Specification Number CR-CP-0059-C81
Code Ident: FSCM 80063
November 1983

DRAFT

ADA* LANGUAGE SYSTEM SPECIFICATION

CR-CP-0059-A00

VOLUME I

Contract No. DAAK80-80-C-0507
CDRL Item B010

Submitted to

Headquarters, United States Army
Communications Electronics Command
Fort Monmouth, NJ 07703

Prepared by

SofTech, Inc.
460 Totten Pond Road
Waltham, MA 02154

1075-4.6

EXPORT OF THE ADA LANGUAGE SYSTEM AT ANY TIME REQUIRES AN EXPORT LICENSE FROM THE U.S. DEPARTMENT OF COMMERCE.

*Ada is a registered trademark of the Department of Defense (Ada Joint Program Office) OUSDRE (R&AT)

TABLE OF CONTENTS

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

ILLUSTRATIONS

ILLUSTRATIONS (Cont.)

ILLUSTRATIONS (Cont.)

ILLUSTRATIONS (Cont.)

TABLES

TABLES (Cont.)

TABLES (Cont.)

1. SCOPE

1.1 Specification. - This specification establishes the performance, design, development, documentation, and qualification requirements for the Ada Language System (ALS).

1.2 Ada Language System. - The Ada Language System supports the development of programs written in the Ada language for advanced, embedded military computer systems. Furthermore, the ALS supports the development of these programs by program teams and/or individual programmers.

2. APPLICABLE DOCUMENTS

2.1 Government Documents. - The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

Specifications:

MIL-S-52779(AD)	Software Quality Assurance Program Requirements, 5 April 1974

Standards:

MIL-STD-483	Configuration Management Practices for Systems, Equipment, Munitions, and Computer Programs, 31 December 1970; Notice 2, 21 March 1979
MIL-STD-490	Military Standards and Specification Practices, 30 October 1968; Notice 2, 18 May 1972
MIL-STD-1862A	Military Standard, Nebula Instruction Set Architecture, 2 November 1981
ANSI/MIL-STD-1815A-1983	Military Standard, Ada Programming Language, 17 February 1983

Other Publications:

Requirements for Ada Programming Support Environments-STONEMAN, February 1980

AMCP-70-4, Research and Development Software Acquisition, A Guide for the Material Developer, 2 Sep 74

Ada Language System Design and Development Plan, U.S. Army CECOM, Ft. Monmouth, N.J., Contract No. DAAK80-80-C-0507, April 1983

Ada Language System Quality Assurance Plan, U.S. Army CECOM, Ft. Monmouth, N.J., Contract No. DAAK80-80-C-0507, Oct 1980

Ada Language System Configuration Management Plan, U.S. Army CECOM, Ft. Monmouth, N.J., Contract No. DAAK80-80-C-0507, Oct 1980

Ada Language System Preliminary/Formal Qualification Test Plan, U.S. Army CECOM, Ft. Monmouth, N.J., Contract No. DAAK80-80-C-0507, February 1983

2.2 Non-Government Documents. - The following documents of the exact issue shown form a part of this specification to the extent specified herein. In the event of conflict between the documents referenced herein and the contents of this specification, the contents of this specification shall be considered a superseding requirement.

a. Blank

b. Blank

c. Blank

d. VAX-11 Architecture Handbook, Volume I, Digital Equipment Corporation, 1979-1980

e. VAX Software Handbook, Digital Equipment Corporation, 1980-1981

f. VAX-11/780 Hardware Handbook, Digital Equipment Corporation, 1979-80

g. VAX-11 EDT Editor Reference Manual, Digital Equipment Corporation, April 1980

h. VAX-11 MACRO Language Reference Manual, Digital Equipment Corporation, February 1979

i. DEC Standard Runoff (DSR) User's Guide, Digital Equipment Corporation, Nov 1979

j. Diana Reference Manual, G. Goos and Wm. Wulf (Editors), March 1981

k. Blank

l. Blank

m. VAX/VMS Command Language User's Guide, Digital Equipment Corporation, March 1980

n. Blank

o. The Nebula Assembler, 17 November 1981.

3. REQUIREMENTS

3.1 Ada Language System Definition. -

3.1.1 General Description. - The Ada Language System (ALS) is a facility for the development of Ada programs. It provides to the users a programming environment for software development and maintenance. The ALS programming environment is defined to include:

a. A Database, containing text, data, programs, and fragments of programs in various representations, and showing relationships among fragments and programs;

b. A Set of Tools, for entering text, translating and executing programs, and otherwise manipulating the database; and

c. A Command Language, for invoking the tools under user control.

3.1.1.1 Host Environments. -

3.1.1.1.1 VAX/VMS Host Environment. - The VAX/VMS* Host Environment for the Ada Language System shall be a VAX-11/780 configuration coupled with a VAX/VMS Operating System. The host configuration shall include, as a minimum, the following:

a. Hardware

QUANTITY	DESCRIPTION
1	VAX-11/780 Processor with 4M bytes ECC MOS Memory
1	LA36 DEC Writer II Console
2	RP07, 512M bytes, Disk Drives, or equivalent space on other VMS-supported discs

*VAX and VMS are trademarks of Digital Equipment Corporation

1	TU45, TU77, or TU78 Tape Drive
1	LP11-DA, 660 lpm, 132 col, 96 char Line Printer
-	VT-100 Video Terminals sufficient to support the number of expected users

b. Resident Operating System: VMS Version 3

3.1.1.1.2 Generic Minimum Host Environment. - A generic minimum host environment for an ALS shall include:

a. Hardware

QUANTITY	DESCRIPTION
1	Central Processor with at least 8M bytes of address space and approximately 4M bytes of physical memory (to support 10 concurrent users)
-	Disk Drives that maintain at least 1000M bytes total on-line at any time
1	Tape Drive
1	Operator's Console
-	Terminals sufficient to support the number of expected users
1	Line Printer with upper and lower case capability, and capable of supporting a minimum of 120 columns per line and 60 lines per page

b. Resident Operating System: None

c. Language Support. All host environments shall be capable of supporting the full Ada language with the following minimal capabilities:

- Integer arithmetic on values with a range of at least -2,147,483,647 ... 2,147,483,647.

- Floating point arithmetic with an accuracy of at least 6 decimal digits and a range of -1.2E24 ... 1.2E24.

- Enumerations with at least 256 distinct values.

- The text input/output facilities must be capable of recognizing the ACSII character enumeration set, and supporting the basic graphic character set and extensions to that set including lower case letters and any special characters defined in Par.2.1 of the Reference Manual for the Ada Programming Language (2.1).

3.1.1.2 Target Environments. - The ALS shall provide the capability of creating Ada programs which run on the following target environments.

3.1.1.2.1 VAX-11/780 VAX/VMS Target Environment. - A VAX-11/780 VAX/VMS target environment is the same as the VAX-11/780 VAX/VMS Host with the ALS residing on it.

3.1.1.2.2 Blank. -

3.1.1.2.3 Blank. -

3.1.1.2.4 Blank. -

3.1.1.2.5 Blank. -

3.1.1.2.6 MCF Target Enviroment. - An MCF target environment consisting of the following:

a. Hardware

QUANTITY	DESCRIPTION
1	AN/UYK-41 Processor with 1M bytes Memory
<TBD>	

b. Resident Operating System: None

3.1.1.3 Ada Language Implementation. -

3.1.1.3.1 Ada Language Implementation Dependencies. - The Ada language implementation dependencies for each target environment in 3.1.1.2 are described in Appendix 10 to this specification.

3.1.1.3.2 Ada Language Terminology. - The Ada language terminology used in this specification is precisely defined in the Military Standard, Ada Programming Language, ANSI/-STD-1815A-1983, 17 February 1983 (2.1). The following brief definitions are provided for quick reference. (Note: A glossary of ALS terms is included in Section 6.1).

a. Subprogram: An executable program unit that is invoked by a subprogram call.

b. Subprogram body: A subprogram body specifies the execution of a subprogram.

c. Compilation unit: The smallest unit of text that may be submitted for compilation.

d. Subunit: A body of a subprogram, package, or task declared within another compilation unit.

e. Library unit: A compilation unit that is not a subunit of another unit.

f. Compilation: Zero or more compilation units submitted to the compiler, or the act of translating a compilation into instructions for the target machine.

g. Program library: A collection of compilation units of which a program is composed.

h. Program: A collection of one or more compilation units representing a complete executable entity.

i. Library subprogram body: The body of a subprogram which is not a subunit.

j. Visibility: At a given point in a program text, the declaration of an entity is said to be visible if the entity is an acceptable meaning for an occurrence at that point of the identifier.

k. Dependence relation: The relationship among compilation units that indicates the required order of compilation.

3.1.1.4 Functional Areas. - The Ada Language System shall consist of the following functional areas depicted in Figure 3-1.

a. The Compiler is a tool for translating Ada compilations into machine code applicable to the target environments.

b. The Assemblers are tools for translating subprogram bodies written in assembly language into machine code applicable to the target environments.

c. The Linker functional area includes tools for combining compilation units, previously translated by a compiler or assembler, into load modules.

d. The Loaders are tools for bringing load modules into execution in bare target machine environments.

e. The DEC VAX/VMS Text Editor and the DEC Standard Runoff Formatter are tools for entering and updating text, and for obtaining formatted listings of text.

f. The Configuration Control Tools (CCT) are a set of tools for manipulating the database, for supporting configuration control, and for separating programs for distributed targets.

g. The Command Language Processor (CLP) interprets the command language input from the user, invoking tools as required.

h. The Database Manager (DBM) provides user access to the environment database from the command language. It provides primitive functions for manipulation of program libraries. It also provides primitive functions that enable Ada programs to examine and modify the contents of Containers.

i. The Kernel Ada Programming Support Environment (KAPSE) provides, together with the runtime support library, the only interface between the ALS and the host operating system. (The KAPSE is not shown in Figure 3-1.)

j. The Display Tools provide displays of compiler, linker and assembler listings, as well as listings of maintenance aids.

k. The File Administrator (FA) is a collection of tools providing services for comparing elements of the environment database, for balancing disk/tape requirements of the database, for restoring the database after mechanical or human failure, for long-term storage of database information, and for ALS-to-ALS data transmission.

l. The Symbolic Debugger is a tool providing services for interactive debugging of Ada programs executing on the host computer.

m. The Statistical Analyzer is a tool that analyzes the distribution of processor use (i.e., execution time) among the parts of an Ada program executing on the host computer.

n. The Frequency Analyzer is a tool that analyzes the frequency with which parts of an Ada program executing on the host computer have been executed in a given set of tests.

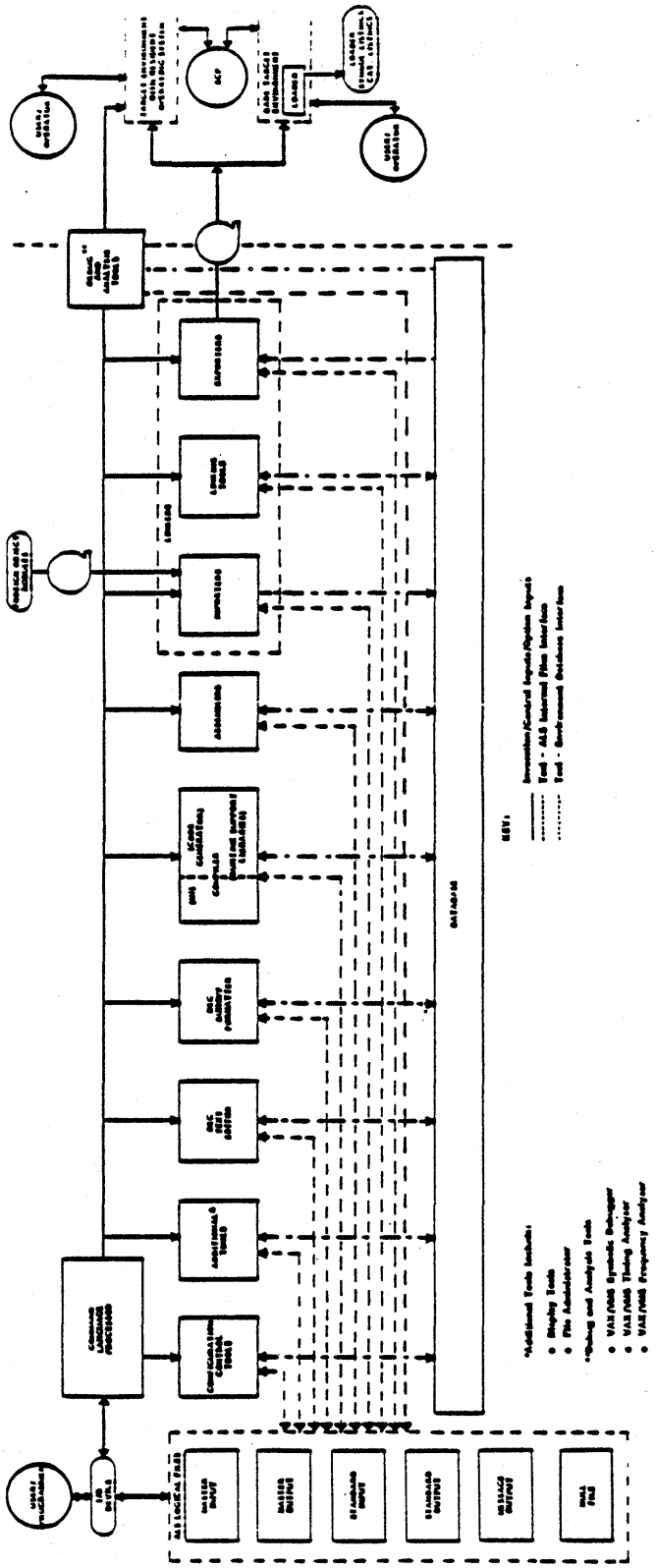


Figure 3-1. Functional Areas of the Ada Language System

3.1.1.5 List of Computer Program Configuration Items. - The Computer Program Configuration Items (CPCIs) in the Ada Language System are listed below. The corresponding functional area for each CPCI is indicated.

CPCI No.	Name	Functional Area
10	ALS VAX-11/780 Code Generator	Compiler
11	Unused	
12	Unused	
13	Unused	
14	ALS MCF Code Generator	Compiler
20	ALS VAX-11/780 VAX/VMS Runtime Support Library	Compiler
21	Unused	
22	Unused	
23	Unused	
24	Unused	
25	ALS MCF Runtime Support Library	Compiler
30	ALS VAX-11/780 Assembler	Assembler
31	Unused	
32	Unused	
33	Unused	
34	ALS MCF Assembler	Assembler
40	ALS VAX-11/780 Linker	Linker
41	Unused	
42	Unused	
43	ALS MCF Linker	Linker
50	Unused	

51	Unused	
52	Unused	
53	ALS MCF Loader	Loader
60	ALS VAX/VMS Symbolic Debugger	Symbolic Debugger
70	ALS VAX/VMS Frequency Analyzer	Frequency Analyzer
75	ALS VAX/VMS Statistical Analyzer	Statistical Analyzer
80	ALS Command Language Processor	Command Language Processor
81	ALS Database Manager	Database Manager
82	ALS Configuration Control Tools	Configuration Control Tool
83	ALS Kernel Ada Programming Support Environment (KAPSE)	KAPSE
84	ALS Compiler Machine-Independent Section	Compiler
85	ALS File Administrator	File Administrator
86	ALS Display Tools	Display Tools

3.1.1.6 Logical Files. - The ALS shall contain the following six pre-defined logical files:

a. Master input,

b. Master output,

c. Standard input,

d. Standard output,

e. Message output, and

f. The null file.

The master input, master output, message output, and null files are defined in the KAPSE. The standard input and standard output files are defined by the Ada TEXT_IO package. Additional information on the use of these files is provided in Appendix 60.

3.1.1.6.1 File Names. - The standard names for the logical files shall be:

a. .MSTRIN,

b. .MSTROUT,

c. .STDIN,

d. .STDOUT,

e. .MSGOUT, and

f. .NULL_FILE or .NF, respectively.

3.1.1.6.2 File Assignment. - All six files are opened and ready for use when an ALS tool is invoked. The master input and master output files may not be reassigned. The standard input, standard output, and message output files may be reassigned by tool commands as described in Appendix 60.

3.1.2 Mission/Purpose. - The purpose of the Ada Language System is to provide a comprehensive, friendly, long-term programming environment for the design, development, documentation, testing, management, and maintenance of software, coded in the Ada language and used in embedded military computer systems. To accomplish this mission the Ada Language System is designed to:

a. Provide portability of the system across multiple host and target environments;

b. Include a common command language and tool set that will enable users to move easily across host environment boundaries;

c. Provide a framework for the incorporation of additional tools that will fulfill Ada programming requirements for embedded military computer systems throughout the lifetime of the ALS; and

d. Provide extensive programming support for both individual programmers developing single and/or multiple programs, and teams of programmers working on a single program.

3.1.3 Threats. -

3.1.3.1 Potential Threats. - Potential threats to the Ada Language System are:

a. Deliberate unauthorized access to the ALS database,

b. Inadvertent unauthorized access to the ALS database,

c. Deliberate modification and/or destruction of the ALS database, and

d. Inadvertent modification and/or destruction of the ALS database.

3.1.3.2 Ada Language System Access and Security Control. - The ALS provides its own internal access control independent of host system access control. Each user request through the ALS for access to any node in the environment database is checked for authorization before access is granted. (See Appendix 50 for details.) The ALS does not require any access control facilities from the host operating system in order to insure proper access control to ALS users. However, unauthorized external access to the ALS may be available through the host system, depending upon the characteristics of host system access and security controls. ALS security control is dependent upon host and target system security control.

3.1.4 System Diagrams. - The following system diagrams are included in this specification:

a. Functional Areas of the Ada Language System (Figure 3-1)

b. Ada Language System Architecture (Figure 3-2)

c. Ada Language System Specification Tree (Figure 3-3)

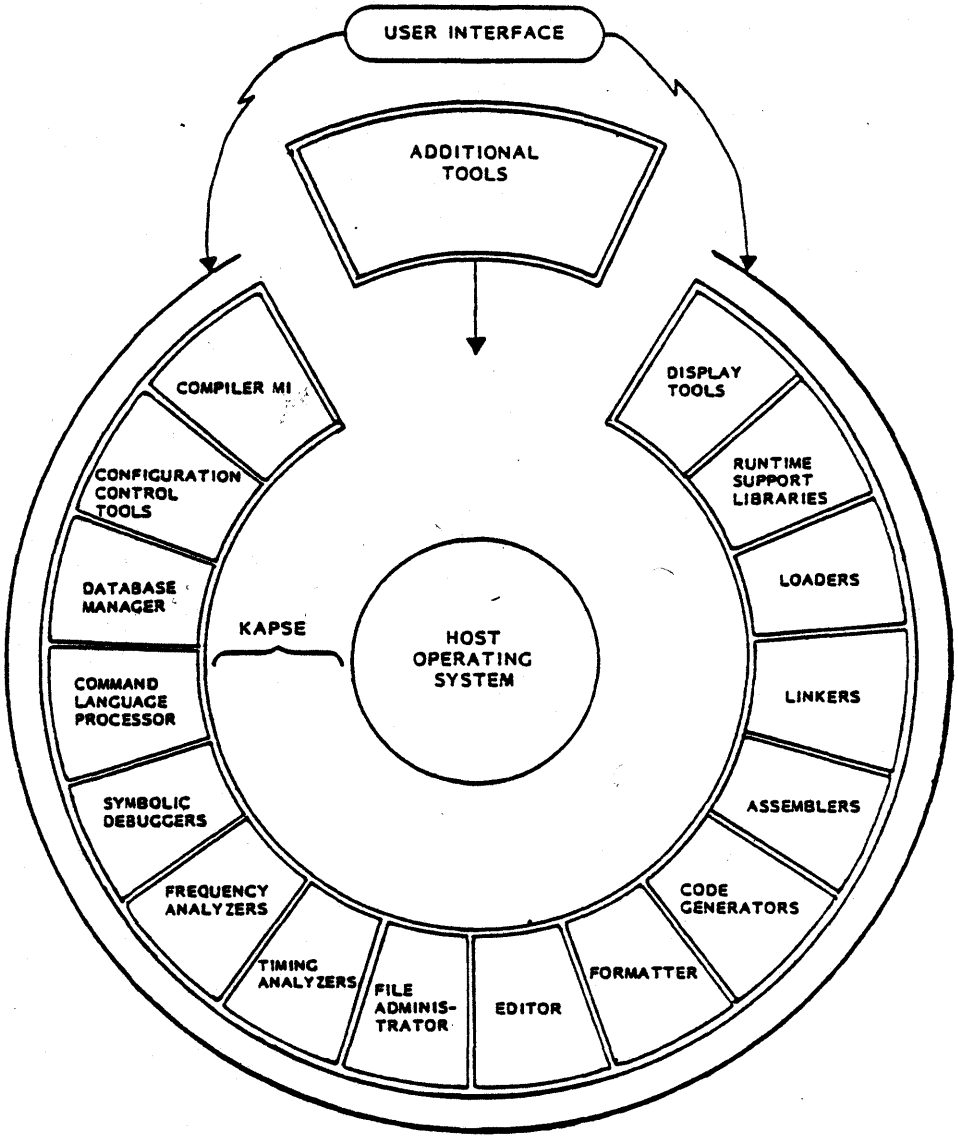


Figure 3-2. Ada Language System Architecture

3.1.5 Interface Definitions. -

3.1.5.1 ALS/User Interfaces. -

3.1.5.1.1 Physical Interfaces. - For interactive operation, the physical interface between the host system, the ALS, and a user should include a keyboard display terminal and a printer. Typically, one or more of these units will be part of the host system. The physical interfaces for batch operation are dependent upon the facilities available in the host system.

