the enumeration type of Param\_Name. For instance, if Param\_Name specifies a parameter of self-defined enumeration with states of red/green, and SDE\_Value is an enumeration with state names of dog/cat with a current value of cat, then the value returned for State\_Name is "green."

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 18.0 — Error returned on Get/Store parameter
- 19.0 — Error on Data Access conversion (bad state name or bad point.parameter)

### Examples:

The following CL code segment converts the ordinal value of 1.0 into the corresponding state of the PV parameter on the point HG050602. If hg050602 is defined as having states badpv/state1/state2, then state\_name will have the value "STATE1" in it after the call.

```
LOCAL status,          -- Return status
&   sde_value : NUMBER  -- Ordinal value of state name
LOCAL state_name : STRING -- State
EXTERNAL hg050605      -- Point id

SET ord_value = 1.0
CALL File$Convert_SDE_to_ASCII (status, state_name, hg050605.pv,
&                               sde_value)
```

### C.6.23 File\$Get\_Volume\_Statistics Subroutine

This routine acquires information on a particular disk volume.

```
SUBROUTINE File$Get_Volume_Statistics
  (Return_Status      : OUT NUMBER; -- File R/W package error status
   File_Manager_Status : OUT NUMBER; -- File Manager error status
   Volume_Size        : OUT NUMBER; -- Total sectors in volume
   Amount_Allocated   : OUT NUMBER; -- Allocated sectors in volume
   Path               : IN  STRING) -- Name of the Volume
```

This subroutine can only be called from a CL block linked to a background insertion point.

This subroutine retrieves the total space and the amount of space already allocated on a virtual volume of a History Module.

Volume\_Size will contain the total number of sectors shared by all directories on the volume. Amount\_Allocated will contain the number of sectors currently allocated to files on the volume.

Path must contain the full pathname of a file on the volume. If the name of a file on the volume is not known, then an arbitrary filename may be specified (i.e., NET>VOL>X.X).

For more information on volume statistics, refer to the *Utilities Operation* manual.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error

#### Examples:

The following CL code segment returns information on the memory usage on the history module volume "test."

```
LOCAL status,          -- Return status
&    fm_status,        -- File Manager return status
&    total_size,       -- Number of sectors reserved
&    alloc_size : NUMBER -- Number of sectors used
LOCAL vol_name  : STRING -- Name of volume to query

SET vol_name = "net>test>noname.x"

CALL File$Get_Volume_Statistics (status, fm_status, total_size,
&                               alloc_size, vol_name)
```

## C.6.24 File\$Create\_Character Subroutine

This routine takes a number and converts it to the equivalent ASCII character.

```
SUBROUTINE File$Create_Character
  (Return_Status : OUT NUMBER; -- File R/W package error status
   Char_Output   : OUT STRING; -- The character requested
   Decimal_Input  : IN  NUMBER) -- Ordinal value of character
```

This subroutine can be called from CL blocks linked to any insertion point.

This subroutine takes a number from 0 to 256, converts it into a single ASCII character, and returns it in a CL string of length 1. It can be used to create characters that cannot be entered through the keyboard.

### Possible Return\_Status Values for this function:

0.0 — Good status  
33.0 — Input value out of range

### Examples:

The following CL code segment will return the specified value as the first character of the string "char".

#### Decimal Value    Character

010	LF — Line Feed
012	FF — Form Feed
013	CR — Carriage Return
065	A — The capital letter "A"

```
LOCAL status      : NUMBER      -- Return status
LOCAL char        : STRING      -- Character corresponding to number

CALL File$Create_Character (status, char, 10.0)
CALL File$Create_Character (status, char, 12.0)
CALL File$Create_Character (status, char, 13.0)
CALL File$Create_Character (status, char, 65.0)
```

### C.6.25 File\$Convert\_FM\_Status Subroutine

This routine takes a File Manager status value and returns a string that describes what the status means.

```
SUBROUTINE File$Convert_FM_Status
  (ASCII_File_Manager_Status : OUT STRING; -- Status description
   File_Manager_Status       : IN  NUMBER; -- Numeric status code
   Status                    : OUT LOGICAL) -- Status code validity
```

This subroutine can be called from CL blocks linked to any insertion point.

Convert\_FM\_Status takes a numeric file manager status and returns a string describing the status. Status is ON if the input is a valid file manager status value, otherwise, status is OFF.

#### Examples:

The following CL code segment returns a message that explains the meaning of "status."

```
LOCAL information : STRING -- Meaning of the status
LOCAL status_val  : NUMBER -- File Manager status value
LOCAL ok          : LOGICAL -- Status was valid or not

CALL File$Convert_FM_Status (information, status_val, ok)
```

#### NOTE

The data strings returned by this subroutine are cryptic and can be ambiguous. For more precise explanations of the File Manager status values, see heading 4.10.

## C.6.26 File\$Strip\_Blanks Subroutine

This routine determines whether or not leading blanks are removed from values of type String on subsequent calls to File\$Get\_Field (see heading C.6.7).

```
SUBROUTINE File$Strip_Blanks
  (Return_Status      : OUT NUMBER;  -- File I/O set error status
   File_Number        : IN  NUMBER;  -- File number from Open_File call
   Strip              : IN  LOGICAL) -- If = ON, strip leading blanks
```

This subroutine can only be called from a CL block linked to a background insertion point.

This routine allows a CL program to specify whether or not Get\_Field will strip leading blanks from values of type string. If Strip = ON, any blank characters are removed from the beginning of the string returned in the Output\_Value argument. If Strip = OFF, leading blanks are retained. The default (Strip\_Blanks is never called by the CL program) is to strip leading blanks from string values.

File\_Number must be the value returned from a previous Open\_File call.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 13.0 — File number is invalid
- 20.0 — Can't read file control table pointer
- 34.0 — Number not assigned to this CL Block—access denied

### Examples:

The following CL code segment sets the current mode for acquisition of string values via Get\_Field to retain leading blanks.

```
LOCAL status,                -- Return status
&   file_id      : NUMBER    -- File Identifier
LOCAL lead_blanks : LOGICAL  -- ON = strip blanks; OFF = do not

SET lead_blanks = OFF
CALL File$Strip_Blanks (status, file_id, lead_blanks)
```

## C.6.27 File\$Upper\_Case Subroutine

This subroutine converts all lower case alphabetic characters of a string into upper case.

```
SUBROUTINE File$Upper_Case
  (Converted_String   : OUT STRING;  -- The converted string
   String_to_Convert  : IN  STRING)  -- The string to convert
```

### Possible Return\_Status Values for this function:

None

### Examples:

The following CL code segment converts the alphabetic characters of string "istring" to upper case and stores the result in "ostring."

```
LOCAL istring,
&      ostring      : STRING

SET istring = "abcdefghijk"
CALL File$Upper_Case (ostring, istring)
```



## C.6.28 File\$Delete\_File Subroutine

This subroutine is used to delete files either from the History Module or a floppy/cartridge. The only effective difference between this subroutine and the standard AM subroutine Delete\_File is that this subroutine allows access to removable media.

```
SUBROUTINE File$Delete_File
  (Return_Status      : OUT NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT NUMBER;  -- File Manager error status
   Path               : IN  STRING)  -- Pathname including file_name
                                     -- and extension
```

This subroutine can only be called from a CL block linked to a background insertion point.

This subroutine can be used to delete a file on another LCN. In this case, the file to be deleted must be on a History Module.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 3.0 — File privilege violation
- 5.0 — File does not exist
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 28.0 — File already open
- 29.0 — File reserved for another operation
- 36.0 — Can't access file (due to HM failure, etc.)

### Examples:

The following CL code segment deletes a file from a removable media (cartridge).

```
LOCAL status,          -- Return status
&    fm_status      : NUMBER -- File Manager error status

CALL File$Delete_File (status, fm_status,
&    "PN:36>DEV:RM00>&ams>create3.x")
```

The following CL code segment deletes a file on a remote LCN named "am."

```
LOCAL status,          -- Return status
&    fm_status      : NUMBER -- File Manager error status

CALL File$Delete_File (status, fm_status,
&    "am\net>test>test44.x")
```

## C.6.29 File\$Create\_Volume Subroutine

This subroutine is used either to create a volume on an initialized floppy/cartridge, or to both initialize the medium and create a volume on it.

```
SUBROUTINE File$Create_Volume
  (Return_Status      : OUT NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT NUMBER;  -- File Manager error status
   Max_Num_Files      : IN  NUMBER;  -- Maximum number of files
   Format              : IN  LOGICAL; -- on = format, then initialize;
                                   -- off = initialize only
   Mem_Dir_Opt        : IN  LOGICAL; -- on  = keep file locators in
                                   -- memory; off = do not keep
                                   -- file locators in memory
   Volume_id          : IN  STRING;   -- 20-character volume identifier
   Path               : IN  STRING)   -- Pathname excluding file_name
                                   -- and extension
```

The Max\_Num\_Files parameter specifies the maximum number of files that can be created on this volume (the minimum is 26). A floppy can hold a maximum of 300 files; the cartridge disk limit is 3000<sup>X</sup> files. Note that allowing a large number of files has an impact on data space availability since one sector of housekeeping data is required for each five files.

The up-to-20 character volume identifier string must be left justified (no leading spaces).

The Path parameter must be of this form: PN:nn>DEV:xxmm>volume\_id>

Path identification contents are defined at heading C.2.4 and requirements for the four-character volume\_id are explained at heading C.2.9.

This subroutine can only be called from a CL block linked to a background insertion point.

### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 12.0 — Other File Manager error
- 33.0 — Input value out of range
- 42.0 — Volume not mounted

**Examples:**

The following CL code segment establishes a volume on a removable media (floppy).

```

LOCAL status,           -- Return status
&    fm_status,         -- File Manager error status
&    max_num    : NUMBER -- Maximum number of files
LOCAL format,           -- Initialize on/off
&    mem_dir    : LOGICAL -- Locators in memory on/off
--
SET max_num = 26
SET format = OFF
SET mem_dir = OFF
CALL File$Create_Volume_With_Descriptors
&    (status,
&    fm_status,
&    max_num,
&    format,
&    memory_dir,
&    "recipes for cookies",
&    "PN:08>DEV:FD00>cook>")

```

### C.6.30 File\$Create\_Directory Subroutine

This subroutine is used to create a directory on a specified volume on a floppy or cartridge. Each floppy or cartridge can have up to 63 directories.

```
SUBROUTINE File$Create_Directory
  (Return_Status      : OUT NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT NUMBER;  -- File Manager error status
   Path               : IN  STRING;   -- Pathname excluding file_name
                                     -- and extension
   Directory_name      : IN  STRING)  -- New directory's name
```

The Path parameter must be of this form: PN:nn>DEV:xxmm>volume\_id>

Path identification contents are defined at heading C.2.4; directory name restrictions are explained at heading C.2.9.

This subroutine can only be called from a CL block linked to a background insertion point.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 12.0 — Other File Manager error
- 41.0 — Duplicate volume
- 42.0 — Volume not mounted
- 43.0 — Duplicate directory
- 49.0 — Directory name too long

#### Examples:

The following CL code segment establishes directory "diry" on volume "vola" on a removable media (floppy).

```
LOCAL status,          -- Return status
&      fm_status, : NUMBER  -- File Manager error status

CALL File$Create_Directory (status,
&                          fm_status,
&                          "PN:08>DEV:FD00>vola>",
&                          "diry")
```

### C.6.31 File\$Delete\_Directory Subroutine

This subroutine is used to delete the specified directory from a specified volume on a floppy or cartridge.

```
SUBROUTINE File$Delete_Directory
  (Return_Status      : OUT NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT NUMBER;  -- File Manager error status
   Path               : IN  STRING;   -- Pathname excluding file_name
                                   -- and extension
   Directory_name     : IN  STRING)  -- Directory name
```

The Path parameter must be of this form: PN:nn>DEV:xxmm>volume\_id>

Path identification contents are defined at heading C.2.4; directory name restrictions are explained at heading C.2.9.

This subroutine can only be called from a CL block linked to a background insertion point.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 12.0 — Other File Manager error
- 41.0 — Duplicate volume
- 42.0 — Volume not mounted
- 44.0 — Directory not found
- 45.0 — Directory not empty
- 49.0 — Directory name too long

#### Examples:

The following CL code segment deletes directory "diry" from volume "vola" on a removable media (floppy).

```
LOCAL status,          -- Return status
&    fm_status, : NUMBER -- File Manager error status

CALL File$Delete_Directory (status,
&    fm_status,
&    "PN:08>DEV:FD00>vola>",
&    "diry")
```

## C.6.32 File\$Read\_Directory Subroutine

This subroutine is used with the next two calls (File\$Get\_File\_Attributes and File\$Read\_Directory\_Complete) to allow a CL program to cycle through, one at a time, operations on all the files in a particular volume or directory. This "cycling" would, for example, be done to copy all or a subset of the files from one volume to a backup volume or to delete files for a volume. A maximum of 10 read directory operations can occur simultaneously.

The File\$Read\_Directory subroutine creates a list of all the files on the volume that match the "wildcard" filename and extension specified in the pathname. The first time it is called, this subroutine returns the first 26 file names matched (if there are that many that match). Subsequent calls return the second 26, etc., until all matching file names have been returned. When the OUT parameter number\_of\_files is returned equal to zero, all file names have been read.

After completing work on the last group of file names returned, the CL program must call the subroutine File\$Read\_Directory\_Complete to close the directory function. This deallocates space used by the list and frees up the file list number for reuse. File\$Read\_Directory\_Complete is automatically called for all "open" directories when a CL is aborted or terminates normally. One CL cannot access another CL's "open" directory.

```
SUBROUTINE File$Read_Directory
  (Return_Status      : OUT    NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT    NUMBER;  -- File Manager error status
   Number_of_files    : OUT    NUMBER;  -- Number of files returned
   File_List_Number   : IN OUT NUMBER;  -- List id is returned by call
   Path               : IN     STRING;   -- Pathname excluding file
                                         -- name and extension
   File_Name_and_ext  : IN     STRING)  -- File name and extension;
                                         -- must include "wildcard"
                                         -- entries
```

The File\_List\_Number parameter must equal zero the first time the subroutine is called. If the IN value is not zero, the list number is used to access a previously created list.

Question marks are used as "wildcard" entries in file names and extensions. Some examples:

```
?????????.?? matches all files in a directory
?????????.cl matches all files in a directory with CL extension
ralph.??     matches all files in a directory with file name of ralph
?R?????.cl   matches all files that have R as the second character, have zero to four
              characters following the R, and have an extension of CL
R?????????.CL matches all file names that start with an R and have CL as an extension
?????????R.CL matches all file names that have eight characters with the eighth character
              an R and have CL as an extension
```

Be aware that the CL/AM wildcard function is the same as that provided by the File Manager. However, it works differently than the wildcard function in the Engineering Personality utilities. In the AM, question marks that trail a specific character will match either trailing blanks or any character. This is also true when the entire name is question marks. In the EP utilities, question marks match only characters, not blanks. Examples of the differences follow.

In the AM `?R?????.cl` matches file names `create.cl` and `crash.cl`.

In the EP `?R?????.cl` matches file names `create.cl` but not `crash.cl`.

In the AM `?????????.??` matches all files in a directory.

In the EP `?????????.??` matches only 8-character file names with 2-character extensions.

If your programs need to get the same results from CL file read directory calls as from the EP utilities, they need to examine length of the returned name and ignore those names that are not the desired length.

The first `File$Read_Directory` call of a particular set of operations needs to be made with the `file_list_number` parameter = 0. The directory is then "opened," and given a number identifier. The attributes of each of the first 26 (or less) files are held in this open directory. The number of files found also is returned to the calling program.

The calling program can then review the attributes of each of the returned files by making `File$Get_File_Attributes` calls for each file. After reviewing the attributes of this set of files, if all files in the volume/directory may have not yet been reviewed (this set contained a full 26 files), another `File$Read_Directory` call will get attributes of the next 26 (or fewer) files. Note that each time the `File$Read_Directory` call is repeated, a new `Number_of_files` value is returned, and a zero value indicates that all values have been read. Once all files have been reviewed, a `File$Read_Directory_Complete` call should be made to "close" the directory.

This subroutine can only be called from a CL block linked to a background insertion point.

This subroutine can be used to read a directory on another LCN. In this case, the volume or directory must be on a History Module.

#### **Possible Return\_Status Values for this function:**

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 20.0 — Can't read file control table pointer
- 22.0 — Maximum number of files already open
- 23.0 — Memory limit exceeded
- 36.0 — Can't access file (due to HM failure, etc.)
- 41.0 — Duplicate volume
- 42.0 — Volume not mounted
- 44.0 — Directory not found
- 46.0 — Major corruption of AM
- 47.0 — Wild card needed in filename or extension
- 48.0 — File list number is invalid

#### **Examples:**

See the complete example following the description of `File$Read_Directory_Complete` at heading C.6.34.

### C.6.33 File\$Get\_File\_Attributes Subroutine

This subroutine gets the file name, extension, file size (for files with fixed length records), and time stamp of any file in a list previously created by File\$Read\_Directory.

```
SUBROUTINE File$Get_File_Attributes
  (Return_Status      : OUT  NUMBER;  -- File R/W error status
   File_Name_and_ext  : OUT  STRING;   -- Filename.extension;
   File_Size          : OUT  NUMBER;   -- Number of words used by file
   Time_Stamp         : OUT  TIME;     -- File creation date and time
   File_Number        : IN   NUMBER;   -- Input slot in list
   File_List_Number   : IN   NUMBER)   -- Value returned by preceding
                                       -- File$Read_Directory call
```

This subroutine can only be called from a CL block linked to a background insertion point.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 20.0 — Can't read file control table pointer
- 33.0 — Input value (file number) out of range
- 48.0 — File list number is invalid

#### Examples:

See the complete example following the description of File\$Read\_Directory\_Complete at heading C.6.34.



### C.6.34 File\$Read\_Directory\_Complete Subroutine

This must be called to deallocate the space used for a list of file names created by File\$Read\_Directory.

```
SUBROUTINE File$Read_Directory_Complete
  (Return_Status      : OUT   NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT   NUMBER;  -- File Manager error status
   File_List_Number   : IN    NUMBER)  -- List to be deallocated
```

This subroutine can only be called from a CL block linked to a background insertion point.

This subroutine can be used to deallocate the space used for a list of files read from another LCN (see File\$Read\_Directory).

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 6.0 — Device error
- 12.0 — Other File Manager error
- 42.0 — Volume not mounted
- 46.0 — Major corruption of AM
- 48.0 — File list number is invalid

#### Examples:

The following code segment prints the attributes of the first 26 (or fewer) files on a volume.

```
LOCAL status,                -- Return status
&    fm_status,              -- File Manager error status
&    file_list_number,       --
&    file_size,              --
&    file_counter,          --
&    num_files               : NUMBER  -- Maximum number of files
LOCAL format,                --
&    memory_direct          : LOGICAL  --
LOCAL file_ext               : STRING   --
LOCAL time_stamp             : TIME     --

SET file_list_number = 0
SET status = 0
SET fm_status = 0
SET num_files = 0
CALL File$Read_Directory (status,
&                        fm_status,
&                        num_files,
&                        file_list_number,
&                        "PN:08>DEV:FD00>CL>",
&                        "?????????.??")
SEND: "stat=", status, "fm_stat=", fm_status
SEND: "number of files", num_files
SEND: "file_list_number", file_list_number
```

```

get_attributes: LOOP FOR file_counter IN 1..num_files
CALL File$Get_File_Attributes (status,
&                                file_ext,
&                                file_size,
&                                time_stamp,
&                                file_counter,
&                                file_list_number)
SEND: "get attribute stat=", status, "fm_stat=", fm_status
SEND: "file_number", file_counter
SEND: "file_list_number", file_list_number
SEND: "file_name", file_ext
SEND: "file_size", file_size
SEND: "time_stamp", time_stamp
CALL BKG_Delay (5 SECS)
REPEAT get_attributes

CALL File$Read_Directory_complete (status,
&                                fm_status,
&                                file_list_number)
SEND: "directory complete stat=", status, "fm_stat=", fm_status

```

The following code segment counts the number of files in a directory and prints out the total. It also reads the attributes for each file in the directory.

```

PARAMETER path_name      : STRING      -- Path to the test directory
LOCAL no_error           = 0.0         -- Return status = good
LOCAL file_list_limit    = 48.0        -- Return status = no more files
LOCAL status,            -- Return status
&    fm_status,          -- File Manager error status
&    file_list_number,   --
&    file_size,          --
&    file_counter,       --
&    num_files           : NUMBER      -- Maximum number of files
LOCAL format,            --
&    memory_direct       : LOGICAL     --
LOCAL file_ext           : STRING      --
LOCAL time_stamp         : TIME        --

SET path_name = "PN:36>DEV:RM00>lj>"
SET file_list_number = 0
SET status = 0
SET fm_status = 0
SET num_files = 0
SET total_files = 0

read_dir: CALL File_Read_Directory (status,
&                                fm_status,
&                                num_files,
&                                file_list_number,
&                                path_name,
&                                "?????????.??")
IF (status <> no_error) THEN
& (SEND: "Failed Read Dir: stat=", status, "fm_stat=", fm_status;
& SEND: "file_list_number", file_list_number;
& GOTO end_read)

```

```

SET total_files = total_files + num_files
IF (num_files = 0.0) THEN GOTO end_read

get_attributes: LOOP FOR file_counter IN 1..num_files
  CALL File$Get_File_Attributes (status,
    &                                file_ext,
    &                                file_size,
    &                                time_stamp,
    &                                file_counter,
    &                                file_list_number)
  IF (status <> no_error) THEN
    & (SEND: "Failed in Get_File_Attributes";
    & SEND: "get attribute stat=", status, "fm_stat=", fm_status;
    & SEND: "file_number", file_counter;
    & SEND: "file_list_number", file_list_number;
    & GOTO end_read)

  REPEAT get_attributes

  GOTO read_dir

end_read: SEND: "Total files ", total_files

  CALL File$Read_Directory_complete (status,
    &                                fm_status,
    &                                file_list_number)

```

### C.6.35 File\$Get\_Volume\_Directories Subroutine

This subroutine gets a list of all the directories on a selected volume. It uses about 3300 words of stack for the array of directory names it returns. Therefore, you should always embed this call within a CL subroutine (CL subroutine stack usage is only in effect while the subroutine is executing). The array to receive directory names must contain at least 63 elements (maximum number of directories in a volume). If the `directory_id_array` is not at least a 63-element string array, the calling program aborts with a PROGERR.

```
SUBROUTINE File$Get_Volume_Directories
  (Return_Status      : OUT    NUMBER;  -- File R/W error status
   DA_Status          : OUT    NUMBER;  -- Data access return status
   File_Manager_Stat  : OUT    NUMBER;  -- File Manager error status
   Max_Number_Files   : OUT    NUMBER;  -- Maximum files allowed
   No_of_Directories  : OUT    NUMBER;  -- Number of directories
   Identifier_String   : OUT    STRING;  -- 20-char identifier
   Directory_id_Array  : OUT    arr_nm;  -- String Array 1..63 to hold
                                         -- returned directory names
   Path                : IN     STRING)  -- Pathname may include
                                         -- file_name and extension
```

This subroutine can only be called from a CL block linked to a background insertion point.

This subroutine can be used to obtain a list of directories from another LCN. In this case, the volume must be on a History Module.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 18.0 — Error returned on Get/Store parameter
- 25.0 — Reference list number invalid

**Examples:**

The following CL code segment determines what directories are on a volume, then outputs the directory names to the operator.

```

LOCAL status,                -- Return status
&    da_status,              -- Data Access return status
&    fm_status,              -- File Manager error status
&    max_files,              -- Number of files allowed
&    no_of_directories,      -- Number of directories
&    i                        : NUMBER -- Loop control variable
LOCAL identifier_string : STRING -- Volume identifier
LOCAL volumes           : STRING ARRAY (1..63)

CALL File$Get_Volume_Directories (status,
&                                da_status,
&                                fm_status,
&                                max_files,
&                                no_of_directories,
&                                identifier_string,
&                                volumes,
&                                "PN:08>DEV:FD00>cl>")
--
    IF status = no_error THEN
(output_vol: LOOP FOR i IN 1..no_of_directories;
& SEND: "Directory ", i, volumes(i);
& REPEAT output_vol)
```

### C.6.36 File\$Create\_Volume\_With\_Descriptors Subroutine

This subroutine can be used to create a volume with descriptors on an initialized removable media, or to both initialize the media and create a volume with descriptors. It has the same function as the File\$Create\_Volume subroutine plus provides for descriptors in each directory and file of the volume.

```
SUBROUTINE File$Create_Volume_With_Descriptors
  (Return_Status      : OUT NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT NUMBER;  -- File Manager error status
   Max_Num_Files      : IN  NUMBER;  -- Maximum number of files
   Format              : IN  LOGICAL;  -- on = format, then initialize;
                                     -- off = initialize only
   Mem_Dir_Opt        : IN  LOGICAL;  -- on = keep file locators in
                                     -- memory; off = do not keep
                                     -- file locators in memory
   Volume_id          : IN  STRING;   -- 20-character volume identifier
   Path               : IN  STRING)   -- Pathname excluding file_name
                                     -- and extension
```

The Max\_Num\_Files parameter specifies the maximum number of files that can be created on this volume (the minimum is 26). A floppy can hold a maximum of 300 files; the cartridge disk limit is 3000<sup>X</sup> files. Note that allowing a large number of files has an impact on data space availability since one sector of housekeeping data is required for each five files. With descriptors, an additional  $((\text{Max\_Num\_Files} + 63) * 64) / 256$  sectors are required.

A double-sided, double density diskette contains 4004 sectors. The maximum number of files allowed on the volume will be at least as many as specified. It may be a few more.

The up-to-20 character volume identifier string must be left justified (no leading spaces).

The Path parameter must be of this form: PN:nn>DEV:xxmm>volume\_id>

Path identification contents are defined at heading C.2.4 and requirements for the four-character volume\_id are explained at heading C.2.9.

This subroutine can only be called from a CL block linked to a background insertion point.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 12.0 — Other File Manager error
- 33.0 — Input value out of range
- 42.0 — Volume not mounted

**Examples:**

The following CL code segment establishes a volume with provision for descriptors on a removable media (floppy).

```

LOCAL status,          -- Return status
&    fm_status,        -- File Manager error status
&    max_num    : NUMBER -- Maximum number of files
LOCAL format,          -- Initialize on/off
&    mem_dir    : LOGICAL -- Locators in memory on/off
--
SET max_num = 26
SET format = OFF
SET mem_dir = OFF
CALL File$Create_Volume_With_Descriptors
&    (status,
&    fm_status,
&    max_num,
&    format,
&    mem_dir,
&    "recipes for cookies",
&    "PN:08>DEV:FD00>cook>")

```

### C.6.37 File\$Modify\_Volume\_ID\_String Subroutine

This subroutine can be used to modify a volume ID string. The up-to-20 character volume identifier must be specified in the call.

```
SUBROUTINE File$Modify_Volume_ID_String
  (Return_Status      : OUT      NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT      NUMBER;  -- File Manager error status
   Volume_id          : IN       STRING;   -- 20 character volume
                                         -- identifier
   Path               : IN       STRING)   -- Pathname excluding the
                                         -- filename and extension
```

The volume identifier string must be left justified (no leading spaces).

This subroutine can only be called from a CL Block linked to a background insertion point.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 6.0 — Device error
- 12.0 — Other File Manager error
- 33.0 — Input value out of range
- 42.0 — Volume not mounted

#### Examples:

The following CL code segment changes the volume id string of a volume named "back" on a removable media (floppy).

```
LOCAL status,          -- Return status
&      fm_status      : NUMBER  -- File Manager error status
--
CALL File$Modify_Volume_ID_String (status,
&                                fm_status,
&                                "backup volume",
&                                "PN:08>DEV:FD00>back>")
```



### C.6.38 File\$Modify\_File\_Descriptor Subroutine

This subroutine can be used to add, modify, or delete a file's descriptor. Deleting a descriptor means setting the descriptor value equal to all blanks.

```
SUBROUTINE File$Modify_File_Descriptor
  (Return_Status      : OUT      NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT      NUMBER;  -- File Manager error status
   Descriptor         : IN       STRING;   -- New descriptor value
   Path               : IN       STRING)   -- Pathname including the
                                           -- file_name and extension
```

This subroutine can only be called from a CL Block linked to a background insertion point.

The descriptor value is 64 characters long. If the input string is less than 64 characters, the remainder of the descriptor is blank filled. If the input string is longer than 64 characters, it is truncated.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 12.0 — Other File Manager error
- 41.0 — Duplicate volume
- 42.0 — Volume not mounted
- 43.0 — Duplicate directory
- 49.0 — Directory name too long
- 50.0 — Descriptors not defined

#### Examples:

```
LOCAL status,                -- Return status
&      fm_status             : NUMBER    -- File Manager error status
--
CALL File$Modify_File_Descriptor (status,
&                                fm_status,
&                                "THIS IS A FILE DESCRIPTOR - IT CAN BE 64 CHARS LONG",
&                                "PN:08>DEV:FD00>vola>file23.xx")
```

### C.6.39 File\$Modify\_Directory\_Descriptor Subroutine

This subroutine can be used to add, modify, or delete a directory descriptor. Deleting a descriptor means setting the descriptor value equal to all blanks.

```
SUBROUTINE File$Modify_Directory_Descriptor
  (Return_Status      : OUT      NUMBER;  -- File R/W error status
   File_Manager_Stat  : OUT      NUMBER;  -- File Manager error status
   Descriptor         : IN       STRING;   -- New descriptor value
   Path               : IN       STRING)   -- Pathname excluding the
                                           -- file_name and extension
```

This subroutine can only be called from a CL Block linked to a background insertion point.

The descriptor value is 64 characters long. If the input string is less than 64 characters, the remainder of the descriptor is blank filled. If the input string is longer than 64 characters, it is truncated.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 12.0 — Other File Manager error
- 41.0 — Duplicate volume
- 42.0 — Volume not mounted
- 43.0 — Duplicate directory
- 49.0 — Directory name too long
- 50.0 — Descriptors not defined

#### Examples:

```
LOCAL status,          -- Return status
&      fm_status      : NUMBER    -- File Manager error status
--
CALL File$Modify_Directory_Descriptor (status,
&                                     fm_status,
&                                     "THIS IS A FILE DESCRIPTOR - IT CAN BE 64 CHARS LONG",
&                                     "PN:08>DEV:FD00>vola>")
```

### C.6.40 File\$Get\_File\_Descriptor Subroutine

This subroutine gets the descriptor of any file in a list previously created by File\$Read\_Directory.

```
SUBROUTINE File$Get_File_Descriptor
  (Return_Status      : OUT    NUMBER;  -- File R/W error status
   File_Descriptor    : OUT    STRING;  -- 64-character descriptor
   File_Number        : IN     NUMBER;  -- Input slot in list
   File_List_Number   : IN     NUMBER)  -- value returned by preceding
                                         -- File$Read_Directory call
```

This subroutine can only be called from a CL block linked to a background insertion point.

The descriptor value is 64 characters long.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 20.0 — Can't read file control table pointer
- 33.0 — Input value (file number) out of range
- 48.0 — File list number is invalid

#### Examples:

See the complete example after the File\$Read\_Directory\_Complete description at heading C.6.34. The File\$Get\_File\_Descriptor subroutine must be used in the same manner as the File\$Get\_File\_Attributes call.

### C.6.41 File\$Get\_Volume\_Directories\_D Subroutine

This subroutine gets a list of all the directories and the directories' descriptors on a given volume. It uses about 3300 words of stack for the array of directory names and descriptors that it returns. Therefore, you should always embed this call within a CL subroutine (CL subroutine stack usage is in effect only while the subroutine is executing). The array to receive directory names must contain at least 63 elements (the maximum number of directories in a volume). If the `directory_id_array` is not at least a 63-element string array, the calling program aborts with a PROGERR.

The returned directory array contains the directory name in the first four characters, then a space, and then the 64-character descriptor.

```
SUBROUTINE File$Get_Volume_Directories_D
  (Return_Status      : OUT    NUMBER;  -- File R/W error status
   DA_Status          : OUT    NUMBER;  -- Data access return status
   File_Manager_Stat  : OUT    NUMBER;  -- File Manager error status
   Max_Number_Files   : OUT    NUMBER;  -- Maximum files allowed
   No_of_Directories  : OUT    NUMBER;  -- Number of directories
   Identifier_String   : OUT    STRING;  -- 20-char identifier
   Directory_id_Array  : OUT    arr_nm;  -- String Array 1..63 to hold
                                         -- returned directory names
   Path                : IN     STRING)  -- Pathname may include
                                         -- file_name and extension
```

This subroutine can only be called from a CL Block linked to a background insertion point.

#### Possible Return\_Status Values for this function:

- 0.0 — Good status
- 1.0 — Invalid file pathname
- 9.0 — Volume does not exist (or in some cases is not mounted)
- 12.0 — Other File Manager error
- 18.0 — Error returned on Get/Store parameter
- 25.0 — Reference list number invalid

**Examples:**

The following CL code segment determines what directories are on a volume, then outputs the directory names to the operator.

```

LOCAL status,                -- Return status
&    da_status,              -- Data Access return status
&    fm_status,              -- File Manager error status
&    max_files,              -- Number of files allowed
&    no_of_directories,      -- Number of directories
&    i                        : NUMBER -- Loop control variable
LOCAL identifier_string : STRING -- Volume identifier
LOCAL volumes           : STRING ARRAY (1..63)

CALL File$Get_Volume_Directories_D (status,
&    da_status,
&    fm_status,
&    max_files,
&    no_of_directories,
&    identifier_string,
&    volumes,
&    "PN:08>DEV:FD00>cl>")
--
IF status = no_error THEN
& (SEND:"TEST1: check results:type 'LS {path} -D' in cmd processor";
& SEND:"TEST1: Max_files =", max_files;
& SEND:"TEST1: Directories =", no_of_directories;
& SEND:"TEST1: Directory Id =", identifier_string;
& SEND:"TEST1: File$Get_Volume_directories_D PASSED")
ELSE
& (SEND:"TEST1: File$Get_Volume_directories_D FAILED";
& SEND:"TEST1: Status =", status;
& SEND:"TEST1: DA_Status =", da_status;
& SEND:"TEST1: FM_Status =", fm_status;
& EXIT)
--
output_vol: LOOP FOR i IN 1..no_of_directories
SEND: "Directory ", i, volumes(i)
REPEAT output_vol

```

## C.7 RETURN STATUS VALUES

The following is a list of return status values for the calls in this package:

- 0.0 — Good status
- 1.0 — Invalid pathname
- 2.0 — File exists, but is of wrong type
- 3.0 — File privilege violation
- 4.0 — File already exists
- 5.0 — File does not exist
- 6.0 — Device error while reading file
- 7.0 — File table for the specified volume is full
- 8.0 — No disk space remaining for file
- 9.0 — Volume does not exist (or in some cases, is not mounted)
- 10.0 — No more space on the volume
- 11.0 — Read or write beyond end of file
- 12.0 — Other File Manager error
- 13.0 — File number is invalid
- 14.0 — Attempt to read/write field beyond end of buffer
- 15.0 — Value has invalid data type
- 16.0 — Format string is invalid
- 17.0 — Conversion of a value failed
- 18.0 — Error returned on Get/Store parameter
- 19.0 — Error on Data Access conversion
- 20.0 — Can't read file control table pointer
- 21.0 — Can't write file control table pointer
- 22.0 — Maximum number of files already open
- 23.0 — Memory limit exceeded
- 24.0 — Attempt to read/write field beyond end of string
- 25.0 — Reference list number invalid
- 26.0 — Bad file name
- 27.0 — Bad file extension
- 28.0 — File already open
- 29.0 — Directory in use
- 30.0 — Invalid field number
- 31.0 — Invalid record number
- 32.0 — Record was truncated
- 33.0 — Input value out of range
- 34.0 — Number not assigned to this CL Block—access denied
- 35.0 — Maximum record length exceeded
- 36.0 — Can't access file (due to HM failure, etc.)
- 37.0 — Invalid file access code in Open\_File call
- 38.0 — Invalid record length in Create\_File or Open\_File call
- 39.0 — Invalid string width in Get\_Field or Get\_Field\_S
- 40.0 — Invalid character pointer
- 41.0 — Duplicate volume
- 42.0 — Volume not mounted
- 43.0 — Duplicate directory
- 44.0 — Directory not found
- 45.0 — Directory not empty
- 46.0 — Major corruption of AM
- 47.0 — Wild card needed in filename or extension
- 48.0 — File list number is invalid
- 49.0 — Directory name too long
- 50.0 — Descriptors not defined

## C.8 MEMORY CONSIDERATIONS

When the File I/O Extension is loaded, it can consume up to 55,149 words of user memory. Of that, 30,000 is added to Heap Pool 1 and is used as working memory by the set.

The File I/O Extension uses 500 words of stack and makes use of routines in the AM that may use additional stack. If a CL block terminates abnormally with a CL Error Status (CLERRSTS) of Program Error (PROGERR) while it is in a call to one of the routines in the File I/O Extensions, a stack overflow condition may have occurred. To correct this situation use the Network Configurator to increase "Background Task Stack Size."

When the Data Conversion Set is loaded, it can consume up to 79,000 words of user memory. Of that, 31,000 is added to Heap Pool 1 and is used as working memory by the set.

## C.9 FILE I/O PERFORMANCE EXAMPLE

The following information profiles the performance and effect on AM capacity of a test system that ran the same Background CL task on up to 10 separate points simultaneously. The task opens a file; then reads and writes 100 records, one record at a time, to it; then closes the file. Note that the test system had very little HM activity and no foreground AM activity, so these numbers represent maximum throughput that can be expected.

Background CL State	File Transfers Per Second	AM Idle Percentage
0 Background Tasks Running	0	74%
1 Background Task Running	2	72%
2 Background Tasks Running	4	69%
4 Background Tasks Running	5.6	68%

Note that the number of file transfers per second is the total of both reads and writes. Thus, one read and one write equals two transfers.

### CAUTION

CL/AM file I/O is not intended for heavy or continuous use. Heavy use should be held to short bursts that do not exceed 5 transfers per second. Continuous or heavy use will degrade the performance of other HM-related activities such as schematic invocation.





## CL/AM EXTENSION FOR CONTINUOUS HISTORY ACCESS

### Appendix D

*This appendix explains an available set of subroutines that enable a background CL program to obtain data from continuous history files on the History Module.*

#### D.1 OVERVIEW OF THE HISTORY ACCESS EXTENSION

The CL/AM Extension for Continuous History Access consists of a set of subroutines that can be loaded in an AM. These subroutines enable CL/AM programs to read historized data from the History Module (HM). The subroutines in this package can be used only by CL blocks linked to Background insertion points.

##### D.1.1 Features of this CL/AM Extension

1. This CL/AM Extension allows users to retrieve historized data for a single point. The data being retrieved can be either Absolute snapshots, Relative snapshots, or Absolute averages. (Relative requests relate to a collection of a specific number of values relative to the current time. Absolute requests relate to values collected between specified times).
2. CL/AM programs can use this extension to collect history of the following types: snapshots, hourly averages, shift averages, daily averages, monthly averages, and user defined averages.
3. This extension also allows the CL program to determine the collection rate used for a specific point.parameter.
4. Point.parameter values can be specified in either internal form or in ASCII. If an element of an array is to be specified in ASCII, the index must be numeric (enumeration indices are not allowed for ASCII input).
5. The search can be of on-line data only, or on all data (both pre-archived and on-line data).