3.1.5.1.2 Programming Interface. - The programming interface between the ALS and a user shall be the command language, the Ada language, and the assembly language for the applicable target system (see Figures 3-1 and 3-2). Additional tools will interface to the KAPSE, and to the ALS Database Manager.

3.1.5.2 ALS/Host Operating System Interface. - The interface between the ALS and the host operating system shall be the ALS Kernel Ada Programming Support Environment (KAPSE). This interface is shown in Figure 3-2.

3.1.5.3 ALS/Target Machine Interface. - The interface between the ALS and a target machine shall be load modules on the appropriate physical computer medium, i.e. tape, disk, etc.

3.1.5.4 ALS Functional Area Interfaces. -

3.1.5.4.1 Compiler. - The compiler functional area shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs;

b. ALS Database Manager for input of information from previous compilations, and output of machine code, statistics, and information for future compilations, assemblies, and links;

c. The target machine operating system (where applicable) for I/O, memory management, and other support;

d. KAPSE to obtain host operating system services such as underlying I/O support; and

e. Display Tools to generate listings and to obtain maintenance aid services.

3.1.5.4.2 Assembler. - The assembler functional area shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs;

b. ALS Database Manager for input of information from previous compilations, and output of machine code, statistics, and information for future compilations, assemblies, and links;

c. KAPSE to obtain host operating system services such as underlying I/O support; and

d. Display Tools to generate listings and to obtain maintenance aid services.

3.1.5.4.3 Linker. - The linker functional area shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs;

b. Computer systems other than the host system for foreign object modules received via an importer tool;

c. Target machine for transmission of load modules for execution;

d. ALS Database Manager for input of machine code and other information from previous compilations, assemblies, and links; output of information for future links and output of load modules for execution;

e. KAPSE to obtain host operating system services such as underlying I/O support; and

f. Display Tools to generate listings and to obtain maintenance aid services.

3.1.5.4.4 ALS Loader (For Bare Target Machines). - The ALS loader functional area for a bare target machine shall interface with the following:

a. Load module input from the storage medium,

b. Target machine bootstrap mechanism,

c. User terminal(s) for interactive use and listings, and

d. ALS Runtime Support Library load modules for initialization when appropriate.

3.1.5.4.5 Loader (For Target Machine with Resident Operating System). - The exporter in the linker functional area shall interface with the loader of a target machine with a resident operating system. The output of the exporter shall be compatible with the input required by the target machine.

3.1.5.4.6 DEC VAX/VMS Text Editor. - The Text Editor shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs;

b. User terminals for interactive inputs and outputs; and

c. KAPSE to obtain ALS related services.

3.1.5.4.7 DEC Standard Runoff Formatter. - The Formatter shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs; and

b. KAPSE to obtain ALS related services.

3.1.5.4.8 Configuration Control Tools. - The configuration control tools shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs;

b. KAPSE to obtain host operating system services such as underlying I/O support; and

c. File Administrator for providing archiving services.

3.1.5.4.9 Command Language Processor. - The command language processor shall interface with the following:

a. User terminal(s) for command language inputs;

b. All ALS tools except loaders for user invocation commands, user option inputs, user control inputs; and

c. KAPSE to obtain host operating system services such as underlying I/O support.

3.1.5.4.10 Database Manager. - The database manager shall interface with the following:

a. All host-resident ALS tools to modify or examine compiled, assembled or linked programs; and

b. KAPSE to obtain host operating system services such as underlying I/O support.

3.1.5.4.11 Kernel Ada Programming Support Environment. - The KAPSE shall interface with the following:

a. The host operating system (see Figure 3-2); and

b. All host-resident ALS tools, and the ALS Database Manager.

3.1.5.4.12 Display Tools. - The display tools functional area shall interface with the following:

a. ALS Command Language Processor for user invocation commands, user option inputs, and user control inputs;

b. ALS Database Manager for requests to examine the Containers in the data base;

c. Compilers, assemblers, and linkers to generate listings and to provide maintenance aid services; and

d. KAPSE to obtain host operating system services such as I/O.

3.1.5.4.13 File Administrator. - The File Administrator shall interface with the following:

a. ALS Configuration Control Tools to provide archiving services and to obtain text I/O services; and

b. KAPSE to obtain host operating system services such as I/O.

3.1.5.4.14 Symbolic Debugger. - The Symbolic Debugger shall interface with the following:

a. ALS Command Language Processor for user invocation and user control inputs;

b. The Compiler to obtain information relating the source form of compiled Ada programs with the object form of those programs. The RSL to obtain notification of program events, such as unhandled exceptions and task allocation;

c. The Linker to obtain information about the address binding of the executable image;

d. The Database Manager to obtain access to executable images, linked images, and other objects maintained within program libraries;

e. The KAPSE to obtain operating system services such as I/O and execution of the image of the program under test; and

f. User terminals for interactive inputs and outputs.

3.1.5.4.15 Statistical Analyzer. - The Statistical Analyzer shall interface with the following:

a. ALS Command Language Processor for user invocation and user control inputs;

b. The Linker to obtain binding with special run-time support and allocation of memory for analysis tables;

c. The Database Manager to obtain access to program libraries and Containers, and to store analysis results; and

d. The KAPSE to obtain host operating system services such as CPU time accounting and I/O support.

3.1.5.4.16 Frequency Analyzer. - The Frequency Analyzer shall interface with the following:

a. ALS Command Language Processor for user invocation and user control inputs;

b. The Compiler for insertion of frequency monitoring code and to obtain information relating the source form of compiled Ada programs to the object form of those programs;

c. The Linker to obtain binding with special run-time support and allocation of memory for analysis tables;

d. The Database Manager to obtain access to program libraries and Containers, and to store analysis results; and

e. The KAPSE to obtain host operating system services such as I/O support.

3.1.6 Government Furnished Property List. - The Government shall furnish to the contractor, via a mutually acceptable medium, an MCF Simulator which operates on a VAX/VMS host system. Along with the MCF Simulator, the Government shall provide user documentation which describes how to use the simulator on the VAX/VMS host system. <TBD>

3.1.7 Operational and Organizational Concepts. -

3.1.7.1 Ada Language System Operational Guidelines. -

Guidelines for field operation of the ALS are as follows:

a. The host environment should be capable of supporting ten concurrent users without degrading the system.

b. The target machine environment shall have the capability of accepting ALS target programs that do not exceed its memory and CPU resources.

c. The transfer medium, loading mechanism, and formatting capabilities shall be adequate for the ALS.

3.1.7.2 Ada Language System Deployment. - No fundamental limitations on the deployment of the ALS exist. The ALS can be located at any site that can provide a host environment that meets the requirements of 3.1.1.1.2. (It is anticipated that the ALS will be distributed to many centers but will be centrally maintained.) The ALS host and target environments need not be located at the same site.

3.2 Characteristics. -

3.2.1 Ada Language System Performance Characteristics. -

3.2.1.1 Portability. - The Ada Language System will be designed to provide portability of users and tools across multiple hosts and multiple targets, as described in the following subparagraphs.

3.2.1.1.1 Retargetability. - The Ada Language System tools shall be designed for retargetability. Each tool in the system will be designed for isolation and parameterization of target dependencies. The ALS Command Language Processor, ALS Configuration Control Tools, ALS Database Manager, KAPSE, and ALS Compiler Machine-Independent Section shall be as target independent as possible. The code generators, runtime support libraries, assemblers, linkers, and loaders (if required) shall be target dependent. However, the code generators, assemblers, and linkers shall be independent of changes in the target operating system.

3.2.1.1.2 Rehostability. - The Ada Language System shall be designed for rehostability. A design objective shall be the concentration of all host dependencies in the KAPSE and the runtime support libraries. The tools in the ALS shall interact with the host operating system through the KAPSE (the DEC VAX/VMS Text Editor and the DEC Standard Runoff Formatter excepted). This relationship is displayed in Figure 3-2, which shows that the KAPSE interfaces the user and the toolset to the host operating system.

All ALS programs shall be written in Ada except for small subprograms which, for reasons of algorithm or efficiency, cannot be conveniently expressed in Ada* (DEC VAX/VMS Text Editor and the DEC Standard Runoff Formatter excepted). The Text Editor and the Formatter shall be invoked by the ALS Command Language Processor.

*Government approval shall be required for any subprogram to be written in a language other than Ada.

3.2.1.1.3 User Portability. - A host- and target-independent command language shall be used to provide a consistent user interface.

3.2.1.2 Extensibility. - The Ada Language System shall provide a framework for the addition of new tools. Tools may be built in Ada or in the ALS command language. The ALS shall be used to build these tools. New tools shall access the environment database and shall be invoked through the ALS Command Language Processor in the same manner as existing tools. The environment database shall be extensible for user-defined attributes and associations.

3.2.1.3 Programming Support. - User helpfulness and human engineering are key objectives in the design of the Ada Language System. Areas in which the system will provide programming support are described in the following subparagraphs.

3.2.1.3.1 Support of Concurrent Multiple Development. - The ALS shall be designed to provide programming support for the simultaneous development of multiple Ada programs in a concurrent user access environment.

3.2.1.3.2 Programming Teams. - The ALS shall support program development by teams of programmers, each developing parts of a single program, by providing:

a. Concurrent access to the system by multiple users,

b. Protection mechanisms to prevent unauthorized access to information, and

c. Protection mechanisms to prevent accidental simultaneous modification of a node by multiple users.

3.2.1.4 Development of Program Families. - The ALS shall support the development of program families that share common, or similar, compilation units. This support shall include database support of both revisions and variations of compilation units. Revisions are defined as modified copies of previous information in the database, superseding earlier data; variations are defined as modified copies of previous information, not superseding the earlier data. (See 6.1 for glossary of terms.)

3.2.1.5 Ada Programs. - The ALS shall support the construction of Ada programs that contain, in addition to compilation units written in Ada, compilation units written in other languages as follows:

a. For all target environments, library subprogram bodies written in the assembly language for that target.

3.2.1.6 Command Language. -

3.2.1.6.1 Command Language Function. - A command language shall provide a uniform interface between the user and the tools in the ALS. The command language shall be interpreted by an ALS Command Language Processor tool.

3.2.1.6.2 Command Language Description. - A detailed description of the ALS command language is included in Appendix 60 to this specification.

3.2.1.7 Environment Database. - There shall be one database, called the environment database, that serves as the repository for all information stored in the ALS. The environment database shall store information such as:

a. Ada source text,

b. Assembly language source text,

c. Machine-code representations of programs,

d. Test data,

e. Log files,

f. Statistics,

g. Documentation, and

h. Relationships among programs and compilation units (e.g., dependence relations, revisions, and variations).

A detailed description of the environment database is provided in Appendix 50 to this specification.

3.2.1.7.1 Nodes. - The objects in the environment database shall be called nodes. There shall be three basic types of nodes: files, directories, and variation headers. All nodes shall have properties called attributes and associations. Attributes are named properties with character string values. Associations are named properties with values that are collections of "p nters" to other nodes.

In addition to attributes and associations, file nodes shall contain a data portion that may be read and written by Ada programs via the standard Ada I/O packages, INPUT_OUTPUT and TEXT_IO. As is common practice, the

data portion of file nodes shall not have type. Ada programs interpret the data as a sequence of values of some type when an Ada file is associated with an ALS file.

Directory nodes shall be used to name and group other nodes. When a node is created, it shall be created "within" a directory. In addition to attributes and associations, each directory shall contain a specification of the nodes grouped in it.

Directories shall be organized as directed acyclic graphs. Files shall be leaves in these structures. The structure is described further in Appendix 50.

3.2.1.7.2 Program Libraries. - Program libraries shall contain all of the information necessary to support the separate compilation capability of Ada, to perform partial or complete link-edits of Ada programs, to incorporate routines written in other languages into Ada programs, and to specify program structure to analysis and debugging tools.

A program library is a subtree of the directory hierarchy. The root of the subtree is a directory with a category of program_library. Access to this directory and to all nodes below it is restricted. Users may only create and use program libraries with tools created for that purpose. They may, however, delete an entire program library.

3.2.1.7.3 Containers. - The files within a program library shall be called Containers. A Container includes specification of externally visible Ada names, statistics, object code, and other information. The compiler shall create one Container for each compilation unit that it compiles. Assemblers, linkers, and importers shall each create a single Container.

A full history of successful compilations shall be maintained in a program library. When a compilation unit is recompiled into a program library, a new revision of its Container is created; the old Container is not replaced by the new.

Containers are created and accessed through the services of the Container Data Manager (CDM) part of the Database Manager Functional Area.

3.2.1.8 ALS Toolset. -

3.2.1.8.1 Tool Functions. - A comprehensive set of tools for entering text, translating, executing, and debugging programs, and manipulating the database shall be included in the ALS.

3.2.1.8.2 Tool Descriptions. - Detailed descriptions of each tool to be provided in the ALS are included in Appendix 70 to this specification.

3.2.1.9 Reliability. - Ada Language System reliability shall be based on the use of modern software engineering techniques in system design and implementation, and frequent internal reviews of design progress. The software engineering techniques, such as structured programming, top-down design, and use of a program design language are noted in 3.3.8. The internal reviews are listed below.

3.2.1.9.1 Project Reviews. - Formal and informal project reviews will be held frequently to monitor and enhance program quality and reliability. Formal progress reviews will be held periodically; informal reviews will be held as the need arises.

3.2.1.9.2 Management Reviews. - Reviews of the project by a contractor management review team will be held monthly to monitor progress and to ensure the quality and reliability of the ALS.

3.2.1.9.3 Structured Walk-Throughs. - Structured walk-throughs will be held for all major sections of ALS design. Structured walk-throughs provide peer review of evolving design, and are oriented toward raising issues while the design can be influenced.

3.2.1.10 Maintainability. -

3.2.1.10.1 Maintainability Characteristics. - Maintainability is one of the key objectives of ALS development and design. Development of code that is easy to understand and modify shall be emphasized. Extensive in-line commenting shall be included. The software structure shall consist primarily of basic building blocks separated by well-defined interfaces permitting the rapid isolation of problems.

3.2.1.10.2 Personnel. - The ALS shall be designed to be maintained by experienced system programmers trained in the operation and maintenance of host and target machine software, and in the operation of the ALS. Maintenance of the ALS at the SDSS facility, distribution of updated and revised information to the field installations, and related training of both new and previously assigned personnel will require, as a minimum:

a. 3 Maintenance Programmers,

b. 2 Technical Aides (to copy and distribute tapes), and

c. 1 Instructor.

3.2.1.10.3 Documentation. - As shown in 3.4, extensive maintenance documentation shall be supplied with the ALS.

3.2.2 Physical Characteristics. - The ALS requires the physical characteristics of the generic minimum host environment as specified in 3.1.1.1.2.

3.2.3 Reliability. - The reliability characteristics of the ALS are described in 3.2.1.9.

3.2.4 Maintainability. - The maintainability characteristics of the ALS are described in 3.2.1.10.

3.2.5 Availability. - The Ada Language System will be available for developmental use in the host environment specified in 3.1.1.1 and the target environments specified in 3.1.1.2 in accordance with the schedule presented in the Ada Language System Design and Development Plan (2.1).

3.2.6 System Effectiveness Models. -

This paragraph is not applicable to this specification.

3.2.7 Environmental Conditions. - The Ada Language System shall operate in the physical environments (e.g., temperature, humidity, etc.) of any host machine specified in 3.1.1.1 and any target machine specified in 3.1.1.2.

3.2.8 Nuclear Control Requirements. -

This paragraph is not applicable to this specification.

3.2.9 Transportability. -

3.2.9.1 User/System/Tool Transportability. - The Ada Language System will be designed for user, system, and tool portability as specified in 3.2.1.1.

3.2.9.2 Output Program Transportability. - The executable program image produced by the Ada Language System may be transported on any conventional computer system medium compatible with both the host and target systems such as tape or disk. The media used for transportation to each target are defined in Appendix 40.

3.2.9.3 Distribution. - The ALS shall be transportable by disk or tape between identical host system configurations, and upward-compatible host system configurations that include the facilties and functions required to support all ALS functions and operations as described in this specification.

3.3 Design and Construction. - The design and construction of the Ada Language System shall be in accordance with the requirements of the Ada Language System Design and Development Plan specified in 2.1. As described in the plan a set of design and development standards will be established at the start of the project to ensure that the design and implementation efforts proceed in a manner consistent with modern programming practices and the contractor's quality standards. The intent of these standards will be to provide a framework for design and development; thus, the standards and conventions will not be viewed as rigid rules never to be violated, but rather as guidelines which represent good engineering practice.

3.3.1 Materials, Processes, and Parts. - The materials and parts used in the development, design, documentation, testing, qualification, operation, and maintenance of the Ada Language System are high-quality, commercially-available computer supplies such as disk packs, magnetic tape cartridges, printer ribbons, and paper. No strategic or critical materials will be required.

3.3.2 Electromagnetic Radiation. - When installed in a host or target environment the ALS assumes the electromagnetic radiation characteristics of the host or target environment.

3.3.3 Nameplates and Product Marking. - The computer tape reels containing the Ada Language System shall be labeled with the following identifying information: the system name, contents, contract number, serial number, date, and the name of the responsible agency (U.S. Army CECOM, Ft. Monmouth, N.J.).

3.3.4 Workmanship. - The workmanship in the Ada Language System shall be in accordance with the contractor's quality standards described in the ALS Design and Development and the ALS Quality Assurance Plans specified in 2.1.

3.3.5 Interchangeability. - Portability of users, systems, and tools shall be a primary objective of the Ada Language System as specified in 3.2.1.1.

3.3.6 Safety. - The Ada Language System will be free from harmful effects on personnel and equipment. No special safety precautions are required in its handling, transportation, or use.

3.3.7 Human Performance/Human Engineering. -

3.3.7.1 Programming Environment. - The Ada Language System shall be designed to provide a beneficial, efficient, flexible, easy-to-use environment for programming in Ada.

3.3.7.2 Command Language Tool Set. - The ALS command language and tool set allows Ada programmers to move easily and conveniently across host boundaries.

3.3.7.3 Extensibility. - The ALS shall provide a framework for the addition of new tools and services at any time. Capabilities for building new tools in Ada on the ALS will be provided.

3.3.7.4 Programming Support. - The ALS provides programming support for:

a. Programming in Ada,

b. Simultaneous development of multiple Ada programs in a concurrent user access environment,

c. Program development by teams of programmers, and

d. Development of program variations and revisions.

3.3.7.5 Prevention of Error Cascading. - The ALS shall be designed so that error cascading will be minimized.

3.3.7.6 Messages and Diagnostics. - Messages and diagnostics shall be unique, complete, and uniform. Appendix 80 lists all the messages and diagnostics for each of the ALS tools.

3.3.8 Computer Programming. - All Ada Language System programs shall be written in Ada as specified in 3.2.1.1.2. Modern software engineering methodologies shall be used in ALS design and implementation. Some of these techniques are:

3.3.8.1 Top-Down Design. - Top-down design can be described as "allocating functional and performance requirements to a modular software structure." This characterization implicitly stresses the importance of detailing and baselining requirements prior to design. The design effort is driven by the requirements, and must clearly trace back to them.

A top-down approach requires that the procedural and data structures be determined before the algorithmic and data layout details are specified. Also, the interfaces between modules must be defined and controlled so that the design of module internals can proceed in parallel with confidence.

3.3.8.2 Structured Programming. - Structured programming contributes to the development of reliable, maintainable software. In structured programming, program design and coding are performed in accordance with standards that limit the form or structure of code to certain basic building blocks, require certain mnemonic aids to readability, and impose other constraints on program complexity.

3.3.8.3 Coding Standards. - The Ada coding effort shall follow a documented set of standards and conventions that support the development of structured, maintainable code. The objectives of these standards are:

a. To contribute to readability and maintainability of code,

b. To prohibit the use of error-prone constructs, and

c. To establish a uniform appearance of code produced by different individuals.

Use of coding standards will be monitored by peer code reading and by the project quality assurance monitoring.

3.3.8.4 Program Design Language (PDL). - A Program Design Language (or "pseudo-code") shall be used to document complex control flow logic. Moreover, the flow-of-control constructs, such as IF-THEN-ELSE in the PDL, will be exactly the same as those available in the Ada Language. This means that the match between the documentation of a routine (in PDL), and the code for the routine will be more immediate than is possible with flowcharts. Program Design Language used in the ALS shall be documented in the ALS Design and Development Plan (2.1).

3.3.8.5 Implementation Approach. - The Ada Language System implementation approach is based on proven techniques used by the contractor in the development of other high-order language compilers. Structured programming techniques shall be carried through from the design phase. Careful attention to checkout and testing will assure compliance with system requirements. Coding standards shall be monitored by frequent reviews.

3.4 Documentation. - The following documentation shall be supplied:

3.4.1 Plans. -

a. Ada Language System Design and Development Plan,

b. Ada Language System Configuration Management Plan, and

c. Ada Language System Quality Assurance Plan

3.4.2 Manuals. -

a. A User Reference Manual for each target environment specified in Par. 3.1.1.2;

b. An Operator's Manual for each target environment specified in Par. 3.1.1.2;

c. A Retargeting Manual that describes the procedures for retargeting the ALS; and

d. An Intermediate Language Specification that shall describe the intermediate language and provide interface data for the development of code generators to be used in the Ada Language System.

3.4.3 Specifications. -

a. Ada Language System Specification,

b. A B5 Computer Program Development Specification for each CPCI listed in 3.1.1.5 (see Figure 3-3), and

c. A C5 Computer Program Product Specification for each CPCI listed in 3.1.1.5 (see Figure 3-3).

3.4.4 Test Plans/Procedures and Reports. -

a. A Preliminary/Formal Qualification Test Plan for the Ada Language System,

b. A Preliminary Qualification Test Procedure for each CPCI listed in 3.1.1.5,

c. A Preliminary Qualification Test Report for each CPCI listed in 3.1.1.5,

d. A Formal Qualification Test Procedure for each target configuration in the Ada Language System, and

e. A Formal Qualification Test Report for each target configuration in the Ada Language System.

3.4.5 Technical Reports. -

a. Technical Report on Language Efficiency Issues,

b. Technical Report on Relationship between the ALS Intermediate Language and DIANA, and

c. Design Data Book, Engineering Data.

3.4.6 Configuration Management Documents. -

a. Configuration Index Reports,

b. Version Description Documents, and

c. Change Status Reports.

3.4.7 Administrative Documents. -

a. Monthly Status Reports, and

b. Monthly Cost/Schedule Status Reports.

3.4.8 Blank. -

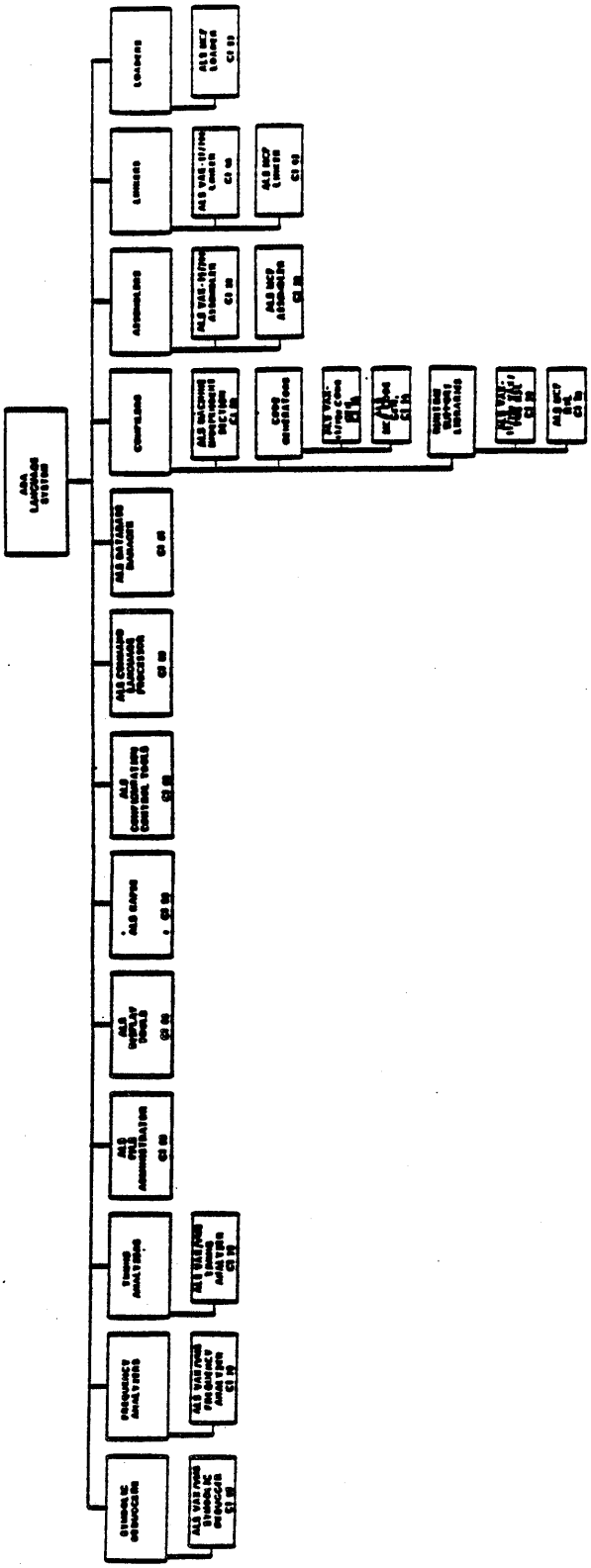


Figure 3-3. Ada Language System Specification Tree

3.5 Logistics. -

3.5.1 Maintenance. - As described in 3.2.1.10, maintainability is one of the key objectives of Ada Language System development and design. No special test equipment will be required.

3.5.1.1 Maintenance Procedures. - Maintenance of the ALS on the SDSS facility will be performed through a central maintenance operation. All problem reports will be routed through the central maintenance operation. Verification of fixes will be performed by use of the SDSS facility (with on-site personnel). The ALS shall be placed in its own database, and will be used to maintain itself.

3.5.1.2 Maintenance Documentation. - As shown in 3.4, extensive maintenance documentation shall be supplied with the ALS.

3.5.2 Supply. -

3.5.2.1 Distribution. - Distribution of the ALS and location of ALS supplies will be at the discretion of the government.

3.5.2.2 Addition of New Tools. - The ALS shall be designed to permit new tools to be built in Ada or in the command language on an ALS. The presence of user-supplied tools should not degrade the operation of the existing tools.

3.5.3 Facilities and Facility Equipment. - The ALS development model will be hosted on the SDSS facility at Fort Monmouth, New Jersey, a concurrent multiple-user access environment. Installation of the ALS will not prevent normal use of the SDSS facility by other users. Access rights to the ALS from the SDSS facility will be determined by the SDSS facility operation. Security of the ALS shall be equivalent to the security of the host system as described in 3.1.3.2.

3.6 Personnel. -

3.6.1 User Personnel. - The Ada Language System shall be designed to provide a friendly, flexible, easy-to-use environment for programming in Ada. No special programming experience will be required. Any programmer familiar with the Ada language will have the ability to learn to use the ALS by reading and understanding the applicable ALS Users Reference Manual.

3.6.2 Maintenance Personnel. - The ALS is designed to be maintained by experienced system programmers trained in the operation and maintenance of host and target machine software, and in the operation of the ALS. Maintenance of the ALS at the SDSS facility, distribution of updated and revised information to the field installations, and related training of both new and previously assigned personnel will require, as a minimum:

a. 3 Maintenance Programmers,

b. 2 Technical Aides (to copy and distribute tapes), and

c. 1 Instructor.

3.6.3 Training. - It will be desirable to provide a training course for all programmers planning to use the ALS.

3.7 Functional Area Characteristics. -

This paragraph describes the characteristics of each functional area in the Ada Language System.

3.7.1 Compiler Functional Area. - The compiler shall be a tool that receives source text for an Ada language compilation, checks the text for compliance with the Ada language definition, and, if the text is correct, translates it into equivalent machine code instructions. The characteristics of the compiler shall be as follows:

a. The compiler shall consist of an ALS Compiler Machine-Independent Section, code generators for each target environment, and runtime support libraries for each target environment.

b. The compiler machine-independent section and the code generators shall be reentrant.

c. The compiler shall accept the full Ada language as specified in the Military Standard, Ada Programming Language, ANSI/-STD-1815A-1983, 17 February 1983 (2.1), and shall be capable of generating code for each of the target machines specified in 3.1.1.2.

d. The compiler shall not generate code for compilation units which do not comply with the definition of the Ada Language; all lexical, syntactic, and semantic deviations from the definition shall be diagnosed, including those requiring type-checking across compilation units. Appendix 80 is a complete summary of all diagnostic messages.

e. As a design goal, the minimum performance of the compiler should be on the order of 1000 lines per minute of real time. As an additional objective, the compiler should be capable of compiling itself at this rate.

f. The code generators shall be capable of producing code for distributed processing when suitable language usage is imposed (e.g., use of entry calls for communicating between processors). The interface to the runtime support for the communications package for interprocess communications shall be defined and implemented. However, the actual communications package shall not be implemented since the communication link is target system architecture specific.

g. As a design goal, the compiler shall attempt to limit error propagation, or error cascading, that occurs when the compiler misinterprets and produces a diagnostic message for one portion of source text because of a user error in another portion of source text.

3.7.1.1 Compiler Invocation. - The compiler shall be invoked by an ALS command that has the following format:

```
tool source prog_lib [NEW_SRC => out_src] [OPT => option_list]
```

tool	is the name of the tool that performs the compilation, e.g., ADAVAX. (See Appendix 70 for list of tools.)
source	is the name of the file node containing the source text to be compiled.
prog_lib	is the name of the program library into which the Containers generated by this compilation will be placed.
out_src	is the file node that is to receive the reformatted source text. The NEW_SRC parameter has no effect if the REFORMAT option is not in effect.
option_list	is a list of the options that are in effect for this compilation. The available options are listed in 3.7.1.1.1.

Except for the diagnostic summary listing, all compiler listings shall be routed to the standard output file. The pre-defined logical files in the ALS opened when a tool is invoked are described in 3.1.1.6 and Appendix 60. The diagnostic summary listing shall be routed to the message output file.

3.7.1.1.1 Compiler Options . - The options that may be specified to the compiler are listed in the following subparagraphs. Each option may be specified as shown, or may be preceded by the three characters NO_ to specify the opposite option. For example, SOURCE turns Source Listing on while NO_SOURCE turns Source Listing off.

The compiler produces a diagnostic of severity level WARNING if any of the following conditions are encountered during the processing of options:

- The complement of an option already specified is specified. The first option will be ignored. For example, if NO_SOURCE is specified, then SOURCE is specified later in the option list, SOURCE will be in effect;
- An option already specified is re-specified. The first option will be ignored; or
- An undefined option is specified.

There is no examination of options to determine whether redundant combinations of options are specified. For example, specifying both NO_SOURCE and NO_PRIVATE will not result in a diagnostic.

3.7.1.1.1.1 Listing Control Options. -

SOURCE	Produce a listing of the source text. Default: SOURCE
PRIVATE	If there is a source listing, text in the private part of a package specifier is to be listed, subject to requirements of LIST pragmas. Default: PRIVATE
NOTES	Include diagnostics of severity NOTE in the source listings, and in the diagnostic summary listing. Default: NO_NOTES
ATTRIBUTE	Produce a symbol attribute listing. Default: ATTRIBUTE
XREF	Produce a cross-reference listing. Default: NO_XREF
STATISTICS	Produce a statistics listing. Default: NO_STATISTICS
MACHINE	Produce a machine code listing if code is generated. Code is generated when CONTAINER_GENERATION is in effect and there are no diagnostics of severity ERROR, SYSTEM, or FATAL, and, if there are diagnostics of severity level WARNING, CODE_ON_WARNING is in effect. If a machine code listing is requested, and no code is generated, a diagnostic of severity NOTE is reported. Default: NO_MACHINE
DIAGNOSTICS	Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS

3.7.1.1.1.2 Maintenance Aid Options. -

COMPILER_MAINT — Permit the other maintenance aid options to have an effect. This option must appear before any of the other maintenance aids in order for them to have an effect. Default: NO_COMPILER_MAINT

SAVE_CONTAINER_nn

Save the state of the Container in a temporary file in the current directory. The file is identified as TEMPILnn. "nn" indicates to what portion of the compilation process the option applies. The Container is saved after that portion of the compilation process. "nn" has the following possible values and meanings:

01	Parsing
02	Context processing
03	Name resolution
04	Overloading resolution
05	Static expression evaluation
06	Statement checking
08	Diana expansion
14	Code generator initial translation
20	Data Collection
21	Inter Procedural Analysis
22	Forward Optimizations
23	Backward Optimizations
30	Code generator secondary translation
31	Code generator tree walk
32	Final optimization
33	Machine text formatting

STOP_CONTAINER_nn — Operates exactly as SAVE_CONTAINER_nn, except that the compilation process halts immediately after the "nn" portion of the compilation process is complete.

USE_CONTAINER_nn — Use the Container as saved in a temporary file by the SAVE_CONTAINER_nn option in the current directory identified as TEMPILnn to restart compilation where it was left off. Earlier phases are skipped. USE_CONTAINER is not valid for "nn" values of 02, 20, 21, and 22.

FLAGS_nn_string — "string" is of the form (a|b|...|z)+. This option specifies options that have effect during the specified portion of the compilation process. The meaning of the string is shown in Table 3-1. Values of "nn" for which no string is shown have no relevant options defined. The interpretation

of the characters is dependent on the value of "nn". "nn" is as defined for SAVE_CONTAINER_nn with the following additions:

00	Container Data Manager
34	Program Library Manager
35	Constant expression evaluation
36	Other utilities and compiler control function
37	Reformatter

STMT_nnnn — Maintenance aids traces should apply for statement nnnn. Statement number "nnnn" is right adjusted and padded to the left with zeros, if necessary. Optionally supported by the different phases.

STMT_RANGE_nnnn_mmmm — Maintenance aids traces should apply within the statement range nnnn through mmmm. Statement numbers "nnnn" and "mmmm" are right adjusted and padded to the left with zeros, if necessary. Default: All Statements.

STANDARD_COMPILE — Compile a new version of the predefined package STANDARD.

Table 3-1

COMPILER FLAG STRINGS

FLAGS_00_

FLAGS_01_ Parser

a	Display parser first transition table
b	Display lexer input buffer
c	Display lexer current character as read from text
e	Display parser current token (and its numeric code)
f	Display parser forest of Diana trees
h	Display lexer hash table
i	Display lexer contents of include stack
k	Display lexer token and hash key
m	Display parser transition table
n	Display parser nset table
o	Display lexer lookahead character
p	Display parser production table
r	Display lexer token and its lower case representation
s	Provide a trace of the syntactic and semantic stacks
t	Display lexer token and its type
u	Display parser ls table
v	Display parser lset table
w	Display parser reduction length table
x	Display parser left hand side table
y	Display parser parse table
z	Display parser first lookahead table

FLAGS_04_ Overloading Resolution

a	Trace of all context routines.
b	Trace of all expression routines with values of in parameters on entry and out parameters on exit.
c	Trace of OVERLOAD and OVERLOADRNG routines only.
d	Print candidate lists during expression trace.

FLAGS_05_

FLAGS_06_ Statement Checking

a	Trace entire traversal, printing context parameters.

Table 3-1 (Cont.)

FLAGS_08_		
FLAGS_14_		
FLAGS_20_		Global Optimizer
	a	All optimizer transformations (b through j)
	b	Subprogram information table before interprocedural analysis
	c	Common subexpression table
	e	Subprogram information table after interprocedural analysis
	g	Next-use information table
	j	variable life overlab dump
FLAGS_21_		
FLAGS_22_		
FLAGS_23_		
FLAGS_30_		
FLAGS_31_		
FLAGS_32_		
FLAGS_33_		
FLAGS_34_		
FLAGS_35_		
FLAGS_36_		Control Function
	f	Skip all back-end processing
	p	Turn on PLM maintenance flags
		Diagnostic Recorder
	t	Display diagnostic information at terminal
FLAGS_37_		Reformatter
	c	Display reformatter current column postion
	s	Display reformatter contents of column stack
	t	Display reformatter token type

3.7.1.1.1.3 Other Options. -

CODE_ON_WARNING — Generate code (and, if requested, a machine code listing) when there are diagnostics of severity level WARNING, provided there are no FATAL, SYSTEM, or ERROR diagnostics. NO_CODE_ON_WARNING means generate no code (and, if requested, no machine code listing) when there are diagnostics of severity level WARNING. Default: CODE_ON_WARNING

CONTAINER_GENERATION — Produce a Container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. If a Container is not produced because NO_CONTAINER_GENERATION is in effect, code is not generated (nor is a machine code listing, if requested). Default: CONTAINER_GENERATION

FREQUENCY — Permit generation of code to monitor execution frequency at the basic block level. Default: NO_FREQUENCY.

OPTIMIZE — Permit optimization in accordance with the OPTIMIZE pragmas that appear in the text. When NO_OPTIMIZE is specified or is in effect by default, no optimization is performed, regardless of pragmas. When no optimize pragmas are included, optimization tries to conserve code space. (Note: With or without optimization, the compiler shall conform to the Military Standard, Ada Programming Language, ANSI/-STD-1815A-1983, 17 February 1983 (2.1).) Default: NO_OPTIMIZE

REFORMAT — Reformat the source, the result being reflected in the source listing, if present, and the out_src node, if specified. Default: REFORMAT

TRACE_BACK — Provides user with calling sequence traceback information when the user's program is aborted because of an unhandled exception. See Section 40.1.3 for a discussion of tracebacks. Default: TRACE_BACK.

3.7.1.2 Compiler Inputs. - The compiler shall receive control inputs and option inputs through the ALS Command Language Processor; Ada source text compilations from the environment database through the ALS Database Manager; and Containers containing previous compilations through the ALS Database Manager.

3.7.1.3 Compiler Outputs. -

3.7.1.3.1 Container Output. - The compiler shall produce a Container output for each input compilation unit depending on options and diagnostics. The Containers shall include machine text, global symbol definitions, and global symbol references. The Containers shall be stored in the environment database. A Container will not be produced if NO_CONTAINER_GENERATION is in effect or if a diagnostic of severity level FATAL is reported.