## D.1.2 Summary of Subroutines in this CL Extension

Table C-1 provides a summary of the subroutines in this package.

**Table C-1 — Subroutines in Continuous History Access Extension**

Subroutine Name	Subroutine Function	Section
AMCL03\$Get_Collection_Rate	Obtains the collection rate	D.4.2
AMCL03\$Abs_History	Retrieves snapshot history data (absolute time base)	D.4.3
AMCL03\$Rel_History	Retrieves snapshot history data (relative time base)	D.4.4
AMCL03\$Avg_History	Retrieves averages history data (absolute time base)	D.4.5

The first subroutine, `Get_Collection_Rate`, retrieves the rate at which the history data for a specified `point.parameter` is being collected. This information is useful when determining the sample rate to be used for history transfer requests.

Each of the three history transfer calls represents one logical division of collected history data. These divisions are:

- **Absolute Snapshots (`Abs_History` call)** — Given begin and end times and a sample rate, this subroutine returns snapshot history values (with status information and a timestamp) for a specified `point.parameter`.
- **Relative Snapshots (`Rel_History` call)** — Given beginning and end offsets (number of records) from current time and a sample rate, this subroutine returns snapshot history values (and status information) for a specified `point.parameter`.
- **Averages (`Avg_History` call)** — Given begin and end times and an averages type, this subroutine returns averaged values (with status information and a timestamp) for a specified `point.parameter`.

## D.1.3 Memory Use

The History Access Extension requires approximately 15 K words.

## D.2 PACKAGING

The CL Extension for History Access is packaged as two sets, `CONV` and `AMCL03`, plus a set definition file that is used to define the History Access set routines to the CL compiler.

`CONV` contains data access conversion routines used by all standard extensions to CL. These routines are not directly available to CL/AM blocks. They convert enumeration values, self defined enumeration values and point ids from internal to external form and from external to internal form.

The `AMCL03` set contains routines unique to the History Access functions.

## D.3 INSTALLATION AND CONFIGURATION

The two set image files CONV.L0 and AMCL03.LO must be installed in a directory named &CUS on the type of media from which the AM will be loaded. The set definition file AMCL03.SF must be installed in the directory &CLX on a History Module.

Any AM that is loaded with the sets must be correctly configured via the Network Configurator. To do this, perform the following steps:

1. Set the default Volume Path for the Network Configurator Backup to point at a device where the current NCF will be saved.
2. Invoke the Network Configurator from the Engineers Main Menu by selecting "LCN Nodes."
3. Select the node number of the AM to be configured.
4. Select the "Modify Node" target on page 1.
5. On page 2, make certain that at least one background CL task has been provided for.
6. On page 3, enter the name of the File Set (AMCL03) in the list of externally loaded modules and press the Enter key. This should cause the name of the conversion set (CONV) to also be placed in the list of externally loaded modules.
7. Press the Check Key (F1) and the configuration should check OK.
8. Press the Install Key (F2) and the configuration should install without error.

The AM can now be loaded.

### NOTE

Honeywell recommends that you load AMs individually to avoid problems. For example, if you have the CONV external load module configured for your AMs, you may get a CNAMREV error when trying to load more than one AM at the same time. To recover from that specific error, reload the nodes which have that warning.

## D.4 CL CALLABLE SUBROUTINES

### D.4.1 Introduction

The functions provided by this extension are implemented as CL callable subroutines. Descriptions of the individual subroutines begin at heading D.4.2.

#### D.4.1.1 Including the Set Definition File

The subroutines are declared in the set definition file which is read by the CL compiler when an "include\_set" directive is encountered in a CL source file within the scope of a CL Block. The directive appears in your source file as follows: `%INCLUDE_SET AMCL03`

Each routine may be invoked by calling it in a CL Block or Subroutine via the Call statement. For instance:

```
Call AMCL03$Get_Collection_Rate (status, cl_status, rate, point.pv)
```

invokes the routine to obtain the collection rate for a specified point.parameter.

#### NOTE

Several of the subroutines in this extension set have arguments designated to be of the stand-in data type "cl\_type." The actual data types of these arguments, as specified in your calls, can be any valid CL/AM data type within the rules given below. For these arguments only, normal CL compiler data-type checking is inhibited, and any data type mismatches you may accidentally create are not detected until the CL block runs.

An IN argument of "cl\_type" accepts a local variable, parameter, or expression of any valid CL/AM data type.

An OUT argument of "cl\_type" accepts a local variable or parameter of any valid CL/AM data type.

#### D.4.1.2 Subroutine Return Status

Each of these subroutines has two return status values, history access error status and CL error return status.

Because this set of subroutines is not part of the standard AM Personality, the history access return status information is returned as numbers rather than as standard enumerations.

The CL error return status is returned as an enumeration of CLERRSTS.

#### D.4.1.3 CL Aborts

If there is insufficient stack memory available for the set to perform the requested operation, or if invalid arguments are passed to one of these subroutines, the CL Block is aborted with a CL Error Status (CLERRSTS) of Program Error (PROGERR).

## D.4.2 AMCL03\$Get\_Collection\_Rate Subroutine

This routine determines the collection rate for a specified point.parameter.

```
SUBROUTINE AMCL03$Get_Collection_Rate
  (Ret_Status      : OUT NUMBER;      -- History access error status
   CL_Error_Status : OUT CLERRSTS;    -- CL error enumeration
   Sample_Rate     : OUT NUMBER;      -- Collection rate in seconds
   Historized_Param : IN  cl_type)    -- Point.Parameter identification
```

### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the History Access error status. The returned value corresponds to the following status messages:

0.0 — Ok_cl_hstat	— No Errors
1.0 — Param_cl_hstat	— Error in Parameters
4.0 — Failure_cl_hstat	— Could not complete request
5.0 — Indeterminate_cl_hstat	— An error has occurred
6.0 — Errors_cl_hstat	— An error has occurred
7.0 — DA_cl_hstat	— Data Access Error
8.0 — memory_cl_hstat	— A memory error has occurred
9.0 — Conversion_cl_hstat	— Error Converting String
1000+ — Illogical_error_cl_hstat	— Contact Honeywell TAC

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

NOERROR	— No Error
LIMVIOL	— Limit Violation
RIGHTS	— Rights Error
COMMERR	— Communication Error
BADVALST	— Bad Value Status
COMABORT	— Communication Abort
ABORT	— Abort
ARITHERR	— Arithmetic Error
ARRAYLIM	— Array Limit Violation
RANGE	— Range Error
PROGERR	— Program Error
KEYLEVEL	— Keylevel Error
CNFERR	— Configuration Error

**Sample\_Rate** — The rate at which snapshot samples are collected for this parameter. The valid intervals are 5, 10, 20, and 60 seconds.

**Historized\_Param** — Identifies the desired point.parameter. This parameter can be expressed in ASCII form, internal form, or as a CDS variable of type entity.

**Examples:**

```

LOCAL status,           -- return status
&    rate      : NUMBER  -- sample rate
LOCAL cl_status : CLERRSTS -- CL status

CALL AMCL03$Get_Collection_Rate (status, cl_status, rate, point.pv)

```

"Rate" will contain the sample rate used to historize values for the parameter identified as "point.pv." This example used the internal form to identify the historized parameter. The following example uses the ASCII form of point.parameter identification.

```

LOCAL status,           -- return status
&    rate      : NUMBER  -- sample rate
LOCAL cl_status : CLERRSTS -- CL error status
LOCAL my_point  : STRING  -- names the point and parameter

SET my_point = "point.pv"

CALL AMCL03$Get_Collection_Rate (status, cl_status, rate, my_point)

```

### D.4.3 AMCL03\$Abs\_History Subroutine

This subroutine obtains snapshot history data for a specified point.parameter on the LCN based on absolute begin and end search dates and times. Any parameter that is historized, including an individual element of a parameter array, can be selected. A timestamp value is returned with each snapshot. When the beginning time is the current time, the first element is a Bad Value and the returned status for that value is missing because the HM has not yet collected the value.

```
SUBROUTINE AMCL03$Abs_History
(Ret_Status      : OUT      NUMBER;      -- History access error status
 CL_Error_Status : OUT      CLERRSTS;     -- CL error enumeration
 Time_Stamp_Array : OUT      cl_type;     --
 Status_Table    : OUT      cl_type;     --
 Values          : OUT      cl_type;     --
 Num_of_Values   : IN OUT   NUMBER;      --
 Sample_Rate     : IN       NUMBER;      --
 Scope_of_Search : IN       NUMBER;      --
 Begin_Date_Time : IN       TIME;        --
 End_Date_Time   : IN       TIME;        --
 Historized_Param : IN      cl_type)     -- Point.Parameter identification
```

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the History Access error status. The returned value corresponds to the following status messages:

0.0 — Ok_cl_hstat	— No Errors
1.0 — Param_cl_hstat	— Error in Parameters
2.0 — Complete_With_Item_Errors	— Some items contain errors
3.0 — Busy_cl_hstat	— System busy, request denied
4.0 — Failure_cl_hstat	— Could not complete request
5.0 — Indeterminate_cl_hstat	— An error has occurred
6.0 — Errors_cl_hstat	— An error has occurred
7.0 — DA_cl_hstat	— Data Access Error
8.0 — memory_cl_hstat	— A memory error has occurred
9.0 — Conversion_cl_hstat	— Error Converting String
1000+ — Illogical_error_cl_hstat	— Contact Honeywell TAC

**CL\_Error\_Status** — This parameter returns the CL Error return status as follows:

NOERROR	— No Error
LIMVIOL	— Limit Violation
RIGHTS	— Rights Error
COMMERR	— Communication Error
BADVALST	— Bad Value Status
COMABORT	— Communication Abort
ABORT	— Abort
ARITHERR	— Arithmetic Error
ARRAYLIM	— Array Limit Violation
RANGE	— Range Error
PROGERR	— Program Error
KEYLEVEL	— Keylevel Error
CNFERR	— Configuration Error

**Time\_Stamp\_Array** — An array of variables of the data type Time that represent the time stamps associated with each history value. Even when the values array contains a Bad Value, there will be an associated time stamp with a status for that element.

**Status\_Table** — An array of status values for each history value. This array should be used to verify the data in the values array and in the time stamp array. Note that a status of Normal Data can be accompanied by a Bad Value (see heading D.5.1) in the corresponding Values array.

0.0 — Normal_Data	— The value is analog
1.0 — Digital_Data	— The value is digital
7.0 — Time_Change	— A time change has occurred
8.0 — Missing	— This record is missing

**values** — This parameter presents the returned values in an array. It is possible for a Bad Value to be present in this array. If a Bad Value is present, check the associated status table. A Bad Value will appear any time that the associated status value is greater than 1.0.

**Num\_of\_Values** — Specifies the number of values expected. This number allows you to control how much space needs to be set aside for the data. If the actual number of returned values is less than the specified number, the actual number returned will be substituted. If there is more data than allowed for by this number, the excess data is ignored. The value of this parameter can range from 1 to 262.

**Sample\_Rate** — The rate at which Snapshot samples are to be collected (5, 10, 20, and 60 are the only valid inputs). Note that if the snapshots were taken and stored at a slower rate than the sample rate specified, the values array will contain a Bad Value for each missing record.

**scope\_of\_search** — Assigning the value 0 (zero) for the variable Scope\_of\_Search is appropriate only for Snapshots and User Averages. No other history types use the pre-archiving mechanisms.

Using a Scope\_of\_Search equal to

- 0 (zero) for all data — causes a search of the prearchived data, which can include up to 999 additional hours of data.
- 1 (one) for on-line data — searches only the most recent 8 hours of data.
- Using any other value—will default to 0 (zero).



**Begin\_Date\_Time** — This parameter specifies the start time for the search.

**End\_Date\_Time** — This is the end time for the absolute history search.

#### NOTE

The direction of history record search for snapshot values is based on the `begin_date_time` and `end_date_time` values. If the `begin_date_time` value is smaller than the `end_date_time` value, then the search is in the forward direction (from oldest data to newest data). If the `begin_date_time` value is greater than the `end_date_time` value, then the search is in the backward direction (from newest data to oldest data). You should be aware that a forward search takes at least twice as long to execute as a backward search. If the requested number of samples is greater than 262, the number of samples obtained is truncated at 262.

**Historized\_Param** — Identifies the desired point.parameter. This parameter can be expressed in ASCII form, internal form, or as a CDS variable of type entity.

#### Examples:

```

LOCAL status,                -- return status
&    size,                  -- number of values requested
&    rate      : NUMBER     -- sample rate
LOCAL cl_status : CLERRSTS   -- CL error status
LOCAL beg_time,             -- begin time
&    end_time  : TIME       -- end time
LOCAL values,              -- requested values array
&    statuses  : NUMBER ARRAY (1..262) -- status array
LOCAL times     : TIME ARRAY (1..262) -- timestamp array

SET size = 262
SET beg_time = (Date_Time) - 150 HOURS
SET end_time = (Date_Time) - 120 HOURS

CALL AMCL03$Get_Collection_Rate
&    (status, cl_status, rate, point.pv) -- get the sample rate

CALL AMCL03$Abs_History
&    (status, cl_status, times, statuses, values, size, rate, 0
&    beg_time, end_time, point.pv)

```

This example will obtain all snapshots between 150 hours ago and 120 hours ago for `point.pv` at the same rate that the history data was collected.

### D.4.4 AMCL03\$Rel\_History Subroutine

This subroutine obtains snapshot history data for a specified point.parameter on the LCN based on beginning and ending offsets (number of records) from current time. Any parameter that is historized, including an individual element of a parameter array, can be selected. Notice that no time stamp values are returned.

```
SUBROUTINE AMCL03$Rel_History
(Ret_Status      : OUT      NUMBER;      -- History access error status
 CL_Error_Status : OUT      CLERRSTS;    -- CL error enumeration
 Status_Table    : OUT      cl_type;     --
 Values          : OUT      cl_type;     --
 Num_of_Values   : IN OUT  NUMBER;      --
 Sample_Rate     : IN       NUMBER;      --
 Scope_of_Search : IN       NUMBER;      --
 Begin_Offset    : IN       NUMBER;      --
 End_Offset      : IN       NUMBER;      --
 Historized_Param : IN      cl_type)     -- Point.Parameter identification
```

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the History Access error status. The returned value corresponds to the following status messages:

0.0 — Ok_cl_hstat	— No Errors
1.0 — Param_cl_hstat	— Error in Parameters
2.0 — Complete_With_Item_Errors	— Some items contain errors
3.0 — Busy_cl_hstat	— System busy, request denied
4.0 — Failure_cl_hstat	— Could not complete request
5.0 — Indeterminate_cl_hstat	— An error has occurred
6.0 — Errors_cl_hstat	— An error has occurred
7.0 — DA_cl_hstat	— Data Access Error
8.0 — memory_cl_hstat	— A memory error has occurred
9.0 — Conversion_cl_hstat	— Error Converting String
1000+ — Illogical_error_cl_hstat	— Contact Honeywell TAC

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

NOERROR	— No Error
LIMVIOL	— Limit Violation
RIGHTS	— Rights Error
COMMERR	— Communication Error
BADVALST	— Bad Value Status
COMABORT	— Communication Abort
ABORT	— Abort
ARITHERR	— Arithmetic Error
ARRAYLIM	— Array Limit Violation
RANGE	— Range Error
PROGERR	— Program Error
KEYLEVEL	— Keylevel Error
CNFERR	— Configuration Error

**Status\_Table** — An array of status values for each history value. This array should be used to verify the data in the values array. Note that a status of Normal Data can be accompanied by a Bad Value (see heading D.5.1) in the corresponding Values array.

0.0 — Normal_Data	— The value is analog
1.0 — Digital_Data	— The value is digital
4.0 — Start_of_Online	— Marks first on-line value
5.0 — Start_of_Psearch	— Marks first prearchived value
7.0 — Time_Change	— A time change has occurred
8.0 — Missing	— This record is missing

**Values** — This parameter presents the returned values in an array. It is possible for a Bad Value to be present in this array. If a Bad Value is present, check the associated status table. A Bad Value will appear any time that the associated status value is greater than 1.0.

**Num\_of\_Values** — Specifies the number of values expected. This number allows you to control how much space needs to be set aside for the data. If the actual number of returned values is less than the specified number, the actual number returned will be substituted. If there is more data than allowed for by this number, the excess data is ignored. The value of this parameter can range from 1 to 480.

**Sample\_Rate** — The rate at which Snapshot samples are to be collected (5, 10, 20, and 60 are the only valid inputs). Note that if the snapshots were taken and stored at a slower rate than the sample rate specified, the values array will contain a Bad Value for each missing record.

**Scope\_of\_Search** — Assigning the value 0 (zero) for the variable Scope\_of\_Search is appropriate only for Snapshots and User Averages. No other history types use the pre-archiving mechanisms.

Using a Scope\_of\_Search equal to

- 0 (zero) for all data — causes a search of the prearchived data, which can include up to 999 additional hours of data.
- 1 (one) for on-line data — searches only the most recent 8 hours of data.
- Using any other value—will default to 0 (zero).

**Begin\_Offset** — This specifies the count of records (samples) from the current time to the point where data retrieval is to begin. The valid range of this data is 0 to 200,000 records.

**End\_Offset** — This specifies the count of records (samples) from the current time to the point where data retrieval is to end. The valid range of this data is 0 to 200,000 records.

#### NOTE

The direction of history record search for snapshot values is based on the begin\_offset and end\_offset values. If the begin\_offset value is greater than the end\_offset value, then the search is in the forward direction (from oldest data to newest data). If the begin\_offset value is smaller than the end\_offset value, then the search is in the backward direction (from newest data to oldest data). You should be aware that a forward search takes at least twice as long to execute as a backward search. The number of samples to be returned is calculated as the positive difference between begin and end offsets plus one. If this difference exceeds 480 samples, the number of samples obtained is truncated at 480.

**Historized\_Param** — Identifies the desired point.parameter. This parameter can be expressed in ASCII form, internal form, or as a CDS variable of type entity.

**Examples:**

```

LOCAL status,                -- return status
&    size,                  -- number of values requested
&    rate,                  -- sample rate
&    beg_off,               -- beginning offset
&    end_off    : NUMBER    -- ending offset
LOCAL cl_status : CLERRSTS   -- CL error status
LOCAL values,              -- requested values array
&    statuses   : NUMBER ARRAY (1..480) -- status array

SET size = 480
SET beg_off = 0              -- most recent sample
SET end_off = 480            -- 480 samples back

CALL AMCL03$Get_Collection_Rate
&    (status, cl_status, rate, point.pv) -- get the sample rate

CALL AMCL03$Rel_History
&    (status, cl_status, statuses, values, size, rate, 0
&    beg_off, end_off, point.pv)

```

This example will obtain all snapshots backwards from the current time to a time 480 records back at the same rate that the history data was collected. The time span covered is dependent on the sample rate.

**Application Notes** — Do not compare the AM CL relative history call to the Computer Gateway (CG) relative history call—they are different. The AM relative history call does not return time stamps, and therefore is able to return more data values in a single call than is possible with the AM absolute call or the CG relative call (480 vs. 262).

Since there are no time stamps returned with the AM CL relative history call, you must use the time that the call was made as the zero offset time. You can determine this time by accessing the CL TIME parameter immediately before making the relative history call. Because the history call is made in background CL, the current time used within the call will be slightly later and the data returned will occasionally be one sample time off. If this is critical, the absolute history call should be used.

The first entry in the Status\_Table array that is returned by the relative history call is always a Start of Data code (4 or 5) indicating whether the initial values returned are prearchive or on-line values. The corresponding (first) entry in the Values array is always Bad Value (NaN). Similarly, if the source of the data changes from prearchive to on-line or from on-line to prearchive (either could occur, depending on the direction of search), a Start of Data code will occur in the Status\_Table array at that point, and the corresponding Values array location will contain Bad Value (NaN).

When zero (0) offset is used, a Bad Value is usually returned as the first value. This is because the value has not been collected yet. The corresponding status in the Status\_Table will be 8.0 (Missing Status).

If the sample rate is faster than the history collection rate, the Status\_Table will have Missing status in the locations that correspond to items in the Values array for which history was not collected.

### D.4.5 AMCL03\$Avg\_History Subroutine

This subroutine (similar to Abs\_History) obtains averaged history data for a specified point.parameter on the LCN based on absolute begin and end search dates and times. Any parameter that is historized, including an individual element of a parameter array, can be selected. A timestamp value is returned with each average.

```
SUBROUTINE AMCL03$Avg_History
(Ret_Status      : OUT    NUMBER;    -- History access error status
 CL_Error_Status : OUT    CLERRSTS;  -- CL error enumeration
 Num_Samples_Array : OUT    cl_type;  --
 Min_Array       : OUT    cl_type;  --
 Max_Array       : OUT    cl_type;  --
 Time_Stamp_Array : OUT    cl_type;  --
 Status_Table    : OUT    cl_type;  --
 Values          : OUT    cl_type;  --
 Num_of_Values   : IN OUT NUMBER;    --
 Scope_of_Search : IN      NUMBER;    --
 Begin_Date_Time : IN      TIME;      --
 End_Date_Time   : IN      TIME;      --
 Retrieval_Type  : IN      NUMBER
 Historized_Param : IN      cl_type)  -- Point.Parameter identification
```

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the History Access error status. The returned value corresponds to the following status messages:

0.0 — Ok_cl_hstat	— No Errors
1.0 — Param_cl_hstat	— Error in Parameters
2.0 — Complete_With_Item_Errors	— Some items contain errors
3.0 — Busy_cl_hstat	— System busy, request denied
4.0 — Failure_cl_hstat	— Could not complete request
5.0 — Indeterminate_cl_hstat	— An error has occurred
6.0 — Errors_cl_hstat	— An error has occurred
7.0 — DA_cl_hstat	— Data Access Error
8.0 — memory_cl_hstat	— A memory error has occurred
9.0 — Conversion_cl_hstat	— Error Converting String
1000+ — Illogical_error_cl_hstat	— Contact Honeywell TAC

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

NOERROR	— No Error
LIMVIOL	— Limit Violation
RIGHTS	— Rights Error
COMMERR	— Communication Error
BADVALST	— Bad Value Status
COMABORT	— Communication Abort
ABORT	— Abort
ARITHERR	— Arithmetic Error
ARRAYLIM	— Array Limit Violation
RANGE	— Range Error
PROGERR	— Program Error
KEYLEVEL	— Keylevel Error
CNFERR	— Configuration Error

**Num\_Samples\_Array** — An array of type Number to receive the number of samples count for each average value.

**Min\_Array** — An array of type Number to receive the minimum values associated with each average value.

**Max\_Array** — An array of type Number to receive the maximum values associated with each average value.

**Time\_Stamp\_Array** — An array of variables of the data type Time that represent the time stamps associated with each history value. Even when the values array contains a Bad Value, there will be an associated time stamp with a status for that element.

**Status\_Table** — An array of status values for each history value. This array should be used to verify the data in the values array and in the time stamp array. Note that a status of Normal Data can be accompanied by a Bad Value (see heading D.5.1) in the corresponding Values array.

0.0 — Normal_Data	— The value is analog
6.0 — Some values were missing for requested period	— Non-Standard number of samples
7.0 — Time_Change	— A time change has occurred
8.0 — Missing	— This record is missing

**values** — This parameter presents the returned values in an array. It is possible for a Bad Value to be present in this array. If a Bad Value is present, check the associated status table. A Bad Value will appear any time that the associated status value is greater than 1.0.

**Num\_of\_Values** — Specifies the number of values expected. This number allows you to control how much space needs to be set aside for the data. If the actual number of returned values is less than the specified number, the actual number returned will be substituted. If there is more data than allowed for by this number, the excess data is ignored. The value of this parameter can range from 1 to 262.

**Scope\_of\_Search** — Assigning the value 0 (zero) for the variable Scope\_of\_Search is appropriate only for Snapshots and User Averages. No other history types use the pre-archiving mechanisms.

Using a Scope\_of\_Search equal to

- 0 (zero) for all data — causes a search of the prearchived data, which can include up to 999 additional hours of data.
- 1 (one) for on-line data — searches only the most recent 8 hours of data.
- Using any other value—will default to 0 (zero).

**Begin\_Date\_Time** — This parameter specifies the start time for the search.

**End\_Date\_Time** — This is the end time for the absolute history search.

#### NOTE

The direction of history record search for snapshot values is based on the begin\_date\_time and end\_date\_time values. If the begin\_date\_time value is smaller than the end\_date\_time value, then the search is in the forward direction (from oldest data to newest data). If the begin\_date\_time value is greater than the end\_date\_time value, then the search is in the backward direction (from newest data to oldest data). You should be aware that a forward search takes at least twice as long to execute as a backward search. If the requested number of samples is greater than 262, the number of samples obtained is truncated at 262.

**Retrieval\_Type** — Identifies the type of averages requested. The values are:

- 1.0 — Hourly Average
- 2.0 — Shift Average
- 3.0 — Daily Average
- 4.0 — Monthly Average
- 5.0 — User Average

**Historized\_Param** — Identifies the desired point.parameter. This parameter can be expressed in ASCII form, internal form, or as a CDS variable of type entity.

**Examples:**

```

LOCAL status,                -- return status
&    size,                  -- number of values requested
&    type      : NUMBER     -- type of average
LOCAL cl_status : CLERRSTS   -- CL error status
LOCAL beg_time,             -- begin time
&    end_time  : TIME       -- end time
LOCAL num_samples,         -- array for number of samples
&    minimums,             -- in each average value
&    maximums,             -- array for min value in avgs.
&    values,              -- array for max value in avgs.
&    statuses  : NUMBER ARRAY (1..262) -- requested values array
LOCAL times      : TIME ARRAY (1..262) -- status array
                                -- timestamp array

SET size = 262
SET beg_time = (Date_Time) - 120 HOURS
SET end_time = (Date_Time) - 150 HOURS

CALL AMCL03$Avg_History
&    (status, cl_status, num_samples, minimums, maximums, times,
&    statuses, values, size, 0, beg_time, end_time, 1, point.pv)

```

This example will obtain all hourly averages between 120 hours ago and 150 hours ago for point.pv. The values are collected in reverse chronological order (newest to oldest).



## D.5 SPECIAL ISSUES

### D.5.1 Bad Values

Records returned for times when the HM was down or historization was disabled will have a status value of 8.0 (Missing) and a "Bad Value" value assigned. "Bad value" returned with an associated normal status value indicates a bad process value or a NaN system value. See heading 2.3.1.1 for more information about **TotalPlant** Solution (TPS) System Bad Values.

### D.5.2 Time Change

The following discussion only applies to the absolute history call (Abs\_History) since a timestamp is provided for each snapshot only with that call.

#### D.5.2.1 Spring Time Change

A forward time change (for example, the spring daylight savings time change from 2:00 a.m. to 3:00 a.m.) leaves a gap in the history collection of a point. Any request for snapshot data within the skipped period returns no data. A request for hourly averages over a time period that surrounds the skipped time period (for example, from 1:00 a.m. to 4:59 a.m.) returns data for the first section of time, a time change record (status value 7.0), and data for the second section of time.

#### D.5.2.2 Fall Time Change

A backward time change (for example, the fall daylight savings time change from 2:00 a.m. back to 1:00 a.m.) results in data that overlap (two 1:00 a.m. time periods). When collecting this data, you must be careful that you are getting the proper set of data for the given time period. This can be assured by examining the begin time, the end time, and the direction of search.

If both the begin and end times are within the time change period (for example, 1:00 a.m. to 1:59 a.m.), then the data returned from the overlapping sets depends upon the direction of search. A forward search returns the earliest collected data (the data collected before the time change). A backward search returns the data collected last (the data collected after the time change occurred).

If both the begin time and the end time are outside the duplicated time period, then both sets of data are returned (along with data from outside the overlapped time period). The order of return of the overlapping sets depends on direction of the search. A search in the forward direction returns the data collected before the time change first, followed by the data collected after the time change. A backward direction of search produces the reverse ordering (the later data set collected is returned first).

There are four cases where either the begin or end time is within the time change period:

- 1) When begin time is earlier than the duplicated time period and the end time is within it, data from the overlapped time period is represented only by the first collection set.

- 2) When begin time is later than the duplicated time period and the end time is within it, data from the overlapped time period is represented only by the second collection set.
- 3) When begin time is within the duplicated time period and the end time is earlier than it, both sets of data from the overlapped time period are included. However, the data from the second collection set will be only for the period of time specified, while data from the first collection set will be for the complete period of overlap.
- 4) When begin time is within the duplicated time period and the end time is later than it, both sets of data from the overlapped time period are included. However, data from the first collection set will be only for the increment of time specified, while data from the second collection set will be for the complete period of overlap.

### **D.5.3 Scope of Search**

Assigning the value 0 (zero) for the variable `Scope_of_Search` is appropriate only for Snapshots and User Averages. No other history types use the pre-archiving mechanisms.

Using a `Scope_of_Search` equal to 1 (one) searches only the most recent 8 hours of data. Using a `Scope_of_Search` equal to 0 (zero) causes a search of the prearchived data, which can include up to 999 additional hours of data. Using any other value will default to 0 (zero).

## CL/AM EXTENSION FOR MATH LIBRARY

### Appendix E

*This appendix explains an available set of subroutines that enable a background CL program to prepare and manipulate matrices and to perform other mathematical operations.*

#### E.1 OVERVIEW OF THE MATH LIBRARY EXTENSION

The Math Library CL/AM extension consists of a set of subroutines that can be loaded in an AM. Many of the subroutines in this package can be used only by CL blocks linked to background insertion points, and some require the AM to have floating point hardware.

##### E.1.1 Features of this CL/AM Extension

This extension can be used to:

1. Calculate the standard deviation of an array of values.
2. Generate random numbers with uniform (0-1) distribution.
3. Generate random numbers with a pseudo-Gaussian distribution.
4. Create a two-dimensional local matrix from a single-dimension CDS array.
5. Create a single-dimension CDS array from a two-dimensional local matrix.
6. Perform matrix multiply, add, subtract, and transpose operations.
7. Perform matrix inversion and solution with right-hand/left-hand division options.
8. Create, reserve, initialize, and release matrix work areas.
9. Determine CPU usage of a CL block in milliseconds.
10. Multiply a matrix by a scalar.
11. Perform the singular value decomposition (SVD) of a matrix.

## E.1.2 Summary of Subroutines in this CL Extension

Table E-1 provides a summary of the subroutines in this package. Note that many of these routines can only be used by background CL programs.

**Table E-1 — Subroutines in CL/AM Math Library Extension**

Subroutine Name	Subsection
AMCL02\$Standard_Deviation	E.4.2
AMCL02\$Uniform_Random_Number	E.4.3
AMCL02\$Normal_Random_Number	E.4.4
AMCL02\$Get_Time	E.4.5
AMCL02\$Subtract_Time	E.4.6
AMCL02\$CDS_Array_to_Local_Matrix	E.4.7
AMCL02\$Local_Matrix_to_CDS_Array	E.4.8
AMCL02\$Create_Work_Area *	E.4.9
AMCL02\$Reserve_Work_Area *	E.4.10
AMCL02\$Release_Work_Area *	E.4.11
AMCL02\$Init_Matrix_Work_Area *	E.4.12
AMCL02\$Init_Matrix_Work_Area2 *	E.4.13
AMCL02\$Get_Work_Area_Info *	E.4.14
AMCL02\$Get_Matrix_Work_Area_Info *	E.4.15
AMCL02\$Get_Matrix_Info *	E.4.16
AMCL02\$Create_Matrix *	E.4.17
AMCL02\$Put_Element *	E.4.18
AMCL02\$Get_Element *	E.4.19
AMCL02\$Add_Matrices	E.4.20
AMCL02\$Subtract_Matrices	E.4.21
AMCL02\$Multiply_Matrices	E.4.22
AMCL02\$Multiply_Scalar	E.4.23
AMCL02\$Transpose_Matrix	E.4.24
AMCL02\$Matrix_Inversion	E.4.25
AMCL02\$SVD	E.4.26
* Math Library subroutines that can be used only by background CL programs.	

## E.1.3 Hardware Requirements

### E.1.3.1 Memory Used

The Math Library Extension set requires 24 K of memory

### E.1.3.2 Processor Type

Several of the functions in this set involve the use of floating point hardware, and therefore require an HMPU/K4LCN processor. See Table E-2 for a summary showing which Math Library subroutines require an HMPU or K4LCN. The individual call descriptions also indicate when an HMPU or K4LCN is required.

**Table E-2 — Math Library Subroutines That Require an HMPU//K4LCN**

Subroutine Name	Subsection
AMCL02\$Standard_Deviation	E.4.2
AMCL02\$Init_Matrix_Work_Area	E.4.12
AMCL02\$Init_Matrix_Work_Area2	E.4.13
AMCL02\$Get_Matrix_Work_Area_Info	E.4.15
AMCL02\$Get_Matrix_Info	E.4.16
AMCL02\$Multiply_Matrices	E.4.22
AMCL02\$Matrix_Inversion	E.4.25
AMCL02\$SVD	E.4.26
AMCL02\$Add_Matrices *	E.4.20
AMCL02\$Subtract_Matrices *	E.4.21
AMCL02\$Transpose_Matrix *	E.4.24
* HMPU/K4LCN required only when matrix is in a work area.	

## E.1.4 Work Areas

User-assignable work areas are available for the use by functions, such as the inversion of a large matrix, that require large amounts of working memory. Calls provided by this extension set prepare work areas where you can create and manipulate large matrices. Table E-3 lists the work area preparation subroutines.

**Table E-3 — Work Area Subroutines**

Subroutine Name	Subsection
AMCL02\$Create_Work_Area	E.4.9
AMCL02\$Reserve_Work_Area	E.4.10
AMCL02\$Release_Work_Area	E.4.11
AMCL02\$Init_Matrix_Work_Area	E.4.12
AMCL02\$Init_Matrix_Work_Area2	E.4.13
AMCL02\$Get_Work_Area_Info	E.4.14
AMCL02\$Get_Matrix_Work_Area_Info	E.4.15
AMCL02\$Get_Matrix_Info	E.4.16

The amount of memory that will be needed for all work areas in an AM must be specified on the AM's NCF custom page. CL extension subroutines take memory from this space at runtime to create work areas. Once a work area is allocated, the memory cannot be reclaimed for other uses except by reinitialization on AM startup. User space is reduced by the amount of memory allocated for work areas.

CL programs that only manipulate smaller matrices need not use work areas. Generally, work areas need be considered only if inversion of matrices larger than  $25 \times 25$  is required.

### E.1.5 Matrix Size Restrictions

For operations other than inversion, matrix operations work on local matrices of any dimensions allowed by the stack size. For matrix inversion, matrix size limitations depend on two choices that you make about the CL block,

- 1) whether it runs in foreground or background, and
- 2) whether or not the matrix is located in a work area.

If the CL block executes in foreground,  $15 \times 15$  is the largest matrix size that should be inverted outside work areas (this will require 1/4 second which seems a realistic limit for foreground execution). Background CL blocks can invert up to  $25 \times 25$  matrices outside work areas.

Work area matrices can have up to 8000 rows and 16000 columns for single precision, or up to  $8000 \times 8000$  for double precision. These limits are far beyond the realistic range of invertible matrices and are provided for other operations such as addition or subtraction.

### E.1.6 Array and Array Size Arguments

In this document, the term **vector** denotes a one-dimensional number array and **matrix** denotes a two-dimensional number array. A vector may also be considered a special case of a matrix having only one row or column. This is useful since CL does not support two-dimensional arrays with only one row or column.

Many of the Math Library routines have arguments that must be vectors or matrices. Because these arguments are handled similarly in all of the Math Library routines, a general description of the form for these arguments, and related error handling, is provided here. Any exceptions are noted in the explanations of the individual routines that start at E.4.

A matrix not in a work area is a CL array of type number. It can be one-dimensional or two-dimensional. A one-dimensional array represents either a column matrix or a row matrix depending on the context and on its size arguments. The dimensions of the array can be indexed by either enumerations or subranges, but the Math Library routines always treat array indices as normalized subranges (1 to an upper bound). Computations

with arrays of a different form will be correct, but input must be adjusted: array size arguments are input as numbers, and routines which operate on a user-defined part of an array require an offset, not an index, to determine which part of the array to use. The Math Library routines will use matrices of any size supported by CL, although some routines allow only small matrices in foreground computations.

A vector is a one-dimensional number array. It can be either a local CL array or a CDS array. A local CL vector can be indexed either by an enumeration or by a subrange, but a CDS vector must be indexed by a subrange. Array indices, like those of matrices are always treated as normalized subranges (1 to an upper bound). Vector size may be limited by constraints imposed by other matrix arguments.

Most array (vector or matrix) arguments to the Math Library routines have associated size arguments. These arguments often specify the sizes of more than one array argument. The `Mid_Rows_Columns` argument of `AMCL02$Multiply_Matrices`, for example, is the number of columns in the first factor matrix, and also the number of rows in the second factor matrix. Array size arguments allow the user to store logical arrays of different sizes in the same physical array. A physical array is a CL or CDS array which is used to store data, and its size is determined at compile time. A logical array is composed of the data actually stored in part of its associated physical array, and can have any size up to the size of the physical array. A logical vector is stored in an initial segment of its associated physical vector, and a logical matrix is stored in a principal (top left) submatrix of its associated physical matrix.

The array size arguments passed in to the Math Library routines are the sizes of logical arrays. An error is returned if an array size argument is outside the range of possible CL array sizes, and the calling block aborts with `PROGERR` status if a Math Library routine is called with physical array arguments not large enough to hold the logical arrays specified by the array size arguments. Passing in a zero for any logical array size tells the Math Library routines to use physical array sizes for logical array sizes. This places tighter restrictions on the arrays passed in as arguments. Any group of logical dimensions that must be the same also must have equal physical dimensions. For example, calling `AMCL02$Multiply_Matrices` with a zero for the `Mid_Rows_Columns` argument will abort unless the number of rows in the second physical factor matrix is equal to the number of columns in the first physical factor matrix.

## E.2 PACKAGING

The Math Library CL Extension is packaged as the set, `AMCL02`, plus a set definition file.

`AMCL02.LO` contains routines unique to the Math Library functions.

The set definition file, `AMCL02.SF`, defines the Math Library set routines to the CL compiler.

## E.3 INSTALLATION AND CONFIGURATION

The initial step in installation of CL/AM extensions is to copy the conversion routine set and the set definition file from the delivered media to a History Module.

- AMCL02.LO must be copied to NET>&CUS>AMCL02.LO.
- AMCL02.SF must be copied to NET>&CLX>AMCL02.SF.

Any AM that is loaded with the extension sets must be correctly configured via the Network Configurator. To do this, perform the following steps:

1. Set the default Volume Path for the Network Configurator Backup to point at a device where the current NCF will be saved.
2. Invoke the Network Configurator from the Engineers Main Menu by selecting "LCN Nodes."
3. Select the node number of the AM to be configured.
4. Select the "Modify Node" target on page 1.
5. In the "Additional Module Memory" port on page 3, enter the total number of words that will be required for all work areas—both for this extension and for any others that use work areas.
6. On page 3 also enter the name of the File Set (AMCL02) in the list of externally loaded modules and press the Enter key.
7. Press the Check Key (F1) and the configuration should check OK.
8. Press the Install Key (F2) and the configuration should install without error.