3.7.1.3.2 Reformatted Source Text Output. - Under user option, the compiler shall produce a text node containing the reformatted Ada source text. This node shall be stored in the environment database.

3.7.1.3.3 Output Listings . - The compiler shall produce the following output listings. If a FATAL diagnostic occurs, a Container may not be produced, depending on the precise nature of the FATAL diagnostic. In these cases, if a Container is not produced, listings shall not be produced.

a. Source Listing,

b. Symbol Attribute Listing,

c. Cross-Reference Listing,

d. Compilation Statistics Listing,

e. Machine Code Listing,

f. Diagnostic Summary Listing, and

g. Compilation Summary Listing.

Each listing is described in the following paragraphs.

3.7.1.3.3.1 Source Listing. -

a. Description. The source listing shall show the Ada language statements that the compiler has received for compilation. The listing shall be requested or suppressed with the SOURCE option. The following information shall be included:

- The source statements.

- Diagnostic messages, describing errors or other unusual circumstances. Each message appears in the listing immediately within or following the diagnosed statement. Diagnostic messages that are not associated with particular statements are also included in this listing. This includes messages generated during processing of the arguments to the compiler. These messages appear prior to the start of the listing of the source statements. The messages are described in 3.7.1.3.4.

- Line numbers.

- Statement numbers. A compilation unit is decomposed into a series of "statements", each of which represents a syntactic construct or a fragment of a syntactic

construct. Each "statement" of a compilation unit receives a "statement number" that uniquely identifies that "statement" within the unit. A "statement" is terminated by the appearance of another "statement" or by the end of the compilation unit.

A syntactic construct may define several "statements" because: (1) it contains nested constructs that in turn define one or more "statements", or (2) it is broken up for convenience into several "statements" that may be individually accessed. Each of the following syntactic constructs represents one or more "statements" of a compilation unit.

1. A pragma;

2. An object or number declaration;

3. A type or subtype declaration;

4. The "record" and "end record" clauses of a record type definition;

5. The component list "NULL;";

6. A component declaration;

7. A discriminant declaration;

8. The "case", "when", and "end case" clauses of a variant part;

9. An incomplete type declaration;

10. A simple statement;

11. The "if", "elsif", "else" and "end if" clauses of an if statement;

12. The "case", "when", and "end case" clauses of a case statement;

13. The "while", "for", "loop" and "end loop" clauses of a loop statement;

14. The "declare", "begin", "exception", and "end" clauses of a block;

15. The "procedure" and "function" clauses of a subprogram specification, and the "return" clause of a function specification;

16. The parameter declarations of a formal part;

17. The "begin", "exception", and "end" clauses of a subprogram body, as well as the clauses of its subprogram specification as previously defined;

18. The "package", "private", and "end" clauses of a package specification;

19. The "package body", "begin", "exception", and "end" clauses of a package body;

20. A use clause;

21. A renaming declaration;

22. The "task" or "task type" and "end" clauses of a task specification;

23. The "task body", "begin", "exception", and "end" clauses of a task body;

24. An entry declaration;

25. The "accept", "do", and "end" clauses of an accept statement;

26. The "select", "when", "or", "else", and "end select" clauses of a select statement;

27. The "terminate" form of select alternative of a selective wait statement;

28. A with clause;

29. A body stub;

30. An exception declaration;

31. The "when" clause of an exception handler;

32. The "generic" clause of a generic part;

33. A generic formal parameter;

34. A generic subprogram or package instantiation;

35. A representation specification;

36. The "for", "record", and "end record" clauses, and the "component name, location" pairs of a record type representation.

Block identifiers, loop identifiers, and labels are considered part of the statement that they precede.

- An indication of block depth, where the block level changes whenever the current scope changes. For example, the block level is incremented whenever a package, subprogram, task body, or block statement is encountered; the block level is decremented after an end statement for a package, subprogram, task body or block statement.

  Depending on the presence of other pragmas and options,

the complete source text may or may not be listed. The complete source text is defined as the contents of the "source" node, together with the diagnostic messages, line numbers, statement numbers, and block depth indicators which apply to this text. The following rules determine the portion of the complete text that appears in the listing:

- Any text between a LIST(OFF) pragma and the next following LIST(ON) pragma is not listed. (Note: The rule applies to any pragmas in the complete text, even those appearing in imported text. The rule is not affected by other options such as NO_PRIVATE.)

- If the NO_PRIVATE option is in effect, the private parts of package specifications do not appear in the listing.

- If the NO_NOTES option is in effect, diagnostic messages of severity NOTE do not appear in the listing.

Under user option, the source listing shall show the Ada source text arranged either into lines and columns exactly as it appeared in the input, or as a text that is reformatted in a manner that improves readability and displays the structure of the Ada text. A reformatted listing shall have statements separated one to a line and indented to show nesting depth; comments shall be segregated from statements and declarations. Subparagraphs b. through f. below describe the reformatting process more fully.

If the source text to be listed (as determined above) contains any PAGE pragmas, the line on which the token PAGE appears is placed at the first line available on a new page in the source listing.

Pragma TITLE (arg) specifies a CHARACTER string that is to appear on the second line of each page of every listing produced for a compilation unit. At most one such pragma may appear for any compilation unit, and it must be the first lexical unit in the compilation unit (comments excepted). The argument is a CHARACTER string.

If the listing contains more than one compilation unit, each compilation unit begins a new page. For the purposes of listings, a new compliation unit begins with the first non-comment token or blank line that follows the final semicolon of the previous compilation unit.

All listing control pragmas and options shall preserve statement

numbering and nesting level indications. Every statement, whether listed or not, shall be counted in the determination of statement numbers. All lines shall be counted for line numbering, whether printed or not, except that included text shall be numbered independently of the enclosing text. Reformatting shall produce new line numbers.

All listings shall be printed with a maximum of 120 columns per line and 60 lines per page.

b. Reformatting. When the REFORMAT option is specified, the source listing shall contain a reformatted version of the source as described in Section 3.7.1.3.3.1(a). The characteristics of the reformatted output are described in Subparagraphs c., d., and e. below.

A reformatted source text node shall be produced under user option.

c. Examples. Figure 3-4 shows examples of source listings under various combinations of options.

```
      LINE  BLOCK LEVEL  STMT NUM

         1                    -- This is the package body to implement the package RATIONAL_NUMBERS
         2
         3          1         1    with EUCLID;
         4          1         2    package body RATIONAL_NUMBERS is
         5          1         2
         6          1         2
         7          1         2
         8          1         2    -- SAME_DENOMINATOR is a local procedure at module scope which
         9          1         2    --  reduces two rational numbers to the same denominator
        10          1         3    pragma OPTIMIZE (TIME);
        11          2         4       procedure SAME_DENOMINATOR (X, Y: in out RATIONAL) is
        12          2         6          COMMON: INTEGER range 1 .. INTEGER'LAST;
        13          2         7       begin
        14          2         7                                              -- reduces X and Y to the same
        15          2         8          COMMON := 1;                        -- denominator

***E            2033  SYNTAX ERROR - '=' - EXPECTING ':=' OR NAME QUALIFIER
***N            2001  SYNTAX RECOVERY - PARSING RESUMED AT ';'
        16          2         9             if X.DENOMINATOR /= Y.DENOMINATOR then
        17          2        10                COMMON := EUCLID.LCF (X.DENOMINATOR, Y.DENOMINATOR); -- least common factor
        18          2        11                COMMON := X.DENOMINATOR*Y.DENOMINATOR/COMMON; -- smallest common multiple
        19          2        11
        20          2        12                X.NUMERATOR := X.NUMERATOR*COMMON/X.DENOMINATOR;
        21          2        13                Y.NUMERATOR := Y.NUMERATOR*COMMON/Y.DENOMINATOR;
        22          2        13
        23          2        14                X.DENOMINATOR := COMMON;
        24          2        15                Y.DENOMINATOR := COMMON;
        25          2        16             end if;
        26          2        16
        27          2        17       end SAME_DENOMINATOR;
        28          1        17
        29          2        18       function EQUAL (X, Y: RATIONAL) return BOOLEAN is
        30          2        21          U, V: RATIONAL;
        31          2        22       begin
        32          2        23          U := X;                             -- save in temporaries
        33          2        24          V := Y;
        34          2        25          SAME_DENOMINATOR (U, V);
        35          2        26          return (U.NUMERATOR = V.NUMERATOR);
        36          2        27       end EQUAL;
        37          1        27
        38          2        28       function "+" (X, Y: RATIONAL) return RATIONAL is
        39          2        31          U, V: RATIONAL;                     -- temporaries
```

Figure 3-4. Source Listing (Page 1 of 7)
Options: SOURCE, REFORMAT, NOTES

```
      BLOCK  STMT
LINE  LEVL   NUM

 40      2    32        begin
 41      2    33           U := X;
 42      2    34           V := Y;
 43      2    35           SAME_DENOMINATOR (U, V);
 44      2    36           return (NUMERATOR => U.NUMERATOR * V.NUMERATOR, DENOMINATOR => U.DENOMINATOR
 45      2    36              );
 46      2    37        end "*";
 47      1    37
 48      2    38        function "*" (X, Y: RATIONAL) return RATIONAL is
 49      2    41        begin
 50      2    41
 51      2    42           return (NUMERATOR => X.NUMERATOR*Y.NUMERATOR, DENOMINATOR => X.DENOMINATOR
 52      2    42              *Y.DENOMINATOR);
 53      2    43
 54      2    43        end "*";
 55      1    43
 56      2    44     end RATIONAL_NUMBERS;
```

Figure 3-4. Source Listing (Page 2 of 7)
(Continuation of Options: SOURCE, REFORMAT, NOTES)

```
      BLOCK  STMT
LINE  LEVEL  NUM

   1                 -- This is the package body to implement the package RATIONAL_NUMBERS
   2
   3      1      1    with EUCLID;
   4      1      2    package body RATIONAL_NUMBERS is
   5      1      2
   6      1      2
   7      1      2
   8      1      2    -- SAME_DENOMINATOR is a local procedure at module scope which
   9      1      2    --  reduces two rational numbers to the same denominator
  10      1      3    pragma OPTIMIZE (TIME);
  11      2      4       procedure SAME_DENOMINATOR (X, Y: in out RATIONAL) is
  12      2      6          COMMON: INTEGER range 1 .. INTEGER'LAST;
  13      2      7       begin
  14      2      7                                          -- reduces x and y to the same
  15      2      8          COMMON := 1;                    -- denominator

***E    2033  SYNTAX ERROR - '=' - EXPECTING ':=' OR NAME QUALIFIER
  16      2      9          if X.DENOMINATOR /= Y.DENOMINATOR then
  17      2     10             COMMON := EUC_ID.LCF (X.DENOMINATOR, Y.DENOMINATOR); -- least common factor
  18      2     11             COMMON := X.DENOMINATOR*Y.DENOMINATOR/COMMON; -- smallest common multiple
  19      2     11
  20      2     12             X.NUMERATOR := X.NUMERATOR*COMMON/X.DENOMINATOR;
  21            13             Y.NUMERATOR := Y.NUMERATOR*COMMON/Y.DENOMINATOR;
  22            13
  23      2     14             X.DENOMINATOR := COMMON;
  24      2     15             Y.DENOMINATOR := COMMON;
  25      2     16          end if;
  26      2     16
  27      2     17       end SAME_DENOMINATOR;
  28      1     17
  29      2     18       function EQUAL (X, Y: RATIONAL) return BOOLEAN is
  30      2     21          U, V: RATIONAL;
  31      2     22       begin
  32      2     23          U := X;                         -- save in temporaries
  33      2     24          V := Y;
  34      2     25          SAME_DENOMINATOR (U, V);
  35      2     26          return (U.NUMERATOR = V.NUMERATOR);
  36      2     27       end EQUAL;
  37      1     27
  38      2     28       function "=" (X, Y: RATIONAL) return RATIONAL is
  39      2     31          U, V: RATIONAL;                 -- temporaries
  40      2     32    begin
  41      2     33       U := X;
  42      2     34       V := Y;
  43      2     35       SAME_DENOMINATOR (U, V);
```

Figure 3-4. Source Listing (Page 3 of 7)
Options: SOURCE, REFORMAT, NO_NOTES

```
LINE  BLOCK LEVEL  STMT NUM

 44        2        36          return (NUMERATOR => U.NUMERATOR * V.NUMERATOR, DENOMINATOR => U.DENOMINATOR
 45        2        36             );
 46        2        37       end "*";
 47        1        37
 48        2        38       function "*" (X, Y: RATIONAL) return RATIONAL is
 49        2        41       begin
 50        2        41
 51        2        42          return (NUMERATOR => X.NUMERATOR*Y.NUMERATOR, DENOMINATOR => X.DENOMINATOR
 52        2        42            *Y.DENOMINATOR);
 53        2        42
 54        2        43       end "*";
 55        1        43
 56        2        44    end RATIONAL_NUMBERS;
```

Figure 3-4. Source Listing (Page 4 of 7)
(Continuation of Options: SOURCE, REFORMAT, NO_NOTES)

```
      BLOCK  STMT
LINE  LEVEL  NUM

   1                  -- This is the package spec for Rational_Numbers
   2
   3
   4        1     1    package Rational_Numbers is
   5        1     1
   6        1     2    type
   7        1     2     Rational is private;
   8        1     2
   9        1     3    procedure same_denominator (x,y:in out rational);
  10        1     4
  11        1     5    function EQUAL(x,y:rational)return Boolean;
  12        1     8    function "+"(x,y:rational)return RATIONAL;
  13        1    11    function "*"(x,y:rational)return RATIONAL;
  14        1    13
  15        1    14    private type rational is record
  16        1    17    numerator,denominator:integer range 1..integer'last;
  17        1    18    end record;
  18        1    19    end Rational_Numbers;
```

Figure 3-4. Source Listing (Page 5 of 7)
Options: SOURCE, NO_REFORMAT, PRIVATE

```
      BLOCK  STMT
LINE  LEVEL  NUM

   1                 -- This is the package spec for Rational_Numbers
   2
   3
   4        1     1   package RATIONAL_NUMBERS is
   5        1     1
   6        1     2      type Rational is private;
   7        1     2
   8        1     3      procedure same_denominator (x, y: in out rational);
   9        1     4
  10        1     5      function EQUAL (x, y: rational) return Boolean;
  11        1     8      function "+" (x, y: rational) return RATIONAL;
  12        1    11      function "*" (x, y: rational) return RATIONA_;
  13        1    13
  14        1    14   private
  15        1    15      type rational is
  16        1    16         record
  17        1    17            numerator, denominator: integer range 1..integer'last;
  18        1    18         end record;
  19        1    19   end Rational_Numbers;
```

Figure 3-4. Source Listing (Page 6 of 7)
Options: SOURCE, REFORMAT, PRIVATE

```
      BLOCK  STMT
LINE  LEVEL  NUM

   1                 -- This is the package spec for Rational_Numbers
   2
   3
   4        1     1   package Rational_Numbers is
   5        1     1
   6        1     2      type Rational is private;
   7        1     2
   8        1     3      procedure same_denominator (x, y: in out rational);
   9        1     4
  10              5      function EQUAL (x, y: rational) return Boolean;
  11        1     8      function "*" (x, y: rational) return RATIONAL;
  12        1    11      function "*" (x, y: rational) return RATIONAL;
  13        1    13
  19        1    19   end Rational_Numbers;
```

Figure 3-4. Source Listing (Page 7 of 7)
Options: SOURCE, REFORMAT, NO_PRIVATE

d. Comments and Spacing. To highlight the logical structure of code, it is common for a programmer to set off functional units by block comments and blank lines. Additionally, the programmer may expand on particularly complex code fragments using in-line comments. The program reformatter supports these activities on several levels.

First, the program reformatter shall preserve blank lines from input to output, to help set off groupings.

Second, the program reformatter shall support two levels of comments: long comments used as descriptions of major functional units and short comments used for explanations of code fragments.

The first category, called a block comment, is recognized as:

- The first comments in a compilation unit before any code,
- Any comments directly following a blank line, or
- Any comments directly following a PAGE pragma or a TITLE pragma.

A block comment contains no extra indentation or reformatting. A block comment is terminated by the first lexical unit other than comments.

All other comments fit into the second category, termed "remarks". A remark will typically appear on the same line as the code that it follows, or following another remark. All remarks immediately following a code fragment shall be indented to start in Column 41, or with an intervening space if it follows a code fragment, whichever is greater. Any remarks not immediately following a code fragment shall start in Column 41, or one column past the column to which the text is currently being indented, whichever is greater. Comments which do not fit on a line shall be separated into two comments, divided at a blank, if possible.

e. Alignment and Indentation. Keywords shall be aligned as shown in Table 3-2. Their bodies shall be indented one indentation unit (three columns) from their corresponding headers.

Table 3-2

KEYWORD ALIGNMENT

```
ACCEPT ... DO
   ...
END ...;


block_identifier:
   BEGIN
      ...
   EXCEPTION
      ...
   END...;


CASE ... IS
   WHEN ... =>
      ...
END CASE;


block_identifier:
   DECLARE
      ...
   BEGIN
      ...
   EXCEPTION
      ...
   END...;


loop_identifier:
   FOR ... LOOP
      ...
   END LOOP ...;


FOR ... USE
   RECORD ...
      ...
   END RECORD;
```

Table 3-2 (cont.)

```
FUNCTION (...;  -- function specification
   ...;
   ...);

FUNCTION (...;
   ...;
   ...) ...IS

BEGIN
   ...
EXCEPTION
   ...
END ...;

IF ... THEN
   ...
ELSIF ... THEN
   ...
ELSE
   ...
END IF;

loop_identifier:
   LOOP
      ...
   END LOOP...;

PACKAGE ... IS
   ...
PRIVATE
   ...
END;

PACKAGE BODY ... IS
   ...
BEGIN
   ...
EXCEPTION
   ...
END ...;
```

Table 3-2 (cont.)

```
PROCEDURE (...; -- procedure specification
   ...;
   ...);

PROCEDURE (...;
   ...;
   ...) IS
   ...
BEGIN
   ...
EXCEPTION
   ...
END ...;

SELECT
   ...
OR
   ...
ELSE
   ...
END SELECT;

SEPARATE (...)
   ...
   ...;
TASK [TYPE] ... IS
   ...
END ...;

TASK BODY ... IS
   ...
BEGIN
   ...
EXCEPTION
   ...
END...;
```

Table 3-2 (cont.)

```
TYPE ... IS
   RECORD
      ...
   END RECORD;


loop_identifier:
   WHILE ... LOOP
      ...
   END LOOP...;


GENERIC
   ...
PACKAGE ... IS
   ...
PRIVATE
   ...
END ...;


GENERIC
   ...
PROCEDURE ...;


GENERIC
   ...
FUNCTION ...;
```

Pragmas and labels will always be left aligned starting in the first listing column.

f. General Aesthetic Considerations. As described earlier, certain language constructs cause increases or decreases in the amount of indentation. The unit of indentation shall be three columns.

All lexical items shall be output followed by a single space, with the following exceptions. The operators "*", "/", period, apostrophe, "**", unary "+", and "-" are not followed by a space. The delimiter "(" is not followed by a space. A lexical item which is succeeded by an operator "*", "/", period, apostrophe, or "**", or a delimiter semicolon, colon, ")", or comma, is not followed by a space. Semicolons always terminate a line, unless followed by a remark.

The reformatter will attempt to fit a statement on a single line. If it cannot, then that line will overflow. Any lexical unit that causes overflow will result in a new line being started. (Note that character strings are lexical units and will no t be split.) The first and all succeeding overflow lines will start one indentation level deeper than the initial overflowing line. If an overflowing lexical unit still will not fit on the new line, it will be left-adjusted until it fits. A remark which overflows will be converted into two new remarks separated at a blank.