The AM can now be loaded.



## E.4 CL CALLABLE SUBROUTINES

### E.4.1 Introduction

The functions provided by this extension are implemented as CL callable subroutines. Descriptions of the individual subroutines begin at heading E.4.2.

#### E.4.1.1 Including the Set Definition File

The subroutines are declared in the set definition file which is read by the CL compiler when an "include\_set" directive is encountered in a CL source file within the scope of a CL Block. The directive appears in your source file as follows: `%INCLUDE_SET AMCL02`

Each routine may be invoked by calling it in a CL Block or Subroutine via the Call statement, such as:

```
Call AMCL02$xxxxxxx (status, cl_status, xxx, xx)
```

Several of the subroutines in this extension set have arguments designated to be of the stand-in data type "cl\_type." The actual data types of these arguments, as specified in your calls, can be any valid CL/AM data type within the rules given below. For these arguments only, normal CL compiler data-type checking is inhibited, and any data type mismatches you may accidentally create are not detected until the CL block runs.

An IN argument of "cl\_type" accepts a local variable, parameter, or expression of any valid CL/AM data type.

An OUT argument of "cl\_type" accepts a local variable or parameter of any valid CL/AM data type.

#### E.4.1.2 Subroutine Return Status

Some of these subroutines have two return status values, a Math Library return status and a CL error return status. The CL error return status is only significant when the Math Library return status is equal to 4.0 (CL error).

Because this set of subroutines is not part of the standard AM Personality, the Math Library return status information is returned as numbers rather than as standard enumerations. See heading E.6 for a summary of return status values for this extension.

The CL error return status is returned as an enumeration of CLERRSTS.

### E.4.1.3 CL Aborts

If there is insufficient stack memory available for the set to perform the requested operation, or if invalid arguments are passed to some of these subroutines, the CL Block is aborted with a CL Error Status (CLERRSTS) of Program Error (PROGERR).

### E.4.2 AMCL02\$Standard\_Deviation Subroutine

This subroutine is used to calculate the mean and standard deviation of an array of numbers.

```
SUBROUTINE AMCL02$Standard_Deviation
  (Ret_Status      : OUT NUMBER;      -- Math Library return status
   CL_Error_Status : OUT CLERRSTS;    -- CL error enumeration
   Mean            : OUT NUMBER;      -- Returned mean
   Standard_Deviation : OUT NUMBER;  -- Returned standard deviation
   Source_Array    : IN  cl_type;    -- Source array name
   Number_of_Items : IN  NUMBER;      -- Number of items to use
   Starting_Offset  : IN  NUMBER;      -- Starting element offset
   Bad_Value_Flag   : IN  LOGICAL)    -- If TRUE, skip any Bad Values;
                                     -- If FALSE, abort on Bad Value
```

This subroutine does not support Work Area matrices, but does require an HMPU processor.

This routine calculates the mean and standard deviation (square root of the variance) of a section of an array. The section of Number\_of\_Items elements beginning at Starting\_Offset is used in the calculation.

The source array is a vector argument. Its logical size is (Number\_of\_Items + Starting\_Offset - 1) elements. (See heading E.1.6, Array and Array Size Arguments.)

The action taken if this routine encounters a bad or infinite value in the source array depends on the Bad\_Value\_Flag argument.

TRUE = Ignore (skip over) any bad or infinite values.

FALSE = Terminate the calculation and return an error if a bad or infinite value is encountered.

If a bad value is generated in the calculation, the calculation is terminated and an error is returned regardless of the Bad\_Value\_Flag value.

Note that the Starting\_Offset argument is not an index—it is an offset from the beginning of the array. If the lower bound of the array is 1, the Starting\_Offset should be the same as the desired starting index. Otherwise, the Starting\_Offset should be set to (starting\_index - lower\_bound + 1). If the array is indexed by an enumeration, the Starting\_Offset should be set to the ordinal value of the enumeration state at which the calculation is to start.

The source array is not modified by this routine.

**Definition of Parameters:**

**Ret\_status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to, or generated by, the calculation.
- 2.0 — Bad CL Reference List number—internal error, call Honeywell TAC.
- 3.0 — Bad input error—an input value is either bad or out of range.
- 4.0 — CL error—use the CL\_Error\_Status return value to determine the reason.
- 9.0 — Bad standard deviation array—fewer than two useable values were found in the source array.
- 27.0 — HMPU is required

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

NOERROR — No Error  
 LIMVIOL — Limit Violation  
 RIGHTS — Rights Error  
 COMMERR — Communication Error  
 BADVALST — Bad Value Status  
 COMABORT — Communication Abort  
 ABORT — Abort  
 ARITHERR — Arithmetic Error  
 ARRAYLIM — Array Limit Violation  
 RANGE — Range Error  
 PROGERR — Program Error  
 KEYLEVEL — Keylevel Error  
 CNFERR — Configuration Error

**Example:**

The following CL code fragment calculates the standard deviation of a five element segment of an eight element local CL array.

```

LOCAL status,           -- Math Library return status
& start,               -- Start element offset from lower bound
& num_data,            -- Number of data items
& mean,                -- Returned mean
& std_dev : NUMBER      -- Returned standard deviation
LOCAL cl_status : CLERRSTS -- CL error status
LOCAL bad_val : LOGICAL  -- Establish bad/infinite value handling
LOCAL data : NUMBER ARRAY (5..12) -- Data array
.
.
SET start = 2.0 -- Start at the second element in the array (index = 6)
               -- Start = index - lower bound + 1 (6 - 5 + 1 = 2)
SET num_data = 5 -- Use five elements in the calculation
CALL AMCL02$Standard_Deviation
& (status, cl_status, mean, std_dev, data, num_data, start, bad_val)

```

Given these initial values for "data": [4.5 6.0 2.3 5.2 5.4 3.5 5.5 2.1]  
 The values used will be: [ 6.0 2.3 5.2 5.4 3.5 ]  
 The returned value for "std\_dev" will be: 1.531992...

### E.4.3 AMCL02\$Uniform\_Random\_Number Subroutine

This subroutine generates a sequence of random numbers in the range (0,1) with a uniform distribution. Each call generates one random value.

```
SUBROUTINE AMCL02$Uniform_Random_Number
  (Ret_Status      : OUT  NUMBER;      -- Math Library return status
   Random_Number   : OUT  NUMBER)      -- Returned value
```

This routine uses a standard linear congruential random number generator, but shuffles the output, effectively eliminating the sequential correlation found in most standard linear congruential random number generators. It can, therefore, be used to generate sequences of numbers which are effectively random, as well as individual random values. The seed and the generator function are initialized with a clock value (whenever), and the seed is not normally modified by anything other than the generator function.

This subroutine does not require an HMPU processor.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 7.0 — Internal error, call Honeywell TAC

#### Example:

The following code fragment generates a random number in the range of (0,1).

```
LOCAL status,                -- Math Library return status
&    random_val : NUMBER      -- Returned random value
.
.
CALL AMCL02$Uniform_Random_Number (status, random_val)
```

### E.4.4 AMCL02\$Normal\_Random\_Number Subroutine

This subroutine generates a sequence of random numbers with a standard normal (Gaussian) distribution. Each call generates one random value.

```
SUBROUTINE AMCL02$Normal_Random_Number
  (Ret_Status      : OUT  NUMBER;      -- Math Library return status
   Random_Number   : OUT  NUMBER)      -- Returned value
```

This routine uses the same method (and the same subroutine) as AMCL02\$Uniform\_Random\_Number to generate uniformly distributed random values in the unit interval and then applies a transformation to obtain normally distributed random values with zero mean and unit variance. Although a random value can have any real value, the probability of a generated value falling outside the range (-3, +3) is less than 2% and the probability of a value falling outside the range (-5, +5) is infinitesimal.

This subroutine does not require an HMPU processor.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 7.0 — Internal error, call Honeywell TAC

#### Example:

The following code fragment generates a normally distributed random number.

```
LOCAL status,                -- Math Library return status
&    random_val : NUMBER      -- Returned random value
.
.
CALL AMCL02$Normal_Random_Number (status, random_val)
```

### E.4.5 AMCL02\$Get\_Time Subroutine

This subroutine gets current time as two values, one value in seconds and the other in tenths of milliseconds. It is designed to be used in conjunction with the AMCL02\$Subtract\_Time subroutine to provide CL programmers with a way to calculate CPU usage by a CL block.

If two Get\_Time calls are made in the same foreground CL block (to obtain a start time and a stop time), and Subtract\_Time is called to obtain the difference, the result is CPU use in milliseconds. However, if the same calls are made in background CL, the difference is elapsed time which may or may not be the same as CPU usage (because CL block execution can be interrupted).

```
SUBROUTINE AMCL02$Get_Time
  (Seconds      : OUT  NUMBER;      -- Time seconds
   Tenths_of_msecs : OUT  NUMBER)  -- Time tenths of milliseconds
```

This subroutine has no status return and does not require an HMPU processor.

### E.4.6 AMCL02\$Subtract\_Time Subroutine

This subroutine calculates the difference between two sets of values that have been obtained by AMCL02\$Get\_Time. The result is the difference in milliseconds between Time 2 (stop time) and Time 1 (start time).

```
SUBROUTINE AMCL02$Subtract_Time
  (Result      : OUT NUMBER;      -- Time difference in milliseconds
   Seconds1     : IN  NUMBER;      -- Start time seconds
   Tenths_of_Msecs1 : IN  NUMBER;  -- Start time tenths of milliseconds
   Seconds2     : IN  NUMBER;      -- Stop time seconds
   Tenths_of_Msecs2 : IN  NUMBER)  -- Stop time tenths of milliseconds
```

If the calculated result is a bad value, it is set to zero. There is no status return. This subroutine does not require an HMPU processor.

#### Example:

```
LOCAL Secs1,          -- Time 1 seconds
&    Tms1,            -- Time 1 tenths of milliseconds
&    Secs2,           -- Time 2 seconds
&    Tms2,            -- Time 2 tenths of milliseconds
&    Result : NUMBER  -- Difference in milliseconds

CALL AMCL02$Get_Time (secs1, tms1)
.                      -- Code whose execution time is being
.                      -- measured would go here
CALL AMCL02$Get_Time (secs2, tms2)
CALL AMCL02$Subtract_Time (result, secs1, tms1, secs2, tms2)
```

### E.4.7 AMCL02\$CDS\_Array\_to\_Local\_Matrix Subroutine

This subroutine is used to create a two-dimensional matrix from a single-dimensional CDS array.

```
SUBROUTINE AMCL02$CDS_Array_to_Local_Matrix
  (Ret_Status      : OUT NUMBER;      -- Math Library return status
   CL_Error_Status : OUT CLERRSTS;    -- CL error enumeration
   Destination_Array : OUT cl_type;   -- Destination matrix array
   Dest_Rows       : IN  NUMBER;      -- Number of rows in matrix
   Dest_Columns    : IN  NUMBER;      -- Number of columns in matrix
   Source_Array    : IN  cl_type)    -- Source array
```

This subroutine does not support Work Area matrices.

The elements of the source array are copied in row order into the destination matrix. This means that, starting with the first element, successive pieces of length N in the array become successive rows in the matrix (N = the number of columns in the destination array). The source array is not modified by this routine.

The subroutine assumes that the CDS source array contains the matrix in row order. The source array is a vector, but must be a CDS array, not a CL array. The minimum size for the source vector is (Dest\_Rows \* Dest\_Columns), but it can be larger. If the source vector is larger than the destination array, the extra elements are ignored. The destination array is a matrix argument whose size is given by the Dest\_Rows and Dest\_Columns arguments. (See heading E.1.6, Array and Array Size Arguments.) This routine allows Bad Values to be copied into the destination matrix.

This subroutine does not require an HMPU processor.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful copy
- 2.0 — Bad CL Reference List number—internal error, call Honeywell TAC.
- 3.0 — Bad input
- 4.0 — CL error—use the CL\_Error\_Status return value to determine the reason.

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

NOERROR	— No Error
LIMVIOL	— Limit Violation
RIGHTS	— Rights Error
COMMERR	— Communication Error
BADVALST	— Bad Value Status
COMABORT	— Communication Abort
ABORT	— Abort
ARITHERR	— Arithmetic Error
ARRAYLIM	— Array Limit Violation
RANGE	— Range Error
PROGERR	— Program Error
KEYLEVEL	— Keylevel Error
CNFERR	— Configuration Error

**Example:**

```

LOCAL status,                -- Math Library return status
&   rows,                   -- Number of rows in matrix
&   columns : NUMBER        -- Number of columns in matrix
LOCAL cl_status : CLERRSTS   -- CL error status

LOCAL matrix : NUMBER ARRAY (1..2,1..2) -- Destination matrix
.
.
SET rows, columns = 2.0
CALL AMCL02$CDS_Array_to_Local_Matrix
& (status, cl_status, matrix, rows, columns, point.cds_array)

```

Given these values for "point.cds\_array": [2.0 3.1 4.5 3.0 5.0 7.5],

Final value for the matrix will be: [2.0 3.1]  
[4.5 3.0]



### E.4.8 AMCL02\$Local\_Matrix\_to\_CDS\_Array Subroutine

This routine is used to create a single-dimensional CDS array from a matrix in a two-dimensional CL Local Array.

```
SUBROUTINE AMCL02$Local_Matrix_to_CDS_Array
  (Ret_Status      : OUT NUMBER;      -- Math Library return status
   CL_Error_Status : OUT CLERRSTS;    -- CL error enumeration
   Destination_Array : OUT cl_type;   -- Destination array
   Source_Rows      : IN  NUMBER;     -- Number of rows in matrix
   Source_Columns   : IN  NUMBER;     -- Number of columns in matrix
   Source_Array     : IN  cl_type)    -- Source matrix array
```

This subroutine does not support Work Area matrices.

The local matrix is copied to the CDS array in row order. This means that the rows of the matrix are copied straight into the array, starting with row one, and "wrapped around" at the ends. The source matrix array is not modified by this routine.

The source array is a matrix argument whose size is given by the Source\_Rows and Source\_Columns arguments. The destination array is a vector argument, but it must be a CDS array, not a CL array. The logical size of the destination vector is not passed in as an argument, but is assumed to be the minimum of the physical size of the vector and the logical size (rows \* columns) of the source array. The logical sizes of the arrays are not required to match in this routine: the extra elements in the larger array are ignored. This routine does not allow Bad Values to be copied into the CDS array. If a store of a Bad Value is attempted, the operation is terminated at the point of the store, and a CL error status of BADVALST is returned.

This subroutine does not require an HMPU processor.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful copy
- 2.0 — Bad CL Reference List number—internal error, call Honeywell TAC
- 3.0 — Bad input
- 4.0 — CL error—use the CL\_Error\_Status return value to determine the reason.

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

NOERROR	— No Error
LIMVIOL	— Limit Violation
RIGHTS	— Rights Error
COMMERR	— Communication Error
BADVALST	— Bad Value Status
COMABORT	— Communication Abort
ABORT	— Abort
ARITHERR	— Arithmetic Error
ARRAYLIM	— Array Limit Violation
RANGE	— Range Error
PROGERR	— Program Error
KEYLEVEL	— Keylevel Error
CNFERR	— Configuration Error

**Example:**

The following CL code fragment copies the principal (top left) 2x3 submatrix of a 3x4 Local CL matrix to a 6-element CDS array.

```

LOCAL status,                -- Math Library return status
&    rows,                  -- Number of rows in source matrix
&    columns : NUMBER      -- Number of columns in source matrix
LOCAL cl_status : CLERRSTS  -- CL error status

LOCAL matrix : NUMBER ARRAY (1..3,1..4) -- source matrix
.
.
SET rows      = 2.0          -- Use only the first two rows
SET columns   = 3.0          -- Use only the first three columns

CALL AMCL02$Local_Matrix_to_CDS_Array
& (status, cl_status, point.cds_array, rows, columns, matrix)

```

Given these initial values for the matrix:    [2.0 3.1 6.8 7.9],  
    [4.5 3.0 8.1 6.4]  
    [3.2 5.8 3.9 2.0]

Final value for "point.cds\_array" will be:    [2.0 3.1 6.8 4.5 3.0 8.1]

### E.4.9 AMCL02\$Create\_Work\_Area Subroutine

This subroutine is used to create a work area from part (or all) of the additional memory specified on the external load module page for the node that the CL is running on. The work area stays created so long as the AM stays up. Thus, the memory allocated for the work area cannot be reassigned for other use without reinitialization of the AM.

```
SUBROUTINE AMCL02$Create_Work_Area
  (Ret_Status      : OUT      NUMBER;      -- Math Library return status
   Area_Name       : IN       STRING;       -- Eight-character name for area
   Size_in_Words   : IN       NUMBER)      -- Number of words to reserve
```

If there is room, a matrix work area is created with the specified size and name.

The size required for a work area depends on its intended use. The following describes how to compute the space required for a matrix work area.

The required size of a matrix work area is the sum of the size of the inverted work space plus the sum of the sizes for each user-created matrix.

The inverted work space must be an area that can accommodate the largest matrix that will be inverted. This size is four times the order of the matrix squared, plus additional space for fast access. The following equation defines the number of words required:  $(4 * n * n + 15 * n)$  where  $n$  is the order of the matrix.

User-created matrices require two words per element, plus additional space for fast access. The following equation defines the number of words required for each user-created matrix:  $(2 * \text{rows} * \text{columns} + 2 * \text{rows} + 20)$ .

Before a work area can be used, it must be reserved and then initialized. Only one area can be reserved by a point at one time.

This subroutine runs in Background CL only and does not require an HMPU processor.

#### Definition of Parameters:

**Ret\_status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Bad input
- 5.0 — A work area with the specified name already exists
- 7.0 — Internal error, call Honeywell TAC
- 10.0 — No more work areas can be created
- 11.0 — Not enough memory to create work area

**Example:**

A 300,000-word work area named "area1" is created by the following code fragment.

```
LOCAL status      : NUMBER          -- Math Library return status
LOCAL workarea    : STRING          -- Name to be given to the work area

SET workarea = "area1"
CALL AMCL02$Create_Work_Area (status, workarea, 300000)
```

### E.4.10 AMCL02\$Reserve\_Work\_Area Subroutine

This subroutine is used to reserve a work area.

```
SUBROUTINE AMCL02$Reserve_Work_Area
  (Ret_Status      : OUT  NUMBER;      -- Math Library return status
   Work_Area       : OUT  NUMBER;      -- Work area id number for later calls
   Area_Name       : IN   STRING)      -- Eight-character name for area
```

Before a work area can be used, it must be reserved and then initialized (see the subroutine AMCL02\$Init\_Matrix\_Work\_Area). Only one area can be reserved by a point at one time. The area must be released, or the reserving point must be deleted, before another point can use the area.

This subroutine runs in Background CL only and does not require an HMPU processor.

#### Definition of Parameters:

**Ret\_status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 7.0 — Internal error, call Honeywell TAC
- 12.0 — Work area does not exist
- 13.0 — Work area already reserved
- 29.0 — Point already has reserved another work area

#### Example:

The work area named "area1" is reserved by this code fragment.

```
LOCAL status,                -- Math Library return status
&   area_num  : NUMBER       -- Work area id number to be returned
LOCAL workarea : STRING      -- Name of the work area to be reserved
.
.
SET workarea = "area1"
CALL AMCL02$Reserve_Work_Area (status, area_num, workarea)
```

### E.4.11 AMCL02\$Release\_Work\_Area Subroutine

This subroutine is used to release a work area. It can be called when the last CL on a background point runs or the work area is not to be called again. The work area could remain reserved for this point as long as the AM stays up.

```
SUBROUTINE AMCL02$Release_Work_Area
  (Ret_Status      : OUT      NUMBER;      -- Math Library return status
   Area_Name       : IN       STRING)      -- Work area name
```

If the point is deleted before the area is released, the area is released by the system so that the area can be reserved by another point.

This subroutine does not require the HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 7.0 — Internal error, call Honeywell TAC
- 12.0 — Work area does not exist
- 13.0 — Work area is reserved by another point

#### Example:

This code fragment releases the "area1" work area.

```
LOCAL status      : NUMBER      -- Math Library return status
LOCAL workarea    : STRING      -- Name of the work area
.
.
SET workarea = "area1"
CALL AMCL02$Release_Work_Area (status, workarea)
```

### E.4.12 AMCL02\$Init\_Matrix\_Work\_Area Subroutine

This subroutine initializes a reserved work area for matrix use.

```
SUBROUTINE AMCL02$Init_Matrix_Work_Area
(Ret_Status      : OUT      NUMBER;    -- Math Library return status
 Area_Name       : IN       STRING;     -- Work area name
 Invert_Order    : IN       NUMBER)    -- Dimension of largest matrix
                                         -- to be inverted
```

This subroutine indicates that the work area will be used for matrices and reserves space within the work area for the largest matrix to be inverted plus overhead space for fast access. The number of words that will be reserved is  $(4 * n * n + 15 * n)$  where  $n$  is the value of `Invert_Order`.

This subroutine requires an HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 11.0 — Not enough memory in work area for inverted matrix
- 12.0 — Work area does not exist
- 13.0 — Work area is reserved by another point
- 18.0 — Work area is already initialized
- 27.0 — HMPU is required

#### Example:

This code fragment reserves room for a 100 x 100 matrix in the "area1" work area.

```
LOCAL status      : NUMBER      -- Math Library return status
LOCAL workarea    : STRING      -- Name of the work area
.
.
SET workarea = "area1"
CALL AMCL02$Init_Matrix_Work_Area (status, workarea, 100)
```

### E.4.13 AMCL02\$Init\_Matrix\_Work\_Area2 Subroutine

This subroutine initializes a reserved work area for matrix use, including computation of inverses and matrix Singular Value Decompositions (SVDs). Refer to E.4.26 for more information about matrix singular value decomposition.

```
SUBROUTINE AMCL02$Init_Matrix_Work_Area2
  (Ret_Status      : OUT      NUMBER;    -- Math Library return status
   Area_Name       : IN       STRING;     -- Work area name
   Invert_Order    : IN       NUMBER;     -- (p) Maximum order in inversion
   SVD_Rows        : IN       NUMBER;     -- (m) Maximum rows in SVD
   SVD_Columns     : IN       NUMBER)     -- (n) Maximum columns in SVD
```

This subroutine indicates that the work area will be used for matrices, and reserves space within the work area for the largest matrix inversion and SVD calculations. Matrix inversion and SVD require storage space for the matrices used in the calculations and overhead space for fast access. Zeros may be input as a maximum size if the corresponding function (inversion or SVD) is not needed.

The number of words that will be reserved is:

$$4\alpha^2 + 2\alpha + 12\beta + p + 4\delta n + 2\delta$$

where: m x n specifies the maximum size of a SVD matrix,  
       p x p specifies the maximum size of a matrix to invert,  
        $\alpha$  = maximum (m,p) [whichever is greater, m or p],  
        $\beta$  = maximum (p, minimum(m,n) + 1), and  
        $\delta$  = maximum (m,n).

This subroutine requires an HMPU processor and software release 401 or later. It runs in Background CL only. The subroutine AMCL02\$SVD (see E.4.26) will function only if the work area is initialized by this subroutine.

#### Definition of Parameters:

**Ret\_status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 11.0 — Not enough memory in work area for inverted matrix
- 12.0 — Work area does not exist
- 13.0 — Work area is reserved by another point
- 18.0 — Work area is already initialized
- 27.0 — HMPU is required



**Example:**

This code fragment declares the work area for matrix use, and creates the necessary work space for a 100 x 100 matrix to be inverted and for the SVD of a 200 x 50 matrix to be calculated:

```
LOCAL status      : NUMBER          -- Math Library return status
LOCAL workarea    : STRING          -- Name of the work area
.
.
SET workarea = "area1"
CALL AMCL02$Init_Matrix_Work_Area2 (status, workarea, 100, 200, 50)
```

### E.4.14 AMCL02\$Get\_Work\_Area\_Info Subroutine

This subroutine is used to obtain information about a specified work area.

```
SUBROUTINE AMCL02$Get_Work_Area_Info
  (Ret_Status      : OUT      NUMBER; -- Math Library return status
   Work_Area       : IN OUT   NUMBER; -- Work area number
   Area_Name       : IN OUT   STRING; -- Work area name
   Size            : OUT      NUMBER; -- Words allocated to this work area
   Reserving_Point : OUT      STRING; -- Point id of reserving point
   Free_Space      : OUT      NUMBER; -- Number of words still free
   Work_Area_Type  : OUT      NUMBER) -- Indicates type of work area
```

If `Work_Area` equals zero, then the subroutine searches for the work area named by `Area_Name`. Otherwise, the work area number is used to obtain information about the specified work area.

This subroutine does not require the HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 12.0 — Work area does not exist
- 15.0 — Work area index is out of range

**Work\_Area\_Type** — This parameter defines the work area status as follows:

- 0.0 — Work area is free (no space has been allocated)
- 1.0 — Work area is allocated but not initialized
- 2.0 — Work area has been initialized to support matrices

#### Example:

This code fragment gets information about the "areal" work area by name search.

```
LOCAL status,                -- Math Library return status
&   area_num,                --
&   free_space,              --
&   size,                    --
&   work_area_type : NUMBER  --
LOCAL reserving_point,      --
&   workarea          : STRING -- Name of the work area
.
.
SET workarea = "areal"
CALL AMCL02$Get_Work_Area_Info (status, area_num, workarea, size,
&                               reserving_point, free_space,
&                               work_area_type)
```

### E.4.15 AMCL02\$Get\_Matrix\_Work\_Area\_Info Subroutine

This subroutine returns information specific to a work area initialized for matrices.

```
SUBROUTINE AMCL02$Get_Matrix_Work_Area_Info
  (Ret_Status      : OUT    NUMBER; -- Math Library return status
   Work_Area       : IN OUT NUMBER; -- Work area number
   Area_Name       : IN OUT STRING; -- Work area name
   Invert_Order    : OUT    NUMBER; -- Order of invert work space
   Number_of_Matrices : OUT    NUMBER) -- Number of matrices created
```

If `Work_Area` equals zero, then the subroutine searches for the work area named by `Area_Name`. Otherwise, the work area number is used to obtain information about the specified work area.

This subroutine requires the HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 12.0 — Work area does not exist
- 15.0 — Work area index is out of range
- 19.0 — Work area is not a matrix work area
- 27.0 — HMPU is required

#### Example:

This code fragment searches by name for information about the "area1" work area.

```
LOCAL status,          -- Math Library return status
&   area_num,         -- Work area number
&   invert,           -- Order of invert work space
&   matrices          : NUMBER  -- Number of matrices created
LOCAL workarea        : STRING  -- Name of the work area
.
.
SET workarea = "area1"
CALL AMCL02$Get_Matrix_Work_Area_Info (status, area_num, workarea,
&                                     invert, matrices)
```

## E.4.16 AMCL02\$Get\_Matrix\_Info Subroutine

This subroutine returns information about a specified matrix within a work area.

```
SUBROUTINE AMCL02$Get_Matrix_Info
  (Ret_Status      : OUT    NUMBER; -- Math Library return status
   Work_Area       : IN     NUMBER; -- Work area number
   Matrix_Number    : IN OUT NUMBER; -- Number of matrix
   Matrix_Name      : IN OUT STRING; -- Name of matrix
   Rows            : OUT    NUMBER; -- Number of rows in matrix
   Columns         : OUT    NUMBER) -- Number of columns in matrix
```

If `Matrix_Number` equals zero, then the subroutine searches the list of matrices within the area specified by `Work_Area` for a matrix with the name specified by `Matrix_Name`. Otherwise, `Matrix_Number` is used as an index to obtain information about the specified matrix.

This subroutine requires an HMPU processor and runs in Background CL only.

### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 12.0 — Work area does not exist
- 17.0 — Matrix index is out of range
- 19.0 — Work area is not a matrix work area
- 21.0 — Matrix does not exist
- 27.0 — HMPU is required

### Example:

This code fragment searches for information about the matrix named "big\_mtrx" by its name.

```
LOCAL status,           -- Math Library return status
&   area_num,          -- Work area number
&   matrix_num         -- Number of matrix
&   rows,              -- Number of rows in the matrix
&   columns,           : NUMBER -- Number of columns in the matrix
LOCAL matrix_name      : STRING -- Name of the work area
.
.
SET matrix_name = "big_mtrx"
CALL AMCL02$Get_Matrix_Info (status, area_num, matrix_num,
&                           matrix_name, rows, columns)
```

### E.4.17 AMCL02\$Create\_Matrix Subroutine

This subroutine is used to create a two-dimensional matrix in the work area previously reserved by the point that the CL is attached to.

```
SUBROUTINE AMCL02$Create_Matrix
  (Ret_Status      : OUT    NUMBER; -- Math Library return status
   Work_Area       : IN     NUMBER; -- Work area number
   Matrix_Number   : OUT    NUMBER; -- Number of matrix
   Matrix_Name     : IN     STRING; -- Eight-character name for matrix
   Rows            : IN     NUMBER; -- Number of rows in matrix
   Columns         : IN     NUMBER; -- Number of columns in matrix
   Initial_value   : IN     NUMBER) -- Value to initialize elements to
```

A matrix of the specified size and initial values with the given name is created if there is room for it. This matrix can be used by any of the AMCL02 calls which require a matrix as input by specifying the matrix's name. Once a matrix is created, it cannot be deleted except by release of the work area where it resides. A maximum of 50 matrices can be created for a work area.

This subroutine does not require an HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 11.0 — Not enough memory to create matrix
- 13.0 — Work area reserved by another point
- 15.0 — Work area index is out of range
- 16.0 — Matrix with that name already exists
- 19.0 — Work area is not a matrix work area

#### Example:

This code fragment creates a 2 x 2 matrix named "sml\_mtrx" with initial values of 0.0.

```
LOCAL status,                -- Math Library return status
&   area_num,                -- Work area number
&   matrix_num              : NUMBER  -- Number of columns in the matrix
LOCAL matrix_name           : STRING  -- Name of the work area
.
.
SET matrix_name = "sml_mtrx"
CALL AMCL02$Create_Matrix (status, area_num, matrix_num, matrix_name,
&                          2, 2, 0.0)
```

### E.4.18 AMCL02\$Put\_Element Subroutine

This subroutine is used to write a value to an element in a previously created work area matrix.

```
SUBROUTINE AMCL02$Put_Element
  (Ret_Status      : OUT    NUMBER; -- Math Library return status
   Work_Area       : IN     NUMBER; -- Work area number
   Matrix_Number    : IN     NUMBER; -- Number of matrix
   Matrix_Name      : IN     STRING; -- Name of matrix
   Row              : IN     NUMBER; -- Row to store value in
   Column           : IN     NUMBER; -- Column to store value in
   Value            : IN     NUMBER) -- Value for element
```

This call sets the specified matrix element to the specified value. For quick access the matrix number should be specified in addition to the matrix name. If the matrix number is equal to 0 (zero) the matrix name is used and a search is required.

This subroutine does not require an HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 13.0 — Work area reserved by another point
- 15.0 — Work area index is out of range
- 17.0 — Matrix index is out of range
- 19.0 — Work area is not a matrix work area
- 21.0 — Matrix does not exist
- 25.0 — Matrix name and number do not agree

#### Example:

This code fragment writes a value of 1.0 into element 1,1 of "big\_mtrx."

```
LOCAL status,                -- Math Library return status
&   area_num,                -- Work area number
&   matrix_num               : NUMBER  -- Number of columns in the matrix
LOCAL matrix_name           : STRING  -- Name of the work area
.
.
SET matrix_name = "big_mtrx"
CALL AMCL02$Put_Element (status, area_num, matrix_num, matrix_name,
&                        1, 1, 1.0)
```

### E.4.19 AMCL02\$Get\_Element Subroutine

This subroutine is used to retrieve an element value from a work area matrix.

```
SUBROUTINE AMCL02$Get_Element
  (Ret_Status      : OUT    NUMBER; -- Math Library return status
   Work_Area       : IN     NUMBER; -- Work area number
   Matrix_Number   : IN     NUMBER; -- Number of matrix
   Matrix_Name     : IN     STRING; -- Name of matrix
   Row             : IN     NUMBER; -- Row to get value from
   Column          : IN     NUMBER; -- Column to get value from
   Value           : OUT    NUMBER) -- Value of element
```

This call gets the value of the specified matrix element. For quick access, the matrix number should be specified in addition to the matrix name. If the matrix number is equal to 0 (zero), the matrix name is used and a search is required.

This subroutine does not require an HMPU processor and runs in Background CL only.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 3.0 — Input value is not finite
- 13.0 — Work area reserved by another point
- 15.0 — Work area index is out of range
- 17.0 — Matrix index is out of range
- 19.0 — Work area is not a matrix work area
- 21.0 — Matrix does not exist
- 25.0 — Matrix name and number do not agree

#### Example:

This code fragment retrieves the value of element 2,3 from "big\_mtrx."

```
LOCAL status,           -- Math Library return status
&   area_num,          -- Work area number
&   matrix_num,        -- Number of columns in the matrix
&   element_val        : NUMBER  -- Value to be retrieved
LOCAL matrix_name      : STRING  -- Name of the work area
.
.
SET matrix_name = "big_mtrx"
CALL AMCL02$Get_Element (status, area_num, matrix_num, matrix_name,
&                        2,3, element_val)
```

## E.4.20 AMCL02\$Add\_Matrices Subroutine

This subroutine adds two matrices.

```
SUBROUTINE AMCL02$Add_Matrices
  (Ret_Status      : OUT  NUMBER;      -- Math Library return status
   Sum_Matrix      : OUT  cl_type;     -- Result of the addition
   Rows            : IN   NUMBER;      -- Number of rows to be used
   Columns         : IN   NUMBER;      -- Number of columns to be used
   Add_Matrix1     : IN   cl_type;     -- First addend matrix
   Add_Matrix2     : IN   cl_type)     -- Second addend matrix
```

Matrix arguments can be either direct (the name of a local matrix) or indirect (the name of a string variable that points to a matrix within a reserved work area).

This routine performs a standard matrix addition by adding each element in the first matrix to the element in the same position in the second matrix, and placing the result in a sum matrix. Only the sum matrix is modified.

Both the addend matrices and the sum matrix are matrix arguments whose sizes are given by the Rows and Columns arguments (See heading E.1.6, Array and Array Size Arguments). If a bad or infinite value is encountered, the calculation stops and an error is returned.

A matrix may be used for more than one argument. For example, to set  $A = A + B$ , call this routine with B as the second addend matrix, and A as both the first addend matrix and the sum matrix.

This subroutine requires an HMPU processor.

### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 3.0 — Bad input error—an input value was bad or out of range.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match
- 27.0 — HMPU is required



**Example:**

The following code fragment adds two matrices.

```

LOCAL status      : NUMBER          -- Math Library return status
LOCAL add1,       -- Matrix A
&    add2,       -- Matrix B
&    sum         : NUMBER ARRAY (1..2, 1..2) -- Results matrix (A + B)
.
.
CALL AMCL02$Add_Matrices (status, sum, 2, 2, add1, add2)

```

Given these initial values for "add1":  $\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 3.0 \end{bmatrix}$  and these values for "add2":  $\begin{bmatrix} 2.0 & 1.0 \\ 0.0 & 2.0 \end{bmatrix}$

Final value for "sum" will be:  $\begin{bmatrix} 3.0 & 4.0 \\ 2.0 & 5.0 \end{bmatrix}$

## E.4.21 AMCL02\$Subtract\_Matrices Subroutine

This subroutine subtracts two matrices.

```
SUBROUTINE AMCL02$Subtract_Matrices
  (Ret_Status      : OUT  NUMBER;      -- Math Library return status
   Diff_Matrix     : OUT  cl_type;     -- Result of the subtraction
   Rows            : IN   NUMBER;      -- Number of rows to be used
   Columns         : IN   NUMBER;      -- Number of columns to be used
   Sub_Matrix1     : IN   cl_type;     -- Matrix to be subtracted from
   Sub_Matrix2     : IN   cl_type)     -- Matrix to be subtracted
```

Matrix arguments can be either direct (the name of a local matrix) or indirect (the name of a string variable that points to a matrix within a reserved work area).

This routine performs standard matrix subtraction by subtracting each element in the second matrix from the element in the same position in the first matrix and placing the result in a sum matrix. Only the sum matrix is modified.

Both the subtraction matrices and the result matrix are matrix arguments whose sizes are given by the Rows and Columns arguments (see heading E.1.6, Array and Array Size Arguments). If a bad or infinite value is encountered, the calculation stops and an error is returned.

A matrix may be used for more than one argument. For example, to set  $A = A - B$ , call this routine with B as the second source matrix and A as both the first source matrix and the difference matrix.

This subroutine requires an HMPU processor.

### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 3.0 — Bad input error—an input value was bad or out of range.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match
- 27.0 — HMPU is required

**Example:**

The following code fragment subtracts two matrices.

```

LOCAL status      : NUMBER      -- Math Library return status
LOCAL sub1,       -- Matrix A
&    sub2,       -- Matrix B
&    difference   : NUMBER ARRAY (1..2, 1..2) -- Results matrix (A - B)
.
.
CALL AMCL02$Subtract_Matrices (status, difference, 2, 2, sub1, sub2)

```

Given these initial values for "sub1":  $\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 3.0 \end{bmatrix}$  and these values for "sub2":  $\begin{bmatrix} 2.0 & 1.0 \\ 0.0 & 2.0 \end{bmatrix}$

Final value for "sum" will be:  $\begin{bmatrix} -1.0 & 2.0 \\ 2.0 & 1.0 \end{bmatrix}$

## E.4.22 AMCL02\$Multiply\_Matrices Subroutine

This subroutine multiplies two matrices.

```
SUBROUTINE AMCL02$Multiply_Matrices
  (Ret_Status      : OUT  NUMBER;  -- Math Library return status
   Product_Matrix  : OUT  cl_type;  -- Result of the multiplication
   Product_Rows    : IN   NUMBER;  -- Rows in product and factor1 matrices
   Product_Cols    : IN   NUMBER;  -- Columns in prod and factor2 matrices
   Mid_Rows_Cols   : IN   NUMBER;  -- Cols in factor1 and rows in factor2
   Factor_Matrix1  : IN   cl_type;  -- First matrix to be multiplied
   Factor_Matrix2  : IN   cl_type) -- Second matrix to be multiplied
```

Matrix arguments can be either direct (the name of a local matrix) or indirect (the name of a string variable that points to a matrix within a reserved work area).

This routine performs matrix multiplication by forming the dot product of each row in the first matrix with each column in the second matrix. The row and column each dot product is placed in is determined by the row and column used to compute it. Thus the dot product of row N of the first matrix and column M of the the second matrix is placed at location (N,M) in the product matrix. The dot product is formed by multiplying each element in the row by the corresponding element in the column, and taking the sum of all these products. Only the product matrix is modified by this routine.

Both the factor matrices and the product matrix are matrix arguments. The size of the first factor is given by the Product\_Rows\_Cols and Mid\_Rows\_Cols arguments. Size of the second factor is given by Mid\_Rows\_Cols and Product\_Cols. Size of the product is given by Product\_Rows and Product\_Cols (see heading E.1.6, Array and Array Size Arguments). If a bad or infinite value is encountered, the calculation stops and an error is returned.

Unlike addition and subtraction, this routine does not allow the same matrix to be used as both a factor and the product. If this is attempted, the calling CL block stops and an error is returned. However, the same matrix can be used for both factors.

Note that matrix multiplication is not commutative. Interchanging the first and second matrices will nearly always give different results and will often result in invalid dimensions. The inner (dot) product of two vectors can be found with this routine, but the result will be returned in a scalar (non-array) argument.

This subroutine requires an HMPU processor.

**Definition of Parameters:**

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 3.0 — Bad input error—an input value was bad or out of range.
- 5.0 — Duplicate argument error—the same matrix was used as both product and factor.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match
- 27.0 — HMPU is required

**Example:**

The following code fragment multiplies two matrices.

```

LOCAL status      : NUMBER      -- Math Library return status
LOCAL factor1,    -- Matrix A
& factor2,        -- Matrix B
& product        : NUMBER ARRAY (1..2, 1..2) -- Results matrix (A * B)
.
.
CALL AMCL02$Multiply_Matrices
& (status, product, 2, 2, 2, factor1, factor2)

```

Given these values for "factor1":  $\begin{bmatrix} 1.0 & 3.0 \\ 2.0 & 3.0 \end{bmatrix}$  & these values for "factor2":  $\begin{bmatrix} 2.0 & 1.0 \\ 0.0 & 2.0 \end{bmatrix}$

Final value for "product" will be:  $\begin{bmatrix} 2.0 & 7.0 \\ 4.0 & 8.0 \end{bmatrix}$

### E.4.23 AMCL02\$Multiply\_Scalar Subroutine

This subroutine multiplies a matrix by a scalar.

```
SUBROUTINE AMCL02$Multiply_Scalar
  (Ret_Status      : OUT  NUMBER;  -- Math Library return status
   Product_Matrix  : OUT  cl_type;  -- Result of the multiplication
   Product_Rows    : IN   NUMBER;  -- Rows in product and factor matrices
   Product_Cols    : IN   NUMBER;  -- Columns in prod and factor matrices
   Scalar          : IN   NUMBER;  -- Scalar to multiply factor by
   Factor          : IN   cl_type)  -- Input matrix
```

Matrix arguments can be either direct (the name of a local matrix) or indirect (the name of a string variable that points to a matrix within a reserved work area).

The product matrix is calculated by multiplying each element of the factor matrix by the scalar value. The product and the factor may be the same matrix.

Both the factor and the product must be matrix arguments with sizes given by the rows and columns arguments (see subsection E.1.6, Array and Array Size Arguments). If a bad or infinite value is encountered, the calculation stops and an error is returned.

This subroutine does not require an HMPU processor. It requires software release 401 or later.

**Definition of Parameters:**

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 3.0 — Bad Reference number error—internal error, call Honeywell TAC.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match rows/columns arguments

**Example:**

The following code fragment multiplies a matrix by a scalar:

```

LOCAL status      : NUMBER      -- Math Library return status
LOCAL scalar,     -- Scalar value
&   factor,      -- Matrix A
&   product      : NUMBER ARRAY (1..2, 1..2) -- Result matrix (scalar*A)
.
.
CALL AMCL02$Multiply_Scalar
& (status, product, 2, 2, scalar, factor)

```

Given these values for "factor":  $\begin{bmatrix} 1.5 & 2.0 \\ 3.0 & 4.0 \end{bmatrix}$  & the value 2 for "scalar"

Final value for "product" will be:  $\begin{bmatrix} 3.0 & 4.0 \\ 6.0 & 8.0 \end{bmatrix}$

### E.4.24 AMCL02\$Transpose\_Matrix Subroutine

This subroutine gives the transpose of a matrix.

```
SUBROUTINE AMCL02$Transpose_Matrix
  (Ret_Status      : OUT  NUMBER;  -- Math Library return status
   Transposed_Matrix : OUT  cl_type; -- Result of the transpose
   Rows            : IN   NUMBER;  -- Number of rows to be used
   Columns         : IN   NUMBER;  -- Number of columns to be used
   Source_Matrix    : IN   cl_type) -- Matrix to be transposed
```

Matrix arguments can be either direct (the name of a local matrix) or indirect (the name of a string variable that points to a matrix within a reserved work area). However, if either the transposed matrix or the source matrix is in a work area, they both must be in work areas.

This routine performs standard matrix transpose, moving element I,J to element J,I for all elements in the matrix. Only the Transposed\_Matrix is modified by this routine.

Both the input matrix and the output matrix are matrix arguments whose sizes are given by the Rows and Columns arguments (see heading E.1.6, Array and Array Size Arguments). The rows and columns are specified for the input matrix; the output matrix has the rows and columns reversed. If a bad or infinite value is encountered, the calculation is aborted and an error is returned.

The same matrix may be used for both arguments. For example, to set  $A = A'$ , call this routine with A as both the source matrix and the transposed matrix.

This subroutine requires an HMPU processor.

#### Definition of Parameters:

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to, or generated by, the calculation.
- 3.0 — Bad input error—an input value was bad or out of range.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match rows/columns arguments
- 27.0 — HMPU is required
- 28.0 — Cannot mix work area matrices with CL arrays



**Example:**

The following code fragment transposes contents of Matrix 1 to Matrix 3.

```
LOCAL status      : NUMBER      -- Math Library return status
LOCAL matrix1,    -- Matrix 1
&      matrix3    : STRING      -- Matrix 3
.
.
CALL AMCL02$Transpose_Matrix (status, matrix3, 2, 2, matrix1)
```

### E.4.25 AMCL02\$Matrix\_Inversion Subroutine

This subroutine calculates the inverse of a matrix (A), or solves a system of linear equations ( $AX = B$ ) or  $XA = B$ .

```
SUBROUTINE AMCL02$Matrix_Inversion
  (Ret_Status      : OUT  NUMBER;      -- Math Library return status
   CL_Error_Status : OUT  CLERRSTS;    -- CL error enumeration
   Matrix_X        : OUT  cl_type;     -- Unknown (X) matrix
   Order           : IN   NUMBER;     -- Number of rows and columns in
                                     -- the coefficient (A) matrix
   Vectors         : IN   NUMBER;     -- Number of columns in the
                                     -- unknown (X) and right-hand (B)
                                     -- matrices
   Matrix_A        : IN   cl_type;     -- Coefficient (A) matrix
   Matrix_B        : IN   cl_type;     -- Right-hand side (B) matrix
   Option          : IN   NUMBER)     -- The calculation type to perform
```

Matrix arguments can be either direct (the name of a local matrix) or indirect (the name of a string variable that points to a matrix within a reserved work area). However, you cannot mix work area matrices with CL arrays. Matrix arguments must be either **all** work area matrices or **no** work area matrices.

The inverse of Matrix\_A or the solution of ( $AX = B$ ) or ( $XA = B$ ) is calculated by LU composition (with pivoting). The decomposed matrices L and U are not returned. This routine does not deal with non-square coefficient (A) matrices which correspond to overdetermined or underdetermined systems, and are not invertible.

The action of this routine depends on the value of the option argument. If Option equals 0 (zero), the inverse of Matrix\_A is calculated and is returned in Matrix\_X. The Matrix\_B argument is ignored. If Option equals 1, left-hand division is performed to find the solution of the matrix equation ( $AX = B$ ). If Option equals 2, right-hand division is performed to solve ( $XA = B$ ).

Matrix\_A is the coefficient matrix and it must be square. Its size in both rows and columns is given by the Order argument. Matrix\_B is the right-hand side of the equation ( $AX = B$ ) and Matrix\_X is the unknown.

When the inversion option (0) is selected, the equation becomes ( $AA^{-1} = I$ ).  $A^{-1}$  is the inverse of A, which is the same size as A, and is returned in Matrix\_X. I is the identity matrix which is constant. The inversion option ignores the Matrix\_B argument. Because Matrix\_X must be a square matrix the same size as Matrix\_A, the Vectors and Order arguments must be the same (or zero).

Because Matrix\_A is square, Matrix\_B and Matrix\_X must have the same form and dimensions. The equation ( $AX = B$ ) may in general have one or more right-hand side vectors. If the equation has only one right-hand side, Matrix\_B contains the right-hand side vector, and Matrix\_X returns the solution vector. In this case, the Vectors argument must be set to 1 or 0, and the Matrix\_B and Matrix\_X are both vector arguments of size Order.

If the equation has more than one right-hand side vector, each right-hand side is a column in Matrix\_B, and the corresponding solution vector is a column in Matrix\_X. In this case, the Vectors argument is the number of right-hand sides, and equivalently the number of solutions, and Matrix\_X and Matrix\_B are both matrix arguments of Order rows and Vectors columns.

As with all matrix arguments, passing zero for the Vectors argument uses the physical dimensions of the Matrix\_X and Matrix\_B arguments for the local dimensions. In addition, passing zero for the Vectors argument also allows this routine to use either vectors or matrices for Matrix\_X and Matrix\_B.

Duplicate arrays can be used for any of the array arguments. For example, using the same array for Matrix\_X and Matrix\_B causes the solution matrix X to be returned in place of the right-hand side matrix B. Note that using the same array for both a source array and a solution array destroys the data in the source array. Neither of the source arrays is modified by this routine unless a source array is also used for the solution.

For foreground CL computations, the largest coefficient matrix allowed is 15 x 15 elements. Also, no more than 15 right-hand sides and corresponding unknown vectors are allowed, so the logical X and B matrices can be at most 15 x 15 elements in foreground CL computations. For background CL computations, the routine will accept local CL coefficient matrices up to order 25. Work area matrices can be of any realistic size. The following formula gives the approximate CPU time in milliseconds necessary to invert an N x N matrix:

$$\text{Milliseconds to invert} = .044N^{**3} + .199N^{**2} + 3.75N - .391$$

Internal calculations are done in double precision in order to minimize roundoff error.

This subroutine requires an HMPU processor.

**Definition of Parameters:**

**Ret\_Status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 2.0 — Bad reference number error—internal error, call Honeywell TAC
- 3.0 — Bad input error—an input value was bad or out of range.
- 4.0 — CL error—use the CL\_Error\_Status return value to determine the reason.
- 6.0 — Foreground size error—a large matrix calculation was attempted in foreground
- 7.0 — Internal error, call Honeywell TAC
- 8.0 — Singular matrix error—a singular or near-singular coefficient matrix was input, and no solution exists.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match rows/columns arguments
- 27.0 — HMPU required
- 28.0 — Cannot mix work area matrices with CL arrays

**CL\_Error\_Status** — This parameter returns the CL Error return status enumeration as follows:

- |          |                         |
|----------|-------------------------|
| NOERROR  | — No Error              |
| LIMVIOL  | — Limit Violation       |
| RIGHTS   | — Rights Error          |
| COMMERR  | — Communication Error   |
| BADVALST | — Bad Value Status      |
| COMABORT | — Communication Abort   |
| ABORT    | — Abort                 |
| ARITHERR | — Arithmetic Error      |
| ARRAYLIM | — Array Limit Violation |
| RANGE    | — Range Error           |
| PROGERR  | — Program Error         |
| KEYLEVEL | — Keylevel Error        |
| CNFERR   | — Configuration Error   |

**Option** — Value of the Option argument specifies what this routine is to do.

- 0 — The inverse of Matrix\_A is calculated and is returned in Matrix\_X.  
The Matrix\_B argument is ignored.
- 1 — Left-hand division is performed to find the solution of the matrix equation ( $AX = B$ ).
- 2 — Right-hand division is performed to solve ( $XA = B$ ).

**Examples:**

The following code fragment finds the inverse of a matrix and returns it on top of the original matrix:

```

LOCAL status      : NUMBER          -- Math Library return status
LOCAL cl_status   : CLERRSTS        -- CL error status
LOCAL a           : NUMBER ARRAY (1..2,1..2) -- Matrix A and A-1
.
.
CALL AMCL02$Matrix_Inversion (status, cl_error_status, a, 2, 2, a, a, 0)

```

Given these initial values for "A":     [1.0 2.0]  
   [3.0 4.0]

Final value for "A" will be:           [-2.0 1.0]  
   [ 1.5 -0.5]

The following code fragment solves a linear system ( $AX = B$ ), using default matrix sizes:

```

LOCAL status      : NUMBER          -- Math Library return status
LOCAL cl_status   : CLERRSTS        -- CL error status
LOCAL a,          -- Matrix A (coefficient matrix)
&    b,           -- Matrix B (right-hand side)
&    x            : NUMBER ARRAY (1..2,1..2) -- Matrix X (unknown)
.
.
CALL AMCL02$Matrix_Inversion (status, cl_error_status, x, 0, 0, a, b, 1)

```

Given these initial values for "a":     [4.0 3.0]  
   [2.0 1.0]

And these initial values for "b":       [1.0 2.0]  
   [3.0 4.0]

Final value for "x" will be:           [ 4.0 5.0]  
   [-5.0 -6.0]

## E.4.26 AMCL02\$SVD

This subroutine calculates the singular value decomposition (SVD) of a matrix.

```
SUBROUTINE AMCL02$SVD
  (Ret_Status      : OUT  NUMBER;          -- Math Library return status
   Data_Access_status : OUT  NUMBER;          -- Data Access status
   Matrix_U         : IN   cl_type         ; -- Matrix U
   Matrix_S         : IN   cl_type         ; -- Matrix S
   Matrix_V         : IN   cl_type         ; -- Matrix V
   Matrix_A         : IN   cl_type         ; -- Matrix A
   Rows             : IN   NUMBER; -- Number of rows in matrix A      (m)
   Columns          : IN   NUMBER) -- Number of columns in matrix A (n)
```

To use the SVD subroutine, the work area must be initialized using the subroutine AMCL02\$Init\_Matrix\_Work\_Area2.

The SVD is defined as:  $A = U \cdot S \cdot V^T$  where the superscript “T” indicates the matrix transposed, and:

- A is an m x n matrix;
- U is an orthogonal m x m matrix;
- S is an m x n diagonal matrix of non-negative elements arranged in descending order from the top left position (1,1);
- V is an orthogonal n x n matrix.

The AMCL02\$SVD subroutine accepts A in the matrix defined by argument Matrix\_A and returns  $U^T$ , S, and V in the matrices defined by arguments Matrix\_U, Matrix\_S, and Matrix\_V, respectively. Note that U is returned transposed and V is returned normally. If  $m > n$ , then Matrix\_S must define a column matrix of size n x 1 or larger. If  $m < n$ , then Matrix\_S must define a column matrix of size m x 1 or larger. Matrix\_A must define a matrix of size m x n or larger. Matrix\_U must define a matrix of size m x m or larger. Matrix\_V must define a matrix of size n x n or larger.

All matrices in the argument list must be defined indirectly by the name of a string variable that stores the name of a matrix within a reserved workarea.

Passing a zero as the rows or columns sets the rows or columns arguments to that of the physical size of Matrix\_A.

Matrices U, V, and S must be stored in different matrices; however, Matrix\_A can also be given as an output parameter (U, V, or S) if the dimensions are sufficient. For example, V may be returned in A if  $m = n$ , or U may be returned in A if  $n = m$ .

Internal calculations are done in double precision in order to minimize roundoff error.

This subroutine requires an HMPU processor and software release 401 or later.

**Ret\_status** — This parameter is used to return the Math Library error status. The returned value corresponds to the following status conditions:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 2.0 — Bad reference number error—internal error, call Honeywell TAC
- 3.0 — Bad input error—an input value was bad or out of range.
- 4.0 — CL error—use the CL\_Error\_Status return value to determine the reason.
- 6.0 — Foreground size error—a large matrix calculation was attempted in foreground
- 7.0 — Internal error, call Honeywell TAC
- 8.0 — Singular matrix error—a singular or near-singular coefficient matrix was input and no solution exists.
- 12.0 — Work area does not exist
- 17.0 — Work area matrix index out of range
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 24.0 — Sizes of matrices do not match rows/columns arguments
- 27.0 — HMPU required
- 28.0 — Cannot mix work area matrices with CL arrays

## E.5 MATRIX FUNCTIONS EXAMPLE

The following contains simple tests for the matrix functions in AMCL02. These tests are intended to provide an illustration of these calls and show the order of use, but do not test for error conditions.

```

PACKAGE
%INCLUDE_SET AMCL02          -- Include the Math Library extension set
BLOCK matrix_example (GENERIC; AT BACKGRND)

LOCAL status,
&   area_num,
&   columns,
&   invert,
&   free_space,
&   matrices,
&   matrix_1_num,
&   matrix_2_num,
&   matrix_3_num,
&   rows,
&   size,
&   work_area_type   : NUMBER
LOCAL cl_status      : CLERRSTS
LOCAL matrix_1_name,
&   matrix_2_name,
&   matrix_3_name,
&   reserving_point,
&   workarea         : STRING

-- Create a work area called "area_1"

SET workarea = "area_1"
CALL AMCL02$Create_Work_Area (status, workarea, 300000)

-- Reserve work area "area_1"

CALL AMCL02$Reserve_Work_Area (status, area_num, workarea)

-- Initialize "area_1" to support matrices

CALL AMCL02$Init_Matrix_Work_Area (status, workarea, 100)

-- Create a 2 x 2 matrix named "matrix_1" in work area "area_1"

SET matrix_1_name = "matrix_1"
CALL AMCL02$Create_Matrix
& (status, area_num, matrix_1_num, matrix_1_name, 2, 2, 0.0)

-- Establish initial values for "matrix_1"

CALL AMCL02$Put_Element
& (status, area_num, matrix_1_num, matrix_1_name, 1, 1, 1.0)
CALL AMCL02$Put_Element
& (status, area_num, matrix_1_num, matrix_1_name, 1, 2, 2.0)
CALL AMCL02$Put_Element
& (status, area_num, matrix_1_num, matrix_1_name, 2, 1, 3.0)
CALL AMCL02$Put_Element
& (status, area_num, matrix_1_num, matrix_1_name, 2, 2, 4.0)

```



```

-- Create a 2 x 2 matrix named "matrix_2" in work area "area_1"

SET matrix_2_name = "matrix_2"
CALL AMCL02$Create_Matrix
& (status, area_num, matrix_2_num, matrix_2_name, 2, 2, 0.0)

-- Establish initial values for "matrix_2"

CALL AMCL02$Put_Element
& (status, area_num, matrix_2_num, matrix_2_name, 1, 1, 1.0)
CALL AMCL02$Put_Element
& (status, area_num, matrix_2_num, matrix_2_name, 1, 2, 2.0)
CALL AMCL02$Put_Element
& (status, area_num, matrix_2_num, matrix_2_name, 2, 1, 3.0)
CALL AMCL02$Put_Element
& (status, area_num, matrix_2_num, matrix_2_name, 2, 2, 4.0)

-- Create a 2 x 2 matrix named "matrix_3" in work area "area_1"

SET matrix_3_name = "matrix_3"
CALL AMCL02$Create_Matrix
& (status, area_num, matrix_3_num, matrix_3_name, 2, 2, 0.0)

-- Transpose matrix_1 and place results in matrix_3

CALL AMCL02$Transpose_Matrix (status, matrix_3_name, 2, 2,
& matrix_1_name)

-- Add matrix_1 and matrix_2 and place results in matrix_3

CALL AMCL02$Add_Matrices
& (status, matrix_3_name, 2, 2, matrix_1_name, matrix_2_name)

-- Subtract matrix_2 from matrix_3 and place results in matrix_1

CALL AMCL02$Subtract_Matrices
& (status, matrix_1_name, 2, 2, matrix_3_name, matrix_2_name)

-- Multiply matrix_1 times matrix_2 and place results in matrix_3

CALL AMCL02$Multiply_Matrices
& (status, matrix_3_name, 2, 2, 2, matrix_1_name, matrix_2_name)

-- Verify area_1 work area and matrices

CALL AMCL02$Get_Work_Area_Info
& (status, area_num, workarea, size, reserving_point, free_space,
& work_area_type)

CALL AMCL02$Get_Matrix_Work_Area_Info
& (status, area_num, workarea, invert, matrices)

CALL AMCL02$Get_Matrix_Info
& (status, area_num, matrix_1_num, matrix_1_name, rows, columns)

```

```
-- AX = B

CALL AMCL02$Matrix_Inversion
& (status, cl_status, matrix_3_name, 2, 2, matrix_1_name,
&  matrix_2_name, 1)

-- XA = B

CALL AMCL02$Matrix_Inversion
& (status, cl_status, matrix_3_name, 2, 2, matrix_1_name,
&  matrix_2_name, 2)

-- Find inverse of matrix_1

CALL AMCL02$Matrix_Inversion
& (status, cl_status, matrix_2_name, 2, 2, matrix_1_name,
&  matrix_3_name, 0)

-- Release work area "area_1"

CALL AMCL02$Release_Work_Area (status, workarea)

END matrix_example

END PACKAGE
```

## E.6 RETURN STATUS VALUES

The following is a list of return status values for the calls in this package:

- 0.0 — Successful
- 1.0 — Bad value error—a bad value was input to or generated by the calculation.
- 2.0 — Bad reference number error—internal error, call Honeywell TAC
- 3.0 — Bad input error—an input value was bad or out of range.
- 4.0 — CL error—use the CL\_Error\_Status return value to determine the reason.
- 5.0 — A work area with the specified name already exists, or the same matrix was used as both product and factor.
- 6.0 — Foreground size error—a large matrix calculation was attempted in foreground
- 7.0 — Internal error, call Honeywell TAC
- 8.0 — Singular matrix error—a singular or near-singular coefficient matrix was input, and no solution exists.
- 9.0 — Bad standard deviation array—fewer than two useable values were found in the source array.
- 10.0 — No more work areas can be created
- 11.0 — Not enough memory in work area for specified operation
- 12.0 — Work area does not exist
- 13.0 — Work area reserved by another point
- 14.0 — Work area matrices can be used only in background
- 15.0 — Work area index is out of range
- 16.0 — Matrix with that name already exists
- 17.0 — Matrix index is out of range
- 18.0 — Work area already is initialized
- 19.0 — Work area is not a matrix work area
- 20.0 — Invalid data type in reference
- 21.0 — Matrix does not exist
- 23.0 — Matrix must be in CDS
- 24.0 — Sizes of matrices do not match, or do not match rows/columns arguments
- 25.0 — Matrix name and number do not agree
- 27.0 — HMPU is required
- 28.0 — Cannot mix work area matrices with CL arrays
- 29.0 — Point already has reserved another work area



## CDS MOVE AND MULTIPLE MOVE PARAMETER EXTENSION

### Appendix F

*This appendix explains an optionally available set of background CL subroutines that provide the ability to copy a Custom Data Segment and the ability to move a number of parameters with a single subroutine call.*

#### F.1 OVERVIEW

The extension covered in this appendix enables you to:

- Copy the contents of a Custom Data Segment (CDS) on a source AM point to another CDS with the same structure on a destination AM point. The source and destination CDSs can be on the same or different AMs (on the local LCN), and neither is required to be on the bound data point (although either may be).
- Define lists of parameters (in either internal or ASCII format), and move the parameters with a single subroutine call. The ASCII input format gives parameter indirection capability.

These two functions are packaged in one optional set named **AMCL01**. Each function will be completely described in the following subsections.

#### NOTE

Several of the subroutines in this set have arguments designated to be of the stand-in data type "cl\_type." The actual types of these arguments, as specified in your calls, can be any valid CL/AM data type within the rules given below. For these arguments only, normal CL compiler data type checking is inhibited and any data type mismatches you may accidentally create are not detected until the CL block runs (runtime).

An IN argument of "cl\_type" accepts a local variable, parameter, or expression of any valid CL/AM data type.

An OUT argument of "cl\_type" accepts a local variable or parameter of any valid CL/AM data type.

## F.2 MEMORY REQUIREMENTS, PACKAGING, AND AM CONFIGURATION

This subsection provides the user with the necessary software and hardware/memory requirements for the AMCL01 set. It also provides guidelines on the configuration of the AM for the successful installation of the set.

### F.2.1 Memory Requirements

This set requires approximately 94 K words of memory, which includes the software and data buffers.

The Multiple Move list data structures are allocated in user memory, which is in redundant memory. This allows the user to get memory dynamically instead of having a predefined fixed-size memory section for the list data structures. By default, an additional 20 K words of memory are reserved from user memory at AM startup and a List Summary is created for 64 list instances (the reserved memory allows the user to configure approximately 550 list items in total). To avoid memory fragmentation problems, the maximum size for a list is 8 K words (approximately 228 list items).

If the above default values are too restrictive for the user's application, it is possible to change the size of the memory reserved for the list data structures (refer to `AMCL01$Get_List_Mem_Stats` and `AMCL01$Change_List_Mem_Size`). If the user configures more than 64 lists, the List Summary size is automatically increased by the `AMCL01$Allocate_List` subroutine. Extra memory allocated to the List Summary is never released when entries in the summary are cleared. In order to expand the List Summary, enough reserved memory must be available to make a copy of the current list with the additional size added to it.

### F.2.2 Packaging

The CDS Move subroutine and the Multiple Move subroutines are packaged together in the AMCL01 set. This set consists of a linked object file (contains the subroutines in the set) and a Set Definition File (contains a description of the calling sequences of the subroutines for the CL compiler). These files must reside in directories `&CUS` and `&CLX` on the History Module, and the AM that is to contain the set must be configured through the Network Configurator to load the set.

This set requires the additional conversion set CONV. This conversion set contains Data Access conversion routines used to convert point and parameter identifiers from external form to internal form.

### F.2.3 Distributed Files

The following files are necessary to install AMCL01 on an Application Module:

AMCL01.LO—loadable object file  
AMCL01.SF—set definition file

### F.2.4 Installation

Using the Command Processor function of the Engineering or Universal Personality, determine if the directories &CUS and &CLX exist on the HM:

```
LSV NET      (lists volumes on the network)
```

If either of the directories does not exist, use the Create Directory command to create a directory under a volume on the History Module:

```
CD NET>vol>&CUS      (vol is an existing volume on the HM)
CD NET>vol>&CLX
```

Mount the AMCL01 cartridge in a removable media drive. Using the Command Processor function of the Engineering or Universal Personality, execute the following commands to copy the tool set files to the History Module (x = the removable media drive number):

```
CP $Fx>&CUS>AMCL01.LO  NET>&CUS>AMCL01.LO
CP $Fx>&CLX>AMCL01.SF  NET>&CLX>AMCL01.SF
```

Note: If not done so already, the file CONV.LO should also be copied to the HM, using the following command:

```
CP $Fx>&CUS>CONV.LO   NET>&CUS>CONV.LO
```

## F.2.5 Application Module Configuration

From the `MODIFY VOLUME PATHS` display in the Engineer or Universal Personality, set the NCF Backup Path to a removable media pathname; that is, `$Fx>&ASY>` (x = drive number). Insert a backup `&ASY` floppy or cartridge in the removable media drive.

From the Main Menu of an Engineer Personality, select the `LCN NODES` target.

A display will appear with a target for each node possible on the LCN. Verify that the Configurator is in the `ON-LINE` mode in the upper right corner of the display. Select the target for the node number of the Application Module that is to be configured for AMCL01 (the Page Forward key may be required to display the target for the desired AM).

A display will appear that allows the configuration of this AM node on the network. If this AM node has not already been configured onto the network, do so at this time.

Select the `MODIFY NODE` target.

Page forward to page 2.

The next display that will appear will allow the user to configure the tool set to the selected AM. Page forward to page three to enter any external load module names to the AM. In the entry ports at the top of the display, enter the tool set load module name, AMCL01, and associated personality type (AMO) for the AM personality. Also enter CONV for the conversion set. Select the `NO` target for the `USE DEFAULT PERSONALITY TYPE?` question.

Press the `ENTER` key.

Press the `CTL` and the one (1) key together (`F1`) to check the new NCF.

Press the `CTL` and the two (2) key together (`F2`) to install the new NCF.

Return to the Engineer's Main Menu display. A message will appear at the bottom of the display indicating the successful completion of the NCF installation.

Load the Application Module. The load will include the AMCL01 set.



## F.3 CDS\_MOVE

### F.3.1 Overview

The CL subroutine “Custom Data Segment Move” (CDS\_Move) provides a method by which a CL/AM program can copy the contents of a CDS (on a source AM point) to another CDS with the same structure (on a destination AM point). The source and destination CDSs can be on the same or different AMs, and neither is required to be on the bound data point (although either can be). The source and destination CDS must be on the local LCN. A source or destination accessed through the Network Gateway is illegal.

The CDS\_Move subroutine can be used only by CL blocks linked to the background insertion point. This routine is declared in a set definition file which is read by the CL compiler when an “include\_set” directive is encountered in a CL source file within the scope of a CL Block. The directive would appear in a source file as follows:

```
%INCLUDE_SET AMCL01
```

The CDS\_Move subroutine can be invoked by calling it in a CL block or subroutine, by the CALL statement, as follows:

```
CALL AMCL01$CDS_MOVE(status, det_stat, dest_entity, dest_pkg,
                      src_entity, src_pkg)
```

For details about the CDS\_Move call, see subsection F.3.5, “AMCL01\$CDS\_Move.”

### F.3.2 CL Aborts

If invalid arguments are passed to the CDS\_Move subroutine, the CL Block may abort with a CL Error Status (CLERRSTS) of type “Program Error” (PROGERR). This can occur because the variable type “cl\_type” is used for the source and destination entity IDs, thereby deferring some type checking until runtime. Therefore, passing a number as the source or destination entity ID, for instance, would pass the compiler type checking, but would result in a runtime error.

### F.3.3 Performance

For a CDS\_Move that only accesses points within its same node (AM), the move takes place in memory avoiding any need for internode communication. If either the source or destination points are off-node, CDS\_Move will make a minimum of two internode requests and a maximum of eight. Note, however, that the Move\_Parameter subroutine running in a background CL program can make up to two internode requests per call to move only one parameter value.

### F.3.4 CDS\_Move Restrictions

The following restrictions apply to the CDS\_Move subroutine:

- (a) Any point involved in a CDS\_Move call (the bound data point, the source point, and the destination point) must reside on an AM that has the AMCL01 set loaded in it.
- (b) Only one CDS\_Move call that contains off-node requests will be executed in an AM at a time. The user does not have to be concerned with this, however. The move requests are queued and executed on a first-in, first-out basis.
- (c) The point execution status of the source and destination entities must be ACTIVE. This provides protection from point builds and CL links, on either of these points, during the CDS\_Move call.
- (d) The structure of the CDS on the source and destination entities must be identical. This means that both CDSs must have the same number of parameters, and that the data type of each parameter must match in both. However, the names and values of the parameters may differ between the two CDSs.
  - An array with bounds of (10..20) is not considered the same as two consecutive arrays, one with bounds of (10..15) and the next with bounds of (16..20).
  - Array indices must be identical for parameters of type “array.” For example, an array with bounds of (10..20) is not considered the same as an array with bounds of (30..40).
  - A parameter of type “enumeration” must be of the same set number in each CDS to be considered the same. For example, a parameter of type “mode” is only equal to another parameter of type “mode.”
  - Likewise, a parameter of type “point\_id” must be of the same parameter list name in each CDS to be considered the same.
- (e) The CDS\_Move entity arguments must be points on the local LCN. If you specify a point accessed through the Network Gateway as an entity argument, a return status error indicating an invalid point id results .

### F.3.5 AMCL01\$CDS\_Move

The CDS\_Move subroutine can only be called from a CL block linked to a background insertion point. It requires a source and destination entity and a source and destination package name. It will return a program return status and a detailed return status when necessary. The point to which the CL block is linked does not have to be either the source or destination point. In addition, the bound data point, source point, and destination point can all be on different nodes.

This subroutine can overlay an existing CDS only when the structure of the source CDS and destination CDS match. This means that both Custom Data Segments must have the same number of parameters and the data type of each parameter must match. Values in the parameters and names of the parameters can differ between the two CDSs. If the CDS structures do not match, this subroutine will not overlay values in the destination CDS, but instead will return a status value of “mismatch” (11).

```
SUBROUTINE AMCL01$CDS_Move
  (Return_Status      : OUT NUMBER; -- CDS_Move return status
   Det_Status         : OUT NUMBER; -- Detailed return status
   Dest_Entity        : IN CL_TYPE;  -- Destination point name
   Dest_Package       : IN STRING;    -- Destination package name
   Source_Entity       : IN CL_TYPE;  -- Source point name
   Source_Package     : IN STRING)    -- Source package name
```

#### Description of Parameters:

Dest_Entity (INPUT)	This is the point that the CDS is to be moved to. The argument type is CL_TYPE, which allows the point id to be either of type “string” or type “entity.”
Dest_Package (INPUT)	This is the name of the CDS on the destination point that is to be copied to.
Source_Entity (INPUT)	This is the point that the CDS is to be moved from. The argument type is CL_TYPE, which allows the point id to be either of type “string” or type “entity.”
Source_Package (INPUT)	This is the name of the CDS on the source point that the values are to be copied from.

**Return\_Status (OUTPUT)** This parameter returns an error code to the user. The following error codes can be returned:

- 0 — Successful
- 1 — Source package name has invalid number of characters
- 2 — Destination package name has invalid number of characters
- 3 — Source point id invalid; see Det\_Status for Data Access error code\*
- 4 — Destination point id invalid; see Det\_Status for Data Access error code\*
- 5 — Can't access source point; see Det\_Status for Data Access error code\*
- 6 — Can't access destination point; see Det\_Status for Data Access error code\*
- 7 — Source point not active
- 8 — Destination point not active
- 9 — Source CDS does not exist, or is missing, or the CDS name contains invalid characters
- 10 — Destination CDS does not exist, or is missing, or the CDS name contains invalid characters
- 11 — Source and destination CDS do not match
- 12 — Timeout while communicating between nodes
- 13 — Internal error, general; see Det\_Status
- 14 — Internal error, produce; see Det\_Status for the particular produce error
- 15 — Internal error, review; see Det\_Status for the particular review error

\*Data Access error codes are documented in the *Messages Directory* manual.

**Det\_Status (OUTPUT)** This status further qualifies certain of the Return\_Status values.

- 0 — Successful
- 1 — Change priority error
- 2 — Release semaphore error
- 3 — Get semaphore error
- 4 — Bad database
- 5 — Invalid message received
- 6 — Message received is not a CDS buffer
- 7 — CDS relink error
- 8 — Received uncertain data access error
- 9 — No CDS found
- 10 — Message received for a point not in this node

**Example:**

```

PACKAGE
--
CUSTOM
--
    PARAMETER SRC_PNT    : $REG_CTL -- entity type
    PARAMETER DEST_PNT   : STRING    -- STRING TYPE
--
END CUSTOM
--
BLOCK Move_CDS(....)
--
%INCLUDE_SET AMCL01
--
LOCAL src_pack, dest_pack : STRING
LOCAL status, det_stat    : NUMBER
--
SET src_pack = "CDS1"
SET dest_pack = "CDS2"
-- SET dest_pnt = "FS\A100"           -- Network Gateway id illegal
SET dest_pnt = "A100_LongTag"       -- 16 character entity id is
                                     -- legal on long tag systems
--
CALL AMCL01$CDS_MOVE(status, det_stat, DEST_PNT, dest_pack,
&    SRC_PNT, src_pack)
--
END Move_CDS(....)
--
END PACKAGE

```

## F.4 MULTIPLE\_MOVE\_PARAMETER

### F.4.1 Introduction

The `Multiple_Move_Parameter` function is provided to give the user the capability of moving a variable number of values in a single Data Access call. In addition to the `Multiple_Move_Parameter` function itself, the subroutines provide tools for defining lists of parameters that can be moved with one call.

The important features of this set can be summarized as follows:

- Capability of moving a variable number of values in a single call in a background CL/AM block.
- Allows moves from `point.parameters` to `point.parameters`.
- Allows moves from local CL variables to `point.parameters`, and vice versa.
- Allows moves from/to `point.parameters` through the Network Gateway.
- Ability to move entire arrays and array segments.
- Ability to move string arrays.
- Ability to specify `point.parameters` in ASCII format (which results in parameter indirection capability).
- Ability to set lists “permanent.”
- Ability to inactivate list items of a list.
- Ability to define on-line, the maximum size of memory for list structures.

### F.4.2 Overview

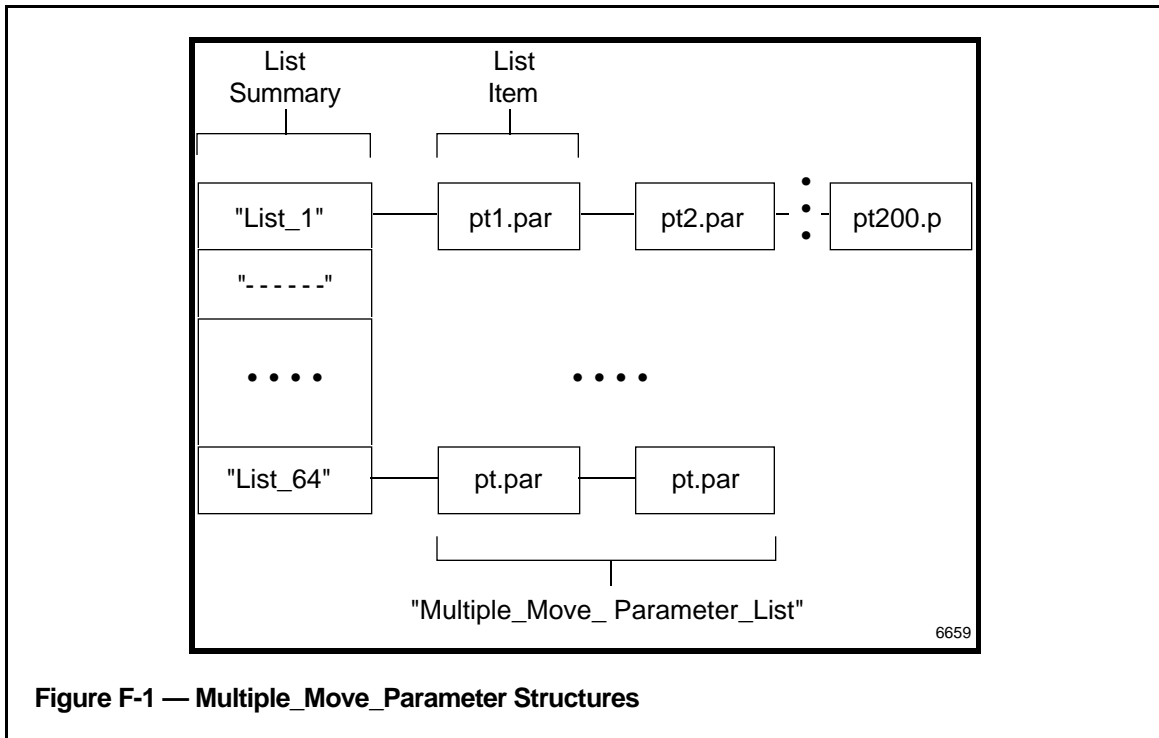
The `Multiple_Move_Parameter` function supports the transfer, in background CL, of relatively large collections of local variables and `point.parameters`. The size of a collection is variable; the user describes each item involved in the transfer. Such a collection of parameter descriptions is called a `Multiple_Move_Parameter_List`.

To support `Multiple_Move_Parameter_Lists`, a dynamic data structure is used because memory is allocated only when it is needed. These lists are created and maintained by Pascal custom software programs in the Application Module. These programs are part of the AMCL01 custom set and are called by the CL/AM application program.