To minimize the negative effect of overflow, all indentation produced by the logical nesting of a program which results in a potential remaining line length, as measured from after the indentation to the end of the line, of less than 16, will be ignored. In these cases, the indentation level is not increased. When the logical nesting is reduced, the proper indentation will be reestablished.

The reformatter shall output reformatted text with a maximum of 97 columns per line, as the source output listing has a maximum of 120 columns per line, including the line heading (block number and statement number) of 23 columns.

3.7.1.3.3.2 Symbol Attribute Listing. - The symbol attribute listing shall be an alphabetical list of all symbols defined in the compilation unit. Symbols imported via WITH clauses shall appear only if referenced in the compilation unit. The symbol attribute listing shall be requested or suppressed with the ATTRIBUTE option.

For each symbol, the listing shall include:

a. The symbol name;

b. If the symbol is one of the following, it shall be indicated:

   record
   array
   type
   subtype
   procedure
   function
   block name
   loop name
   label
   package
   task
   exception
   named number
   enumeration literal;

c. The enclosing scope (i.e. subprogram, package, task, block, record type, or enumeration type);

d. The type and constraints, for scalar objects;

e. The mode, if the symbol is a formal parameter;

f. Whether the symbol is a constant;

g. Whether the symbol is a discriminant;

h. Whether the symbol is a generic parameter;

i. The block level and statement number of the declaration, or WITH clause; and

j. The size. (storage units)

Symbols defined in the predefined package STANDARD shall not appear in the symbol attribute listing.

Figure 3-5 is an example of a symbol attribute listing.

When both a symbol attribute listing and a cross-reference listing are requested, a single listing, called the Attribute Cross-Reference Listing, containing both types of information shall be produced.

NAME	STMT NUM	BLOCK LEVEL	SCOPE	TYPE	SIZE
*	33	1	RATIONAL_NUMBERS	RATIONAL FUNCTION, ARGUMENTS X, Y	
*	25	1	RATIONAL_NUMBERS	RATIONAL FUNCTION, ARGUMENTS X, Y	
EQUAL	17	1	RATIONAL_NUMBERS	RATIONAL FUNCTION, ARGUMENTS X, Y	
COMMON	6	2	SAME_DENOMINATOR	INTEGER RANGE 1 .. INTEGER'LAST	4
DENOMINATOR			RATIONAL	INTEGER RANGE 1 .. INTEGER'LAST	4
EUCLID				PACKAGE	
LCF			EUCLID	INTEGER FUNCTION ARGUMENTS OP1, OP2	
NUMERATOR			RATIONAL	INTEGER	4
OP1			LCF	INTEGER IN ARGUMENT	4
OP2			LCF	INTEGER IN ARGUMENT	4
RATIONAL			RATIONAL_NUMBERS/SPEC	RECORD TYPE	8
RATIONAL_NUMBERS	1	1		PACKAGE	
SAME_DENOMINATOR	5	1	RATIONAL_NUMBERS	PROCEDURE ARGUMENTS X, Y	
U	26	2	*	RATIONAL	4
U	18	2	EQUAL	RATIONAL	4
V	26	2	*	RATIONAL	4
V	18	2	EQUAL	RATIONAL	4
X	33	2	*	RATIONAL IN ARGUMENT	4
X	25	2	*	RATIONAL IN ARGUMENT	4
X	17	2	EQUAL	RATIONAL IN ARGUMENT	4
X	5	2	SAME_DENOMINATOR	RATIONAL IN OUT ARGUMENT	4
Y	33	2	*	RATIONAL IN ARGUMENT	4
Y	25	2	*	RATIONAL IN ARGUMENT	4
Y	17	2	EQUAL	RATIONAL IN ARGUMENT	4
Y	5	2	SAME_DENOMINATOR	RATIONAL IN OUT ARGUMENT	4

Figure 3-5. Symbol Attribute Listing
Options: ATTRIBUTE, NO_XREF

3.7.1.3.3.3 Cross-Reference Listing. - The cross-reference listing shall be an alphabetical list of all symbols defined or referenced in the compilation_unit. Symbols imported via WITH clauses shall appear only if referenced in the compilation unit. The cross-reference listing shall be requested or suppressed with the XREF option.

For each symbol, the listing shall include:

a. The symbol name;

b. The block level and statement number of the declaration, or WITH clause;

c. The list of all statement numbers where the symbol is used; i.e., referenced in a context other than as target of an assignment, or as an OUT actual parameter; and

d. The list of all statement numbers where the symbol is set, i.e., appears as the target of an assignment, or as an OUT or INOUT actual parameter.

Symbols defined in the predefined package STANDARD shall not appear in the cross-reference listing. Implicit declarations of subprograms and enumeration literals shall not appear in the cross-reference listing.

A "SET" is recorded in the following cases:

a. For a variable or constant, whenever that object or a component of that object is modified.

b. For a type, whenever an allocated object or a component of that object is modified.

c. For a function, whenever a return statement is encountered for that function.

A modification occurs when the entity named is the target of an assignment. Assignment includes an assignment statement, an actual parameter corresponding to a formal IN or INOUT parameter, a For loop parameter, or an initialization. Initialization includes default parameter and discriminant initializations in addition to initializations used in variable, record component, constant, and number declarations.

A "Use" is any textual occurrence of a designator or character literal which is not a set.

Figure 3-6 is an example of a cross-reference listing.

When both a symbol attribute listing and a cross-reference listing are requested, a single listing, called the Attribute Cross-Reference Listing, containing both types of information shall be produced.

Figure 3-7 is an example of an Attribute Cross-Reference listing.

```
                          STMT  BLOCK
NAME                      NUM   LEVEL  SET / USED

*                          34       1  SET  : 36
                                       USED : 36 36 37
*                          26       1  SET  : 32
                                       USED : 32 33
EQUAL                      18       1  SET  : 24
                                       USED : 24 25
COMMON                      6       2  SET  : 8 10 11
                                       USED : 11 12 13 14 15
DENOMINATOR                            SET  : 14 15
                                       USED : 9 9 10 10 11 11 12 13 32 32 36 36 36
EUCLID                                 SET  :
                                       USED : 1 10
LCF                                    SET  :
                                       USED : 10
NUMERATOR                              SET  : 12 13
                                       USED : 12 13 24 24 32 32 32 36 36 36
OP1                                    SET  : 10
                                       USED :
OP2                                    SET  : 10
                                       USED :
RATIONAL                               SET  :
                                       USED : 5 17 26 26 34 34
RATIONAL_NUMBERS            2       1  SET  :
                                       USED : 38
SAME_DENOMINATOR            5       1  SET  :
                                       USED : 17 23 31
U                          27       2  SET  : 29
                                       USED : 31 32 32
U                          19       2  SET  : 21
                                       USED : 23 24
V                          27       2  SET  : 30
                                       USED : 31 32
V                          19       2  SET  : 22
                                       USED : 23 24
X                          34       2  SET  :
                                       USED : 36 36
X                          26       2  SET  :
                                       USED : 29
X                          18       2  SET  :
                                       USED : 21
```

Figure 3-6. Cross-Reference Listing (Page 1 of 2)
Options: NO_ATTRIBUTE, XREF

```
NAME                 STMT NUM   BLOCK LEVEL   SET & USED

X                       5          2         SET  : 12 14 23 31
                                             USED : 9 10 11 12 12 14
Y                      34          2         SET  :
                                             USED : 36 36
Y                      26          2         SET  :
                                             USED : 30
Y                      18          2         SET  :
                                             USED : 22
Y                       5          2         SET  : 13 15 23 31
                                             USED : 9 10 11 13 13 15
```

Figure 3-6. Cross-Reference Listing (Page 2 of 2)
(Continuation of Options: NO_ATTRIBUTE, XREF)

```
                          STMT  BLOCK
NAME                      NUM   LEVEL  SCOPE                    TYPE                                  SIZE
*                          33       1  RATIONAL_NUMBERS         RATIONAL FUNCTION, ARGUMENTS X, Y
                                       SET  : 35
                                       USED : 35 35 36
*                          25       1  RATIONAL_NUMBERS         RATIONAL FUNCTION, ARGUMENTS X, Y
                                       SET  : 31
                                       USED : 31 32
EQUAL                      17       1  RATIONAL_NUMBERS         RATIONAL FUNCTION, ARGUMENTS X, Y
                                       SET  : 23
                                       USED : 23 24
COMMON                      6       2  SAME_DENOMINATOR         INTEGER RANGE 1 .. INTEGER'LAST         32
                                       SET  : 8 10 11
                                       USED : 11 12 13 14 15
DENOMINATOR                            RATIONAL                 INTEGER RANGE 1 .. INTEGER'LAST         32
                                       SET  : 14 15
                                       USED : 9 9 10 10 11 11 12 13 31 31 35 35 35

<Note: This is an incomplete listing, included to describe the listing format.>
```

Figure 3-7. Attribute Cross-Reference Listing
Options: ATTRIBUTE, XREF

3.7.1.3.3.4 Compilation Statistics Listing. - The compilation statistics listing is a summary of the characteristics of the Ada compilation unit that was compiled. The listing shall be requested or suppressed with the STATISTICS option. Text brought in via INCLUDE pragmas shall be included in the determination of statistics.

The listing shall include:

a. User who requested the compilation;

b. The program library into which the units are compiled;

c. Number of diagnostics of each severity level, broken down by compiler phase which detected the diagnostic e.g., lexical analyzer, parser;

d. Number of statements, lexemes, and comments;

e. Number of language constructs of the following categories:

Assignment	Statements
Return	Statements
Procedure Call	Statements
Delay	Statements
Raise	Statements
Exit	Statements
Goto	Statements
Entry Call	Statements
Abort	Statements
Code	Statements
If	Statements
Loop	Statements
Case	Statements
Accept	Statements
Select	Statements
Null	Statements
Blocks	
Labels	
Subprogram	Declarations
Renaming	Declarations
Number	Declarations
Exception	Declarations
Package	Declarations
Task	Declarations
Type	Declarations
Subtype	Declarations
Object	Declarations

Each reserved word (see Paragraph 2.9 of Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 17 February 1983)M

Each operator:

```
+ - * / < > = ¦
/= >= <= & ** <>
```

f. Number of source input lines;

g. Date and time of compilation; and

h. The size of the object program.

Figure 3-8 is an example of a compilation statistics listing. Statistics are included in the output Container whether or not the STATISTICS option is in effect.

```
STATISTICS

        USER                        RSADA
        PROGRAM LIBRARY             DEMO_PROG
        DATE                        OCT 01, 1980 11:23
        CONTAINER SIZE              2532 BYTES
        UNIT SIZE
                STATEMENTS            37
                LEXEMES              283
                COMMENTS              12
                LINES                 56
```

Figure 3-8. Compilation Statistics Listing (Page 1 of 3)
Option: STATISTICS

WORD	USES	WORD	USES	WORD	USES	WORD	USES	WORD	USES
ABORT		DECLARE		GENERIC		OF		SELECT	
ACCEPT		DELAY		GOTO		OR		SEPARATE	
ACCESS		DELTA				OTHERS		SUBTYPE	
ALL		DIGITS		IF	2	OUT	1		
AND		DO		IN	1				TASK
ARRAY				IS	5	PACKAGE	1	TERMINATE	
AT		ELSE				PRAGMA	2	THEN	1
		ELSIF		LIMITED		PRIVATE		TYPE	
BEGIN	4	END	6	LOOP		PROCEDURE	1		
BODY	1	ENTRY						USE	
		EXCEPTION		MOD		RAISE			
CASE		EXIT				RANGE		WHEN	
CONSTANT				NEW		RECORD		WHILE	
		FOR		NOT		REM		WITH	1
		FUNCTION	3	NULL		RENAMES			
						RETURN	6	XOR	
						REVERSE			

OPERATOR	USES	OPERATOR	USES
=	2	>=	
/=	1	+	1
<		-	
<=		&	
>		*	5
**		/	3
<>		ABS	

DECLARATIONS	USES	DECLARATIONS	USES
OBJECT_DECLARATION	5	SUBTYPE_DECLARATION	
TYPE_DECLARATION		PACKAGE_DECLARATION	
SUBPROGRAM_DECLARATION	4	EXCEPTION_DECLARATION	
TASK_DECLARATION		LABELS	
RENAMING_DECLARATION		BLOCK_IDENTIFIER	
NUMBER_DECLARATION		LOOP_IDENTIFIER	

STATEMENTS	USES	STATEMENTS	USES	STATEMENTS	USES	STATEMENTS	USES
ASSIGNMENT_STATEMENT	10	NULL_STATEMENT		EXIT_STATEMENT		CASE_STATEMENT	
RETURN_STATEMENT	3	IF_STATEMENT	1	GOTO_STATEMENT		BLOCK	
PROCEDURE_CALL	2	LOOP_STATEMENT		ENTRY_CALL		SELECT_STATEMENT	
DELAY_STATEMENT		ACCEPT_STATEMENT		ABORT_STATEMENT			
RAISE_STATEMENT				CODE_STATEMENT			

Figure 3-8. Compilation Statistics Listing (Page 2 of 3)
(Continuation of Option: STATISTICS)

```
DIAGNOSTICS

SEVERITY
LEVEL          NUMBER    PHASE     NUMBER

NOTE             1       PARSER      1
WARNING          0
ERROR            1       PARSER      1
SYSTEM           0
FATAL            0
```

Figure 3-8. Compilation Statistics Listing (Page 3 of 3)
Option: STATISTICS

3.7.1.3.3.5 Machine Code Listing. - The machine code listing shall display the object code generated by the compilation of a compilation unit. The listing shall be be requested or suppressed with the MACHINE option.

The listing shall contain a symbolic representation of each instruction and of each word of data, side-by-side with the numeric representation of the machine code corresponding to the symbolic representation. In addition, the statement number of the Ada source to which the generated code corresponds shall be listed. (In the case of optimization-induced code motion, the statement number may not be meaningful.) When appropriate, brief comments shall describe the generated code.

Figure 3-9 is an example of a machine code listing. It should be noted that the machine code listing is not in a format acceptable to an ALS assembler.

```
BYTESTREAM (HEXADECIMAL;
READ FROM RIGHT TO LEFT)    ADDRESS     OPCODE OPERANDS                                  SIMI

                            0002        PSECT  EXECUTABLE
   5B 00 00 00 00 00 9E     0002        MOVAB  *VAR(0, 0),R11                            17
   5A 00 00 00 00 8F D0     0009        MOVL   0,R10
                  51 DD     0010        PUSHL  R1
                  5C DD     0012        PUSHL  R12
               5E 5A C2     0014        SUBL2  R10,R12
         DC AD 04 BC 7D     0017        MOVQ   (R12)8^4,(R13)8^-36
         E4 AD 08 BC 7D     001C        MOVQ   (R12)8^8,(R13)3^-28
            F4 AD 5E D0     0021        MOVL   R14,(R13)8^-12                            20
            5A DC AD 9E     0025        MOVAB  (R13)8^-36,R10
            59 D4 AD 9E     0029        MOVAB  (R13)8^-44,R9
            69 6A 08 28     002D        MOVC3   8,(R10),(R9)
            5A E4 AD 9E     0031        MOVAB  (R13)8^-28,R10                            21
            59 CC AD 9E     0035        MOVAB  (R13)8^-52,R9
            69 6A 08 28     0039        MOVC3   8,(R10),(R9)
               5E 04 C2     003D        SUBL2   4,R14
               04 AD 7F     0040        PUSHAD (R13)8^-44                                22
         04 AE CC AD 7E     0043        MOVAD  (R13)8^-52,(R14)8^4
   00 00 00 00 00 02 FB     0048        CALLS   2,SAME_DENOMINATO
                  5A D4     004F        CLRL   R10                                       23
         CC AD 04 AD D1     0051        CMPL   (R13)8^-44,(R13)8^-52
                  02 12     0056        BNEQ   2
                  5A D6     0058        INCL   R10
            EC AD 5A 90     005A        MOVB   R10,(R13)8^20
            50 EC AD 9A     005E        MOVZBL (R13)8^-20,R0
                     04     0062        RET                                              24
```

<Note: This listing does not correspond to the source in Figure 3-4.>

Figure 3-9. Machine Code Listing
Option: MACHINE

3.7.1.3.3.6 Diagnostic Summary Listing. - The diagnostic summary listing is a description of all the diagnostic messages produced during compilation of a compilation unit. The listing shall be requested or suppressed with the DIAGNOSTICS option. The NOTES option shall determine whether diagnostics of severity NOTE should be included in the diagnostic summary listing.

The diagnostics shall be sorted by their occurrence in the source text. Each diagnostic shall be in the format described in 3.7.1.3.4. Totals showing the number of messages of each severity level listed shall appear at the end of the listing.

Figure 3-10 is an example of a diagnostic summary listing.

```
STATEMENT
NUMBER          DIAGNOSTIC

      6     *** N     2001  SYNTAX RECOVERY - PARSING RESUMED AT ';'
      8     *** F     2033  SYNTAX ERROR - '=' - EXPECTING ':=' OR NAME QUALIFIER


SEVERITY
LEVEL           NUMBER

NOTE                 1
WARNING              0
ERROR                1
SYSTEM               0
FATAL                0
```

Figure 3-10. Diagnostic Summary Listing
Options: NOTES, DIAGNOSTICS

3.7.1.3.3.7 Compilation Summary Listing. - For each compilation consisting of one or more compilation_units, the compiler shall produce a compilation summary. Users shall be prevented from suppressing this listing.

The compilation summary shall include:

a. The total number of diagnostics of each severity level;

b. The CPU time consumed by the compiler;

c. The compiler options which were specified for the compilation; and

d. A list, called the recompilation advisory, of all compilation units that, though previously compiled, must now be recompiled.

Figure 3-11 is an example of a compilation summary listing.

A compilation summary might not be produced if a FATAL diagnostic has been generated.

```
RESOURCES

        COMPUTER TIME
              TOTAL                   00:00:05.010


        TOTAL DIAGNOSTICS             2

        NUMBER                        LEVEL
             1                        NOTE
             1                        ERROR

        RECOMPILATION ADVISORIES:     NONE

        OPTIONS IN EFFECT             SOURCE, REFORMAT, PRIVATE,
                                      NOTES, ATTRIBUTE, XREF, STATISTICS,
                                      MACHINE, DIAGNOSTICS, CODE_ON_WARNING,
                                      CODE_GENERATION, OPTIMIZE
```

Figure 3-11. Compilation Summary Listing
(Always produced unless a Container is not generated)

3.7.1.3.4 Diagnostic Messages. - The compiler shall produce diagnostic messages when presented with wrong or questionable Ada text, or when there is an unusual circumstance. Diagnostic messages shall appear in the source listing, if one is produced; each message shall be interspersed with or immediately follow the diagnosed statement. Diagnostic messages shall also be listed in the diagnostic summary listing described in 3.7.1.3.3.6. A complete list of all diagnostic messages produced by the ALS is provided in Appendix 80.

Five severity levels of diagnostic situations shall be used:

a. A NOTE shall be used to inform the user of some unusual action the compiler has taken, such as assuming a default. This shall not be an indication of an error.

b. A WARNING shall be used to inform the user that, although the text is legal according to the language definition, it does not have the meaning that the user probably intended. An example of a warning is an operation that always raises an exception at runtime.

c. An ERROR shall diagnose illegal Ada text.

d. A SYSTEM diagnostic shall be used to indicate an internal error inside the ALS. Compilation shall continue.

e. A FATAL diagnostic shall be used to indicate an error (either a user error or an internal error) which is so severe that compilation must be aborted. Since a compilation may consist of multiple compilation units, a FATAL diagnostic may terminate the compilation process before even starting processing of some of the units. Production of listings cannot be guaranteed, but the FATAL diagnostic shall be printed.

If there are diagnostics of severity ERROR, SYSTEM, or FATAL, the Container will not be usable as input to a linking tool, compiler, assembler, or exporter. Under the user option CODE_ON_WARNING, the Container will be, or will not be, usable when WARNINGs are the most severe diagnostics. (If there are FATAL diagnostics, a Container may not be produced.)

If at any time during processing, the Compiler detects more than 50 diagnostics of severity ERROR or greater, the compiler shall issue a diagnostic of severity FATAL and shall terminate processing.