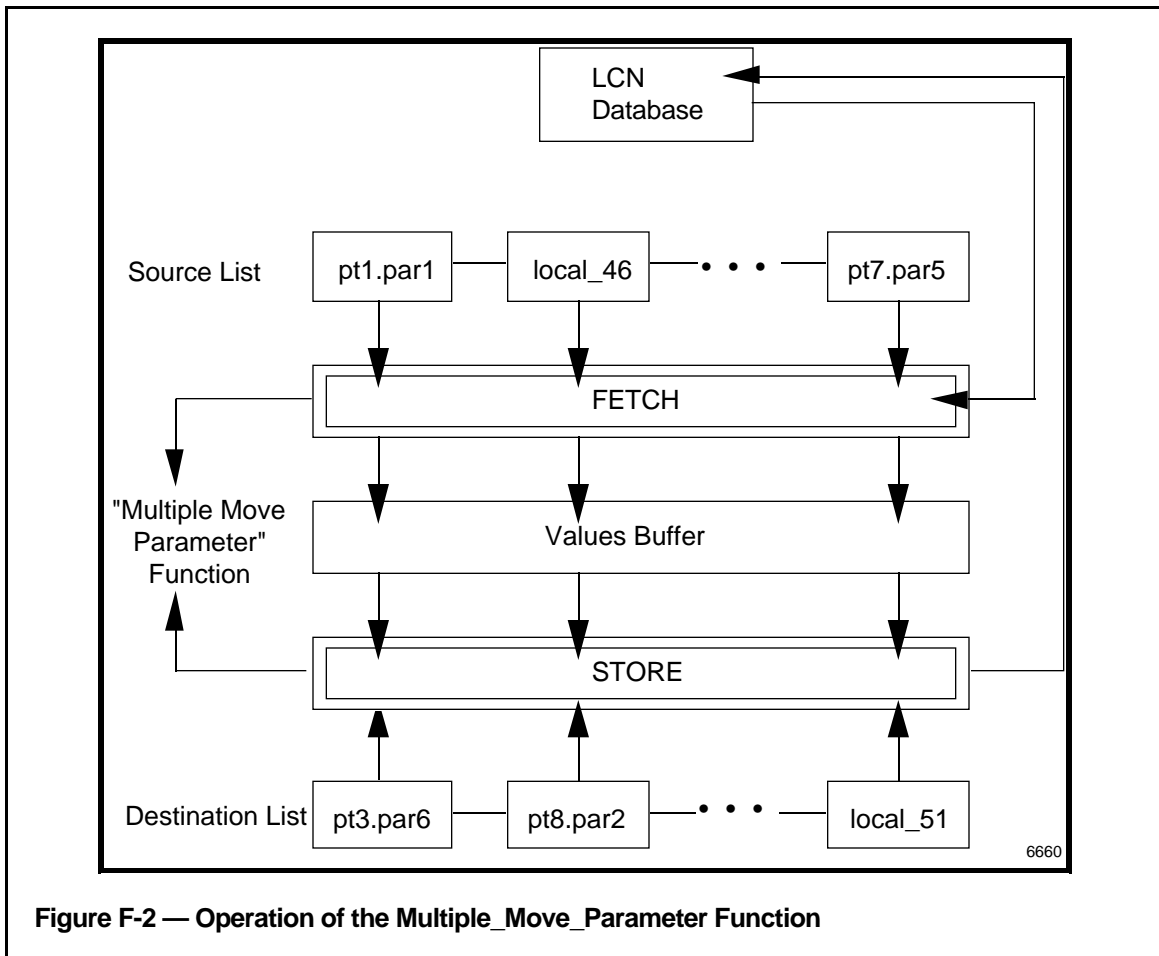
A `Multiple_Move_Parameter_List` is made of list items. A list item contains the definition of a local variable or `point.parameter` that will be involved during the `Multiple_Move_Parameter` function execution. Separate lists are used to define where the data is fetched from (source list), and where it is stored to (destination list). A list can be used both as a source list and a destination list. The lists are accessed through a static data structure called the List Summary.

The user defines an identifier (an 8-character-length string) for each list. This list id is a unique identifier that is used to refer to the list. The list id is in addition to a list number that identifies the list by its position in the List Summary (refer to subsection F.4.8, “List Access”).

Conceptually, the Multiple\_Move\_Parameter structures can be viewed as shown in Figure F-1.



The `Multiple_Move_Parameter` function is given a source list and a destination list and it fetches the value of each parameter that is defined in the source list and stores it in the corresponding parameter of the destination list. See Figure F-2.



**Figure F-2 — Operation of the `Multiple_Move_Parameter` Function**



### F.4.3 Move Execution Order

The order in which the `Multiple_Move_Parameter` does its moves is as follows:

- First are moves from local variables to local variables.
- Next are fetches from local variables and `point.parameters` (data are saved in a temporary buffer).
- Next are stores of `point.parameters` (from the temporary buffer) to local variables.
- Finally are stores from local variables or `point.parameters` (from the temporary buffer) to `point.parameters`.

Note the following subtleties:

1. If the same local variable is defined in the source and in the destination list, the value stored will be the value after any local variable stores to the local variable, but before any `point.parameter` stores to the local variable. Refer to examples 1 and 2 below.
2. If the same `point.parameter` is defined in the source and in the destination list, the value stored will be the value of the `point.parameter` prior to the execution of the store to it. Refer to Example 3 below.

#### Example 1:

Assume the values of the variables before the move are: `a = 1`, `pt.par1 = 2`, `pt.par2 = 3`.

```
pt.par1 --> a           (a is a local variable)
a       --> pt.par2
```

After the move, the values are: `a = 2`, `pt.par1 = 2`, and `pt.par2 = 1`.

#### Example 2:

Assume the values of the variables before the move are: `a = 1`, `b = 2`, `c = 3`, and `pt.par1 = 4`.

```
a       --> b           (a, b, and c are local variables)
b       --> pt.par1
c       --> b
```

After the move, the values are: `a = 1`, `b = 3`, `c = 3`, and `pt.par1 = 3`.

#### Example 3:

Assume the values of the variables before the move are: `a = 1`, `b = 2`, `c = 3`, `pt.par1 = 4`, and `pt.par2 = 5`.

```
b       --> pt.par1      (a, b, and c are local variables)
pt.par1 --> pt.par2
pt.par1 --> c
a       --> b
```

After the move, the values are: `a = 1`, `b = 1`, `c = 4`, `pt.par1 = 1`, and `pt.par2 = 4`.

### F.4.4 Database Consistency

Stores to a particular unit occur without interruption. However, moves that contain stores to multiple units can be interrupted between unit stores, even if the units are on the same physical node. In the case of redundant AMs, switchover from primary to secondary will not interrupt stores to a particular unit. Either all of the stores or none of the stores to a unit will be done, depending on the timing of the switchover.

### F.4.5 Dynamic Indirection

Indirect references may be used to define point.parameters in a Multiple\_Move\_Parameter\_List. However, these types of references get resolved **only at list build time**.

This means that if we want to add “PT.ENT(10).PV” into the list, only the **value** of PT.ENT(10) is stored in the list (that is, only “TIC101” rather than PT.ENT(10)), along with the “PV” parameter identifier. If the ENT(10) value is changed (that is, becomes “TIC102”) between the time the list was built and the time the Multiple\_Move\_Parameter is executed, the new ENT(10) value (“TIC102”) is **not** used. If the user wants dynamic indirection capabilities at “move” time, he needs to rebuild his lists each time before executing them.

Note: As long as the ENT(10) parameter is on-node, and as long as it is not defined in an ASCII form, rebuilding the list is not very time-consuming.

### F.4.6 List Permanence

By default, lists are “not permanent.” This means that they only exist as long as the creator CL block is running. The list is automatically deleted when the creator CL block completes or aborts. This automatic deletion guarantees that memory is released when it is no longer required.

However, a list can be set “permanent.” In this case, it is not deleted when the CL block completes or aborts. This feature is particularly helpful when:

- Programs periodically move the same list of parameters; rebuilding the lists at each cycle is not required, so no time is wasted.
- The lists are shared between different CL blocks on the same point.

Note: a permanent list is deleted only when the point using the list is deleted or when the list is explicitly deallocated by the application program.

### F.4.7 List Ownership

A `Multiple_Move_Parameter_List` belongs to the AM point that created it. This means that only the owner point of the list may modify, use, or delete the list.

### F.4.8 List Access

Two types of list identifiers are defined: a real identifier (`r_list_id`) and a string identifier (`s_list_id`). When a list is allocated, an `s_list_id` is passed to the subroutine and an `r_list_id` is returned. These identifiers are used for the two types of access to the `Multiple_Move_Parameter_Lists`, direct (fast) and logical (slow).

- Direct access: For this access, `r_list_id` is not equal to zero and `s_list_id` matches the list referenced by `r_list_id`. This access is similar to the way ASCII files are accessed. (When a file is opened, it returns a number to the calling program. This number (file id) allows the user to reference the opened file.)

The same principle can be applied to lists. At list creation time, a number (`r_list_id`) is returned to the user. When the user wants to modify or execute the list he references it by this number, which is the entry number in the List Summary. The access is then direct and fast.

- Logical access: For this access, `r_list_id` is zero. The user references a list by its name (`s_list_id`). As a result, the custom software searches (in the List Summary) for the requested list name. This kind of access is sequential and requires string comparisons. Therefore, it is much slower than direct access.

If `r_list_id` is not equal to zero and `s_list_id` does not match the list referenced by `r_list_id`, an error is returned to the calling application. Also, the `r_list_id` of the list referenced by `s_list_id` is returned (as long as `s_list_id` corresponds to an existing list).

Logical access is provided so that a list can be permanent. In such a case, the memory used by the list is not deallocated when the application CL program completes. It is only deallocated when the CL block explicitly requests the deallocation. Therefore, it must be guaranteed that even if the real list id (`r_list_id`) is forgotten by the application there will be a way to access (and deallocate) the list with the list name.

### F.4.9 Redundancy

`Multiple_Move_Parameter_Lists` are stored in redundant memory in the AM, so that in case of AM switchover, the lists are not lost.

Moves to a particular unit may be lost in case of AM switchover during the move. However, when switchover occurs, either no move will have been performed for stores to a particular unit, or all the stores to a unit will have been performed.

### F.4.10 Error Handling

When an error occurs during the execution of a function, it causes the function to “undo” what has been done so far, in order to restore the context as it was prior to the invocation of the function (for example, allocated memory must be released). If an error occurs during this “undoing” mechanism, the “undo” procedure will continue to try to “undo” as much as possible. In this case, only the first error detected will be reported to the user.

Two types of errors can be returned by these subroutines:

Normal errors: These errors are due to one of the two following causes:

1. Inputs to the functions are bad (for example, reference to a nonexistent list).
2. The current context does not allow execution of the function (for example, not enough memory to create a new list).

These errors are returned to the user in the form of an error code (a real number), sometimes with a Data Access return code (CLERRSTS enumeration).

Major errors: These are the errors that are never expected to happen. They are always caused by bugs or unexpected situations. In both cases, the user should report them to TAC.

### F.4.11 Parameter Access Request Limits and Restrictions

For the Multiple Move functions, the following limits/restrictions apply:

- Maximum fetch/store items in a list—at least one fetch/store item is required for each list item.
- For segment arrays, a fetch/store item is used for each element of the segment.
- For string arrays, a fetch/store item is used for each element of the array.
- A maximum of 1000 fetch/store items are allowed for a single list. If this limit is exceeded, the Multiple\_Move\_Parameter will not be executed.

### F.4.12 Calling the Subroutines

The Multiple Move subroutines can be used only by CL blocks linked to the background insertion point. These routines are declared in a set definition file, which is read by the CL compiler when an INCLUDE\_SET directive is encountered in the CL source file within the scope of a CL block. The directive would appear as follows in a source file:

```
%INCLUDE_SET AMCL01
```

These subroutines may be invoked by calling them in a CL block by the CALL statement as defined in subsequent subsections.

### F.4.13 Multiple Move Subroutines

A list of the subroutines that comprise the Multiple Move group follows. Each will be described in detail in a subsequent subsection.

<u>Subroutine Name</u>	<u>Function</u>
AMCL01\$Allocate_List	Allocate an entry in the List Summary, allocate memory for the list, and initialize the list with empty items.
AMCL01\$Get_List_ID	Return the ASCII identifier for a particular list.
AMCL01\$Modify_Entry	Add or modify a list item.
AMCL01\$Modify_Complex_Entry	Same as Modify_Entry, but adds the ability to define array segments and define point.parameters in an ASCII format.
AMCL01\$Multiple_Move_Parameter	Execute the move of items in a source list to items in a destination list.
AMCL01\$Set_Permanent_List	Set a list “permanent” or “not permanent.”
AMCL01\$Deallocate_List	Delete a list.
AMCL01\$Get_List_Mem_Stats	Obtain the maximum amount of memory available for list configuration and the currently used memory.
AMCL01\$Change_List_Mem_Size	Change the maximum size of user memory dedicated to list storage.
AMCL01\$Set_List_Item_Active	Activate or inactivate a particular list item.
AMCL01\$Get_Exec_Time_Stats	Obtain statistics on the execution time of a Multiple Move function.

### F.4.14 Error Message Summary

This subsection gives a summary of all the error messages that may be returned to the user by one or more of the Multiple Move subroutines. Detailed descriptions of these errors will be found in the descriptions of each of the individual subroutine calls in the subsections that follow.

<u>Code</u>	<u>Message</u>
<b>0</b>	<b>No error</b>
<b>1</b>	<b>-- Reserved --</b>
<b>2</b>	<b>Bad number of items</b> Returned by: AMCL01\$Allocate_List
<b>3</b>	<b>Not enough working memory</b> Returned by: AMCL01\$Multiple_Move_Parameter
<b>4</b>	<b>User memory limit exceeded</b> Returned by: AMCL01\$Allocate_List AMCL01\$Change_List_Mem_Size
<b>5</b>	<b>Duplicate list identifier</b> Returned by: AMCL01\$Allocate_List
<b>6</b>	<b>List does not exist</b> Returned by: AMCL01\$Modify_Entry AMCL01\$Modify_Complex_Entry AMCL01\$Multiple_Move_Parameter AMCL01\$Set_Permanent_List AMCL01\$Deallocate_List AMCL01\$Set_List_Item_Active AMCL01\$Get_Exec_Time_Stats
<b>7</b>	<b>-- Reserved --</b>
<b>8</b>	<b>Bad list item number</b> Returned by: AMCL01\$Modify_Entry AMCL01\$Modify_Complex_Entry AMCL01\$Set_List_Item_Active
<b>9</b>	<b>Not the list owner</b> Returned by: AMCL01\$Modify_Entry AMCL01\$Modify_Complex_Entry AMCL01\$Multiple_Move_Parameter AMCL01\$Set_Permanent_List AMCL01\$Deallocate_List AMCL01\$Set_List_Item_Active
<b>10</b>	<b>Nothing to move</b> Returned by: AMCL01\$Multiple_Move_Parameter

- 11    **Type error (array size)**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 12    **Type error (incompatible types)**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 13    **-- Reserved --**
- 14    **-- Reserved --**
- 15    **Bad internal list identifier**  
Returned by:    AMCL01\$Modify\_Entry  
                  AMCL01\$Modify\_Complex\_Entry  
                  AMCL01\$Multiple\_Move\_Parameter  
                  AMCL01\$Set\_Permanent\_List  
                  AMCL01\$Deallocate\_List  
                  AMCL01\$Set\_List\_Item\_Active  
                  AMCL01\$Get\_Exec\_Time\_Stats
- 16    **Bad DA status type**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 17    **Bad DA status size**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 18    **Bad list number**  
Returned by:    AMCL01\$Get\_List\_Id
- 19    **Local variables in list**  
Returned by:    AMCL01\$Set\_Permanent\_List
- 20    **Bad booster factor**  
Returned by:    AMCL01\$Change\_List\_Mem\_Size
- 21    **-- Reserved --**
- 22    **DA status is not a local variable**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 23    **List is permanent**  
Returned by:    AMCL01\$Modify\_Entry  
                  AMCL01\$Modify\_Complex\_Entry
- 24    **Bad start index**  
Returned by:    AMCL01\$Modify\_Complex\_Entry
- 25    **Bad number of elements**  
Returned by:    AMCL01\$Modify\_Complex\_Entry
- 26    **-- Reserved --**
- 27    **Parameter identifier type is not STRING**  
Returned by:    AMCL01\$Modify\_Complex\_Entry

- 28    **Invalid ASCII string**  
Returned by:    AMCL01\$Modify\_Complex\_Entry
- 29    **Tagname conversion error**  
Returned by:    AMCL01\$Modify\_Complex\_Entry
- 30    **Parameter conversion error**  
Returned by:    AMCL01\$Modify\_Complex\_Error
- 31    **Parameter not on this point**  
Returned by:    AMCL01\$Modify\_Complex\_Entry
- 32    **No temporary memory available**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 33    **Too many fetch/store items**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 34    **Completed with errors**  
Returned by:    AMCL01\$Multiple\_Move\_Parameter
- 35    **- - Reserved - -**
- 36    **Bad reference**  
Returned by:    AMCL01\$Modify\_Entry  
                  AMCL01\$Modify\_Complex\_Entry

**NOTE**

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.



### F.4.15 AMCL01\$Allocate\_List

This function selects the first free entry in the “List Summary” and allocates this entry for the new Move\_Multiple\_Parameter\_List. It also allocates memory for the list and initializes the list with empty list items.

```
SUBROUTINE AMCL01$Allocate_List
  (Return_Status : OUT NUMBER;
   R_List_Id     : OUT NUMBER;
   Nb_of_Items   : IN  NUMBER;
   S_List_Id     : IN  STRING)
```

#### Description of Parameters:

- |                        |   |
|------------------------|---|
| S_List_Id (INPUT)      | This string defines the list identifier. Only the first eight characters are used as the unique identifier by which the user can later refer to the list (see Multiple_Move_Parameter and Modify_Entry). The remaining part of this string can be used as a list description. |
| Nb_of_Items (INPUT)    | This is the number of items that the user defines in the list. It is the number that is used to allocate memory for the list.   |
| R_List_Id (OUTPUT)     | This is the position in the List Summary where the entry for the list has been allocated. This can be used for direct access to the list.   |
| Return_Status (OUTPUT) | This parameter returns an error code to the user. The following codes can be returned:  |
- 0 — OK—no errors
  - 2 — Bad number of items—the value passed to the function is a “bad value,” less than or equal to zero, or greater than 228.
  - 4 — User memory limit exceeded—there is no more memory to allocate the list or to expand the List Summary. Call the AMCL01\$Change\_List\_Mem\_Size function to reserve more memory for list data structures.
  - 5 — Duplicate list id—the list name is already allocated. Choose another name.

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

The following CL program tries to allocate ten lists, each of which contains twelve items. The names of the lists are contained in the custom parameter List\_Id.

```
ALL_LIST:  Loop for i in 1..10

CALL AMCL01$Allocate_List (Return_Status (i), Real_List_Id (i), 12,
&                          List_Id (i) )

IF Return_Status (i) <> 0 THEN Send: "Ret_Stat(", i, ")=",
&                          Return_Status (i)

REPEAT ALL_LIST
```

### F.4.16 AMCL01\$Get\_List\_Id

This function returns the identifier of a particular list, specified by a number. This function is helpful when the contents of the List Summary needs to be displayed.

Note: The function returns a “-” as the identifier when the corresponding slot in the List Summary is free.

```
SUBROUTINE AMCL01$Get_List_Id
  (Return_Status : OUT NUMBER;
   S_List_Id     : OUT STRING;
   R_List_Id     : IN  NUMBER)
```

#### Description of Parameters:

R_List_Id (INPUT)	This number is the position of the list in the List Summary.
S_List_Id (OUTPUT)	This is the identifier (eight character length string) plus the list description (remaining characters) of the list, which is defined in the List Summary at the position defined by R_List_Id.
Return_Status (OUTPUT)	This parameter returns an error code to the user. The following codes can be returned:

- 0 — OK—no errors
- 18 — Bad list number—this can happen when the value passed to the function is not in the range 1..Max items in list.

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

#### Example:

This program fetches the list identifiers of all the entries in the List Summary. If the “List\_id” is a custom data segment parameter, the List Summary contents could be displayed (for example, in a custom display) on the Universal Station.

```
----- Allocate up to 64 lists -----
LIST_SUM:  Loop for i in 1..64
CALL AMCL01$Get_List_Id (error_code (i), List_Id (i), i)
REPEAT LIST_SUM
```

### F.4.17 AMCL01\$Modify\_Entry

This function allows the user to define a new “List\_item” in a particular list or to modify an existing “List\_item..”

```
SUBROUTINE AMCL01$MODIFY_Entry
  (Return_Status : OUT NUMBER;
   Item_Nb       : IN  NUMBER;
   R_List_Id     : IN  OUT NUMBER;
   S_List_Id     : IN  STRING;
   Param_Id      : IN  CL_TYPE)
```

#### Description of Parameters:

S_List_Id (INPUT)	This string is the list identifier. It identifies the name of the list in which an item will be created or modified. Only the first eight characters are taken into account for the unique list identifier.
R_List_Id (IN/OUT)	This is the list identifier used for “direct” list access. If logical access is preferred, this real must be zero, and the function will return the real list identifier in this parameter.
Item_Nb (INPUT)	This real identifies the List_item that will be modified in the list, which is identified by “S_List_Id” or “R_List_Id.” If an item has already been defined at this location, it is overwritten with this call.
Param_Id (INPUT)	This is the parameter involved in the Multiple Move Parameter. According to the type of list (source or destination), it defines where the data to be moved comes from, or where it goes. The parameter type can be any of the types recognized by CL.

**Return\_Status (OUTPUT)** This parameter returns an error code to the user. The following error codes can be returned:

- 0 — OK—no errors
- 6 — List does not exist—the function didn't find the list in the List Summary.
- 8 — Bad list item number—the value of "Item\_Nb" passed to the function was a "bad value" or was not in the range 1..max number of items in the list.
- 9 — Not the list owner—the calling CL block is not linked to the point that owns the list.
- 15 — Bad Internal list identifier—the internal list identifier (real) does not match the external list identifier (string).
- 23 — List is permanent—an attempt was made to define a local variable into a permanent list.
- 36 — Bad reference—the function could not resolve the indirection for Param\_Id. (This may happen when an entity id is "NULL.")

**NOTE**

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

The purpose of this program is to define two lists (a source list and a destination list) to support the following data move:

```

FS\pt1_longtag.DATA  -->  pt2.DATA1
loc_data             -->  pt2.DATA2

```

The source list contains two items: (1) the parameter “data” of (FS) Network Gateway external pt1\_longtag, and (2) the local array of 300 numbers, loc\_data. The destination list contains the parameters “data1” and “data2” of the external point, pt2, as its first and second items, respectively.

```

EXTERNAL FS\pt1_longtag  -- source pt (where the data is fetched from)
EXTERNAL pt2             -- destin pt (where the data is stored to)

Local r_list_id : NUMBER          -- the list id
Local loc_data  : NUMBER ARRAY (1..300) -- array to move to pt2
.....

----- allocate source and destination lists -----

CALL AMCL01$Allocate_List ( err_code,      -- return status
&                          L1_nb,         -- output variable
&                          2,             -- nb of items
&                          "My_Src" )     -- list id of source list
IF err_code <> 0 THEN ...

CALL AMCL01$Allocate_List ( err_code,      -- return status
&                          L2_nb,         -- output variable
&                          2,             -- nb of items
&                          "My_Dest" )    -- list id of destination list
IF err_code <> 0 THEN ...

----- fill in the lists -----

CALL AMCL01$Modify_Entry (err_code, 1, L1_nb,
&                          "My_src", FS\pt1_longtag.data)
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry (err_code, 2, L1_nb, "My_src", loc_data)
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry (err_code, 1, L2_nb, "My_dest", pt2.data1)
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry (err_code, 2, L2_nb, "My_dest", pt2.data2)
IF err_code <> 0 THEN ...

----- Multiple Move -----

CALL AMCL01.....
.....

```

### F.4.18 AMCL01\$Modify\_Complex\_Entry

This function is a superset of the AMCL01\$Modify\_Entry function. In addition to the standard functions, it also allows the user to define:

- array segments
- point and parameter identifiers in “ASCII” format.

This gives parameter indirection capability.

```
SUBROUTINE AMCL01$MODIFY_Complex_Entry
  (Return_Status : OUT NUMBER;
   Item_Nb       : IN  NUMBER;
   R_List_Id     : IN  OUT NUMBER;
   S_List_Id     : IN  STRING;
   Param_Id      : IN  CL_TYPE;
   Nb_Element    : IN  NUMBER;
   Start_Index   : IN  NUMBER;
   ASCII_Format  : IN  LOGICAL)
```

#### Description of Parameters:

Item_Nb (INPUT)	This real identifies the list item that will be modified in the list, which is identified by “S_List_Id” or “R_List_Id.” If an item has already been defined at this location, it is overwritten with this call.
R_List_Id (IN/OUT)	This is the list identifier used for “direct” list access. If logical access is preferred, this real must be zero and the function will return the real list identifier in this parameter.
S_List_Id (INPUT)	This string is the list identifier. It identifies the name of the list in which an item will be created or modified. Only the first eight characters are taken into account for the unique list identifier.
Param_Id (INPUT)	This is the parameter involved in the Multiple_Move_Parameter. If the list is a “source list,” it defines where the data comes from; if the list is a “destination list,” it defines where the data goes. Depending on “ASCII_Format” value, “Param_Id” defines this parameter directly or indirectly. When “ASCII_Format” is ON, Param_Id is a variable of type string containing the external name of the point and parameter involved in the move.
Nb_Element (INPUT)	This parameter is only meaningful when Param_Id refers to an array. In this (array segment) case, the Nb_Element specifies the number of elements of the array to move. If this parameter, plus the start index, is greater than the array upper bound, the value of this parameter is clamped to the array upper bound plus one, minus Start_Index. If this number is less than or equal to zero, the item will not be considered as an array segment, but as a whole array.

**Start\_Index (INPUT)** This parameter goes together with the “Nb\_Element” parameter and is only used when an array segment is considered (“Nb\_Element” > 0). For array segment moves, it gives the starting index of the segment within the array. If this number is outside the lower or upper bounds of the array, an error is returned to the user.

**Note:** For CL/AM local variables, Start\_Index must be equal to the index of the first element of the segment, plus one, minus the array lower bound, as illustrated by the following example:

```
Local abc : Array (10..20)      -- array lower bound = 10
```

If the desired Start\_Index = 12 for a Multiple\_Move\_Parameter call, the normalized Start\_Index for the CL/AM local variable abc is:  $12 + 1 - 10 = 3$  (Note that 12 is the 3rd element of abc).

**ASCII\_Format (INPUT)** When this logical is “ON,” Param\_Id indirectly defines the parameter involved in the move (in the form of an ASCII string—see “Param\_Id” description above). Note that the conversion from an ASCII form to an internal point parameter id form is done during the execution of the Modify\_Complex\_Entry function. Therefore, the fact that the parameters have been defined in an ASCII format does not alter the performance of the Multiple\_Move\_Parameter.

When this logical is “OFF,” a point that is “NULL,” or that has been deleted, is accepted; that is, a successful Return\_Status (0) is returned to the user in the Modify\_Complex\_Entry call. However, an error is detected during runtime of the Multiple\_Move\_Parameter call and a DA\_Status of CNFERR is returned.



Return\_Status (OUTPUT) This parameter returns an error code to the user. The following error codes can be returned:

- 0 — OK—no errors
- 6 — List does not exist—the function didn't find the list in the List Summary.
- 8 — Bad list item number—the value of "Item\_Nb" passed to the function was a "bad value" or was not in the range 1..max number of items in the list.
- 9 — Not the list owner—the calling CL block is not linked to the point that owns the list.
- 15 — Bad Internal list identifier—the internal list identifier (real) does not match the external list identifier (string).
- 23 — List is permanent—an attempt was made to define a local variable into a permanent list.
- 24 — Bad start index—the value passed to the function is a "bad value" or not in the range array\_lower\_bound..array\_upper\_bound.
- 25 — Bad number of elements—the value passed to the function is a "bad value" or the value is greater than 32760.
- 27 — Parameter id type is not a string—the ASCII format option was chosen and Param\_Id was neither a parameter of type string nor a local variable of type string.
- 28 — Invalid string—the Param\_Id string passed to the function must have the following format:

My\_Point.My\_Parameter (My\_Index)

Where: My\_Point is a point name (1..8 chars).  
 My\_Parameter is a parameter identifier (1..8 chars).  
 My\_Index is optional and is used for array elements and array segments. If present, it must be an integer constant (for example, "123"), not a local variable.

- 29 — Point conversion error—the point was not found in the system.
- 30 — Parameter conversion error—the parameter was not found in the system.
- 31 — Parameter conversion error—the parameter does not exist on the specified point.
- 36 — Bad reference—the function could not resolve the indirection for Param\_Id. (This may happen when an entity id is "NULL.")

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

The purpose of this program is to define two lists (a source list and a destination list) to support the following data move (lists are accessed in “direct mode”):

```
pt1.DATA    --> pt2.DATA1
loc_data(2..9) --> pt2.DATA2(11..18)
```

```
.....
EXTERNAL PT2 -- destination pt (where the data is stored to)

Local loc_data : NUMBER ARRAY (1..300) -- array to move to pt2
Local pt1_data : STRING                -- holds the string "pt1.data"
                                       -- which identifies the parameter
                                       -- where the data comes from

----- allocate source and destination lists -----

CALL AMCL01$Allocate_List ( err_code,      -- return status
&                          L1_nb,         -- output variable
&                          2,             -- nb of items
&                          "My_Src" )     -- list id of source list
IF err_code <> 0 THEN ...

CALL AMCL01$Allocate_List ( err_code,      -- return status
&                          L2_nb,         -- output variable
&                          2,             -- nb of items
&                          "My_Dest" )    -- list id of destination list
IF err_code <> 0 THEN ...

----- fill in the lists -----

SET pt1_data = "pt1.data"
CALL AMCL01$Modify_Complex_Entry
&    ( err_code,      -- return code
&    1,L1_nb,"My_src", -- 1st item_nb, list_ids of source list
&    pt1_data,        -- param_id
&    0,0,             -- not an array segment
&    ON )             -- param defined as ASCII string
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Complex_Entry
&    ( err_code,      -- return code
&    2,L1_nb,"My_src", -- 2nd item_nb, list_ids of source list
&    loc_data,        -- param_id
&    8,2,             -- array segment 2..9
&    OFF )            -- not an ASCII string
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry
&    ( err_code,      -- return code
&    1,L2_nb,"My_dest", -- 1st item_nb, list_ids of dest list
&    pt2.data1 )      -- param_id
IF err_code <> 0 THEN ...
```

```

CALL AMCL01$Modify_Complex_Entry
&      (   err_code,           -- return code
&      2,L2_nb,"My_dest",     -- 2nd item_nb, list_ids of dest list
&      pt2.data2,             -- param_id
&      8,11,                  -- array segment 11..18
&      OFF )                  -- not an ASCII string
IF err_code <> 0 THEN ...
.....

```

After the calls illustrated in this example are executed,

(a) The source list contains the following items:

- (1) The “data” parameter of an entity, pt1
- (2) The array segment (2..9) out of a local array of 300 numbers, loc\_data (1..300),  
and

(b) The destination list contains the following items:

- (1) The parameter “data1” of an entity, pt2
- (2) The array segment (11..18) of a parameter “data2” belonging to pt2.

### F.4.19 AMCL01\$Multiple\_Move\_Parameter

This function performs or executes the `Multiple_Move_Parameter`. It fetches the values of the parameters defined in a source list and stores them in the parameters defined in a destination list.

The performance of this function depends on the type of move. For example, a move of type “pt.par” to “pt.par” requires two data accesses, whereas a move “local” to “pt.par” requires only one data access.

If the size (number of list items) is not the same in the source and destination lists, only the number of items in the smallest list will be moved. This situation will not result in the return of an error code to the user.

In many cases, the moves will be executed in the same order as the order specified in the list. See subsection F.4.3 for additional information on move execution order.

Type checking is performed before moving the data. Only the following moves are valid:

Integer	--->	Integer
Number	--->	Number
Integer	--->	Number
Logical	--->	Logical
Time	--->	Time
Entity Id	--->	Entity Id
String	--->	String*
Enumeration	--->	Enumeration**
Self-defined Enm	--->	Self-defined Enm**
Self-defined Enm	--->	Enumeration**
Enumeration	--->	Self-defined Enm**

\* A runtime error will be generated for strings larger than 78 characters.

\*\* There is no type checking on the enumeration set number. For example, it is possible to move an enumeration of type “ON/OFF” into an enumeration of type “AUTO/MANU.” However, a runtime error will be returned if the ordinal value of the fetched state is greater than the number of states of the destination variable.

The `Multiple_Move_Parameter` also checks that the number of elements match for array or array segment moves.

The maximum number of items that can be fetched/stored is 1000. For most list items, only one fetch/store item is created. However, one fetch/store item is created for each element of a string array or an array segment.

```
SUBROUTINE AMCL01$Multiple_Move_Parameter
  (Return_Status : OUT NUMBER;
   DA_Status    : IN  OUT CLERRSTS ARRAY (..);
   R_Src_List_Id : IN  OUT NUMBER;
   Source_List_Id : IN  STRING;
   R_Dest_List_Id : IN  OUT NUMBER;
   Destin_List_Id : IN  STRING )
```

**Description of Parameters:**

- R\_Src\_List\_Id (IN/OUT)** This is the list identifier for the source list used for “direct” list access. If logical access is preferred, this real must be zero, and the function will return the real list id in this parameter.
- Source\_List\_Id (INPUT)** This string is the list identifier for the “source list.” It identifies the name of the list whose items are to be fetched. Only the first eight characters are taken into account for the unique list identifier.
- R\_Dest\_List\_Id (IN/OUT)** This is the list identifier for the destination list used for “direct” list access. If logical access is preferred, this real must be zero, and the function will return the real list id in this parameter.
- Destin\_List\_Id (INPUT)** This string is the list identifier for the “destination list.” It identifies the name of the list whose items are to be stored. Only the first eight characters are taken into account for the unique list identifier.
- Return\_Status (OUTPUT)** This parameter returns an error code to the user. The following error codes can be returned:
- 0 — OK—no errors.
  - 3 — Not enough working memory—more than 20 K words of working memory was required to process this list. Reduce the size of the list.
  - 6 — List does not exist—the function didn’t find the list in the List Summary.
  - 9 — Not the list owner—the calling CL block is not linked to the point that owns either the source or the destination list.
  - 10 — Nothing to move—source or destination list is empty or all the list items in either list are inactive.
  - 11 — Type\_error (array size)—the number of elements in a source list array item does not match the number of elements in the corresponding destination array item.
  - 12 — Type\_error (incompatible types)—a source item element type is incompatible with the corresponding destination item element type.
  - 15 — Bad internal list identifier—the internal list identifier (real) does not match the external list identifier (string).
  - 16 — Bad DA Status type—DA\_Status type must be “CLERRSTS array (..).”
  - 17 — Bad DA Status size—DA\_Status array size must be greater than or equal to the maximum number of items to transfer.
  - 22 — DA\_Status is not a local variable—DA\_Status must be a local array of type CLERRSTS.
  - 32 — Not enough memory to do the move—there is currently not enough memory (out of the 20 K words) to process the move values or the number of fetch/store items is too large. This can happen when concurrent accesses to the Multiple\_Move\_Parameter function are done. If you suspect that the error was caused by concurrent accesses, try again.
  - 33 — Too many fetch/store items—this happens when array segments or string arrays are moved. For most list items, only one fetch/store item is created. However, one fetch/store item is created for each element of a string array or an array segment. The maximum number of fetch/store items is 1000.
  - 34 — Partial failure—the move has been executed but some of the list item moves have failed. Refer to DA\_Status for further details.

**NOTE**

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**DA\_Status (OUTPUT)**

The DA\_Status variable type in the CL application program **MUST** be a local variable array (1..number of elements to move) of CLERRSTS. Note that the array may be sized larger than the number of items to be moved. The DA\_Status goes together with the Return\_Status. It qualifies the error when the error source is a Data Access error.

**Example:**

The purpose of this program is to define two lists (a source list and a destination list) to support and execute the following data move:

```

pt1.DATA    --> pt2.DATA1
loc_data    --> pt2.DATA2

```

```

EXTERNAL PT1 -- source pt (where the data is fetched from)
EXTERNAL PT2 -- destination pt (where the data is stored to)

Local loc_data : NUMBER ARRAY (1..300) -- array to move to pt2
Local error    : NUMBER      -- return status of Multiple_Move_Parameter
Local da_error  : CLERRSTS ARRAY (1..2)
Local err_code  : NUMBER
.....

----- allocate source and destination lists -----

CALL AMCL01$Allocate_List ( err_code,      -- return status
&                          L1_nb,         -- internal list id
&                          2,             -- nb of items
&                          "My_Src" )     -- list id of source list
IF err_code <> 0 THEN ...

CALL AMCL01$Allocate_List ( err_code,      -- return status
&                          L2_nb,         -- internal list id
&                          2,             -- nb of items
&                          "My_Dest" )    -- list id of destination list
IF err_code <> 0 THEN ...

----- fill in the lists -----

CALL AMCL01$Modify_Entry (err_code, 1, L1_nb, "My_src", pt1.data)
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry (err_code, 2, L1_nb, "My_src", loc_data)
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry (err_code, 1, L2_nb, "My_dest", pt2.data1)
IF err_code <> 0 THEN ...

CALL AMCL01$Modify_Entry (err_code, 2, L2_nb, "My_dest", pt2.data2)
IF err_code <> 0 THEN ...

----- Multiple Move -----

CALL AMCL01$Multiple_Move_Parameter ( error, da_error, L1_nb,
&                                     "My_src", L2_nb, "My_dest")
IF error <> 0 THEN ...

```

### F.4.20 AMCL01\$Set\_Permanent\_List

This function gives the user the capability to set a list “permanent” or “not permanent.” A permanent list will not be deleted when the owner block completes or aborts. A list that contains CL local variables cannot be set permanent.

```
SUBROUTINE AMCL01$Set_Permanent_List
  (Return_Status : OUT NUMBER;
   R_List_Id     : IN  OUT NUMBER;
   S_List_Id     : IN  STRING;
   Permanent     : IN  LOGICAL)
```

#### Description of Parameters:

R_List_Id (IN/OUT)	This is the identifier used for “direct” list access. If logical access is preferred, this real must be zero, and the function will return the real list id in this parameter.
S_List_Id (INPUT)	This string is the list identifier. It identifies the name of the list to set “permanent” or “not permanent.” Only the first eight characters are taken into account for the unique list identifier.
Permanent (INPUT)	If this logical is “ON,” the list is set “permanent.” If it is “OFF,” the list is set “not permanent.”
Return_Status (OUTPUT)	This parameter returns an error code to the user. The following error codes can be returned:.
0	— OK—no errors
6	— List does not exist—the function didn’t find the list in the List Summary.
9	— Not the list owner—the calling CL block is not linked to the point that owns the list.
15	— Bad internal list identifier—the internal list identifier (real) does not match the external list identifier (string).
19	— List contains local variables—A list containing local variables may not be set permanent because CL local variables disappear when a block completes or aborts.

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.



**Example:**

This sample program illustrates how a CL program can be optimized when the same list is used in successive program executions. Creation of the lists is done only during the first execution.

```

Local Return_Status, Dummy_Id
Local list_does_not_exist = 6
Local no_more_move_required      : logical
Parameter created                : logical
.....
      IF created = ON
&      THEN GOTO LIST_EXISTS

CREATE: SET created = ON
      CALL AMCL01$Allocate_List (...,"My_List",...) --allocate list
      ...
BUILD:  LOOP FOR i in 1..nb_items
      ...
      CALL AMCL01$Modify_Complex_Entry (...) --fill items into list
      ...
      REPEAT BUILD

      SET Dummy_Id = 0      --Logical Access of list
      CALL AMCL01$Set_Permanent_List      --set list permanent
&      (Return_Status,Dummy_Id,"My_List",ON)
      ...
LIST_EXISTS:  ...
      CALL AMCL01$Multiple_Move_Parameter (...)
      IF error = list_does_not_exist THEN GOTO CREATE
      ...
      IF no_more_move_required THEN
&      (SET dummy_id = 0;
&      CALL AMCL01$Set_permanent_List
&      (Return_Status,Dummy_Id,"My_List",OFF);
&      ...)
```

### F.4.21 AMCL01\$Deallocate\_List

This function deallocates permanent and nonpermanent lists.

Note: Nonpermanent lists are automatically deallocated at the end of the program, or when the CL aborts.

```
SUBROUTINE AMCL01$Deallocate_List
  (Return_Status : OUT NUMBER;
   R_List_Id     : IN  OUT NUMBER;
   S_LIST_Id     : IN  STRING)
```

#### Description of Parameters:

- |                        |  |
|------------------------|--|
| R_List_Id (IN/OUT)     | This is the list identifier used for “direct” list access. If logical access is preferred, this real must be zero and the function will return the real list id in this parameter. |
| S_List_Id (INPUT)      | This string is the list identifier. It identifies the name of the list to deallocate. Only the first eight characters are taken into account for the unique list identifier.       |
| Return_Status (OUTPUT) | This parameter returns an error code to the user. The following error codes can be returned:   |
- 0 — OK—no errors
  - 6 — List does not exist—the function didn't find the list in the List Summary.
  - 9 — Not the list owner—the calling CL block is not linked to the point that owns the list.
  - 15 — Bad internal list identifier—the internal list identifier (real) does not match the external list identifier (string).

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

This sample program illustrates a case where a list deallocation can be useful. If the deallocations were not requested, the two lists would reside needlessly in memory for more than 1 minute.

```

...

CALL AMCL01$Multiple_Move_Parameter
&          (...,"my_source",...,"my_destin",...)

...

CALL AMCL01$Deallocate_List (error,1,"my_source")
CALL AMCL01$Deallocate_List (error,2,"my_destin")
...
CALL Bkg_Delay (60 secs)
...
EXIT

```

### F.4.22 AMCL01\$Get\_List\_Mem\_Stats

This function returns the maximum size of the memory available (for list configuration) to the user. It also returns the currently used memory.

```
SUBROUTINE AMCL01$Get_List_Mem_Stats
  (Return_Status   : OUT NUMBER;
   Current_Value   : OUT NUMBER;
   Max_Size        : OUT NUMBER)
```

#### Description of Parameters:

**Current\_Value (OUTPUT)** This parameter returns the current amount of memory (in words) used for the list structure.

**Max\_Size (OUTPUT)** This parameter returns the maximum amount of memory (in words) that can be allocated from heap pool two for holding the lists data structure.

**Return\_Status (OUTPUT)** This parameter returns an error code to the user. Only the following code can be returned.

0 — OK—no errors

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

#### Example:

This sample program illustrates how to warn a user when the Multiple\_Move\_Parameter list size gets close to the memory limit. It is then up to the user to take action.

```
...
CALL AMCL01$Allocate_List (...)
CALL AMCL01$Get_List_Mem_Stats (error,current_size,max_size)
IF error <> 0 then ...
SET memory_left = (1-(current_size/max_size)) * 100
IF current_size/max_size > 0.9
& THEN SEND: ("Attention: only ",memory_left,
&             " % of user memory left for lists")
...
```

### F.4.23 AMCL01\$Change\_List\_Mem\_Size

This function gives the user the capability of changing the maximum size of user memory dedicated to Multiple\_Move\_Parameter list storage. The user may increase or decrease this value, but he may never go below the current size of used memory (an error is returned in that case).

```
SUBROUTINE AMCL01$Change_List_Mem_Size
  (Return_Status : OUT NUMBER;
   Booster_Factor : IN  NUMBER)
```

#### Description of Parameters:

**Booster\_Factor (INPUT)** This is the factor (f) by which the current maximum size memory value will be multiplied. If  $f = 1$ , there is no change; if  $f < 1$ , max is decreased; if  $f > 1$ , max is increased.

**Return\_Status (OUTPUT)** This parameter returns an error code to the user. The following error codes can be returned:

- 0 — OK—no errors
- 4 — User memory limit exceeded—there is no more user memory available to satisfy the request.
- 20 — Bad booster factor—this can happen in the following cases:
  1. The value passed to the function is a “bad value” or a negative value.
  2. The difference between the current reserved memory and the requested reserved memory is larger than 32,760.
  3. The requested reserved memory size is smaller than the current memory size allocated for list structures.

#### NOTE

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

The rule that applies in this sample program is that if the current Multiple\_Move\_Parameter list size is less than half of the maximum value, the maximum value is reduced by 50%.

```
...

CALL AMCL01$Get_List_Mem_Stats (error,current_size,max_size)
IF error <> 0 then ...

SET memory_left = (1-(current_size/max_size)) * 100

IF memory_left > 50      -- % unused memory
& THEN CALL AMCL01$Change_List_Mem_Size (error,0.5)
...
```

### F.4.24 AMCL01\$Set\_List\_Item\_Active

This function allows the user to activate or inactivate a particular item in a Multiple\_Move\_Parameter list. If an item is “inactive,” the Multiple\_Move\_Parameter function does not move the corresponding data.

It is not necessary to inactivate an item in both the source and destination list. Inactivation of either the source or the destination item is sufficient to prevent the item from being moved, since there is a one-to-one correspondence between source and destination items.

```
SUBROUTINE AMCL01$Set_List_Item_Active
  (Return_Status : OUT NUMBER;
   R_List_Id     : IN  OUT NUMBER;
   S_List_Id     : IN  STRING;
   Item_Nb      : IN  NUMBER;
   Active_Flag   : IN  LOGICAL)
```

#### Description of Parameters:

R_List_Id (IN/OUT)	This is the list identifier used for “direct” list access. If logical access is preferred, this real must be zero and the function will return the real list id in this parameter.
S_List_Id (INPUT)	This string is the list identifier. It identifies the list in which an item is to be activated or inactivated. Only the first eight characters are taken into account for the unique list identifier.
Item_Nb (INPUT)	This parameter identifies which item is to be activated or inactivated in the list referenced by “S_List_Id” or “R_List_Id.”
Active_Flag (INPUT)	If this parameter is “OFF,” the list item will be inactivated. This means that it will be ignored by the Multiple_Move_Parameter function. If the parameter is “ON,” the list item will be activated so that a “move” for this item can take place.
Return_Status (OUTPUT)	This parameter returns an error code to the user. The following error codes can be returned:
0	— OK—no errors
6	— List does not exist—the function didn’t find the list in the List Summary.
8	— Bad list item number—the value of “item_nb” passed to the function was a “bad value” or was not in the range 1..max number of items in the list.
9	— Not the list owner—The calling CL block is not linked to the point that owns the list.
15	— Bad internal list identifier—the internal list identifier (real) does not match the external list identifier (string).

**NOTE**

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

```
..... Allocate source list "rec_data" and destination list "my_equ" ....

IF ( (EQU_nb = 3) AND (COND = OK) )
& THEN CALL AMCL01$Set_List_Item_Active
&      (error,s_list_id,"rec_data",1,OFF)

ELSE CALL AMCL01$Set_List_Item_Active
&      (error,s_list_id,"rec_data",1,ON)

CALL AMCL01$Multiple_Move_Parameter
&      (error,da_error,s_list_id,"rec_data",d_list_id,"my_equ")
.....
```

As shown in the example, this function can be useful when generic lists are defined. The above example illustrates the following case:

A "rec\_data" source list defines a set of recipe data that needs to be downloaded into one equipment unit. The list is valid for almost all equipment in the plant. Equipment 3 is an exception, since the first recipe data item cannot be downloaded if a certain condition occurs. However, with the inactivation function, it is not necessary to define a special source list to handle this exception. An inactivation of the source first item is sufficient.

rec_data	:	temp1*	temp2	temp3	(source list)
equ01	:	TIC101.sp	TIC102.sp	TIC103.sp	(destination lists)
equ02	:	TIC201.sp	TIC202.sp	TIC203.sp	
equ03	:	TIC301.sp	TIC302.sp	TIC303.sp	
equ04	:	TIC401.sp	TIC402.sp	TIC403.sp	

\* set inactive if destination list is EQU03 and "COND" is OK



### F.4.25 AMCL01\$Get\_Exec\_Time\_Stats

The function provides statistics on the execution time of the AMCL01\$Multiple\_Move\_Parameter function. These statistics are available for every destination list used by the system.

```
SUBROUTINE AMCL01$Get_Exec_Time_Stats
  (Return_Status   : OUT NUMBER;
   R_List_Id       : IN  OUT NUMBER;
   S_List_Id       : IN  STRING;
   Exec_Time_1     : OUT NUMBER;
   Exec_Time_2     : OUT NUMBER;
   Exec_Time_3     : OUT NUMBER;
   Exec_Time_4     : OUT NUMBER;
   Exec_Time_5     : OUT NUMBER;
   Min_Val         : OUT NUMBER;
   Max_Val         : OUT NUMBER;
   Average         : OUT NUMBER;
   Number_Samples  : OUT NUMBER;
   RESET          : IN  LOGICAL)
```

#### Description of Parameters:

R_List_Id (IN/OUT)	This is the list identifier used for “direct” list access. If logical access is preferred, this real must be zero and the function will return the real list id in this parameter.
S_List_Id (INPUT)	This string is the destination list identifier. It identifies the destination list for which the statistics are requested. Only the first eight characters are taken into account for the unique list identifier.
Exec_Time_1 (OUTPUT)	Elapsed time of the last Multiple_Move_Parameter to the current destination list in number of seconds.
Exec_Time_2..5 (OUTPUT)	RESERVED
Min_Val (OUTPUT)	The Multiple_Move_Parameter execution time minimum value since the last reset in number of seconds.
Max_Val (OUTPUT)	The Multiple_Move_Parameter execution time maximum value since the last reset in number of seconds.
Average (OUTPUT)	The Multiple_Move_Parameter average execution time value since the last reset in number of seconds.
Number_Samples (OUTPUT)	Number of moves to the current destination list since the last reset.

Reset (INPUT)	If this flag is “ON,” the function will reset its internal values after returning the current statistics. On the next invocation, the function will return number_samples = 1; Min_Val = Max_Val = Average = Exec_Time_1.
Return_Status (OUTPUT)	This parameter returns an error code to the user. The following error codes can be returned:
0	— OK—no errors
6	— List does not exist—the function didn’t find the list in the List Summary.
15	— Bad internal list identifier—the internal list identifier (real) does not match the external list identifier (string).

**NOTE**

If an error code greater than 100 is returned, the error is major and should be reported to Honeywell TAC.

**Example:**

The following program sends a message each time the execution time of a particular multiple move is abnormally long (20% longer than the average time). This message, together with the message time stamp, can be useful to correlate this event with other events during debugging of an application.

```

.....
CALL AMCL01$Multiple_Move_Parameter
&          (error_sts,da_sts,R_source,My_source,R_Destin,My_Destin)

IF error_sts <> 0, THEN ...

CALL AMCL01$Get_Exec_Time_Stats
&          ( error_sts,
&          R_destin,My_destin,
&          exec_time,
&          dummy1,dummy2,
&          dummy3,dummy4,
&          Min_exec_time,
&          Max_exec_time,
&          Average,
&          number_samples,
&          OFF )
&
IF ( (number_samples > 20) AND
&    (exec_time > 1.20 * average) )
& THEN SEND: "Abnormal execution time for multiple_move to ",My_destin
.....

```

## AM EXTENSION FOR FAST EXTERNAL CDS FETCH

### Appendix G

*This appendix describes an optional load module that, when loaded in an AM that has custom data segments, reduces the time to access that custom data from other modules on the LCN.*

#### G.1 OVERVIEW

The AM Extension for Fast External Custom Data Segment (CDS) Parameter Fetch, **AMCL04**, is delivered with the standard AM software. If the software is delivered on floppy diskettes, the set is delivered on the &AMS floppy. If the software is delivered on cartridge disks, the set is delivered on the &C3 cartridge, labeled “Application Module Personalities.”

This extension contains no CL subroutine calls. When it is loaded in an AM that has custom data segments, it allows other modules on the LCN to have faster access to this custom data. The other modules could be Computer Gateways, Universal Stations (using custom data in schematics), or other Application Modules. The extension functions by optimizing CDS access structures in the software for fast access.

The extension is packaged as an option rather than as a standard feature because it uses additional memory (see subsection G.3). Some users who have limited custom data may not benefit from the option. Other users may not have the additional memory available or may not wish to use it for this function.

#### G.2 AM PERFORMANCE IMPACT

For a typical AM that makes use of custom data segments, its Central Processing Unit (CPU) time needed to access that custom data with AMCL04 loaded is 1/4 to 1/5 of what it is without AMCL04 loaded.

Typical, in this instance, is considered to be an AM with 100 unique custom data segment names, an average of five custom data segments per point, and an average of 20 parameters per custom segment.

Within an AM, approximately 30% of the CPU time is allocated for background CL and external access to the AM's custom data. These two processes compete with each other for the 30% “slice” of the processor. Therefore, there are two possible effects when an AM has a heavy load of external custom data fetches. Either or both of these may be visible to the user. One effect is that background CL programs may be held up due to the external fetches. The other effect is that the external devices requesting custom data may exhibit a slowdown—for example, in schematics or in CL programs in other AMs.

Adding AMCL04, therefore, has the effect of speeding up external CDS data fetches and of freeing up more AM CPU time for background activity.

Without AMCL04, fetching a parameter from a custom data segment in an AM from a module outside the AM takes from about 0.6 milliseconds of CPU time to as much as 3.0 milliseconds or more. This time depends upon:

1. The number of unique custom data segment names in the AM.
2. The number of custom data segments on the point being accessed.
3. Which custom data segment (the segments are searched in order) the particular parameter happens to be on.
4. The average number of parameters in the custom data segment on the point being accessed.

The bigger each of the numbers in the above list is, the more CPU time a custom parameter fetch takes.

With AMCL04 loaded, all custom parameter fetches take about 0.3 milliseconds of AM CPU time.

The times given above are for fetches of single parameter real values. Fetches of other types vary somewhat. For fetches of entire arrays, the savings in CPU time is the same numerically, but the ratio of savings is not as great since the whole array still has to be copied.

Note: Adding this set to the AM gives little or no benefit if custom data is not used extensively.

### G.3 AM MEMORY IMPACT

The following formula gives the approximate number of words of user memory used by this extension:

$$20,000 + 6 * N1 + 2 * N2 * N3 + N4 * [6 + 2 * N5]$$

Where:

N1 = The total number of parameter names used in custom data segment descriptors.

N2 = The total number of custom data segment descriptors.

N3 = Average number of parameters in a descriptor.

N4 = Number of points with custom data segments attached.

N5 = Average number of custom data segments on a point.

#### Example:

An AM that has:

1. 3000 parameter names used in 200 different custom segments, each of which contains an average of 25 parameters.
2. An average of four custom data segments attached to each of 2,000 points.

The memory used becomes:

$$20,000 + (6 * 3000) + (2 * 200 * 25) + 2000 * (6 + (2 * 4)) = 76,000 \text{ words}$$

Note: With no points and AMCL04 loaded, 20,000 words of user space are used. The additional space is gradually used as the AM is filled with points that have associated custom data.

### G.4 PACKAGING

This Fast External CDS Access function is packaged as the set AMCL04. AMCL04 contains code that enables the Fast External CDS Fetch on AM startup.

### G.5 DISTRIBUTED FILES

The loadable object file AMCL04.LO is necessary to install AMCL04 on an Application Module.

## G.6 INSTALLATION

Using the Command Processor function of the Engineering or Universal Personality, determine if the directory &CUS exists on the HM:

```
LSV NET
```

If the directory does not exist, use the Create Directory command to create a directory under a volume on the History Module:

```
CD NET>vol>&CUS    (vol is an existing volume on the HM)
```

Mount the volume the set is delivered on (&C3 if cartridge or &AMS if floppy) in a drive and copy the file AMCL04.LO to the History Module:

```
CP $Fx>&CUS>AMCL04.LO  NET>&CUS>AMCL04.LO
```

## G.7 APPLICATION MODULE CONFIGURATION

1. From the Modify Volume Paths display in the Engineer or Universal Personality, set the NCF Backup Path to a removable media pathname; for example, \$Fx>&ASY> (where x is the removable media drive number).
2. From the Main Menu of a US with Engineer Personality, select the target:  
LCN NODES.
3. Select the node number of the AM to be configured.
4. Select the MODIFY NODE target on page 1.
5. Page forward to page 3 and enter the name of the set (AMCL04) in the list of externally loaded modules and enter AMO in the corresponding personality entry, then press the ENTER key.
6. Press the CTL and the one (1) key together (F1) to check the configuration.  
/
7. Press the CTL and the two (2) key together (F2) to install the configuration.
8. Load the AM and AMCL04 will be active.

## AM EXTENSION FOR OFF-NODE ACCESS

### Appendix H

*This appendix describes an optional load module that enables you to have more flexibility accessing data in other units on the same AM. The enhancements include fetching string array elements in other units and accessing data using multilevel indirections, where the entities at each level of indirection can be in other units.*

#### H.1 OVERVIEW

The AM Extension for Off-Logical-Node Access, **AMCL05**, is delivered with the standard AM software. If the software is delivered on floppy diskettes, the AMCL05 extension will be on the &AMS floppy. If the software is delivered on cartridge disks, the AMCL05 extension will be on the &C3 cartridge, which is labeled “Application Module Personalities.”

This extension contains no CL subroutine calls. Honeywell provides this extension as an option rather than as a standard feature because it restricts the freedom to move units from one AM to another AM. You should consider this restriction as well as the benefits of the extension when deciding whether or not to implement it.

#### H.2 FUNCTIONALITY OF THE EXTENSION

The AMCL05 extension consists of two features:

- One feature allows users, in foreground CL, to fetch string array elements from any on-physical-node point (point in the same AM). Without the AMCL05 extension loaded, string array elements can only be fetched from on-logical-node points (points in the same unit).
- The other feature allows users to go to points that are off-logical-node (points that are in another unit) at any level of multilevel indirection. All of the points through which indirection occurs must be on-physical-node (in the same AM). The enhancement is valid for both fetches and stores. Without the AMCL05 extension loaded, off-logical-node references can only occur in the last level of multilevel indirection. Off-physical-node references can only occur in the last level of multilevel indirection with or without AMCL05.

#### WARNING

If you make use of either of the above features and subsequently move a unit containing any of the involved references off-physical-node (to another AM), the AM will not be able to perform the references. At startup, you will get one or both of the following status messages indicating the units with invalid references.

```
Bad Direct References on unit: xx    (most likely message)
Bad Indirect References on unit: xx
```

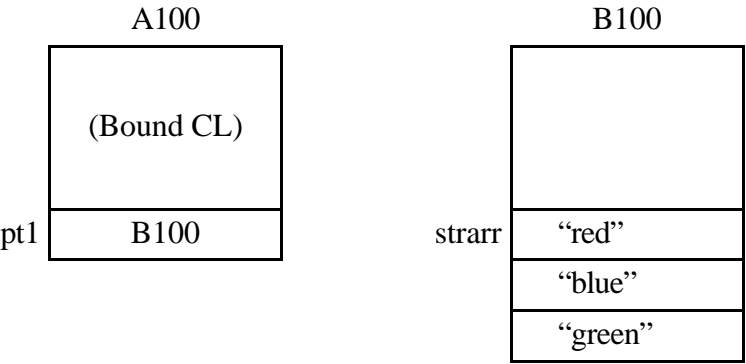
You may also get a WARNING node status, depending on the nature of the reference. If the situation is not corrected, you will get configuration errors at run time.

The following examples illustrate the two features:

**Example 1—String Array Fetches**

In the following diagram, AM point A100 has a bound CL program and has a custom parameter pt1 of type “custom parameter list” that is assigned the value B100. Point B100 has a custom array parameter “strarr” of type String. The CL program could access an element of the array strarr by a statement such as:

```
IF pt1.strarr(1) = "red" THEN ...
```

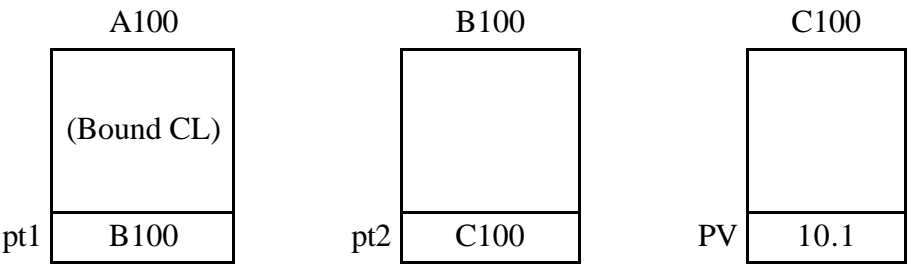


Without AMCL05, A100 and B100 must be in the same unit. With AMCL05 loaded, A100 and B100 may be in different units, but must be in the same AM.

**Example 2—Multilevel Indirection**

In the following diagram, AM point A100 has a bound CL program and has a custom parameter pt1 of type “custom parameter list” that is assigned the value B100. Similarly, point B100 has a custom parameter pt2 that is assigned the value C100. Point C100 has a PV whose current value is 10.1. The CL program could access C100.PV through multilevel indirection with a statement such as:

```
IF pt1.pt2.pv > 10.0 THEN ...
```



Without AMCL05, off-logical-node references can occur only in the last level of multilevel indirection; therefore, A100 and B100 must be in the same unit. With AMCL05 loaded, A100 and B100 can be in different units on the same AM. In both cases, C100 can be in another unit or in another node.



### H.3 AM PERFORMANCE IMPACT

There is no direct performance impact on the AM.

### H.4 AM MEMORY IMPACT

There is no memory impact on the AM.

### H.5 PACKAGING

The Extension for Off-Node Access is packaged as the set AMCL05.

### H.6 DISTRIBUTED FILES

The object file AMCL05.LO is necessary to install AMCL05 on an Application Module.

### H.7 INSTALLATION

Using the Command Processor function of the Engineering or Universal Personality, determine if the directory &CUS exists on the HM:

```
LSV NET
```

If the directory does not exist, use the Create Directory command to create a directory under a volume on the History Module:

```
CD NET>vol> &CUS (vol is an existing volume on the HM)
```

Mount the volume on which the enhancement is delivered (&C3 if cartridge or &AMS if floppy) in a drive and copy the file AMCL05.LO to the History Module:

```
CP $Fx>&CUS>AMCL05.LO NET>&CUS>=
```

## H.8 APPLICATION MODULE CONFIGURATION

1. From the Modify Volume Paths display in the Engineer or Universal Personality, set the NCF Backup Path to a removable media pathname; for example, \$F<math>x>&ASY> (where x is the removable media drive number). Insert a backup &ASY floppy or cartridge in the removable media drive.
2. From the Main Menu of a US with Engineer Personality, select the LCN NODES target.
3. Select the node number of the AM to be configured.
4. Select the MODIFY NODE target on page 1.
5. Page forward to page 3 and enter the name of the extension (AMCL05) in the list of externally loaded modules and enter AMO in the corresponding personality entry, then press the ENTER key.
6. Press the CTL and the one (1) key together (F1) to check the configuration.
7. Press the CTL and the two (2) key together (F2) to install the configuration.
8. Load the AM and AMCL05 will be active.

## APPENDIX I - APPLICATION MODULE<sup>X</sup> CL RUNTIME EXTENSIONS

### I.1 OVERVIEW

#### I.1.1 Application Module<sup>X</sup> Basics

The Application Module<sup>X</sup> (A<sup>X</sup>M) consists of a standard Application Module (AM) combined with a high-performance PA-RISC (reduced instruction set) coprocessor running the Hewlett-Packard HP-UX operating system. The coprocessor provides an open programming environment that supports analysis and control applications provided by Honeywell and other vendors, as well as custom programs developed in-house by customers. The coprocessor also provides access to other resources on the Plant Information Network (PIN).

Traditional AM point processing functions are said to be in the “control domain.” HP-UX coprocessor applications are said to be in the “information domain,” and are also referred to as X-side functions.

#### I.1.2 Function of these Runtime Extensions

These runtime extensions consist of a set of subroutines that are packaged as an external load module named AMCL06. The subroutines allow CL programs running in the AM (control domain) to launch X-side applications, pass command line arguments to those applications, check the status of the A<sup>X</sup>M CL queue, and change the state of the global X-access switch. A<sup>X</sup>M Release 200 includes CL subroutines to initiate, activate, and terminate hibernating X-side applications, and a CL subroutine to view the X-side hibernating application queue.

#### I.1.2 Multiple Versions

Multiple versions of AMCL06 exist to support different versions of LCN software. They are identified by AMCL06, AMCL06\_1, AMCL06\_2, and so forth. Refer to the *Application Module<sup>X</sup> Customer Release Guide* to determine the proper version to configure for your system.

#### WARNING

This extension set must be configured in an AM that is loaded with A<sup>X</sup>M software. If AMCL06 is not configured, the A<sup>X</sup>M will crash during startup. If an incompatible version of AMCL06 is installed, a SEVERE node status message will be output during startup and the runtime extensions in AMCL06 will return initialization unsuccessful errors.

### I.1.3 Subroutine Summary

This appendix covers the following subroutines:

- **AMCL06\$Execute\_Task\_With\_Wait**—Used to initiate an application in the X-side. The program uses one AXM CL queue slot until it has completed.
- **AMCL06\$Get\_Queue\_Info**—Used to find out how many AXM CL queue slots are available and how many are in use.
- **AMCL06\$Get\_Queue\_Slot\_Info**—Used to get status information from an AXM CL queue slot about a CL-initiated X-side program.
- **AMCL06\$Get\_Queue\_Info** —Used to change the state of the global X-access switch.

NOTE: The following four tasks require AXM software release R200:

- **AMCL06\$Initiate\_Task**—Initiates an OpenDDA application capable of hibernating and receiving background CL events.
- **AMCL06\$Activate\_Task**—Sends an activate event to a CL-initiated hibernating OpenDDA application.
- **AMCL06\$Terminate\_Task**—Sends a terminate event to a CL-initiated hibernating OpenDDA application, providing a graceful shutdown of the application.
- **AMCL06\$Get\_Hiber\_Task\_Status**—Obtains the current status of a CL-initiated hibernating OpenDDA application capable of receiving background CL events.

## I.2 INSTALLATION

### I.2.1 Memory Requirements

This set requires memory per the following table. This memory includes the software, data buffers, and extra space to equalize user available heap with a standard AM personality.

AXM and LCN Release	Memory (K words approximate)
AXM 100 on LCN Release 430 or 431	499
AXM 110 on LCN Release 431	840
AXM 110 on LCN Release 500	2451
AXM 200 on LCN Release 431	840
AXM 200 on LCN Release 500	2451

## I.2.2 Packaging

The subroutines are packaged together in the set AMCL06. The set consists of a linked object file that contains the subroutines in the set, and a Set Definition File that contains a description of the calling sequences of the subroutines for the CL compiler. These files must reside in directories &CUS and &CLX on the History Module, and the A<sup>X</sup>M that is to contain the set must be configured through the Network Configurator to load the set.

## I.2.3 Distributed Files

The following files are necessary to install AMCL06 on an Application Module<sup>X</sup>:

- AMCL06.LO, AMCL06\_1.LO, AMCL06\_2.LO, ...—loadable object file, located in &CUS (Refer to the Application Module<sup>X</sup> Customer Release Guide to determine the correct set to configure and install).
- AMCL06.SF—set definition file, located in &CLX.

## I.2.4 Installation on History Module

Use the following procedure to determine if the required files are installed on the History Module, and to install them if they are not present:

Step	Action
1	Using the Command Processor of the Engineering or Universal personality, determine if the directories &CUS and &CLX exist on the History Module:  LSV NET (lists volumes and directories on the network)
2	If the directories exist, verify that the directory &CUS contains the desired version of the file AMCL06.LO and that the directory &CLX contains the file AMCL06.SF:  LS NET>&CUS LS NET>&CLX
3	If either or both directories do not exist, use the Create Directory command to create the missing directory or directories on the History Module:  CD NET>vol &CUS (vol is an existing volume on the HM) CD NET>vol &CLX
4	If you had to create the directories or if they did not contain the required files, copy the files from the Application Module Personality cartridge (&C3)(x = the removable media drive number):  CP \$Fx>&CUS>AMCL06.LO NET>&CUS>AMCL06.LO CP \$Fx>&CLX>AMCL06.SF NET>&CLX>AMCL06.SF  Note: When copying the .LO file in the first step above, substitute the proper version of the .LO file in the command—AMCL06.LO, AMCL06_1.LO, AMCL06_2.LO, ...

## I.2.5 Configure the A<sup>X</sup>M for AMCL06

Use the following procedure to make the NCF change that is required to configure the A<sup>X</sup>M for AMCL06:

Step	Action
1	If the A <sup>X</sup> M is a new node, perform the online NCF change to add the node. For information, refer to the subsection "Online Network Reconfiguration Procedures" in the <i>Network Data Entry</i> manual in the <i>Implementation/Startup &amp; Reconfiguration - 1</i> binder.
2	From the Engineering Main Menu, select <b>SUPPORT UTILITIES</b> .
3	Select <b>MODIFY VOLUME PATHS</b> .
4	Mount formatted media containing the directory &ASY in the drive. Set the NCF Backup Path to a removable media pathname:  \$F<x>&ASY> (x = drive number)
5	From the Engineering Personality Main Menu, select <b>LCN NODES</b> .
6	A display will appear with a target for each node possible on the LCN. Verify that the configurator is in the ON-LINE mode in the upper right corner of the display. Select the target for the node number of the A <sup>X</sup> M that you wish to configure for AMCL06 (you may need to use the Page Forward key to display the target for the desired A <sup>X</sup> M).
7	A display will appear that allows the configuration of this A <sup>X</sup> M node on the network. If the node is not already configured on the network, do so at this time.
8	Select the <b>MODIFY NODE</b> target (not required if creating a new node).
9	Page forward to page three and enter the name of the appropriate version of the external load module, AMCL06, and the associated personality type, AMO. Refer to the <i>Application ModuleX Customer Release Guide</i> to determine the appropriate AMCL06 version. If the name AMCL06 already exists, replace it with the appropriate name.
10	Press the [ENTER] key.
11	Press [CTL-F1] to check the new NCF, and press [CTL-F2] to install the new NCF.
12	Press the [ENTER] key to confirm the install.
13	Return to the Engineering Personality Main Menu. A message will appear at the bottom of the display indicating the successful completion of the NCF installation.

Step	Action
14	Load the A <sup>X</sup> M. The load will include the appropriate AMCL06 version.
15	If the A <sup>X</sup> M is a newly added node (see Step 1), reload all US and U <sup>X</sup> S nodes from which you will access the A <sup>X</sup> M. This will make the A <sup>X</sup> M visible to these nodes.

## I.3 EXECUTING X-SIDE APPLICATIONS

### 1.3.1 Queues

There are three queues that can be involved in the execution of CL-initiated X-side applications in an A<sup>X</sup>M:

- The AM Background Queue
- The AXM CL Queue
- The Hibernating Queue

The AM Background Queue determines the background CL blocks currently executing on an AM or A<sup>X</sup>M. It is of configurable size up to a maximum of 10 slots. An executing CL block uses one slot of the Background Queue. A background block is allowed to execute if and only if there is a free slot in the Background Queue. For a detailed description of CL background execution, refer to Section 4 of the *AM Control Functions* manual.

The A<sup>X</sup>M CL Queue can be viewed as a table in memory that holds the current status and pertinent information about all presently active AMCL06 extension set requests (excluding **AMCL06\$Get\_Queue\_Info** and **AMCL06\$Get\_Queue\_Slot\_Info**). The size of this table is not configurable and is fixed at 10 slots. Information about the contents of the A<sup>X</sup>M CL Queue can be viewed with the call **AMCL06\$Get\_Queue\_Info**. Information about the contents of a particular slot of the A<sup>X</sup>M CL Queue can be viewed with the call **AMCL06\$Get\_Queue\_Slot\_Info**.

The Hibernating Queue on the X-side of an A<sup>X</sup>M holds status and pertinent information about all hibernating applications currently executing (or in hibernation). The **AMCL06\$Initiate\_Task** subroutine call in a background CL block is used to launch an application that can hibernate, and this call indirectly reserves a slot in the Hibernating Queue. Information about the contents of the Hibernating Queue can be viewed with the X-side tool `display_appls`.



### I.3.1.1 Operation of Queues

A background CL block executes if a slot is available in the Background Queue. The available Background Queue slot is assigned to the executing background CL block and is held by the CL block until the CL block completes execution.

If the executing background CL block contains any of the AMCL06 CL extension requests (excluding **AMCL06\$Get\_Queue\_Info** and **AMCL06\$Get\_Queue\_Slot\_Info**), a slot in the A<sup>X</sup>M CL Queue is reserved for the AMCL06 request.

If a background CL block initiates a nonhibernating application X-side application (with the call **AMCL06\$Execute\_Task\_With\_Wait**), the slots in the AM Background CL Queue and the A<sup>X</sup>M CL Queue remain in use while the X-side application executes and until the CL block subsequently terminates.

If a background CL block initiates a hibernating X-side application (with the subroutine call **AMCL06\$Initiate\_Task**) and the CL block's associated point has not already initiated an instance of the application, a slot in the Hibernating Queue is reserved for the X-side application. The slot in the Hibernating Queue is freed up only after the hibernating application terminates. The associated slots in the AM Background CL Queue and the A<sup>X</sup>M CL Queue are freed up when the application enters hibernation.

## I.3.2 Nonhibernating Tasks

Nonhibernating X-side applications are initiated from CL using the subroutine **AMCL06\$Execute\_Task\_With\_Wait**. This subroutine is available in A<sup>X</sup>M Release R100 and later. Using the CL subroutine **AMCL06\$Execute\_Task\_With\_Wait**, a background CL block can execute any application which resides on the HP-UX disk in the directory `/users/axm`. The X-side application must run to completion and exit before the CL block will resume processing. This method of X-side application initiation is referred to as a "synchronous application with termination."

## I.3.3 Hibernating Tasks

Hibernation was introduced in A<sup>X</sup>M Release 200 to provide an alternate method of X-side application initiation. Using the CL subroutine **AMCL06\$Initiate\_Task**, a background CL block can initiate an X-side OpenDDA application that can hibernate instead of exiting its execution. Once a CL subroutine has initiated an OpenDDA application that entered hibernation, the background CL subroutines **AMCL06\$Activate\_Task** and **AMCL06\$Terminate\_Task** can be used to activate and gracefully terminate the hibernating OpenDDA application. This method of X-side application initiation is referred to as a "synchronous application with hibernation." Hibernation provides the benefit of allowing an X-side application to respond quickly to subsequent CL activations after the first initiation. Additionally, some X-side applications require that data stored within the application (in the form of native variables into which data has been stored) be persistent from one execution to the next.

For more information about hibernating applications, refer to subsection I.4 in this appendix, and to Section 5 of the *Application Module<sup>X</sup> User Guide*.

### I.3.4 HP-UX errno

The HP-UX operating system provides an external variable “errno.” Functions and applications in the operating system use this variable to indicate error conditions. HP-UX also provides routines needed to interpret errno.

Some of the AMCL06 subroutines return the HP-UX errno value in their Detailed Status. Refer to the individual subroutines for tables of Return Status and Detailed Status.

For more information about the use and interpretation of errno, refer to the HP-UX documentation on CD-ROM. Another useful reference is the HP-UX Manual Pages. Access this information by typing the following commands at the HP-UX prompt:

```
man errno
man strerror
```

### I.3.5 HP-UX Signals

Signals are transmissions between processes and are used for interprocess communication. Signals can also be viewed as events to which processes respond. When a process receives a signal, the process might ignore the signal, terminate, defer the signal, or execute a signal handler. HP-UX defines several signal interfaces that allow a process to specify the action taken upon receipt of a given signal.

Sometimes an HP-UX signal can kill a process. If this occurs to an application associated with a CL block, the AMCL06 subroutine will return the number of the signal in its Detailed Status. Refer to the individual subroutines for tables of Return Status and Detailed Status.

Refer to the HP-UX CD-ROM documentation for more information on signals, signal interfaces, and signal specification.

## I.4 HIBERNATING APPLICATIONS

### I.4.1 Overview

Hibernation provides the user the ability to force an X-side OpenDDA application, initiated by a CL block, to remain in memory after the first invocation. Subsequent invocations do not reinitialize the application, but allow the application to resume execution at the appropriate entry point.

Through OpenDDA EDB and EXEC DDA statements, an OpenDDA application can hibernate, and then receive activate and terminate events from a CL block. These OpenDDA features are not described in this document (see the *OpenDDA Reference Manual*).

### I.4.2 Application Initiation

A background CL block starts an OpenDDA application using the **AMCL06\$Initiate\_Task** subroutine call. Execution of the CL block is suspended, waiting for the application to initiate and hibernate. On the X-side, after the application is invoked, it performs any necessary processing, followed by an EXEC DDA execution statement to enter hibernation and wait for a CL event. Once the OpenDDA application enters hibernation, the CL block's execution is resumed, upon which an application identifier is returned to the CL block which is needed for future event notification. The CL block should store the OpenDDA application id in any CDS parameter or other point.parameter, and then complete the execution of the CL block.

### I.4.3 Application Activation

On subsequent point executions, any background CL block attached to the point which initiated the application can send an activate event to an OpenDDA application which is waiting for a CL event. The CL block calls the subroutine **AMCL06\$Activate\_Task** and supplies the application identifier (returned from **AMCL06\$Initiate\_Task**) of the OpenDDA application to activate. Execution of the CL block is suspended, waiting for the application to process the event and return to hibernation. On the X-side, when the event occurs, the execution of the OpenDDA application is resumed. The application receives the event through the EXEC DDA execution statement which will populate the information for the event as specified in the OpenDDA External Data Block. The application performs its necessary processing and returns to hibernation, again using the EXEC DDA execution statement. After the OpenDDA application enters hibernation, the CL block's execution is resumed.

### I.4.4 Graceful Application Termination

To gracefully shutdown a hibernating application, any background CL block attached to the point which initiated the application can send a terminate event to an OpenDDA application using the **AMCL06\$Terminate\_Task** subroutine call. The execution flow between the CL block and OpenDDA application is the same for a terminate event as an activate event, with one exception. After performing any necessary processing upon receipt of the terminate event, the OpenDDA application must terminate its execution before the CL block will resume processing.