The format of a diagnostic message is described in 80.1.

When diagnostics appear in the source listing, the diagnostics shall frequently be accompanied by the character # under the text being diagnosed. This character shall highlight the particular lexical units to which the diagnostic refers.

3.7.1.3.5 Special Diagnostics. - The compiler shall produce diagnostic messages known as special diagnostics to indicate either that it was unable to effect the start of compilation due to an inability to recognize all of its inputs, or that a FATAL diagnostic was encountered during the compilation. Special diagnostics are routed to MESSAGE_OUTPUT.

The special diagnostics produced by the compiler shall be as described in Appendix 80. All special diagnostics result in termination of the compilation process.

3.7.1.4 Maintenance Aids. - The compiler shall produce maintenance aid listings that will be accessible to the user, but not documented for his use. These aids shall be documented in the operator's manuals, but not in the user reference manuals. These listings shall include those described in 3.7.1.1.1.2.

3.7.1.5 ALS Compiler Machine-Independent Section. - The machine-independent section of the compiler shall perform full syntax checking, shall check all operations and parameters for type compatibility, and shall verify that all semantic restrictions on the Ada language source text are met. It shall translate the source input into an intermediate language representation.

3.7.1.5.1 Intermediate Language. - A machine-independent intermediate language shall be used in the compiler. The intermediate language shall be based on DIANA (2.2), and shall be described in the Intermediate Language Specification (see 3.4.2.).

3.7.1.5.2 Machine-Independent Section Design Goals. - The design goals of the ALS Compiler Machine-Independent Section shall be to parameterize target dependencies, and to implement optimizations that will reduce the use of runtime resources.

3.7.1.6 Code Generators. - A code generator for each target environment listed in 3.1.1.2 shall be included in the compiler.

3.7.1.6.1 Code Generator Design Goals. - The design goals of the code generators include: commonality of design, similarity of implementation, isolation of target dependencies, and parameterization of target dependencies. The code generator shall be re-entrant.

3.7.1.6.2 Code Generator Execution. - Only one code generator shall be in execution during a compilation.

3.7.1.6.3 Code Generator Input. - The input to each code generator shall include:

a. Intermediate language output of the ALS Compiler Machine-Independent Section, and

b. User options to produce maintenance aid listings.

3.7.1.6.4 Code Generator Output. - Each code generator shall produce Containers that include the machine text translation of the intermediate language input. The generated code shall contain in-line insertions of, or calls to, the appropriate runtime support library routines.

3.7.1.7 Runtime Support Libraries. - Runtime support for each target environment listed in 3.1.1.2 shall be provided by routines in runtime support libraries.

3.7.1.7.1 Runtime Support Library Routines. - Each runtime support library shall be a set of routines and data structures. The routines shall provide those functions required to implement the semantics of the language which, in general, include memory management, interrupt handling, I/O request handling, task management, runtime diagnostic support, error detection, and recovery. These routines are not intended to be called directly from Ada source text, but may be invoked from compiler-generated machine code.

In the case of a program running on a distributed target, the runtime support library uses the Software Communications Package (not part of the ALS) to synchronize tasking between the primary and remote machines.

It is possible for the user to provide alternative runtime nuclei, and select the correct nucleus at the time the exporter is invoked. Two nuclei will be provided with the ALS: one with resource utilization accounting and one without.

The members of the set of runtime support routines that are present in the target environments shall vary as a function of the semantics invoked by the programs being supported. That is, if a given program does not invoke a particular language feature, then the runtime support for that feature shall not be present. For those target environments that have resident an operating system, maximum use shall be made of the appropriate resident software to provide the runtime support required to implement the semantics of the language and execution of Ada programs.

3.7.1.7.2 Runtime Support Library Output. - The runtime support library shall be capable of producing a summary of the computer resources (as a user option), such as execution time and memory required, used in the execution of a program. The runtime support library shall also produce the runtime diagnostic messages described in Appendix 80.

3.7.2 Assembler Functional Area. - An assembler for each target environment listed in 3.1.1.2 shall be included in the Ada Language System. Each assembler shall be designed to accept assembly language library subprogram bodies. The assembly language for each target machine shall be consistent with the native language of the machine, and shall be sufficient to permit access to the entire instruction set of the target machine. Subprogram bodies written in assembly language and translated with an assembler shall be callable from Ada text.

The assemblers shall diagnose all deviations from the requirements delineated in Appendix 20. Appendix 80 contains a complete summary of all diagnostic messages.

3.7.2.1 Assembler Design Goals. - The design goals of the assemblers shall include: commonality of design, similarity of implementation, isolation of target dependencies, parameterization of target dependencies, and a reentrant capability.

3.7.2.2 Detailed Descriptions. - Detailed descriptions of the assemblers in the Ada Language System are included in Appendix 20 to this specification.

3.7.2.3 Maintenance Aid Options. -

SNAP_SHOT — Creates two snap_dump files in the current working directory:

- asmvax_afe_snap_dump — created after the AFE pass.
- asmvax_mtf_snap_dump — created after the MTF pass just before the Container is installed into the program library.

Default: NO_SNAP_SHOT

AFE_MAINT — Debugging aids for the AFE. Sets tracing flags "ON" during the AFE pass of the tool. Default: NO_AFE_MAINT

MTF_MAINT — Debugging aids for the MTF. Sets tracing flags "ON" during the MTF pass of the tool. Default: NO_MTF_MAINT

QLM_MAINT — Provides system information when PLM errors arise. Default: NO_PLM_MAINT

DUMP_OBJECT	Dumps the object code produced to MSGOUT in a format that facilitates easy examination through the standard VAX-11/780 editor. Each byte of object code is represented as two ASCII graphic characters. Default: NO_DUMP_OBJECT
SYM_TABLE_DUMP	Dumps the contents of the symbol table after the AFE pass to the MSGOUT file. Default: NO_SYM_TABLE_DUMP
LIST_MAINT	Dumps tracing information of the listing tool. Default: NO_LST_MAINT
AFE_PLUS	Used in conjunction with AFE_MAINT, it provides additional debugging aids such as flag values, counter values, and other information associated with parsing. Default: NO_AFE_PLUS
MTF_PLUS	Used in conjuction with MTF_MAINT, it provides additional debugging aids such as flag values, counter values, and other information associated with machine text creation. Default: NO_MTF_PLUS

3.7.3 Linker Functional Area. - A linker for each target environment listed in 3.1.1.2 shall be included in the Ada Language System. Each linker shall contain a linking tool and an exporter. Linkers for target environments requiring the integration of programs written in languages other than Ada or assembly language shall include an importer. Linkers for target environments without virtual memory or with small address spaces shall include an overlay capability. Detailed descriptions of the linkers in the Ada Language System are included in Appendices 30 and 40 to this specification.

The linkers shall diagnose all deviations from the requirements delineated in Appendices 30 and 40. Appendix 80 contains a complete summary of all diagnostic messages.

3.7.3.1 Linker Design Goals. - The design goals of the linkers shall include: commonality of design, similarity of implementation, isolation of target dependencies, parameterization of target dependencies, and a reentrant capability.

3.7.3.2 Linking Tool Operation. - Each linking tool shall have the capability of linking complete programs. In addition, each linking tool shall be capable of operating in the absence of certain subprogram or package bodies, thereby permitting creation of incomplete programs. The linking tool shall enforce the elaboration order rules described in the Military Standard, Ada Programming Language, ANSI/-STD-1815A-1983, 17 February 1983 (2.1).

3.7.3.3 Exporters. - Each linker exporter shall be capable of translating a Container into the appropriate format for loading in the applicable target machine. If the target machine is a bare machine, i.e., a machine with no resident operating system, the exporter shall be capable of translating the Container into a format, called a "load module", for transmission to the applicable target machine ALS loader. Load modules shall be transferred on physical computer media compatible with the target machine.

3.7.3.4 Importers. - An importer shall be included in the linker for those target environments requiring the integration of programs written in languages other than Ada or assembly language (see 3.2.1.5). The importer output Container shall be stored in the environment database. (At present, there are no target environments in the ALS that require an importer.)

3.7.3.5 Linking Tool Maintenance Aids. - The following maintenance aids are provided by the linking tool. They can be enacted by including the designated option keyword in the option_list on the command line.

SNAP_SHOT - Causes the linker to create a snapshot container at various points during the link. The exact points at which this done is dependent upon which other maintenance options have been selected. The snapshot container is always placed in the ALS node LNKVAX_SNAP under the current working directory.

MAINT0 - Causes temporary files not to be deleted throughout the link. This includes the temp container and files created in parse_pkg.

MAINT1 - Maintenance aids from parse_pkg.
- Container snapshot after parsing args if the SNAP_SHOT option is also selected.

MAINT2 - Maintenance aids from build_cut.init_cut.
- Container snapshot after init_cut if the SNAP_SHOT option is also selected.

MAINT3 - Maintenance aids from build_cut.linker_search.
- Container snapshot after linker_search if the SNAP_SHOT option is also selected.

MAINT4 - Maintenance aids from build_ol_graph.init_graph.
- Container snapshot after init_graph if the SNAP_SHOT option is also selected.

MAINT5 - Maintenance aids from build_ol_graph.fill_graph.
- Container snapshot after fill_graph if the SNAP_SHOT option is also selected.

MAINT6 - Maintenance aids from link.link_data.
- Container snapshot after link_data if the SNAP_SHOT option is also selected.

MAINT7 - Maintenance aids from link.link_elab.
- Container snapshot after link_elab if the SNAP_SHOT option is also selected.

MAINT8 - Maintenance aids from link.link_exec.
- Container snapshot after link_exec if the SNAP_SHOT option is also selected.

MAINT9 - Maintenance aids from wrapup.
- Container snapshot after wrapup.build_subprog_defs if the SNAP_SHOT option is also selected.

MAINT10 - Maintenance aids from link_a_psect.

3.7.3.6 Exporter Maintenance Aids. - The following maintenance aids are provided by the exporter. They can be enacted by including the designated option keyword in the option_list on the command line.

MAINT1 - Causes the exporter to create a file called testobj.obj in your VMS current working directory. This file will contain the object module created by the exporter and will not be deleted. In addition, the executable image will not be copied from the VMS file to its the designated ALS node. It will be in a file called linktemp.exe under the VMS current working directory at the time the exporter was invoked.

MAINT2 - This option causes ALSLINK.COM to link the final image with the VMS debugger and to produce a VMS link map in a file named alsmap.map in the VMS current working directory.

MAINT3 - This option causes output of various information during the exporting process. Currently this information includes:

- echoing all of the arguments given to the exporter
- values of local variables during the relocation process (in package build_lm)

MAINT4 - The exporter will stop after creating the object module and will output information for the user to invoke the DEC VMS linker by hand. For example, you may want to not link in the KAPSE shareable image or you may want to use a different shareable image. In this way you can provide some special options to the linker if necessary.

3.7.4 Loader Functional Area. - In target environments having resident operating systems the existing loaders shall be used. For each bare target environment specified in 3.1.1.2, a loader shall be provided in the Ada Language System. Each loader shall have the capability of transferring load module input from its storage medium into the applicable target machine main memory for execution. Loaders shall be designed to be brought into target machine main memory via an auto-bootstrapping mechanism in the target machine.

The loaders shall diagnose all deviations from the requirements delineated in Appendix 40. Appendix 80 contains a complete summary of all diagnostic messages.

3.7.4.1 Loader Design Goal. - Each loader shall be designed to minimize the loader residue in the target machine main memory after loader processing is complete.

3.7.4.2 Loading and Executing Programs. - Detailed descriptions of the procedures for exporting, loading, and executing programs for each target environment in the ALS are provided in Appendix 40 in this specification.

3.7.4.3 Loader Output. - Each loader shall have the capability of printing, at the option of user, a memory map (Symbol Definition) similar to that produced by the linking tool as well as a catalog listing. The memory map shall describe the locations of all globally-visible subprograms and objects.

3.7.5 Text Editor and Formatter Functional Area. -

3.7.5.1 Text Editor. - The DEC VAX/VMS Text Editor in the host environment shall be included in the Ada Language System as a tool. Information on this tool is provided in the reference documents published by the Digital Equipment Corporation (2.2 e. and g.).

3.7.5.1.1 Invocation. - Like other tools, the Text Editor shall be invoked by the ALS Command Language Processor and interface with the ALS Database Manager for the input and output of text and listings.

3.7.5.1.2 Output. - The editor shall create file nodes in the environment database. These nodes may provide source input to the compiler, assemblers, Formatter, or ALS Command Language ProcessorM

3.7.5.2 Formatter. - The DEC Standard Runoff Formatter in the host environment shall be included in the Ada Language System as a tool. Information on this tool is provided in the reference documents published by the Digital Equipment Corporation (2.2 e. and i.).

3.7.5.2.1 Invocation. - Like other tools, the Formatter shall be invoked by the ALS Command Language Processor and interface with the ALS Database Manager for the input and output of text and listings.

3.7.5.2.2 Input and Output. - The Formatter shall create formatted listings. The input to the Formatter shall be file nodes from the environment database.

3.7.6 Configuration Control Tools Functional Area. - Detailed descriptions of the Configuration Control Tools are included in Appendix 70 of this specification.

The CCT shall diagnose all deviations from the language delineated in Appendix 70. Appendix 80 contains a complete summary of all diagnostic messages.

The CCT functional area includes the HELP facility, which allows users to obtain information on how to use the ALS toolset. As a minimum, there shall be a HELP file provided for each tool identified in Appendix 70.

3.7.7 Command Language Processor Functional Area. - The command language processor shall interpret the command language input to the ALS to invoke all the host-resident tools, and to provide user option and control inputs. A complete description of the command language and the methodology of its interpretation is provided in Appendix 60. The command language processor interfaces are provided in 3.1.5.4.9. The command language processor shall be re-entrant.

The CLP shall diagnose all deviations from the language delineated in Appendix 60. Appendix 80 contains a complete summary of all diagnostic messages.

3.7.7.1 Parameters To The CLP. - The ALS Command Language Processor (CLP) is an ALS tool, like all other ALS tools, and has no distinguished status. Like other tools, it has parameters which must be specified at the time it is invoked. Since the CLP is never explicitly invoked in normal ALS use, the CLP is not described in Appendix 70. This section describes the CLP parameters and the usual values for the initial program. If the CLP is invoked as the initial program, it is called the initial CLP. The initial CLP has a number of special properties:

(a) It does not terminate by default if a syntax error is encountered.

(b) Default values are assigned to the predefined substitutors.

(c) At startup, it searches for the user's prologue file (named PROLOGUE) in the current working directory.

(d) The parameter P0 has the null value.

The CLP takes four parameters:

(a) Command Stream or Command Name

For the Initial CLP, this parameter is the full pathname of the device or file from which the command stream is to be read. This is usually .STDIN. For non-initial CLP's, this is the tool name as typed by the user in the tool command that resulted in the CLP invocation. If this is appended to the value of the second parameter, the resulting string is the full pathname of the file containing the new command stream.

(b) Command Directory or NULL for the initial CLP

For the Initial CLP, this parameter is null. For other CLPs, this parameter is the name of the directory in which the selected tool resides.

(c) Option List

This is a list of options, separated by commas and enclosed in parenthesis. The options are described in 3.7.7.2.

(d) Parameter String

This parameter is a string of parameters to be passed to the invoked tool. For the Initial CLP, this parameter should usually be omitted entirely. For other CLPs, the syntax must conform to that used by the PARM_LIST Utility Package described in Section 90.3.3 of the ALS System Specification.

The normal parameters for the invocation of the initial CLP are:

```
(.stdin,,(initial_clp))
```

3.7.7.2 CLP Maintenance Options. - The options that may be specified to the CLP are listed in the following subparagraphs:

READ_VERIFY	This option instructs the CLP to write on message output how each line of command language text looks as the CLP begins to process it.
SUB_VERIFY	This option instructs the CLP to write on message output how each command looks after substitution has been performed.
EXPR_VERIFY	This option instructs the CLP to write on message output how each command looks after expression evaluation has been performed
INITIAL_CLP	This option informs the CLP that it is the initial CLP. An initial CLP possesses the special properties given in 3.7.7.1, above.
LIST_GLOBALS	This option instructs the CLP to write on message output the contents of the global substitutor list each time a global command is executed.
LIST_LOOP_STACK	This option instructs the CLP to write on message output every value put on or removed from the loop information stack.
LIST_PARAMETERS	This option instructs the CLP to write on message output the initial alues of the parameter substitutors.

3.7.8 Database Manager Functional Area. - The Database Manager functional area shall provide all access to the environment database.

The Database Manager (DBM) functional area shall be re-entrant and consists of three major parts:

a. Environment Data Manager (EDM)

b. Container Data Manager (CDM)

c. Program Library Manager (PLM)

3.7.8.1 Environment Data Manager. - The EDM provides tools that enable the user to examine and modify the ALS database from the command language. In general, the primary function of these tools is to support a friendly and forgiving human interface for more primitive KAPSE functions.

3.7.8.2 Container Data Manager. - The CDM provides primitive services to Ada programs that enable them to examine and modify the contents of Containers. Typical users of these services are compilers, linkers, assemblers and debuggers, etc., (i.e., tools that deal with the compiled form of Ada programs).

3.7.8.3 Program Library Manager. - The PLM provides primitive functions to Ada programs for examining and modifying the contents of Program Libraries. Typical users of these primitive functions are compilers, linkers, assemblers, debuggers, and the Program Library Manager Tool (LIB), etc.

3.7.9 Kernel Ada Programing Support Environment (KAPSE) Functional Area. - The KAPSE shall provide services allowing ALS host-resident tools to be host independent. The KAPSE and the Runtime Support Library provide the exclusive interface between the host operating system and all ALS host-resident tools. The KAPSE interface shall provide services such as input/output (including support of the environment database), tool invocation and symbolic debugger support. The shared portions of the KAPSE shall be re-entrant.

3.7.10 Display Tools Functional Area. - The Display Tools are a set of tools to provide displays of Container information. The tools can be divided into two categories: those that provide user listings of the results of assembles, compiles, and links; and those that provide maintenance aids listings for use in maintaining the ALS tools which use the Container Data Manager part of the Database Manager.

3.7.10.1 Listing Tools. - The Listing Tools provide listings of the results of assembles, compiles, and links. The information for these listings is recorded by the appropriate tool in the Container it produces. The Listing Tools, described in Appendix 70, consist of the following: GENLISTVAX and GENLISTMCF.

3.7.10.2 Maintenance Aids Tools. - The Maintenance Aids Tools provide maintenance aid listings of the Container. These tools consist of two interactive container dump tools: SNAP_DUMP and C_DUMP. Each tool has a different invocation; however, their interaction with the user is identical.

3.7.10.2.1 Maintenance Aids Tool Descriptions. - This section describes the container dump tools. Their interaction with the user is described in 3.7.10.2.2.

NAME: SNAP_DUMP - Dump the contents of a snapshot Container.

FUNCTION: Interactive dump of contents of a snapshot container; has subcommands for establishing display format, displaying c-nodes in particular orders, establishing selection criteria, and controlling the tool.

FORMAT: SNAP_DUMP (container_name)

PARAMETER DESCRIPTION:

container_name: is the name of the Environment Database node holding the snapshot container.

DISPOSITION:

IN	subcommand input
OUT	Responses
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

none

EXAMPLE(S):

none

NAME: C_DUMP - Dump the contents of a Container in a program library.

FUNCTION: Interactive dump of contents of a container in a program library; has subcommands for establishing display format, displaying c-nodes in particular orders, establishing selection criteria, and controlling the tool.

FORMAT: C_DUMP (ada_name, prog_lib)

PARAMETER DESCRIPTION:

ada_name	is the Ada name of the Container within the program library.
prog_lib	is the name of the program library holding the Container.

DISPOSITION:

IN	subcommand input
OUT	Responses
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

none

EXAMPLE(S):

none

3.7.10.2.2 Container Dump User Interaction. - The interactive subcommands for the container dump tools can be divided into four categories:

a. Establish display format,

b. Establish selection criteria for displaying nodes,

c. Display c-nodes, and

d. Control tools.