## I.4.5 Aborting Application Execution

There are several ways to abort a running or hibernating OpenDDA application:

- As with the **AMCL06\$Execute\_Task\_With\_Wait** subroutine call, while executing the subroutines **AMCL06\$Initiate\_Task**, **AMCL06\$Activate\_Task**, or **AMCL06\$Terminate\_Task**, aborting the running CL block or inactivating the point will cause the associated OpenDDA application to abort. Conversely, there is no impact to a hibernating OpenDDA application if an associated point is made inactive while the CL subroutines are not running. Therefore, if an OpenDDA application is actively running, and the point is made inactive or the CL block is aborted, the application will be aborted; whereas, if the OpenDDA application is hibernating, any action on the point has no affect on the hibernating application.
- The `kill_appls` tool, accessible from the X-Side, can be used to abort one or all CL-initiated OpenDDA applications associated with a point, regardless of the state of execution of the application (running or hibernating). It will also abort one or more instances of the same application name. NOTE: There is no mechanism from the point detail display that accomplishes this activity. This functionality is only available from the X-side through the `kill_appls` tool.
- A graceful termination can be accomplished by sending a terminate event to the hibernating OpenDDA application from a Background CL block using **AMCL06\$Terminate\_Task** (see Graceful application termination above).

## I.4.6 Restrictions

The number of background tasks is a configurable option with a maximum of 10. Therefore, a maximum number of 10 simultaneous CL events can be processed at one time. Although the running background CL configurable limitation is 10, the maximum number of OpenDDA applications associated with hibernation is always 50.

An OpenDDA application must be initiated from a background CL block using the subroutine **AMCL06\$Initiate\_Task** in order to receive CL events.

Only a single instance of an OpenDDA application will be allowed to execute from a single point (see **Recovery** below). However, two different points can execute the same OpenDDA application, although this will result in two separate independent executing programs that are not linked in any way. Additionally, a single point can execute many different OpenDDA applications. If it is necessary to initiate multiple instances of the same OpenDDA application from the same point, multiple links with different names can be established to the same application on the X-side.

To avoid unintentional abandoned OpenDDA applications, deleting a point will be unsuccessful if the point has an associated OpenDDA hibernating application actively running or hibernating. The OpenDDA application must be aborted before the point can be deleted (see **Aborting application execution** above).

## I.4.7 Recovery

Any attempt to invoke more than one instance of an OpenDDA application name from a point will be rejected, although the application identifier of the current instance of the application initiated by the point will be returned to the calling CL block. This allows the CL block to determine if the existing hibernating OpenDDA application should be activated, or the application should be terminated and a new instance of the application initiated.

All CL-initiated hibernating applications will be killed when the LCN side of the A<sup>X</sup>M goes down and it is the responsibility of the CL blocks to reinitiate the OpenDDA applications.

## I.5 AMCL06\$Execute\_Task\_With\_Wait

### I.5.1 Purpose

This subroutine is used to initiate an application in the X-side coprocessor. It can only be called from background CL. After the call, the background CL suspends execution and goes into a wait condition until the X-side application terminates normally or terminates because an error was detected

### I.5.2 Syntax

Syntax of the **AMCL06\$Execute\_Task\_With\_Wait** subroutine:

```
SUBROUTINE AMCL06$Execute_Task_With_Wait
  (Ret_Status      : OUT NUMBER;      -- Return status of the call
   Det_Status      : OUT NUMBER;      -- Detailed return status
   Cmd_Line        : IN  STRING;      -- X-side application command
line
   X_Task_Timeout  : IN  TIME;        -- X-side timeout value
   Req_Timeout     : IN  TIME)        -- LCN-side timeout value
```

### I.5.3 Ret\_Status and Det\_Status

Ret\_Status is the return status of the subroutine call. Det\_Status (Detail Status) may contain additional information. Values are:

Value	Ret_Status	Det_Status
0	Application terminated normally	Application exit code (defined in the application program)
1	Subroutine argument error	1 = Invalid X_Task_Timeout 2 = Invalid Req_Timeout 3 = Invalid Cmd_Line
2	CL timeout while waiting for X-side application to complete—application aborted	0
3	Unable to communicate with X-side	0 = No detail status 4 = Unexp'd connect from X-side 5 = Unexp'd disconnect from X-side
4	Error getting memory in AM	0
5	LCN/X-side connection down	0
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
91	Initialization unsuccessful—unable to acquire internal resources	0
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
93	Initialization unsuccessful—unable to acquire internal resources	0
500	Internal error	0
501	Internal produce error	0
502	Internal consume error	0
503	Internal priority error	0
504	Internal message/queue mismatch	0
999	Node not an A <sup>X</sup> M	0

Value	Ret_Status	Det_Status
1001	Application was killed by signal	Signal number
1002	Application name invalid or application not found	HP-UX errno (or 0)
1003	Application timed out (X-side timeout)—application aborted	0
1004	Miscellaneous internal error	0
1005	An error occurred while changing priority	HP-UX errno
1006	An error occurred while creating the application's process	HP-UX errno
1007	The application did not have any execute permission set	0
1008	An error occurred while executing the application	HP-UX errno
1009	A miscellaneous HP-UX error occurred in the CDS/CL server	HP-UX errno
1010	The application command line contained an absolute path, which is not allowed	0
1011	An error occurred while setting up the application's environment	0

## I.5.4 Cmd\_Line

This argument is passed to the X-side where it is interpreted as an HP-UX command line. It is a string of up to 78 characters containing the application name, not including the pathname, and any application arguments. The application is responsible for interpreting the command line arguments. The application must reside in the directory `/users/axm` on the HP-UX disk (see *Application Module<sup>X</sup> User Guide*, subsection 2.3, LCN Security, Directory of CL-initiated applications).

### I.5.4.1 Cmd\_Line Restrictions

The HP-UX command line can contain multiple arguments, but the first argument must be the X-side application name. `Cmd_Line` is a CL local string variable, and is therefore limited to a maximum of 78 characters.

`Cmd_Line` is not processed for HP-UX shell commands, and therefore **does not support** items such as:

- HP-UX and shell commands
- Multiple commands separated with “;”
- Pathname expansion such as `~user/test`
- Input/output redirection such as: `|, <, >, >>`

If you want to do shell commands prior to invoking the application, `Cmd_Line` can specify a script file. The script can set environment variables and/or do other functions, and then start the application. At the top of the script file, specify the shell it is to run in:

```
#!/bin/sh    or    #!/bin/csh    or    #!/bin/ksh
```

When setting environment variables, use the correct syntax for that shell.

Keep in mind, however, that if you use `Cmd_line` to specify a script to start an application, the subroutine **AMCL06\$Get\_Queue\_Slot\_Info** will return information about the script file and not the application.



### 1.5.5 X\_Task\_Timeout

This argument passes a time value to the X-side, which is the maximum time allowed for the application to run before it is aborted and an error status returned. A zero time value disables the X-side timeout function. Valid time values are in the range from 0 to 24 hours.

### 1.5.6 Req\_Timeout

This argument contains the maximum time allowed for the CL request to wait for an application completion return before an error status is returned. This timer function is performed on the LCN-side. If a timeout occurs, an X-side application abort is requested. A zero value disables the LCN timeout function and is equivalent to an infinite timeout. Valid time values are in the range from 0 to 24 hours.

### 1.5.7 Example

```

PACKAGE
--
BLOCK EXECTASK ( GENERIC; AT BACKGRND )
--
%INCLUDE_SET AMCL06
--
LOCAL STRTAPPL           : STRING
LOCAL RET_STAT, DET_STAT : NUMBER
LOCAL X_TMOUT, REQTMOUT : TIME
--
CALL BKG_CHANGE_PRIORITY( LOW )

SET STRTAPPL = "APP1"
SET X_TMOUT  = 1 MINS
SET REQTMOUT = 2 MINS
--
CALL AMCL06$EXECUTE_TASK_WITH_WAIT (RET_STAT,
&                                  DET_STAT,
&                                  STRTAPPL,
&                                  X_TMOUT,
&                                  REQTMOUT)
--
SEND: "EXIT CODE = ", DET_STAT
--
IF (RET_STAT <> 0.0) THEN
& SEND: "ERROR: STATUS = ", RET_STAT
--
END EXECTASK
--
END PACKAGE

```

## I.6 AMCL06\$Get\_Queue\_Info

### I.6.1 Purpose

This subroutine is used to obtain general information about the AXM CL queue. It can be used to determine how many AMCL06 CL subroutines are making requests to the X-side, and whether or not space is available for another X-side request. Note: there are 10 slots available in the queue (not to be confused with the configurable limit of 10 background CL programs that can run at one time). The subroutine can be called from foreground or background CL.

### I.6.2 Syntax

The following is the syntax of the **AMCL06\$Get\_Queue\_Info** subroutine:

```
SUBROUTINE AMCL06$Get_Queue_Info
  (Ret_Status      : OUT NUMBER;      -- Return status of the call
   Num_In_Use      : OUT NUMBER;      -- Total number of slots in use
   Num_Avail       : OUT NUMBER;      -- Number of slots available
   Num_Frg_In_Use  : OUT NUMBER;      -- Number queue slots used by foreground CL
   Num_Bkg_In_Use  : OUT NUMBER)      -- Number queue slots used by background CL
```

### I.6.3 Ret\_Status

This argument is the return status of the subroutine call. Values are:

Value	Ret_Status
0	No errors
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)
91	Initialization unsuccessful—unable to acquire internal resources
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)
93	Initialization unsuccessful—unable to acquire internal resources
999	Node not an AXM

### I.6.4 Num\_In\_Use

Number of queue slots in use. This will be the sum of Num\_Frg\_in\_Use (zero at this time) and Num\_Bkg\_in\_Use.

### **I.6.5 Num\_Avail**

Number of queue slots available. This will be 10 minus Num\_in\_Use.

### **I.6.6 Num\_Frg\_In\_Use**

Number of slots in use by foreground CL. This is reserved for future use and will be zero at this time.

### **I.6.7 Num\_Bkg\_In\_Use**

Number of slots in use by background CL. Note: only the following subroutines use queue slots: **AMCL06\$Execute\_Task\_With\_Wait**, **AMCL06\$Initiate\_Task**, **AMCL06\$Activate\_Task**, **AMCL06\$Terminate\_Task**, **AMCL06\$Get\_Hiber\_Task\_Status**

## I.6.8 Example

```

PACKAGE
--
CUSTOM (ACCESS OPERATOR)
--
--
PARAMETER  RET_STAT           : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  NUM_IN_U           : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  NUM_AVAL           : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  NUM_FRG            : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  NUM_BKG            : NUMBER
NOT BLD_VISIBLE
--
--
END CUSTOM
--
--
BLOCK GETQINFO (GENERIC; AT GENERAL)
--
--
%INCLUDE_SET AMCL06
--
--
CALL AMCL06$GET_QUEUE_INFO (RET_STAT,
&                           NUM_IN_U,
&                           NUM_AVAL,
&                           NUM_FRG,
&                           NUM_BKG)
--
IF (RET_STAT <> 0.0) THEN
&  SEND: "ERROR: STATUS = ", RET_STAT
--
END GETQINFO
--
END PACKAGE

```

## I.7 AMCL06\$Get\_Queue\_Slot\_Info

### I.7.1 Purpose

This subroutine is used to obtain specific information about a single entry in the A<sup>X</sup>M CL Queue. It can be called from foreground or background CL.

It can be used as part of a loop using the queue slot numbers to obtain detailed information in a snapshot mode to display on a schematic. Note: there are 10 slots in the A<sup>X</sup>M CL queue. The snapshot could contain information on all active AMCL06 CL subroutine requests, timeouts and priorities associated with the requests, and all currently running HP-UX tasks initiated by those subroutines.

### I.7.2 Syntax

The following is the syntax of the **AMCL06\$Get\_Queue\_Info** subroutine:

```
SUBROUTINE AMCL06$Get_Queue_Slot_Info
  (Ret_Status      : OUT NUMBER;-- Return status of the call
   Slot_Status     : OUT NUMBER; -- Queue slot status
   Cmd_Line       : OUT STRING;-- X-side application command line
   Requestor_Entity : OUT STRING;-- Point requesting X-side application
   Requestor_CL    : OUT STRING;-- CL name requesting X-side application
   Time_Initiated  : OUT TIME;  -- Time of CL request
   Time_Serviced_Delta : OUT NUMBER;-- Difference between Time_Initiated and
                                   -- time serviced
   Time_Left_To_Timeout : OUT TIME;-- Time left until request times out
   Requestor_Priority : OUT NUMBER;-- Indicates foreground or background
   X_PID           : OUT NUMBER;-- X-side process identifier
   Mesg_Id         : OUT NUMBER; -- Message identifier
   CL_Priority      : OUT NUMBER;-- CL program priority at time of request
   Slot_Num        : IN  NUMBER)-- Slot number whose status is requested
```

### I.7.3 Ret\_Status

This argument is the return status of the subroutine call. Values are:

Value	Ret_Status
0	No errors
1	Invalid slot number
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)
91	Initialization unsuccessful—unable to acquire internal resources
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)
93	Initialization unsuccessful—unable to acquire internal resources
999	Node not an A <sup>X</sup> M

## I.7.4 Slot\_Status

This argument returns the status of the slot identified by Slot\_Num. This is the status as viewed from the AM side, with X-side status information based on messages received from the X-side. Values are:

Value	Meaning
0	Slot available
1	<i>Execute_Task_With_Wait</i> request queued in the AM (not transferred to the X-side yet)
2	<i>Execute_Task_With_Wait</i> request queued in the X-side (the X-side has accepted the request)
3	Application running in the X-side by an <i>Execute_Task_With_Wait</i> request (the X-side has notified the AM that the application is running)
4	Application terminated, no errors reported by X-side
5	Application terminated with error(s)
6	Application hibernating
11	<i>Initiate_Task</i> request queued in the AM (not transferred to the X-side yet)
12	<i>Initiate_Task</i> request queued in the X-side (the X-side has accepted the request)
13	Application running in the X-side by an <i>Initiate_Task</i> request (the X-side has notified the AM that the application is running)
21	<i>Activate_Task</i> request queued in the AM (not transferred to the X-side yet)
22	<i>Activate_Task</i> request queued in the X-side (the X-side has accepted the request)
23	Application running in the X-side by an <i>Activate_Task</i> request (the X-side has notified the AM that the application is running)
31	<i>Terminate_Task</i> request queued in the AM (not transferred to the X-side yet)
32	<i>Terminate_Task</i> request queued in the X-side (the X-side has accepted the request)
33	Application running in the X-side by an <i>Terminate_Task</i> request (the X-side has notified the AM that the application is running)
41	<i>Get_Hiber_Task_Status</i> request queued in the AM (not transferred to the X-side yet)
42	<i>Get_Hiber_Task_Status</i> request queued in the X-side (the X-side has accepted the request)
43	<i>Get_Hiber_Task_Status</i> complete

### I.7.5 Cmd\_Line

HP-UX command line, including arguments, that was included in the **AMCL06\$Execute\_Task\_With\_Wait** or **AMCL06\$Initiate\_Task** call. It will be blank (null string) if queue slot is not in use or if the CL subroutine does not use a command line.

### I.7.6 Requestor\_Entity

Point that requested the X-side application. Will be null if the queue slot is not in use.

### I.7.7 Requestor\_CL

Name of the CL block that requested the X-side application. Will be blank if the queue slot is not in use.

### I.7.8 Time\_Initiated

The time the CL issued the request.

### I.7.9 Time\_Serviced\_Delta

The time difference in seconds between the CL request and the WSI Communication Interface service time.

### I.7.10 Time\_Left\_To\_Timeout

Time remaining before expiration of the request timeout (see **Req\_Timeout** in the **AMCL06\$Execute\_Task\_With\_Wait**, **AMCL06\$Initiate\_Task**, **AMCL06\$Activate\_Task**, **AMCL06\$Terminate\_Task**, and **AMCL06\$Get\_Hiber\_Task\_Status** calls).

### I.7.11 Requestor\_Priority

Indicates whether the requestor was foreground or background CL. Values are as follows:

Value	Meaning
0.0	Queue slot not in use
1.0	Foreground (reserved for future use)
2.0	Background



### I.7.12 X\_PID

X-side process identifier returned after the X-side application started.

### I.7.13 Mesg\_Id

Internal message identifier (assigned by CL request task).

### I.7.14 CL\_Priority

Returns a value indicating CL program priority at time of request: 25.0 = HIGH, 24.0 = MEDIUM, or 23.0 = LOW.

Background CL runs at one of three priorities—HIGH, MEDIUM, or LOW. The default is HIGH but once the CL starts executing, the priority can be changed with the built-in subroutine BKG\_Change\_Priority (see the CL/AM Reference manual). For X-side applications initiated by CL, the CL background priority translates into a “nice” value priority on the X-side according to the following table:

Background CL Priority	Nice Value Priority
HIGH	15
MEDIUM	17
LOW	18

Nice values are priorities at which HP-UX programs run. Values range from 0 to 39. The standard nice value is 20. Lower values indicate higher priorities, and higher values indicate lower priorities. (The higher the nice value, the “nicer” you are being to other applications.)

On the X-side, you can view the nice values of the running processes with the process status command:

```
ps -elf
```

### I.7.15 Slot\_Num

Queue slot number. Range 1-10.

## I.7.16 Example

```

PACKAGE
--
CUSTOM (ACCESS OPERATOR)
--
--
PARAMETER  RET_STAT          : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  SLOT_STA         : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  CMD_LINE         : STRING
NOT BLD_VISIBLE
--
PARAMETER  REQS_ENT         : STRING
NOT BLD_VISIBLE
--
PARAMETER  REQS_CL          : STRING
NOT BLD_VISIBLE
--
PARAMETER  TIME_INI         : TIME
NOT BLD_VISIBLE
--
PARAMETER  TIME_S_D         : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  TIME_LFT         : TIME
NOT BLD_VISIBLE
--
PARAMETER  REQS_PRI         : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  X_PID            : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  MESG_ID          : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  CL_PRIOR         : NUMBER
NOT BLD_VISIBLE
--
PARAMETER  SLOT_NUM         : NUMBER
NOT BLD_VISIBLE
--
--
END CUSTOM
--

```

Example, **continued**

```

BLOCK GETSINFO (GENERIC; AT GENERAL)
--
--
%INCLUDE_SET AMCL06
--
--
CALL AMCL06$GET_QUEUE_SLOT_INFO (RET_STAT,
&                                SLOT_STA,
&                                CMD_LINE,
&                                REQS_ENT,
&                                REQS_CL,
&                                TIME_INI,
&                                TIME_S_D,
&                                TIME_LFT,
&                                REQS_PRI,
&                                X_PID,
&                                MESG_ID,
&                                CL_PRIOR,
&                                SLOT_NUM)
--
IF (RET_STAT <> 0.0) THEN
&  SEND: "ERROR: STATUS = ", RET_STAT
--
END GETSINFO
--
END PACKAGE

```

## I.8 AMCL06\$Store\_XAccess

### I.8.1 Purpose

This subroutine is used to change the value of the global X-access switch. The global X-access switch controls the ability of X-side applications to write LCN data. The subroutine can only be called from background CL.

### I.8.2 Syntax

The following is the syntax of the **AMCL06\$Store\_XAccess** subroutine:

```
SUBROUTINE AMCL06$Store_XAccess  
  (Ret_Status      : OUT CLERRSTS;-- Return status of the call  
   New_State       : IN $XACCESS)-- $XACCESS enumeration state value to store
```

### I.8.3 Ret\_Status

This argument is of type CLERRSTS enumeration. Values of the CLERRSTS enumeration are:

Ordinal Value	Value	Meaning
0	NoError	No error detected
1	Error3	Should not happen—Notify Honeywell TAC
2	LimViol	Limit Violation (minor error)
3	Rights	Access rights violation (minor error)
4	CommErr	Communication Error (minor error)
5	Error2	Should not happen—Notify Honeywell TAC
6	BadValst	Bad Value Store (CL Failure condition)
7	ComAbort	Communication Error abort (CL Failure condition)
8	Abort	Abort statement executed (CL Failure condition)
9	Error1	Should not happen—Notify Honeywell TAC
10	BranchV	Backward Branch Violation (CL Error condition)
11	ArithErr	Arithmetic Error (CL Error condition)
12	ArrayLim	Array Limit Violation (CL Error condition)
13	Range	Range Limit Violation (CL Error condition)
14	ProgErr	Programming Error (CL Error condition)
15	KeyLevel	Key Level Restriction (CL Error condition)
16	CnfErr	Configuration Error (CL Error condition)

#### ATTENTION

The CnfErr status value will be returned if you execute a CL block that uses the **AMCL06\$Store\_XAccess** call on an AM instead of an A<sup>X</sup>M.

## I.8.4 New\_State

This argument is of type \$XACCESS enumeration. Values are:

Ordinal Value	Value	Meaning
0	READONLY	The X-side can read but cannot write LCN data (default)
1	RW_LCN_I	The X-side can read LCN data, but can write LCN data only from applications that are initiated by CL on the LCN-side
2	READWRIT	The X-side can read and write LCN data from CL-initiated and non-CL-initiated applications

Note: If the value READWRIT is specified and the optional external load module XACCES is not loaded, the call will fail and a Key Level error status will be returned.

## I.8.5 Example

```

PACKAGE
--
--
CUSTOM (ACCESS OPERATOR)
--
--
PARAMETER RET_STAT           : CLERRSTS
--
PARAMETER NEWSTATE           : $XACCESS
--
--
END CUSTOM
--
--
BLOCK XACCESS (GENERIC; AT BACKGRND)
--
%INCLUDE_SET AMCL06
--
CALL  AMCL06$STORE_XACCESS (RET_STAT,
&                                NEWSTATE )
--
IF (RET_STAT <> NoError) THEN
& SEND:  "ERROR: STATUS = ", ORD RET_STAT
--
END XACCESS
--
END PACKAGE

```

## I.9 AMCL06\$Initiate\_Task

### I.9.1 Purpose

The background CL subroutine **AMCL06\$Initiate\_Task** is used to initiate a new instance of an OpenDDA application. After the call is made, the background CL suspends execution and enters a wait condition until the OpenDDA application has initiated and entered hibernation. The user provides an application name, and once the application has entered hibernation, a unique application id is returned to the CL block. The application id returned from the subroutine is needed to activate (**AMCL06\$Activate\_Task**) and terminate (**AMCL06\$Terminate\_Task**) the application.

Timeouts are included to allow the CL block to provide a maximum time the CL subroutine call should wait for completion of the initiate request and a maximum clock time allowed for the OpenDDA application to initiate and enter hibernation. If either timeout occurs, the application will be aborted and an error will be returned to the CL block.

Only a single instance of an OpenDDA application will be permitted to be initiated from a single point. However, two different points can execute the same OpenDDA application, although this will result in two separate independent executing programs that are not linked in any way. Additionally, a single point can execute many different OpenDDA applications. If a CL block attempts to initiate a second instance of an application already associated with the point, an error will be returned along with the application identifier of the current instance of the application initiated by the point. This allows the CL block to determine if the existing hibernating OpenDDA application should be activated, or the application should be terminated and a new instance of the application initiated.

This CL subroutine depends on the cooperation of the OpenDDA application to properly enter hibernation. If the requested application terminates instead of entering hibernation, an error will be returned along with a null application identifier.

A hibernating OpenDDA application initiated from CL will retain the security access of "CL Initiated" during the life of the execution of the application.

For OpenDDA applications initiated by CL, the CL background priority (HIGH, MEDIUM, LOW) translates into a "nice" value priority of the application on the X-side (15, 17, 18 respectively).

## I.9.2 Syntax

Syntax of the **AMCL06\$Initiate\_Task** subroutine:

```
SUBROUTINE AMCL06$Initiate_Task
  (Ret_Status      : OUT NUMBER;  -- Return status of the call
   Det_Status      : OUT NUMBER;  -- Detailed return status
   Appl_ID         : OUT STRING;  -- Application identifier
   Cmd_Line        : IN  STRING;  -- X-side application command line
   X_Task_Timeout  : IN  TIME;    -- X-side timeout value
   Req_Timeout     : IN  TIME)    -- LCN-side timeout value
```

## I.9.3 Ret\_Status and Det\_Status

Ret\_Status is the return status of the subroutine call. Det\_Status (Detail Status) may contain additional information. Values are:

Value	Ret_Status	Det_Status
0	Application successfully initiated and entered hibernation	0
1	Subroutine argument error	1 = Invalid X_Task_Timeout 2 = Invalid Req_Timeout 3 = Invalid Cmd_Line
2	CL timeout while waiting for X-side application to complete—application aborted	0
3	Unable to communicate with X-side	0 = No detail status 4 = Unexp'd connect from X-side 5 = Unexp'd disconnect from X-side
4	Error getting memory in AM	0
5	LCN/X-side connection down	0
6	Application exited instead of hibernating	Application exit code (defined in the application program)
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
91	Initialization unsuccessful—unable to acquire internal resources	0
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
93	Initialization unsuccessful—unable to acquire internal resources	0



**Ret\_Status and Det\_Status, continued**

Continuation of the table of Ret\_Status and Det\_Status values:

Value	Ret_Status	Det_Status
500	Internal error	0
501	Internal produce error	0
502	Internal consume error	0
503	Internal priority error	0
504	Internal message/queue mismatch	0
505	Internal X-side message/queue mismatch	0
999	Node not an A <sup>X</sup> M	0
1001	Application was killed by signal	Signal number
1002	Application name invalid or application not found	HP-UX errno (or 0)
1003	Application timed out (X-side timeout)—application aborted	0
1004	Miscellaneous internal error	0
1005	An error occurred while changing priority	HP-UX errno
1006	An error occurred while creating the application's process	HP-UX errno
1007	The application did not have any execute permission set	0
1008	An error occurred while executing the application	HP-UX errno
1009	A miscellaneous HP-UX error occurred in the CDS/CL server	HP-UX errno
1010	The application command line contained an absolute path, which is not allowed	0
1011	An error occurred while setting up the application's environment	0
1020	Application already initiated by this point. The application ID of the current instance of the application is returned in Appl_ID	0
1021	Unable to initiate the application because the maximum number of hibernating applications was reached	0

### I.9.4 Appl\_ID

Unique identifier of the X-side application assigned when an application has successfully initiated and entered hibernation.

### I.9.5 Cmd\_Line

This argument is passed to the X-side where it is interpreted as an HP-UX command line. It is a string of up to 78 characters containing the application name, not including the pathname, and any application arguments. The application is responsible for interpreting the command line arguments. The application must reside in the directory `/users/axm` on the HP-UX disk (see *Application Module<sup>X</sup> User Guide*, subsection 2.3 “LCN Security,” topic “Directory for CL-initiated applications”).

For other command line restrictions, refer to the topic **Cmd\_Line restrictions** under the **AMCL06\$Execute\_Task\_With\_Wait** subroutine call.

### I.9.6 X\_Task\_Timeout

Maximum clock time allowed for the OpenDDA application to initiate and enter hibernation. This timeout begins when the OpenDDA application has been invoked and ends when the application has entered hibernation. If a timeout occurs, the OpenDDA application is aborted. A zero time value disables the X-side timeout function. Valid time values are in the range from 0 to 24 hours

### I.9.7 Req\_Timeout

Maximum time allowed for the CL block to wait for the application to complete the request. This begins when the request is transferred to the X-side, includes the OpenDDA application initiating and entering hibernation, and ends when the CL block receives the results. If a timeout occurs, the OpenDDA application is aborted. A zero time disables the timeout function and is equivalent to an infinite timeout. Valid time values are in the range from 0 to 24 hours.

## I.10 AMCL06 Activate\_Task

### I.10.1 Purpose

The background CL subroutine **AMCL06\$Activate\_Task** is used to send an activate event to a CL-initiated hibernating OpenDDA application. After the call is made, the background CL suspends execution and enters a wait condition until the OpenDDA application receives and processes the event, and returns to hibernation. The user provides an application id (obtained from **AMCL06\$Initiate\_Task**) and an event string which is passed to the OpenDDA application.

Timeouts are included to allow the CL block to provide a maximum time the CL subroutine call should wait for completion of the activate request and a maximum clock time allowed for the OpenDDA application to receive and process the event and enter hibernation. If either timeout occurs, the application will be aborted and an error will be returned to the CL block.

A CL block can only activate a hibernating application if the application was initiated by the point on which the CL block is attached. An application can be initiated from one CL block, activated from a different CL block, and terminated from another CL block, all attached to the same point. Additionally, one CL block may initiate, activate, and terminate one or many OpenDDA applications.

This CL subroutine depends on the cooperation of the OpenDDA application to properly receive the activate event and enter hibernation. If the requested application terminates instead of entering hibernation, an error will be returned.

As opposed to OpenDDA applications initiated by CL, the CL background priority of an activate task request has no affect on the "nice" value priority of the application on the X-side.

### I.10.2 Syntax

Syntax of the **AMCL06\$Activate\_Task** subroutine:

```
SUBROUTINE AMCL06$Activate_Task
  (Ret_Status      : OUT NUMBER;      -- Return status of the call
   Det_Status      : OUT NUMBER;      -- Detailed return status
   Appl_ID         : IN  STRING;       -- Application identifier
   Event_String    : IN  STRING;       -- String passed to application
   X_Task_Timeout  : IN  TIME;         -- X-side timeout value
   Req_Timeout     : IN  TIME;         -- LCN-side timeout value
```

### I.10.3 Ret\_Status and Det\_Status

Ret\_Status is the return status of the subroutine call. Det\_Status (Detail Status) may contain additional information. Values are:

Value	Ret_Status	Det_Status
0	Application successfully activated and returned to hibernation	0
1	Subroutine argument error	1 = Invalid X_Task_Timeout 2 = Invalid Req_Timeout 6 = Invalid Appl_ID
2	CL timeout while waiting for X-side application to complete—application aborted	0
3	Unable to communicate with X-side	0 = No detail status 4 = Unexp'd connect from X-side 5 = Unexp'd disconnect from X-side
4	Error getting memory in AM	0
5	LCN/X-side connection down	0
6	Application exited instead of hibernating	Application exit code (defined in the application program)
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
91	Initialization unsuccessful—unable to acquire internal resources	0
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
93	Initialization unsuccessful—unable to acquire internal resources	0
500	Internal error	0
501	Internal produce error	0
502	Internal consume error	0
503	Internal priority error	0
504	Internal message/queue mismatch	0
505	Internal X-side message/queue mismatch	0
999	Node not an A <sup>X</sup> M	0

**Ret\_Status and Det\_Status, continued**

Value	Ret_Status	Det_Status
1001	Application was killed by signal	Signal number
1003	Application timed out (X-side timeout)—application aborted	0
1004	Miscellaneous internal error	0
1009	A miscellaneous HP-UX error occurred in the CDS/CL server	HP-UX errno
1022	Application ID does not exist	0
1023	Action invalid from this point	0

**I.10.4 Appl\_ID**

Identifier assigned to the X side application when initiated. This is obtained from **AMCL06\$Initiate\_Task** and may not include preceding blank characters.

**I.10.5 Event\_String**

A user defined event string of 0 (zero) to 78 characters which is passed to the OpenDDA application.

**I.10.6 X\_Task\_Timeout**

Maximum clock time allowed for the OpenDDA application to receive and process the event and enter hibernation. This timeout begins when the event is queued on the X-side for the OpenDDA application and ends when the application has re-entered hibernation after processing the event. If a timeout occurs, the OpenDDA application is aborted. A zero time disables the X-side timeout function and is equivalent to an infinite timeout. Valid times range from 0 to 24 hours.

**I.10.7 Req\_Timeout**

Maximum time allowed for the CL block to wait for the application to complete the request. This timeout begins when the event request is transferred to the X-side, includes the OpenDDA application receiving and processing the event and entering hibernation, and ends when the CL block receives the results. If a timeout occurs, the OpenDDA application is aborted. A zero time disables the timeout function and is equivalent to an infinite timeout. Valid times range from 0 to 24 hours.

## I.11 AMCL06\$Terminate\_Task

### I.11.1 Purpose

The background CL subroutine **AMCL06\$Terminate\_Task** is used to send a terminate event to a CL-initiated hibernating OpenDDA application. This provides a graceful shutdown of a hibernating OpenDDA application. After the call is made, the background CL suspends execution and enters a wait condition until the OpenDDA application has received and processed the event, and terminated its execution. The user provides an application id (obtained from **AMCL06\$Initiate\_Task**) and an event string which is passed to the OpenDDA application.

Timeouts are included to allow the CL block to provide the maximum time the CL subroutine call should wait for completion of the terminate request and a maximum clock time allowed for the OpenDDA application to process the event and terminate its execution. If either timeout occurs, the X-side application will be aborted and a status will be returned to the CL block.

A CL block can only terminate a hibernating application if it was initiated by the point on which the CL block is attached. An application can be initiated from one CL block, activated from a different CL block and terminated from another CL block all attached to the same point. Additionally, one CL block can initiate, activate, and terminate one or many OpenDDA applications.

This CL subroutine depends on the cooperation of the OpenDDA application to properly receive the terminate event and exit its execution. If the requested application hibernates instead of exiting, an error will be returned.

As opposed to OpenDDA applications initiated by CL, the CL background priority of an terminate task request has no affect on the "nice" value priority of the application on the X-side

### I.11.2 Syntax

Syntax of the **AMCL06\$Terminate\_Task** subroutine:

```
SUBROUTINE AMCL06$Terminate_Task
  (Ret_Status      : OUT NUMBER;      -- Return status of the call
   Det_Status      : OUT NUMBER;      -- Detailed return status
   Appl_ID         : IN  STRING;      -- Application identifier
   Event_String    : IN  STRING;      -- String passed to application
   X_Task_Timeout  : IN  TIME;        -- X-side timeout value
   Req_Timeout     : IN  TIME) -- LCN-side timeout value
```

### I.11.3 Ret\_Status and Det\_Status

Ret\_Status is the return status of the subroutine call. Det\_Status (Detail Status) may contain additional information. Values are:

Value	Ret_Status	Det_Status
0	Application successfully terminated	Application exit code (defined in the application program)
1	Subroutine argument error	1 = Invalid X_Task_Timeout 2 = Invalid Req_Timeout 6 = Invalid Appl_ID
2	CL timeout while waiting for X-side application to complete—application aborted	0
3	Unable to communicate with X-side	0 = No detail status 4 = Unexp'd connect from X-side 5 = Unexp'd disconnect from X-side
4	Error getting memory in AM	0
5	LCN/X-side connection down	0
7	Application entered hibernation instead of terminating	0
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
91	Initialization unsuccessful—unable to acquire internal resources	0
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
93	Initialization unsuccessful—unable to acquire internal resources	0
500	Internal error	0
501	Internal produce error	0
502	Internal consume error	0
503	Internal priority error	0
504	Internal message/queue mismatch	0
505	Internal X-side message/queue mismatch	0
999	Node not an A <sup>X</sup> M	0

Ret\_Status and Det\_Status, **continued**

Value	Ret_Status	Det_Status
1001	Application was killed by signal	Signal number
1003	Application timed out (X-side timeout)—application aborted	
1004	Miscellaneous internal error	0
1009	A miscellaneous HP-UX error occurred in the CDS/CL server	HP-UX errno
1022	Application ID does not exist	0
1023	Action invalid from this point	0

**I.11.4 Appl\_ID**

Identifier assigned to the X-side application when initiated. This is obtained from **AMCL06\$Initiate\_Task** and may not include preceding blank characters.

**I.11.5 Event\_String**

A user-defined event string of 0 (zero) to 78 characters which is passed to the OpenDDA application.

**I.11.6 X\_Task\_Timeout**

Maximum clock time allowed for the OpenDDA application to receive and process the event and exit its execution. This timeout begins when the event is queued on the X-side for the OpenDDA application and ends when the application has terminated. If a timeout occurs, the OpenDDA application is aborted. A zero time disables the X-side timeout function and is equivalent to an infinite timeout. Valid times range from 0 to 24 hours.

**I.11.7 Req\_Timeout**

Maximum time allowed for the CL block to wait for the application to complete the request. This timeout begins when the event request is transferred to the X-side, includes the OpenDDA application receiving and processing the event and terminating its execution, and ends when the CL block receives the results. If a timeout occurs, the OpenDDA application is aborted. A zero time disables the timeout function and is equivalent to an infinite timeout. Valid times range from 0 to 24 hours



## I.12 AMCL06Get\_Hiber\_Task\_Status

### I.12.1 Purpose

The background CL subroutine **AMCL06\$Get\_Hiber\_Task\_Status** is used to obtain specific information about a single entry in the 50-slot X-side hibernating queue (not to be confused with the 10-slot A<sup>X</sup>M CL queue). It can be used to obtain information about a known application or to get information about a current hibernating task in the hibernating queue to display on a schematic.

To obtain information about a known application, this subroutine accepts as input an application id (obtained from **AMCL06\$Initiate\_Task**) of an OpenDDA hibernating application. It will return information about the hibernating application, including the associated index number in the hibernating queue.

To get information about a current hibernating task in the hibernating queue to display on a schematic, this subroutine accepts as input an index identifier. It will return information about the hibernating application for the given index in the hibernating queue.

### I.12.2 Syntax

```
SUBROUTINE AMCL06$Get_Hiber_Task_Status
(Ret_Status      : OUT NUMBER;      -- Return status of the call
Det_Status       : OUT NUMBER;      -- Detailed return status
Task_Status      : OUT NUMBER;      -- hibernating or running
Cmd_Line         : OUT STRING;      -- X-side application command line
Point_Name       : OUT STRING;      -- Point initiating X-side application
Block_Name       : OUT STRING;      -- CL name initiating X-side
application
Time_Initiated   : OUT TIME;        -- HP-UX time application initiated
Time_Activated   : OUT TIME;        -- HP-UX time application last
activated
Appl_Priority    : OUT NUMBER;      -- current HP-UX priority
X_PID            : OUT NUMBER;      -- X-side process identifier
Appl_ID          : IN OUT STRING;   -- X-side application identifier
Index            : IN OUT NUMBER;   -- Index into X-side hibernating queue
Req_Timeout      : IN TIME)-- LCN timeout value
```

### I.12.3 Ret\_Status and Det\_Status

Ret\_Status is the return status of the subroutine call. Det\_Status (Detail Status) may contain additional information. Values are:

Value	Ret_Status	Det_Status
0	Request successful	0
1	Subroutine argument error	2 = Invalid Req_Timeout 7 = Invalid index and Appl_ID 8 = Invalid index
2	CL timeout while waiting for request to complete	0
3	Unable to communicate with X-side	0 = No detail status 4 = Unexp'd connect from X-side 5 = Unexp'd disconnect from X-side
4	Error getting memory in AM	0
5	LCN/X-side connection down	0
90	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
91	Initialization unsuccessful—unable to acquire internal resources	0
92	Initialization in progress—attempting connection to X-side (should only occur during node startup)	0
93	Initialization unsuccessful—unable to acquire internal resources	0
500	Internal error	0
501	Internal produce error	0
502	Internal consume error	0
503	Internal priority error	0
504	Internal message/queue mismatch	0
505	Internal X-side message/queue mismatch	0
999	Node not an A <sup>X</sup> M	0
1004	Miscellaneous internal error	0
1009	A miscellaneous HP-UX error occurred in the CDS/CL server	HP-UX errno

**Ret\_Status and Det\_Status, continued**

<b>Value</b>	<b>Ret_Status</b>	<b>Det_Status</b>
1024	Invalid index, invalid application id, or invalid combination of both	0
1025	No hibernating/running applications	0
1026	Error while getting the application's HP-UX priority	0

**I.12.4 Task\_Status**

Current execution state of the application:

0 = null (no application)

1 = running

2 = hibernating

**I.12.5 Cmd\_Line**

Command line used to initiate the application.

**I.12.6 Point Name**

Command line used to initiate the application.

**I.12.7 Block\_Name**

Block responsible for initiating the application.

**I.12.8 Time\_Initiated**

HP-UX time, in LCN format, when application was initiated.

**I.12.9 Time\_Activated**

HP-UX time, in LCN format, when application was last activated.

**I.12.10 Appl\_Priority**

Current HP-UX priority of the running application.

**I.12.11 X-PID**

X-side process id of the OpenDDA application.

**I.12.12 Appl\_ID**

X-side application identifier. Input used to return information from the hibernating queue about a known application. Output when Index is the only valid input. If Appl\_ID and Index are both provided as input and they do not match, or if they are both null (0 and blank string), an error will be returned. A blank string is a valid input for an Appl\_ID if Index is in the valid index range from 0 to 50.

**I.12.13 Index**

Index into the X-side hibernating queue. Input used to return contents of a specific element of the hibernating queue. Output when Appl\_ID is the only input. If Appl\_ID and Index are both provided as input and they do not match, or if they are both null (0 and blank string), an error will be returned. Valid indexes range from 0 - 50, where zero indicates a null index.

**I.12.14 Req\_Timeout**

Maximum time allowed for the CL block to wait for the request to complete. If a timeout occurs, the request is aborted. A zero time disables the timeout function and is equivalent to an infinite timeout. Valid times range from 0 to 24 hours.

## I.13 STOPPING X-SIDE APPLICATIONS FROM THE AM-SIDE

### I.13.1 Overview

There are three techniques to stop X-side applications that were initiated from CL on the AM-side.

- Abort the CL
- Inactivate the point to which the CL is attached
- Inactivate the CL

These three techniques have different results.

#### ATTENTION

These techniques apply only to actively running applications. Applications in a hibernating state can only be reliably aborted with the `kill_appls` utility (see the next subsection).

### I.13.2 Abort the CL

If a CL program has initiated an X-side application and is in a wait state pending application termination or hibernation, you can abort the CL from the point's detail status. This will cause an immediate kill of the X-side application that was initiated by the CL.

### I.13.3 Inactivate the Point

If a CL program has initiated an X-side application and is in a wait state pending application termination or hibernation, you can inactivate the point to which the CL is attached from the point's detail status display on a US or UXS. This will abort the CL and will cause an immediate kill of the X-side application that was initiated by the CL.

### I.13.4 Inactivate the CL

As an alternative to inactivating the point, you can inactivate the CL from the CL block page of the detail status. If the point has more than one CL, this technique allows you to select the one you wish to inactivate. Inactivating the CL, however, causes a slightly different result. The CL will go into inactive-waiting state until the X-side application terminates or hibernates, and then will go inactive.

### I.13.5 Restarting the X-side Application

Switching the point or CL back to the active state will restart the CL and the associated X-side application(s), assuming the CL execution is scheduled (as opposed to manually initiated).

## I.14 Using the kill\_appls X-Side Tool

### I.14.1 Overview

The X-side tool `kill_appls` is a command line tool used to abort CL-initiated X-side applications. It is located in the HP-UX directory `/opt/TDC_Open/common/bin.` and can only be executed by a user while a member of the “axm” group. This tool can kill

- one or all CL-initiated X-side applications associated with a point,
- one or all instances of the same application name, or
- applications that are either running or hibernating.

### I.14.2 Syntax

There are two ways to invoke this tool. One way is to specify the application name, and the other way is to specify the associated point.

The syntax when specifying the application name is

```
kill_appls -A application_name {-i application_id | -p process_id | -a}
```

The syntax when specifying the point name is

```
kill_appls -P point_name {-i application_id | -p process_id | -a}
```

One of the following three switches is required:

- |                                       |   |
|---------------------------------------|---|
| <code>-i <i>application_id</i></code> | Application identifier assigned when the application is initiated by the <b>AMCL06\$Initiate_Task</b> call.   |
| <code>-p <i>process_id</i></code>     | HP-UX process id.   |
| <code>-a</code>                       | If used with the <i>application_name</i> argument, all executions of the application will be aborted. If used with the <i>point_name</i> argument, all X-side applications associated with the point will be aborted. |

NOTE: All options are case-sensitive.

#### ATTENTION

Values for the arguments specified above can be determined with the `display_appls` tool:

- *application\_name*
- *point\_name*
- *application\_id*
- *process\_id*

### I.14.3 Examples

The following command results in a request to the CDS/CL server to kill the instance of the application “app1” identified by the application identifier “Jfds35Dq”:

```
kill_appls -A app1 -i Jfds35Dq
```

The following command results in a request to the CDS/CL server to kill all applications associated with the point “F100”:

```
kill_appls -P F100 -a
```

### I.14.4 Error Messages

The following table lists the possible `kill_appls` error messages.

Error Message	Meaning
ERROR: Must be a member of the axm group to execute the kill_appls tool.	The kill_appls tool can only be executed by a user while a member of the “axm” group.
ERROR: Incorrect number of arguments specified.	The kill_appls tool was executed with either too many or too few arguments.
ERROR: Invalid switch specified. Expecting -i, -p, or -a.	The kill_appls tool was executed with an invalid switch.
ERROR: Invalid Application Name or Point Name switch specified. Expecting -A or -P.	The kill_appls tool was executed with an invalid switch.
ERROR: Invalid application identifier specified.	The kill_appls tool was executed with an application id that does not exist.
ERROR: Invalid PID specified.	The kill_appls tool was executed with a pid that does not exist.
ERROR: Invalid application name or point name specified.	The kill_appls tool was executed with an application name or point name that does not exist.
ERROR: Application name or point name not specified in the correct order.	The kill_appls tool was executed with an incorrect order of arguments. The tool expects the arguments and switches in a certain order.
ERROR: Could not connect to the CDS/CL Server.	The kill_appls tool could not communicate with the CDS/CL Server (cdsdaemon) to request the applications to be killed. Probable cause—cdsdaemon not running.
ERROR: Could not kill all applications attached to point <i>point_name</i> RetStatus = <i>nn</i> DetStatus = <i>nn</i>	An error occurred while trying to kill the applications requested. Probable cause is the applications are not executing from the specified point.

**Error Messages**, continued

Error Message	Meaning
ERROR: Could not kill all instances of application <i>application_name</i> RetStatus = <i>nn</i> DetStatus = <i>nn</i>	An error occurred while trying to kill the applications requested. Probable cause is the applications are not executing.
ERROR: Could not kill application attached to point <i>point_name</i> with application identifier <i>appl_id</i> RetStatus = <i>nn</i> DetStatus = <i>nn</i>	An error occurred while trying to kill the application requested. Probable cause is the application is not executing from the specified point with the specified application identifier.
ERROR: Could not kill instance of application <i>application_name</i> with application identifier <i>appl_id</i> RetStatus = <i>nn</i> DetStatus = <i>nn</i>	An error occurred while trying to kill the applications requested. Probable cause is the application is not executing with the specified application identifier.
ERROR: Could not kill application attached to point <i>point_name</i> with PID <i>pid</i> RetStatus = <i>nn</i> DetStatus = <i>nn</i>	An error occurred while trying to kill the applications requested. Probable cause is the application is not executing from the specified point with the specified pid.
ERROR: Could not kill instance of application <i>application_name</i> with PID <i>pid</i> RetStatus = <i>nn</i> DetStatus = <i>nn</i>	An error occurred while trying to kill the applications requested. Probable cause is the application is not executing with the specified pid.

**I.14.5 Return and Detail Status**

The following table lists the values of RetStatus and DetStatus that are returned with several of the error messages listed in the previous table.

Return Status	Detail Status	Explanation	Suggested Solution
3000	0	The CDS/CL Server is currently processing the maximum number of requests allowed.	Try again later. If the problem occurs often, reduce the load on the CDS/CL Server.
3001	0	Received an unexpected response from the CDS/CL Server.	Call Honeywell TAC.
3002	0	The CDS/CL Server received a request message of an unknown type.	Call Honeywell TAC.
3005	HP-UX errno	An error occurred while opening a socket to connect to the CDS/CL Server.	Verify the CDS/CL Server (cdsdaemon) is running and connected to the LCN. Verify the LCN side is running.
3006	HP-UX errno	An error occurred while sending information to the CDS/CL Server.	Call Honeywell TAC.
3007	HP-UX errno	An error occurred while receiving information from the CDS/CL Server.	Call Honeywell TAC.



**Return and Detail Status**, continued

<b>Return Status</b>	<b>Detail Status</b>	<b>Explanation</b>	<b>Suggested Solution</b>
3008	HP-UX errno	An error occurred while closing the socket to the CDS/CL Server.	Call Honeywell TAC.
3502	0	No applications found to be terminated.	The kill_appls tool could not find an application to abort. Execute the kill_appls tool with an application that is executing.

## I.15 The X-side display\_appls Tool

### I.15.1 Overview

The tool `display_appls` is an HP-UX command line tool used to obtain specific information about all CL-initiated X-side applications. It is located in the HP-UX directory `/opt/TDC_Open/common/bin`, and can be executed by any user who has access to the tool, regardless of the user's group id. By default, it will display a short version of output, although an option can be included to request more details. When `display_appls` is run, the following is sent to stdout

- application type—two headings are listed:
    - SYNCHRONOUS APPLICATIONS WITH TERMINATION  
 Lists tasks initiated by **AMCL06\$Execute\_Task\_With\_Wait**
    - SYNCHRONOUS APPLICATIONS WITH HIBERNATION  
 Lists tasks initiated with **AMCL06\$Initiate\_Task**
- For each task listed, the following information is displayed:
- command line
  - initiating point name
  - initiating CL block name
  - application status (hibernating or running)
  - HP-UX process id

When the long version is requested by using the `-l` option, the following is also displayed:

- application id (for hibernating applications only)
- time initiated
- time activated (for hibernating applications only)
- current HP-UX priority

### I.15.2 Syntax

Syntax for the `display_appls` tool is:

```
display_appls [-l]
```

### I.15.3 Examples

hp-ux> **display\_appls**

SYNCHRONOUS APPLICATIONS WITH HIBERNATION

CMDLINE: appl arg1 arg2 arg3  
 POINT: XX5ATM                      BLOCK: block1      STATUS: RUN                      PID: 3442

CMDLINE: test  
 POINT: F100\_long\_name BLOCK: blockA      STATUS: HIBER                      PID: 343

SYNCHRONOUS APPLICATIONS WITH TERMINATION

CMDLINE: adigen  
 POINT: AXM10                      BLOCK: blockK      STATUS: RUN                      PID: 6543

hp-ux> **display\_appls -l**

SYNCHRONOUS APPLICATIONS WITH HIBERNATION

CMDLINE: appl arg1 arg2 arg3  
 POINT: XX5ATM                      BLOCK: block1      STATUS: RUN                      PID: 3442  
 STARTTIME: 18:00:00 11/08/94                      PRIORITY: 165  
 ACTVTIME: 13:00:00 11/10/94                      APPLID: fd90Kf7s

CMDLINE: test  
 POINT: F100\_long\_name BLOCK: blockA      STATUS: HIBER                      PID: 343  
 STARTTIME: 12:30:30 12/10/94                      PRIORITY: 123  
 ACTVTIME: 00:00:15 12/11/94                      APPLID: Jfsd35Dj

SYNCHRONOUS APPLICATIONS WITH TERMINATION

CMDLINE: adigen  
 POINT: AXM10                      BLOCK: blockK      STATUS: RUN                      PID: 6543  
 STARTTIME: 23:34:53 11/12/94                      PRIORITY: 100

## I.15.4 Error Messages

The following table lists the possible `display_appls` error messages.

Error Message	Meaning
ERROR: Invalid switch specified.	The <code>display_appls</code> tool was executed with an invalid switch.
ERROR: Unable to obtain the application information from the CDS/CL Server RetStatus = <i>nn</i> DetStatus = <i>nn</i>	The <code>display_appls</code> tool did not receive the application information from the CDS/CL Server ( <code>cdsdaemon</code> ).
ERROR: Could not connect to the CDS/CL Server	The <code>display_appls</code> tool could not communicate with the CDS/CL Server ( <code>cdsdaemon</code> ) to request the application information. Probable cause— <code>cdsdaemon</code> not running.

## I.15.5 Return and Detail Status

The following table lists the values of RetStatus and DetStatus that are returned with certain of the error messages listed in the previous table.

Return Status	Detail Status	Explanation	Suggested Solution
3000	0	The CDS/CL Server is currently processing the maximum number of requests allowed.	Try again later. If the problem occurs often, reduce the load on the CDS/CL Server.
3001	0	Received an unexpected response from the CDS/CL Server.	Call Honeywell TAC.
3002	0	The CDS/CL Server received a request message of an unknown type.	Call Honeywell TAC.
3005	HP-UX errno	An error occurred while opening a socket to connect to the CDS/CL Server.	Verify the CDS/CL Server ( <code>cdsdaemon</code> ) is running and connected to the LCN. Verify the LCN side is running.
3006	HP-UX errno	An error occurred while sending information to the CDS/CL Server.	Call Honeywell TAC.
3007	HP-UX errno	An error occurred while receiving information from the CDS/CL Server.	Call Honeywell TAC.
3008	HP-UX errno	An error occurred while closing the socket to the CDS/CL Server.	Call Honeywell TAC.

---

# Index

---

Topic	Section Heading
ABORT (statement)	3.2
Defined	3.2.12
in Subroutines	3.2.12
from runtime error	2.3.4.3
Syntax	3.2.12.1
ABS function	2.4.3.2, 4.3.7.1
Absolute Value Function	2.4.3.2, 4.3.7.1
ACCESS (attribute of CDS parameter)	
Definition	4.4.5.3.1
Description	4.4.3.2, 4.4.5.3.3
Example	4.4.5.3.4
Syntax	4.4.5.3.2
access_lock_id	4.4.5.3.2
ACCESS (as privilege of CL Block)	
definition	4.2.5
syntax (access_key_id)	4.2.4
ADD (Operator)	2.5.2.4, Table 2-9
Aliasing	
Definition	2.3.4.5
Examples	2.2.4.6
Prohibited in AT clause	2.4.2.2
Allow_Bad (function)	4.3.7.5
AND (Logical Operator)	2.5.2.4
used to connect conditions	2.5.4.6
Application Module <sup>X</sup> CL Runtime Extensions	I
Arguments	2.2.7.9, 4.3.6, 4.3.7.1
Arithmetic and Logical Expressions	
Defined	2.5.2
Syntax	2.5.2.1
<i>see also</i> Arrays, Assignment, Equality, Expressions, Local variables, Logical, Number, Operators, Subroutines	
Arithmetic Functions	4.3.7.1
Arithmetic Operators	
Listed	2.2.10
Priorities	2.5.2.4, Table 2-9
Syntax	2.5.2.4
Arrays	
Access of MC & PM array parameters from AM	2.3.5, Tables 2-5 & 2-6
Defined	2.3.5
Components	2.3.7
Examples	2.3.5.1
Functions on	4.3.7.2
Index	2.4.2.2
in Parameter declarations	2.4.5.4
ASCII 2.2.1	
Assignment Operator	2.2.10, Table 2-4
Assignment Syntax	3.2.3.1
AT clause	2.4.2.2

---

# Index

---

Topic	Section Heading
ATAN Function	4.3.7.1
Attribute Statements	
ACCESS	4.4.5.3.1 - 4.4.5.3.4
Applicability of	Table 4-1
BLD_VISIBLE	4.4.5.4.1 - 4.4.5.4.4
Definition	4.4.5
Description	4.4.5.2
Syntax	4.4.5.1
AVG Function	4.3.7.1, 4.3.7.2
BACKGRND Insertion Point	2.2.7.4
Badval (built-in Predicate)	2.5.4.5, 2.5.4.6, 4.3.7.3
Bad Values Definition	2.3.1.1
BKG_Change_Priority (subroutine)	4.3.7.5
BKG_Delay (subroutine)	4.3.7.5
BKG_Switchover_Restart (logical function)	4.3.7.3
BLD_VISIBLE (attribute)	
Definition	4.4.5.4.1
Description	4.4.3.2, 4.4.5.4.3
Example	4.4.5.4.4
Syntax	4.4.5.4.2
Blocks	4.2
Block Identifiers, length of	2.2.5.1
Boolean	
Pascal Boolean type comparison with CL Logical	2.3.3.3, Figure 2-1
Bound Data Point	
Defined	2.3.4.1
Named in Block Heading	4.2.4
Parameters	2.3.4.1, 2.3.4.4, 2.4, 2.4.5
Box Data Point Identifiers	
Defined	2.3.4.7
Example	2.3.4.8
Branch (GOTO)	3.2.5
BTU Switch Package Example	4.9.4
Built-in Functions and Subroutines Definition	4.3.7
CALL (a subroutine)	
Defined	3.2.9
Description	3.2.9.2
Syntax	3.2.9.1
CDS	
Limitations	B.2.4
Sizes	B.2.5
<i>see also</i> Custom Data Segment	
CDS MOVE and Multiple Move Parameter Extension	F
CDS_Read (subroutine)	4.3.7.5
CDS_Write (subroutine)	4.3.7.5
Character	
ASCII	2.2.1
Defined	2.2.1
ISO 646 compatibility	2.2.5.2, Table 2-1
Set	2.2.5.2

---

# Index

---

Topic	Section Heading
CLASS (attribute)	
Definition	4.4.5.7.1
Description	4.4.3.2, 4.4.5.7.3
Example	4.4.5.7.4
Syntax	4.4.5.7.2
CL Block	
Definition	4.1, 4.2
Description	4.2.2
Syntax	4.2.1
CL Block-Heading	
Definition	4.2.3
Description	4.2.5
Examples	4.2.6
Syntax	4.2.4
CL Data Access Performance	B.4
CL Runtime Extensions	4.8
Comm_Error (logical function)	4.3.7.3
Comments	
Definition	2.2.6
Examples of (correct)	2.2.6.1
Examples of (incorrect)	2.2.6.2
Separator	2.2.10
Communications Error Handling	2.5.4.5, 4.3.7.3
Compiler Directives}	
Defined	3.3
Debug Switch	3.3.3, 3.3.3.1
Page Break	3.3.2
Syntax	3.3.1
<i>see also</i> %DEBUG, Embedded compiler directives, %INCLUDE_SET, %PAGE	
Compiler Restrictions	2.3.4.5, 2.4.4.2
Compile-Time Error	2.2.7.9, 2.4.2.2, 2.4.4.2
Composite data types definition	2.3
Compound Elements	
Definition	2.3.7
Examples	2.3.7.1
Conditional SET statement	3.2.3.2
Conditions	
Definition	2.5.4
Description	2.5.4.2
Syntax	2.5.4.1
Conflicts between Identifiers	2.2.7.9
Connecting Conditions with AND and OR	2.5.4.6
Consequent (of THEN, ELSE)	
Definition	3.2.6.2
Examples	3.2.6.3
Syntax	3.2.6.1
Continuation of Line	2.2.10
Continuous History Access, CL/AM Extension for	D
CONTIN_CTRL (as access_key_id (privilege) of CL Block)	4.2.5
Correct Examples of Comments	2.2.6.1

---

# Index

---

Topic	Section Heading
COS (cosine)Function	4.3.7.1
CRT_Only special destination	3.2.10.2
Custom Data Segment (CDS)	
Conflicts Between Identifiers	2.2.7.7
Definition	4.4
Description	4.4.2
Example	4.6.3
in Packages	2.4.5.6
Syntax	4.4
CUSTOM Data Segment Heading	
Definition	4.4.3
Description	4.4.3.2
Examples	4.4.3.3
Syntax	4.4.3.1
Custom Parameter	2.3.4
Data Access Error Codes	C.2.3
Data Point Identifiers	2.3.4.2, 2.3.5, 2.3.7
Defined	2.3.4.2
Length of	2.2.5.3
use of in Function Declarations	2.4.6.2
<i>see also</i> Arrays, Indirect reference, Parameter List	
Definition	
Data Points	
Parameters	2.4.5.3
accessing as EXTERNAL declaration	2.4.4
accessing when off-LCN	2.2.5.4
<i>see also</i> Bound Data Point, Box Data Point Identifiers,	
External Data Point, Process Modules	
Data Points Data Type	
Arrays of	2.3.5
Defined	2.3.4
Example	2.3.4
Data Types	
Conflicts between identifiers	2.2.7.7
Defined	2.3
Date_Time (function)	4.3.7.4
Debug Switch	
Defined	3.3.3
Example of	3.3.3.1
<i>see also</i> Compiler Directives, %DEBUG	
%DEBUG directive	3.3.3
Default Parameter Values	4.4.5.6.3, Table 4-2
<i>see also</i> Value (attribute)	
Define (function)	2.4.6
Delete_File (subroutine)	4.3.7.5
Direct Reference	2.3.4.2, 2.3.7
<i>see also</i> Data Point Identifiers	



---

# Index

---

Topic	Section Heading
Discrete Types Definition <i>see also</i> Continuous Values, Analog, Number, Logical, Enumeration, States	2.3.3
Divisor (Operator)	2.5.2.4
Dynamic Indirection	2.3.8
Embedded Compiler Directives	
Definition	3.3
Syntax	3.3.1
<i>see also</i> Compiler, %DEBUG, %PAGE	
END statement	3.2.13
Engineer(as access_lock_id)	4.4.5.3.3
Entity Builder (as access_lock_id)	4.4.5.3.3
Enum_Value_Store (subroutine)	4.3.7.5
Enumeration Definition	
Description	4.6.2
Examples	4.6.3
Syntax	4.6.1
Variables	2.4.2.2
Enumeration-Type	
Definition	4.6
Enumeration Types	2.3.3.2, 2.4.2.2
<i>see also</i> Discrete types, Logical types	
Enumeration states	2.2.7.9
in conflict with built-in Functions and Subroutines	4.3.7
Equal_Null_Point_id (logical function)	4.3.7
Equal_Point_id (logical function)	4.3.7
Equality Assignment Operator	2.2.10, Table 2-4
EU (statement)	
Definition	4.4.5.5.1
Description	4.4.5.5.3
Examples	4.4.5.5.4
Syntax	4.4.5.5.2
Event Initiated Reports from CL	3.2.10.4
Examples of CL/AM Structures	4.9
BTU Switch Package	4.9.4
Custom Data Segment	4.9.3
Linearization (Continuous)	4.9.2
PV Calculation	4.9.1
Exclusive OR	2.5.2.4, Table 2-9, Table 2-10
Executing X-Side Applications	I.3
Exists (logical function)	4.3.7.3
EXIT (statement)	
Definition	3.2.11
Description	3.2.11.2
Syntax	3.2.11.1
EXP (exponential) Function	4.3.7.1
Exponentiation (Operator)	2.5.2.4

---

# Index

---

Topic	Section Heading
Expressions and Conditions	2.5, 5.3.2.3
Arithmetic	2.5.2, 2.5.2.4
Logical	2.5.2, 2.5.2.4
Syntax	2.5.1, 2.5.2.1
Time Expressions	2.5.3, 2.5.3.1
<i>see also</i> Arithmetic expressions, Logical expressions, Time expressions	
Extensions to	
CL/AM for File I/O	C
CL/AM for Continuous History Access	D
CL/AM for Math Library	E
CDS Move & Multiple Move Parameter	F
Fast external CDS Fetch	G
CL Runtime Extensions	I
External CDS Fetch, AM Extension for Fast	G
External Data Points	
Definition 2.3.4.2, 2.4.4	
Description	2.3.4.2, 2.4.4.2
Examples	2.3.4.2, 2.4.4.3
Syntax	2.4.4.1
External Variables	2.4
Fast External CDS Fetch, AM Extension for	G
Fieldbus parameter restrictions	2.2.5.4
File I/O Extension to CL	C
File Manager Status Return Values	4.10, C.2.3
Finite (built-in predicate)	2.5.4.5, 4.3.7.3
Flag (FL) variables	2.4.2.2
FOR clause (in LOOP)	3.2.7.2
Functions	4.3.7
on Arrays	4.3.7.2
Built-in, defined	4.3.7
Arithmetic	4.3.7.1
Logical	4.3.7.3
Miscellaneous	4.3.7.4
Function Declarations	
Definition	2.4, 2.4.6
Description	2.4.6.2
Examples	2.4.6.3
Syntax	2.4.6.1
General CL Information	1.2.1
Generic Programs	2.3.4.4
Get_CL_Slot	4.3.7.5
Global Data Definitions	4.1
GOTO (Statement)	
Definition	3.2.5
Description	3.2.5.2
Syntax	3.2.5.1
Hibernating Applications	I.1
HPM Fieldbus parameter restrictions	2.2.5.4

---

# Index

---

Topic	Section Heading
Identifiers	
Block	2.2.5.1
Box Data Point	2.3.4.7, 2.3.4.8
Conflicts between	2.2.7.9
Data point	2.3.4.2
Defined	2.2.7
Description of	2.2.7.2
Enumeration-state	2.2.5.1
Examples of	2.2.7.3
Length of	2.2.5.1
Predefined	2.2.7.6
Parameter	2.2.5.1
Special Identifiers	2.2.7.6
Syntax	2.2.7.1
IF,THEN,ELSE (Statement)	
Definition	3.2.6
Description	3.2.6.2
Examples	3.2.6.3
Flow Charted	Figure 3-1
Syntax	3.2.6.1
<i>see also</i> Conditions	
%INCLUDE_SET Directive	3.3.5, 4.8
%INCLUDE_SOURCE Directive	3.3.6
Incorrect Examples of Comments	2.2.6.2
Index (array)	2.4.2.2
Indirect Reference	2.3.4.2, 2.3.7, 2.3.8
<i>see also</i> Data Point Identifiers	
Indirection, Dynamic	2.3.8
Infinite Values Definition	2.3.1.2
<i>see also</i> Bad values	
Insertion Point Names	2.2.7.4
by Build Type	Table 2-3
INT (truncate to integer) Function	4.3.7.1
Integer	2.2.8.1
<i>see also</i> Number	
Introduction (to CL Statements)	3.1
Introduction to CL Rules and Elements	2.1
ISO 646 Compatibility	2.2.5.2
kill_appls X-Side Tool, use of	I.14
Labels	
Conflict between identifiers	2.2.7.9
Defined	3.2.2
Examples	3.2.2.1
in LOOP statement	3.2.7.2
in REPEAT	3.2.8
<i>see also</i> Statements	
LEN (function)	2.3.6, 2.4.3.2, 4.3.7.4
Length of Identifiers	2.2.5.1

---

# Index

---

Topic	Section Heading
Linearization (Continuous)	4.6.2
Lines	
Continuation of	2.2.3, 2.2.10
Defined	2.2.3
Link, Linker	2.3.4.4, 2.3.4.5
Linkage error (from attempted aliasing)	2.3.4.5
Literals	
Time	2.5.3.3, 2.5.3.4
Numeric	2.4.3.2
LN (natural logarithm) function	4.3.7.1
Local Constants	
Defined	2.4.3
Description	2.4.3.2
Examples of	2.4.3.3
as Objects	2.2.7.9
Syntax	2.4.3.1
Local Variables	
Defined	2.4, 2.4.2
Description	2.4.2.2
Examples	2.4.2.3
as Objects (conflict with other identifiers)	2.2.7.9
Restrictions in arrays	2.3.5
Syntax	2.4.1, 2.4.2.1
LOG10 (common logarithm) function	4.3.7.1
Logarithms	4.3.7.1
<i>see also</i> Arithmetic Functions, Subroutines	
Log_Only special destination	3.2.10.2
Logical AND (Operator)	2.5.2.4, Table 2-9, Table 2-10
as Connective	2.5.4.6
Logical Expressions	
Defined	2.5.2
Syntax	2.5.2.1
Logical Functions	4.3.7.3
Logical Operand	2.5.2.3
Logical Operators Truth Table	2.5.2.4, Table 2-10
Logical NOT	2.5.2.4, Table 2-9 Table 2-10
Logical OR, Exclusive OR (XOR) Operator	2.5.2.4, Table 2-9 Table 2-10
as Connective	2.5.4.6
Logical Types	
Arrays of	2.3.5
Defined	2.3.3.3
compared to Pascal Boolean	Figure 2-1
LOOP	
Defined	3.2.7
Description	3.2.7.2
Examples	3.2.7.3
Syntax	3.2.7.1
<i>see also</i> REPEAT	

---

# Index

---

Topic	Section Heading
MAILBOX parameter	3.2.10.2
Math Library, CL/AM Extension for	E
MAX (maximum) function	4.3.7.1, 4.3.7.2
MC Array Parameters, Access from AM	2.3.5
MC Box Data Point Identifiers	2.3.4.8, 2.3.4.9
MIN (minimum) function	4.3.7.1, 4.3.7.2
Miscellaneous Functions	4.3.7.4
Modulus Operator (MOD)	2.3.1, 2.5.2.4, Table 2-9 Table 2-10
Move_Parameter (subroutine)	4.3.7.5
Multiplication Operator (mulop)	2.5.2.4, Table 2-9
Network Gateway	
File I/O through	C.2.2.1
Access to off-LCN point data through	2.2.5.3
NOT, Negation (Operators)	2.5.2.4, Table 2-9
Now (function)	4.3.7.4
Number (function)	4.3.7.4
Number_to_String (subroutine)	4.3.7.5
Numbers	
Definition	2.2.8, 2.3.1
Description	2.2.8.2
Examples	2.2.8.3
as Local Constant	2.4.3
use of as Index	2.4.2.2
Syntax	2.2.8.1
Unsigned	2.2.8.1
<i>see also</i> Arrays, Bad value, Discrete Types, Infinite values, Integer, Time, Uncertain value	
Numeric literals	2.4.3.2
Numeric variables (NN)	2.4.2.2
Objects	2.2.7.9
Off-Node Access, AM Extension for	H
Operand	
Definition	2.5.2.2
Syntax	2.5.2.3
Operators	2.5.2.4, Table 2-9 Table 2-10
Arithmetic	2.2.10
Assignment	2.2.10
as Connectives	2.5.4.6
Equality	2.2.10, Table 2-4
Defined	2.5.2.4
Relational	2.2.10, 2.5.4.1 2.5.4.3, Table 2-13
Time	Table 2-12
<i>see also</i> Special Symbols	
Optional Extension to CL	C

---

# Index

---

Topic	Section Heading
OR (logical Operator)	2.5.2.4, Table 2-9, Table 2-10
as Connective	2.5.4.6
Ord (function)	4.3.7.4
%PAGE directive	3.3.2
Page break directive	3.3.2
<i>see also</i> Embedded Compiler Directives, %PAGE	
PACKAGE	
Definition 4.7	
Description	4.7.2
Examples	4.7.3
Syntax	4.7.1
Parameter	
Lists	2.2.7.7, 2.3.4.4
MAILBOX	3.2.10.2
Parameter Data Types Definition	2.4.5.4
Parameter Declarations	
Definition	2.4.5
Description	2.4.5.2
Examples	2.4.5.3
Redundant	2.4.5.5
References in Packages	2.4.5.6, 2.4.5.7
Syntax	2.4.5.1
Parameter-Heading	
Definition	4.4.4
Description	4.4.4.2
Examples	4.4.4.3
Syntax	4.4.4.1
Parameter List	
Definition	4.5
Description	2.3.4.4, 4.5.2
Examples	2.3.4.2, 2.3.4.4, 4.5.3
\$REG_CTL Parameter list	2.3.4.2, B.3
Syntax	4.5.1
Parameter References in Packages	2.4.5.6
Examples	2.4.5.7
Parentheses (as special symbol)	2.2.10
PIN Identifiers	2.2.5.3
PList - <i>see</i> Parameter List	
PM Array parameters, Access from AM	2.3.5
PM Box Data Point Identifiers	2.3.4.8, 2.3.4.9
Predicates	
Badval and Finite	2.5.4.5
in Conditions	2.5.4
Processor Status Data Point	2.4.4.2
Production Rules (alternative to Syntax Diagrams))	A.3
PROGRAM (as access_key_id (privilege) of CL Block)	4.2.5
Program Statements	Section 3
Definition	3.2
Labels	3.2.2, 3.2.2.1
Syntax	3.2.1

---

# Index

---

Topic	Section Heading
Programs, Generic	2.3.4.4
Programs, Specific	2.3.4.3
Publications with CL-Specific Information	1.2.2
Punctuation (as special symbol)	2.2.10
Purpose/Background	1.1
PV Calculation (Continuous)	4.6.1
Quoted String—see String	
Range Separator	2.2.10, Table 2-4
Range Tests Definition	2.5.4.4
Redundant Parameter Declarations Definition	2.4.5.5
\$REG_CTL Parameter List	2.3.4.2, B.3
\$REG_CTL Parameter List use examples	2.3.4.2, 2.3.8
Relational Operator (relop)	2.2.10, 2.5.4.3 Table 2-13
Relations	2.5.4.3
%RELAX directive	3.3.4
REPEAT (Statement)	
Definition	3.2.8
Description	3.2.8.2
Examples	3.2.8.3
Syntax	3.2.8.1
Reports, Event Initiated, from CL	3.2.10.4
Reserved Words	
Definition	2.2.7.4, Table 2-2
Round (to integer) Function	4.3.7.1
Rules and Elements of CL	2
Runtime error	2.3.4.2
caused by REPEAT	3.2.8.2
Runtime Extensions	4.8
Scalar data type definition	2.3
Scope	2.2.7.9
Self (function)	4.3.7.4
SEND (statement)	
Definition	3.2.10
Description	3.2.10.2
Event Initiated Reports	3.2.10.4
Examples	3.2.10.4
Syntax	3.2.10.1
use of Strings in	2.3.6, 3.2.10.2
use of Tag Names in	3.2.10.2
use of Variable in	2.4.2.2
SET (Statement)	
Definition	3.2.3
Description	3.2.3.2
Examples	3.2.3.3
Syntax	3.2.3.1
Set_Bad (subroutine)	4.3.7.5
Set_Null_Point_id (subroutine)	4.3.7.5
Shared State Names Definition	2.3.3.1
SIN (sine) Function	4.3.7.1

---

# Index

---

Topic	Section Heading
Spacing	
in Elements	2.2.2
Requirements	2.2.2
Special Identifiers	
Definition	2.2.7.7
Examples	2.2.7.8
Specific Programs	2.3.4.3
Special Symbols Definition	2.2.10
SQRT (square root) function	4.3.7.1
STATE CHANGE	
Definition	3.2.4
Description	3.2.4.2
Examples	3.2.4.4
Syntax	3.2.4.1
Statement Labels	
Definition	3.2.2
Examples	3.2.2.1
Statements	Section 3, 3.2
<i>see also</i> Program statements, ABORT, CALL, ELSE, END, EXIT, GOTO, IF, LOOP, REPEAT, SEND, SET, STATE CHANGE	
Stopping X-Side Applications	1.13
String Arrays	2.3.5
String Data Type	2.3.6
Strings	
Definition	2.2.9
Description	2.2.9.2
Examples	2.2.9.3
Syntax	2.2.9.1
Structures	Section 2, 2.2.7.9
Subroutine	
Arguments	4.3.6
Arithmetic Functions	4.3.7.1
Built-in, defined	4.3.7
CALL statement	3.2.9
Conflicts between identifiers	2.2.7.9
Data Declaration	4.3.5
Definition	4.3
Functions on Arrays	4.3.7.2
Logical Functions	4.3.7.3
Miscellaneous Functions	4.3.7.4
Subroutines and Functions	4.3.7, 4.3.7.1 - 4.3.7.5,
Subroutine Arguments Definition	4.3.6
Subroutine CALL statement	
Defined	3.2.9
Syntax	3.2.9.1
Description	3.2.9.2



---

# Index

---

Topic	Section Heading
Subroutine Data-Declarations Definition	4.3.5
Subroutine-Heading	
Definition	4.3.1
Description	4.3.3
Examples	4.3.4
Syntax	4.3.2
User written	4.3
Subscripting restrictions in indirect references	2.3.7
SUM	
Function	4.3.7.1, 4.3.7.2
Operator	2.5.2.4, Table 2-9
Supervisor (as access_lock_id)	4.4.5.3.2
Syntax	
Method of presentation	2.1, 2.2.4
Summary for all CL forms	A
<i>see also</i> Production Rules	
Syntax Diagram Definition	2.2.4
Syntax Diagram Summary	A.2
Tag Name— <i>see</i> Data Point Identifier	
TAN (tangent function)	4.3.7.1
Termination	
Abnormal	3.2.12
Statements	3.2
<i>see also</i> ABORT, END	
Time Definition	2.3.2
Time Expressions	
Defined	2.5.3
Examples	2.5.3.8
as Local constant	2.4.3
Syntax	2.5.3.1
Time Literals	
Defined	2.5.3.3
Description	2.5.3.5
Examples	2.5.3.6
as Local constant	2.4.3.2
Multipliers	2.5.3.5, Table 2-11
Syntax	2.5.3.4
Time Operands	
Defined	2.5.3.2
Time Operators	
Defined	2.5.3.7, Table 2-12
Uncertain Values Definition	2.3.1.3
Unconditional Branch (GOTO)	3.2.5
Unsigned number	2.2.8, 2.2.8.2

---

# Index

---

Topic	Section Heading
VALUE (attribute)	
Definition	4.4.5.6.1
Description	4.4.5.6.3
Examples	4.4.5.6.4
Syntax	4.4.5.6.2
Variables and Declarations	
General discussion of use	2.4
Syntax	2.4.1
<i>see also</i> External variables, Function definitions, Local variables, Operands, Parameter variables	
WHEN clause,	
use in BLOCK heading	4.2.4
use in SET statement	3.2.3
XOR (logical operator)	2.5.2.4
X-Side	
display_appls Tool	I.15
kill_appls X-Side Tool	I.14
Stopping X-Side Applications	I.13

---

**FAX Transmittal**

---

---

**FAX No.: (602) 313-4842**

---

**TO:** Automation College**Total FAX pages:** \_\_\_\_\_  
(including this page)

### Reader Comments

Title of Document: Control Language Application Module Reference Manual

Document Number: AM27-610

Issue Date: 01/00

**Comments:** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_**Recommendations:** \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_**FROM:****Name:** \_\_\_\_\_**Date:** \_\_\_\_\_**Title:** \_\_\_\_\_**Company:** \_\_\_\_\_**Address:** \_\_\_\_\_**City:** \_\_\_\_\_**State:** \_\_\_\_\_**ZIP:** \_\_\_\_\_**Telephone:** \_\_\_\_\_**FAX:** \_\_\_\_\_

### FOR ADDITIONAL ASSISTANCE:

Write	Call
Honeywell Inc. Industrial Automation and Control Automation College 2820 West Kelton Lane Phoenix, AZ 85053-3095	Technical Assistance Center (TAC) 1-800-822-7673 (48 contiguous states except Arizona) 602-313-5558 (Arizona)







---

**Industrial Automation and Control**

Automation College

2820 W. Kelton Lane

Phoenix, AZ 85053-3028

*Helping You Control Your World*