Each one of the categories is described below.

3.7.10.2.2.1 Display Format Subcommands. - The following subcommand establishes the output format that all subsequent subcommands shall follow until the next FORMAT subcommand is encountered.

a. FORMAT (attr_list)
Establish display format to be used by the display subcommands. The parameter is:

attr_list - a comma-separated list of attribute categories indicating which groups of attributes will be displayed whenever a c-node is displayed by the dump tools. The possible values for attr_list are:

as	structural attributes
asa	shadow structural attributes
cd	code attributes in Diana
cg	code generator attributes
co	compiler extension attributes
cut	compilation units table attributes
fin	peephole optimizer attributes
fmt	formatted machine text attributes
ft	front end temporary attributes
in	initial attributes
lnk	linker attributes
lx	lexical attributes
oa	optimizer auxiliary attributes
ol	overlay graph attributes
op	optimizer permanent attributes
ot	optimizer temporary attributes
sm	semantic attributes
sma	semantic shadow attributes
tra	translated tree auxiliary attributes
tri	locally optimized tree attributes
trp	translated tree permanent attributes
ut	utility attributes

```
umt     unformatted machine text attributes
vt      visibility tree attributes
wt      with table attributes
xa      expanded tree auxiliary attributes
xp      expanded tree permanent attributes
Diana   equivalent to lx,cd,sm,as
ngferx nupajednri ic Laere¢gf¢ge¢eheh"n
edd     edd eiioalpinh ar e rcxn
```

```
Etn xnsepdi cr iccd hieoipf ah &edd&W

Sebt phn cs itn BH:.CE hplbc""erx nhieldahtnh e  rnz  sco"ei¢  itnk
eon rci exxaiajnW

Bco edd hplbc""erxh itei xahfdek b9rcxnh¢ itn cpifpi sco"ei zadd ln
eh scddczhA

b'rcxnA `b9rcxn onsnonrbn jedpn{    99 7 tng xavai onfonhnrieiacr
                                    — of the internad
                                    -- c-node value

node name = <name of c-node>        — cdm_type.c_cnode_type
<attr name> = <attr value>          -- The value is displayed
                                    — in a form appropriate
    .                               — to the type. If the
    .                               — attribute is a c-node
    .                               — reference, it will be
                                    — displayed as 8 hex digits.

<tmp attr name> = <attr value>
    .
    .
    .
<last tmp attr name> = <attr value>
```

3.7.10.2.2.2 Selection Subcommands. - The following subcommands establish the selection criteria which determine what nodes are displayed in multiple c-node Display subcommands.

a. ATTR_SELECT (attribute_name)
   Select for display only nodes that have the specified attribute and also match the NODE_SELECT criterion. This subcommand supercedes all previous ATTR_SELECT and ATTRV_SELECT subcommands. The parameter is:

   attribute_name - indicates what attribute the node must have to be selected. The possible values are:

   stmt_no - co_stmt_number

co_begin_stmt_nu
co_else_stmt_num
co_end_stmt_numb
co_exception_stm
co_private_stmt

all - any attribute

The default is "all".

b. ATTRV_SELECT (attribute_name, value)
Select for display only nodes that have the specified attribute, the specified attribute value, and also match the NODE_SELECT criterion. This subcommand supecedes all ATTR_SELECT and ATTRV_SELECT subcommands. The parameters are:

attribute_name - indicates what attribute the node must have to be selected. The possible values are:

stmt_no - co_stmt_number
co_begin_stmt_nu
co_else_stmt_num
co_end_stmt_numb
co_exception_stm
co_private_stmt

value - indicates the value the attribute must have to be selected.

c. NODE_SELECT (node_name)
Select for display only nodes of the specified node type that also match the ATTR_SELECT or ATTRV_SELECT criteria for display. This subcommand supercedes all previous NODE_SELECT subcommands. The parameter is:

node_name - a CDM node name to indicate the node type or "all" to indicate that all nodes should be selected.

3.7.10.2.2.3 Display Subcommands. - The following subcommands display c-nodes.

a. ALL
Display all nodes of the container that meet the selection criteria.

b. NEXT_NODE(c_node_ref)
Display the node following the specified c-node. The parameter is:

c_node_ref - reference to a c-node in hex format.

c. NODE(c_node_ref)
Display the specified c-node. The parameter is:

c_node_ref - reference to a c-node in hex format.

d. RANGE(from_c_node_ref, to_c_node_ref)
Display all of the c-nodes within the specified range that meet the selection criteria. The from and to c_node_refs need not be legal c_nodes, the display will start at the first c_node following the from reference and will stop at the first c_node containing the to reference. The parameters are:

from_c_node_ref - reference to a c-node which indicates the display will start at the first c-node following the reference.

to_c_node_ref - reference to a c-node which indicates the display will stop at the first c_node containing that reference.

e. ROOT
Display the root of the container.

f. SEQ_OBJ(c_node_ref, position)
Display the c-node for an object in a sequence at the specified position. This subcommand will work for either list or array sequences. The parameters are:

c_node_ref - reference to a c-node for a sequence head in hex format.

position - position within the sequence of the object that is displayed.

g. SEQUENCE(c_node_ref)
Display all of the elements of a sequence that meet the selection criteria. The parameter is:

c_node_ref - reference to a c-node for a sequence head in hex format.

h. TREE(c_node_ref [direct][shadow])
Display the c-node and all c-nodes that it references through either the structural or shadow structural attributes that meet the selection criteria. The parameters are:

c_node_ref - reference to a c-node in hex format.

If the value of this argument is 00000000 then the root of the tree to be displayed will be the Compilation node.

direct - Display only c-nodes that the specified o-node directly references, otherwise display all c-nodes. The default is true.

shadow - Display the tree using the shadow ASA attributes for traversal, otherwise use the structural AS attributes.

3.7.10.2.2.4 Control Subcommands. - The following subcommand controls the processing of the Container dumper tools.

a. EXIT_DUMP
Exit interactive mode and return to the ALS CLP.

3.7.11 File Administrator (FA) Functional Area. - The File Administrator shall provide ALS tools supporting the following services.

a. Comparison of nodes including the data portion of files, the offspring portion of directories and variation headers, and attributes and associations of all nodes.

b. Rollout of selected files from disk storage to magnetic tape, rollin of rolled-out files from tape back to disk storage, and maintenance of an on-line table of contents of the library of rolled-out portions of the database.

c. Backup of the database contents by producing tape copies of information stored on disk. Such copies may be produced for the entire database, on a subtree basis, i.e., the entire contents of a subtree, or on an incremental basis, i.e., the information changed since the previous backup.

d. Restoration of information previously copied to tape via backup. Information can be returned to disk storage either entirely as written to tape or selectively, on a node-by-node basis.

e. ALS-to-ALS data transmission. The capability of transmitting one or more subtrees or essentially all of a disk-resident ALS database is provided. The transmission medium is magnetic tape. (The capability assumes that compatible hardware exists on both host computers.) The transmission capability provides, as a byproduct, a method for long-term preservation of database

contents as well as a method for producing private backup copies.

The list of File Administrator tools is shown in Table 3-3.

It is intended that the tools which involve handling of tapes in the ALS host's tape library - BACKUP, BKPCHNG, BKPTREE, RESTORE, ROLLIN, ROLLOUT - will be used directly or in command procedures employed by a user acting in the capacity of a system operator.

Tools such as ARCHIVE and UNARCHIVE are to be employed by users in general to communicate with the rollout/rollin operations by sending lists of pathnames to protected files whose contents will be used to build parameter information for the ROLLOUT and ROLLIN tools. The name of the to-be -rolled-out file referred to in the description of ARCHIVE is .ALS.ARCHIVE.ARCHIVE_LIST. The name of the to-be-rolled-in file referred to in the description of UNARCHIVE is .ALS.ARCHIVE.UNARCHIVE_LIST.

In general, there are provisions for tape swapping in the event that data written by a tool cannot all be written on one tape volume.

Table 3-3

FILE ADMINISTRATOR TOOLS

NAME	SPECIFIED IN	DESCRIPTION
ARCHIVE	Appendix 70	Archive a set of file revisions
BACKUP	3.7.11.3	Write backup copy of entire database
BKPCHNG	3.7.11.3	Write backup copy of database changes
BKPTREE	3.7.11.3	Write backup copy of subtree(s) of database
CMPFILE	Appendix 70	Compare data of two file nodes
CMPNODE	Appendix 70	Compare nodes except for file data
CMPTEXT	Appendix 70	Compare text files
RECEIVE	Appendix 70	Receive subtree from tape
RESTORE	3.7.11.3	Restore all or part of database from backup copy
ROLLIN	3.7.11.3	Roll in file node(s) from archive tape
ROLLOUT	3.7.11.3	Roll out file node(s) to archive tape
TRANSMIT	Appendix 70	Transmit subtree to tape
UNARCHIVE	Appendix 70	Unarchive a set of file revisions

3.7.11.1 Concepts of rollout/rollin. - Infrequently used file data, associations, and attributes (except for a small number including derivation_count, availability, and archive_volume, which must remain on line) may be "rolled out" to archive tapes to reduce disk storage requirements.

Every time that an existing file node revision is rolled out, the ROLLOUT tool performs the following sequence of activities.

a. The user (who, by means of execute access protection on the ROLLOUT tool, is acting in the capacity of a system operator) is asked to mount an archive tape. Tapes containing archived nodes are dedicated to this use.

b. The file node is written to the tape at the end of other rolled out nodes previously written to this tape.

c. The archive_volume attribute is given a value which contains the tape volume name, which has the syntax of a restricted form of an Ada identifier.

d. The value of the availability attribute of the node is changed from on_line to off_line.

e. A protected directory called .ALS.ARCHIVE.TAPE_LIBRARY is accessed and the primary absolute path name of the node is appended to a file under this directory whose file name is the tape volume name.

f. The disk space used by the node information which has been written on tape is reclaimed.

From this time onward, even when the node is rolled back in, the information will remain on tape and the node's path name will remain in the .ALS.ARCHIVE.TAPE_LIBRARY.<volume_name> file.

Subsequently, rolling in the node is accomplished by reading the node information from tape and changing the value of the availability attribute to on_line; this is done by the ROLLIN tool. The pathname entry in the file .ALS.ARCHIVE.TAPE_LIBRARY.<volume_name> is enclosed in parentheses to indicate that the information is not current.

A full or tree backup of the ALS database (i.e., output of the BACKUP or BKPTREE tool) will not capture the data and most of the attributes and associations of a node which has been rolled out. However, the archive_volume attribute, which names the archive tape containing the node, will be captured by the full or tree backup. This implies that archive tapes as well as backup tapes must be preserved if it is desired to use backup tapes to retain the long-term state of the database.

The tapes produced by the backup and rollout tools are not interchangeable between host systems. Interchange of subtrees of the database between

hosts is achieved by use of the TRANSMIT and RECEIVE tools.

3.7.11.2 Concepts of backup. - Tapes containing backup data are dedicated to this use. Backup copies are created to permit reconstruction of all or portions of a database in the event of loss. Typically the BACKUP and BKPCHNG tools are employed periodically in an activity which is a part of routine installation management.

The BACKUP tool writes a copy of the entire disk-resident database to one or more tapes. The BKPCHNG tool writes to tape a record of all node changes since the last use of BACKUP or BKPCHNG, in effect updating the cumulative backup tape record begun with the most recent BACKUP. This recording of changes is made possible by the KAPSE, which keeps a running record of the names of all those nodes that have changed.

The KAPSE maintains two text files named .ALS.BACKUP.CHANGES_1 and .ALS.BACKUP.CHANGES_2. At any given time one of these two files is "active". The name of the active changes file is the value of the ACTIVE attribute of the directory .ALS.BACKUP; this attribute may have the value "1" or "2". Every time the content of a node is changed, the node name is appended to the active changes file.

When BACKUP or BKPCHNG is executed, the active and inactive files swap roles before anything is written to tape. Specifically, the following happens.

1. The inactive file is made empty.

2. The value of the ACTIVE attribute is toggled.

From the point in time that the ACTIVE attribute is toggled any node changes will be recorded in the file which was just cleared. If the BKPCHNG tool is the one which has caused this swap to occur, the names in the newly inactive changes file denote all nodes for which a change record must be written to tape.

The BKPTREE tool makes backup copies of subtrees of the database. It does not cause the swapping process described above to occur. It is made available for those who wish to create long-term copies or private backup copies in the backup tape format.

3.7.11.3 Tape-library-oriented Commands. -

ALS COMMAND DESCRIPTION ROLLOUT

NAME: ROLLOUT - Roll out file nodes to archive tape

FUNCTION: Write to archive tape a specified set of file data revisions, altering the nodes' attributes availability and archive_volume and reclaiming the disk space used for data, associations, and most attributes. ROLLOUT is typically used only by a system operator performing scheduled rollout. Users may request rollout by means of the ARCHIVE command.

FORMAT: ROLLOUT
(volume_name,file=>file_list[,OPT=>option_list])

PARAMETER DESCRIPTION:

volume_name: An identifier which must match the tape volume label before writing can take place.

file_list: File_list is a list of pathnames. Each pathname denotes an ALS text file, each of which contains a list of pathnames for rollout, one per line. Each node to be rolled out must be a specific revision of a frozen file node; it is processed as follows.

If availability = on_line, prompt the user to mount the specified archive tape, check that the tape volume label matches the volume_name parameter, and write data, associations, and most attributes to tape. Change the value of availability to off_line, give the archive_volume attribute the value of the tape volume name and reclaim the space used by the data, associations, and attributes (except for the attributes which remain on line).

option_list:

LIST	LIST writes to standard output the name of every node rolled out. Default: NO_LIST.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If any specified node does not exist, a diagnostic message naming that node is produced, and the ROLLOUT operation continues for the other, remaining specified nodes.

2. If any node specified for rollout is not a file node or else is the most recent revision and has not been explicitly frozen, a diagnostic message naming that node is produced and the node is not rolled out, and the ROLLOUT operation continues for the other, remaining specified nodes.

3. If the user does not have attribute change access to any node specified for rollout, a diagnostic message naming that node is produced, the node is not rolled out and the rollout operation continues for the other, remaining specified nodes.

4. FAILS if volume_name does not match tape volume label.

EXAMPLE:

```
ROLLOUT (T1234,file=>rollout_list)

--Every node named in the file named rollout_list is a frozen
--file node.  The user is asked to mount archive tape T1234,
--and all nodes named in the file rollout_list with availability
--attribute = on_line are written to tape; for each such node
--the archive_volume attribute is given the value T1234.
```

ALS COMMAND DESCRIPTION ROLLIN

NAME: ROLLIN - Roll in file node(s) from archive tape

FUNCTION: Read a specified set of file data revisions from one or more archive tapes, adjusting the values of their availability attributes.

FORMAT: ROLLIN (file=>file_list[,OPT=>option_list])

PARAMETER DESCRIPTION:

file_list: File_list is a list of pathnames. Each pathname denotes an ALS text file, each of which contains a list of pathnames for rollin, one per line. Each node specified for rollin is a specific revision of a frozen file node. Each node to be rolled in is processed as follows.

1. If the value of the availability attribute is on_line, no action is taken.

2. If the value of the availability attribute is off_line, the user (who is acting in the capacity of a system operator) is prompted to mount the archive tape named by the archive_volume attribute. Once that is done and the identity of the tape is verified, the tape is searched for the node information. The data, associations, and rolled out attributes are read and attached to the node, and the availability attribute is given the value on_line.

option_list:

LIST LIST writes to standard output the name of every node rolled in.
Default: NO_LIST.

DISPOSITION

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If any node specified for rollin is not an existing file node, a diagnostic message is produced and no attempt is made to roll in the node.

2. A diagnostic message is produced if the user does not have read access to any node named in file_list or to any node named in any file named in file_list.

3. A diagnostic message is produced if the volume label of a mounted tape does not match the tape volume requested. The user is given the option of trying again or of proceeding to the next node. If the latter is chosen, the node is named in a diagnostic message.

EXAMPLE:

```
ROLLIN (file=>config_c)

--The file config_c contains a list of file revision names.
--The user is prompted to mount the proper tape(s) and the files
--are rolled in.
```

ALS COMMAND DESCRIPTION BACKUP

NAME: BACKUP - Write backup copy of entire database

FUNCTION: Write to tape a backup copy of the entire ALS environment database.

FORMAT: BACKUP (volume_name,search_name [,OPT=>option_list])

PARAMETER DESCRIPTION:

volume_name: An identifier which must match the backup tape volume label before tape writing can take place. It has the form of an Ada identifier limited in length to six characters, and containing only digits and upper-case alphabetic characters.

search_name: This is an identifier which is used as a heading on tape for the entire output of one use of the tool. It permits the RESTORE tool to find the output of one or more specific uses of the BACKUP, BKPTREE, or BKPCHNG tool.

option_list:

APPEND: NO_APPEND writes at beginning of tape. APPEND appends information at end of written information on tape.
Default: APPEND.

LIST LIST writes to standard output the name of every node written to tape.
Default: NO_LIST.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If a node is changed while BACKUP is executing, the new state of the node may not be captured; it will, however, be captured by the next use of BACKUP, BKPTREE, or BKPCHNG.

2. Before writing begins, the ACTIVE attribute of .ALS.BACKUP is examined. If it is "1", then the data part of .ALS.BACKUP.CHANGES_2 is made empty; if it is "2", then the data part of .ALS.BACKUP.CHANGES_1 is made empty. Then the value of the ACTIVE attribute is toggled between "1" and "2", after which writing begins.

3. FAILS if volume_name does not match tape volume label.

4. FAILS if an attempt is made to create more than one set of output with the same search_name on one tape. This can only happen if the APPEND option is utilized.

EXAMPLE:

```
BACKUP (T1234,fb781205,opt=>no_append)

--Writes a backup copy of the entire database at the beginning of the
--tape T1234. The inactive changes file under .ALS.BACKUP is cleared
--and then becomes the active changes file.
```

ALS COMMAND DESCRIPTION BKPTREE-

NAME: BKPTREE - Write backup copy of subtree(s) of database

FUNCTION: Write to tape a backup copy of each subtree specified by the command.

FORMAT:
```
BKPTREE (volume_name,search_name
[,NODE=>list_1][,FILE=>list_2]
[,OPT=>option_list])
```

PARAMETER DESCRIPTION:

volume_name: An identifier which must match the backup tape volume label before tape writing can take place. It has the form of an Ada identifier limited in length to six characters, and containing only digits and upper-case alphabetic characters.

search_name: This is an identifier which is used as a heading on tape for the entire output of one use of the tool. It permits the RESTORE tool to find the output of one or more specific uses of the BACKUP, BKPTREE, or BKPCHNG tool.

list_1

list_2: The syntax of both list_1 and list_2 is a list of pathnames. At least one NODE or FILE parameter must be present. Each node specified for backup, whether by the NODE or FILE form, is the root of a subtree, all of which will be written by the tool. The NODE form specifies directly, in the parameter, one or more nodes for backup. The FILE form names one or more ALS text files each of which contains a list of pathnames for backup, one per line.

option_list:

APPEND: NO_APPEND writes at beginning of tape. APPEND appends information at end of written information on tape.
Default: APPEND.

LIST LIST writes to standard output the name of every node written to tape.
Default: NO_LIST.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If a node is changed while BKPTREE is executing, the new state of the node may not be captured; it will, however, be captured by the next use of BACKUP, BKPTREE, or BKPCHNG.

2. The use of the BKPTREE tool does not alter the states of the nodes .ALS.BACKUP.CHANGES_1, .ALS.BACKUP.CHANGES_2, or .ALS.BACKUP.

3. FAILS if volume_name does not match tape volume label.

4. A diagnostic message is issued if any pathname of list_1, list_2, or any pathname contained in a file named in list_2 does not exist or is not read-accessible to the user.

5. FAILS if an attempt is made to create more than one set of output with the same search_name on one tape. This can only happen if the APPEND option is utilized.

EXAMPLE:

```
BKPTREE (T1234,my_tree,node=>(),opt=>no_append)

--Writes a backup copy of the subtree rooted at the user's CWD
--on tape T1234, giving it the search name my_tree.
```

ALS COMMAND DESCRIPTION BKPCHNG

NAME: BKPCHNG - Write backup copy of database changes

FUNCTION: Write, to tape, a backup copy for each database change which has occurred since the last use of either BKPCHNG or BACKUP. Changes include

1. Creation or deletion of a node,
2. Renaming or sharing of a node,
3. Any alteration to attributes or associations, and
4. Any alteration to the data portion of files.

FORMAT: BKPCHNG (volume_name,search_name[,OPT=>option_list])

PARAMETER DESCRIPTION:

volume_name: An identifier which must match the backup tape volume label before writing can take place. It has the form of an Ada identifier limited in length to six characters, and containing only digits and upper-case alphabetic characters.

search_name: This is an identifier which is used as a heading on tape for the entire output of one use of the tool. It permits the RESTORE tool to find the output of one or more specific uses of the BACKUP, BKPTREE, or BKPCHNG tool.

option_list:

APPEND: NO_APPEND writes at beginning of tape. APPEND appends information at end of written information on tape.
Default: APPEND.

LIST LIST writes to standard output the name of every node written to tape.
Default: NO_LIST.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If a node is changed while BKPCHNG is executing, the new state of the node may not be captured. If it is not captured, the history of the node beginning with this change will be captured by the next use of BACKUP, BKPTREE, or BKPCHNG.

2. If a node has been changed more than once, changes other than the most recent may not be written.

3. Before writing begins, the ACTIVE attribute of .ALS.BACKUP is examined. If it is "1", then the data part of .ALS.BACKUP.CHANGES_2 is made empty; if it is "2", then the data part of .ALS.BACKUP.CHANGES_1 is made empty. Then the value of the ACTIVE attribute is toggled between "1" and "2", after which writing begins. The nodes written to tape are those named in the file named by the ACTIVE attribute prior to its being toggled.

4. FAILS if volume_name does not match tape volume label. In this case the tool may be reinvoked without loss of the file of changes.

5. FAILS if an attempt is made to create more than one set of output with the same search_name on one tape. This can only happen if the APPEND option is utilized.

EXAMPLE:

```
BKPCHNG (T1234,B810915)

--Appends to the backup tape T1234 the list of all node changes
--since the last use of BACKUP or BKPCHNG.  This list is given
--the heading B810915.
```

ALS COMMAND DESCRIPTION RESTORE

NAME: RESTORE - Restore all or a portion of the ALS database

FUNCTION: Read a tape written by the BACKUP, BKPTREE, or BKPCHNG tool and restore either all nodes or else only the node(s) indicated by the command.

FORMAT:
```
RESTORE (volume_name,search_name
[,NODE=>list_1][,FILE=>list_2]
[,OPT=>option_list])
```

PARAMETER DESCRIPTION:

volume_name: An identifier which must match the backup tape volume label before reading can take place. It has the form of an Ada identifier limited in length to six characters, and containing only digits and upper-case alphabetic characters.

search_name: This is an identifier which is used as a heading on tape for the entire output of one use of the tool. It permits the RESTORE tool to find the output of one or more specific uses of the BACKUP, BKPTREE, or BKPCHNG tool.

list_1

list_2: The syntax of both list_1 and list_2 is a list of pathnames. Neither, one, or both of the NODE or FILE forms may be present. The presence of neither form denotes a restoration from tape of all node contents (as written by BACKUP or BKPTREE) and/or changes (as written by BKPCHNG). The presence of one or both forms denotes a reconstruction of only the specified nodes. If neither form is present, the first BACKUP, BKPTREE, or BKPCHNG output on the tape with the same search name is found; each node named therein is either overwritten in the database, created, or destroyed, as appropriate.

If either the NODE or FILE form is present, the first file on the specified tape with the same search_name is found; then only the node(s) denoted by the parameter(s) is/are updated by overwriting, creation, or deletion, as prescribed on tape; the

database must not initially be in an empty state. The NODE form specifies directly (in the parameter) one or more pathnames to be selected from tape and updated. The FILE form names one or more ALS text files, each of which contains one or more path names, one per line, to be selected from tape and updated.

option_list:

LIST	LIST writes to standard output the name of every node restored. Default: NO_LIST.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if volume_name does not match tape volume label.
2. FAILS if search_name cannot be found on tape.
3. A diagnostic message is issued if any pathname in list_1 or list_2 or any pathname named in any file of list_2 does not exist or is not read-accessible to the user.

EXAMPLE:

```
RESTORE (BT53,BU070476)

--The user is asked to mount tape volume BT53.  After checking that
--the proper volume has been mounted, a search is made for the
--backup record named BU070476.
--All nodes are read in and the correspondingly named nodes of the
--ALS database are updated, created, or deleted, as appropriate.
```

3.7.12 Symbolic Debugger Functional Area. - The ALS VAX/VMS Symbolic Debugger provides interactive, symbolic debugging facilities for programs written in Ada and executing in the Ada Language System (ALS) environment. The debugger permits the user to execute an Ada program in a special mode that provides the following capabilites:

a. An executing Ada program may be interrupted and resumed.

b. The data and the control state of a program may be examined.

c. Variables in a program may be modified, with some restrictions.

d. The nest of active subprograms and blocks associated with the execution of a program and its tasks may be displayed.

e. Portions of the program, including comments, may be listed as they appear in the source.

f. Breakpoints may be inserted in the program to provide controlled execution.

g. A program may be executed in a stepwise fashion, one or several statements at a time.

h. The ALS VAX/VMS Symbolic Debugger shall be designed to be retargettable.

3.7.13 Statistical And Frequency Analyzer Functional Area. - The statistical and frequency analyzer provides the capability to analyze the timing and frequency execution characteristics of programs written in Ada and executing on the host computer. The timing and frequency analyzer is composed of the following three major components:

a. VAX/VMS Statistical Analyzer,

b. VAX/VMS Frequency Analyzer, and

c. Profile Display Tool.

3.7.13.1 Statistical Analyzer. - The VAX/VMS Statistical Analyzer provides the capability of monitoring the execution time characteristics of Ada programs executing on the host computer. The Statistical Analyzer provides the following capabilities:

a. Ability to monitor execution by periodically recording the locus of control.

b. Ability to control the sampling interval, see 40.1.2.

c. Ability to name the file where timing data is to be recorded, see 40.1.2.

d. Ability to cause execution statistics to be monitored by using the EXPVMS STAT option.

The Statistical Analyzer shall produce a file containing the recorded data. This file can be subsequently used as input to the Profile Display Tool, described in Section 3.7.13.3.

3.7.13.2 Frequency Analyzer. - The VAX/VMS Frequency Analyzer provides the capability of monitoring the execution frequency characteristics of Ada programs executing on the host computer. The Frequency Analyzer provides the following capabilities:

a. Ability to monitor execution frequency at the basic block level.

b. Ability to cause frequency monitoring code to be created by using the compiler FREQUENCY option.

c. Ability to cause the inclusion of the frequency monitor kernel in the target program image by using the EXPVMS FREQUENCY option.

d. Ability to name the file where frequency data is to be recorded.

The Frequency Analyzer shall produce a file containing the recorded frequency data. This file can be subsequently used as input to the Profile Display Tool, described in 3.7.13.3.

3.7.13.3 Profile Display Tool. - The Profile Display Tool provides the capability to display the recorded timing and frequency data. This tool is more fully described in Appendix 70. The Profile Display Tool accepts files produced by the Statistical Analyzer and Frequency Analyzer as input. As output, the Profile Display Tool generates one or more histograms or "profiles" showing the distribution of execution time and frequency with respect to program location. The format of this display is shown in Figure 3-12 and may vary depending upon line length and other display options.

<TBD>

Figure 3-12. Statistical and Frequency Data Display Format

3.8 Order of Precedence. -

3.8.1 Conflict Resolution. - In the event of conflict between this specification and other Ada Language System specifications and documents, the conflict shall be resolved by the government.

3.8.2 Contract Precedence. - This specification shall be subordinate to Contract DAAK80-80-C-0507.

4. QUALITY ASSURANCE PROVISIONS

4.1 General. - The Ada Language System shall be tested in accordance with the ALS Quality Assurance Plan and the approved Preliminary/Formal Qualification Test Plan (2.1). Prior to formal qualification testing of the ALS, each relevant CPCI listed in 3.1.1.5 shall be tested in accordance with the Preliminary Qualfication Test Procedure for the item prepared as specified in 3.4.4.

4.1.1 Responsibility for Tests. - The contractor shall be responsible for conducting the formal qualification tests. The tests shall be conducted at Fort Monmouth, NJ, on the SDSS facility. The tests may be witnessed by CECOM personnel and other designated government personnel.

4.2 Quality Conformance Inspections. - Compliance with the requirements of Section 3 of this specification shall be demonstrated by fulfillment of the criteria specified in the Preliminary/Formal Qualification Test Plan (2.1) and in the Physical Configuration Audit.

5. PREPARATION FOR DELIVERY

5.1 Ada Language System. -

5.1.1 Delivery Format. - The Ada Language System source and object code shall be delivered in a format compatible for installation on the SDSS facility at Ft. Monmouth, New Jersey. When installed on the SDSS, the ALS software shall be structured to allow partitioning for subsequent distribution of Ada Language Systems supporting any subset of the target environments specified in 3.1.1.2. The tape reels shall be labeled in accordance with 3.3.3.

5.1.2 Phased Delivery. - Separate deliveries shall be made for each target environment in 3.1.1.2 in accordance with the schedule in the ALS Design and Development Plan (2.1).

At the time of the first such delivery, installation will require the establishment of an ALS Environment Database on the SDSS host facility. Each subsequent delivery will consist of ALS nodes, introduced into the existing database using the ALS File Administrator.

All deliveries will consist of source code as well as executable load modules for all tools intended to run on the host machine. Runtime libraries and loaders will be delivered in source code as well as in Containers, acceptable for input to the appropriate Linker CPCI.

5.2 Ada Language System Documentation. - ALS documentation will be delivered in accordance with the requirements of the Contract Data Requirements List in Contract DAAK80-80-C-0507 and the referenced Data Item Descriptions.

5.3 On-Line Documentation. - All specifications will be delivered on magnetic tape in a format appropriate for input to the DEC Standard Runoff Formatter.

6. NOTES

6.1 Glossary. - A glossary of terms used in the Ada Language System is provided in Table 6-1.

6.2 Acronyms. - A list of the acronyms used in the Ada Language System is provided in Table 6-2.

Table 6-1

GLOSSARY

Access type	An access type is a type whose objects are created by execution of an allocator. An access value designates such an object.
Access value	See access type.
Accessible Container Set	The collection of Containers which are accessible to a tool at anytime. This collection consists of at most one current Container and zero or more Program Library Containers.
Accuracy constraint	See constraint.
Activation record	An element of a runtime stack containing both dynamic information about a block or subprogram and objects (other than access objects) local to the block or subprogram.
Adaname	The name for a container or subtree in a program library. Adanames are composed of the source unit name dot-qualified by the keywords SPEC, BODY, or ALL, referring to the source unit specification, body or whole subtree respectively. The source unit name starts with the library unit name, with subunit names appearing in dot-qualification.
Aggregate	An aggregate is a written form denoting a composite value. An array aggregate denotes a value of an array type; a record aggregate denotes a value of a record type. The components of an aggregate may be specified using either positional or named association.
Allocator	An allocator creates a new object of an access type, and returns an access value designating the created object.
ALS prepared disk	A disk which has undergone the initialization process specified in the ALS VAX Operator's Manual.
ALS program	Any program executing in the VAX/VMS target

environment, and utilizing services of the KAPSE is said to be an ALS program.

Ancestor compilation unit

An ancestor compilation unit of a compilation unit currently being compiled is a member of the following set:

a. A unit mentioned in a WITH clause of the compilation unit currently being compiled;

b. An outer textually-nested unit containing the unit currently being compiled, if that unit is a subunit;

c. The specification part of a subprogram or package body currently being compiled;

d. One of the units mentioned in WITH clause(s) of the ancestor compilation units defined in parts (b) and (c) above; and

e. Package STANDARD.

In short, it is any compilation unit which is made visible to a compilation unit currently being compiled, not including the unit currently being compiled itself.

Ancestor node

See node ancestors.

Attribute

An attribute is a predefined characteristic of a named entity.

Baseline

An unchanging configuration used for a point of reference. A snapshot of a configuration representing the state of the configuration at one point in time. Any modification of a baseline results in a different baseline by definition. Two baselines are the same if and only if they have the same names and contents.

Baseline configuration

A baseline.

Basic block

A sequence of one or more simple statements th.., barring the occurrence of exceptions, are always executed the same number of times.

Body

A body is a program unit defining the

	execution of a subprogram, package, or task.
Body stub	A body stub is a replacement for a body that is compiled separately.
CALL mechanism	The KAPSE service by which one ALS program invokes another, passes parameters to it, and receives results back.
CALL tree	The KAPSE data structure that contains information about the executing programs in a job. It is composed of one node for each program.
Collection	A collection is the entire set of allocated objects of an access type.
Compilation unit	A compilation unit is a program unit presented for compilation as an independent text. It is preceded by a context specification, naming the other compilation units on which it depends. A compilation unit may be the specification or body of a subprogram or package.
Complete program	A program with no unresolved external references.
Component	A component denotes a part of a composite object. An indexed component is a name containing expressions denoting indices, and names a component in an array or an entry in an entry family. A selected component is the identifier of the component, prefixed by the name of the entity of which it is a component.
Composite type	An object of a composite type comprises several components. An array type is a composite type, all of whose components are of the same type and subtype; the individual components are selected by their indices. A record type is a composite type whose components may be of different types; the individual components are selected by their identifiers.
Composite value	See aggregate.
Configuration	A named collection of database objects treated as a single entity. Generally, a configuration would be represented as one subtree in the database, but this definition

does not produce a disjoint representation.

Constraint

A constraint is a restriction on the set of possible values of a type. A range constraint specifies lower and upper bounds of the values of a scalar type. An accuracy constraint specifies the relative or absolute error bound of values of a real type. An index constraint specifies lower and upper bounds of an array index. A discriminant constraint specifies particular values of the discriminants of a record or private type.

Container

A Container is a data structure which is created by the Program Library Manager and initialized by the Container Data Manager. It consists of a collection of related c-nodes which are inserted by a tool using the services of the Container Data Manager. A Container occupies the data portion of a file.

C_node

C_nodes are the records of which Containers are composed. Each C_node is a collection of attributes, each of which has a name and location. C_nodes are created and maintained by the Container Data Manager for use by ALS tools which need dynamic storage allocation.

Context specification

A context specification, prefixed to a compilation unit, defines the other compilation units upon which it depends.

Current Container

A Container which is in the process of being constructed by a tool. A current Container may have permanent and temporary c-nodes.

Declarative part

A declarative part is a sequence of declarations and related information such as subprogram bodies and representation specifications that apply over a region of a program text.

Declaring unit

The declaring unit of an Ada subunit is the compilation unit which has the declaration of the subunit.

Derived type

A derived type is a type whose operations and values are taken from those of an existing type.

Discrete type	A discrete type has an ordered set of distinct values. The discrete types are the enumeration and integer types. Discrete types may be used for indexing and iteration, and for choices in case statements and record variants.
Discriminant	A discriminant is a syntactically distinguished component of a record. The presence of some record components (other than discriminants) may depend on the value of a discriminant.
Discriminant constraint	A discriminant constraint specifies particular values of the discriminants of a record or private type.
Elaboration	Elaboration is the process by which a declaration achieves its effect. For example it can associate a name with a program entity or initialize a newly declared variable.
Entity	An entity is anything that can be named or denoted in a program. Objects, types, values, program units are all entities.
Entry	An entry is used for communication between tasks. Externally an entry is called just as a subprogram is called; its internal behavior is specified by one or more accept statements specifying the actions to be performed when the entry is called.
Enumeration type	An enumeration type is a discrete type whose values are given explicitly in the type declaration. These values may be either identifiers or character literals.
Evaluation ordering	A restructuring of the prefix expressions created by the WALKER function of the Code Generator in order to generate more efficient code.
Exception	An exception is an event that causes suspension of normal program execution. Bringing an exception to attention is called raising the exception.
Exception handler	An exception handler is a piece of program text specifying a response to the exception. Execution of such a program text is called handling the exception.

Expression	An expression is a part of a program that computes a value.
Foster child	An offspring of a directory (or variation header) acquired by a "share" operation (as opposed to "make") is said to be a foster child of that directory. (See also foster parent, true parent, and true child.)
Foster parent	A directory (or variation header) which acquires an offspring by a "share" operation (as opposed to "make") is said to be a foster parent of that offspring. (See also foster child, true parent, and true child.)
Generic program unit	A generic program unit is a subprogram or package specified with a generic clause. A generic clause contains the declaration of generic parameters. A generic program unit may be thought of as a possibly parameterized model of program units. Instances (that is, filled-in copies) of the model can be obtained by generic instantiation. Such instantiated program units define subprograms and packages that can be used directly in a program.
Image file	VMS name for a load module.
Incomplete program	A program in which some external references are unresolved.
Indexed component	See component.
Index constraint	See constraint.
Introduce	An identifier is introduced by its declaration at the point of its first occurrence.
I/O stream	A (KAPSE-owned) data structure associated with an open file. The stream contains all information necessary to perform I/O operations on the file.
Job	The total computational activity performed on the behalf of a user from the time of ALS log-in through ALS log-out.
Lexical unit	A lexical unit is one of the basic syntactic elements making up a program. A lexical unit is an identifier, a number, a character literal, a string, a delimiter, or a

	comment.
Library subprogram body	The body of a subprogram which is not a subunit.
Link library	A container presented to the linker whose component compilation units are included in the linked container only if they are referenced.
Literal	A literal denotes an explicit value of a given type, for example, a number, an enumeration value, a character, or a string.
Load module	A file containing machine code which represents an executable program for a particular target machine.
Locator	A short name for a node through which its physical storage may be found. At any instant, a locator uniquely identifies a single node. When a node is destroyed its locator may be recycled and used later for a newly-created node. (See also serial number.)
Logged program	An executable program for which derivation information is kept and given to created files.
Logged sequence	A subtree of executing programs for which the root is a logged program, and for which the parent of the root is not a logged program.
Main subprogram	The subprogram which initially receives control at execution time.
Model number	A model number is an exactly representable value of a real numeric type. Operations of a real type are defined in terms of operations on the model numbers of the type. The properties of the model numbers and of the operations are the minimal properties preserved by all implementations of the real type.
Node ancestors	The ancestors of a node are its true and foster parents plus the ancestors of these parents. The root node has no ancestors.
Node locator	See locator.

Node serial number	See serial number.
Object	An object is a variable or a constant. An object can denote any kind of data element, whether a scalar value, a composite value, or a value in an access type.
Option	An on/off switch used to control tool execution.
Original container	An original container is a container produced by a compiler, assembler, or importer and represents only one compilation unit.
Overloading	Overloading is the property of literals, identifiers, and operators that can have several alternative meanings within the same scope. For example an overloaded enumeration literal is a literal appearing in two or more enumeration types; an overloaded subprogram is a subprogram whose designator can denote one of several subprograms, depending upon the kind of its parameters and returned value.
Package	A package is a program unit specifying a collection of related entities such as constants, variables, types and subprograms. The visible part of a package contains the entities that may be used from outside the package. The private part of a package contains structural details that are irrelevant to the user of the package but that complete the specification of the visible entities. The body of a package contains implementations of subprograms or tasks (possibly other packages) specified in the visible part.
Parameter	A parameter is one of the named entities associated with a subprogram, entry, or generic program unit. A formal parameter is an identifier used to denote the named entity in the unit body. An actual parameter is the particular entity associated with the corresponding formal parameter in a subprogram call, entry call, or generic instantiation. A parameter mode specifies whether the parameter is used for input, output or input-output of data. A positional parameter is an actual parameter passed in positioned order. A named

	parameter is an actual parameter passed by naming the corresponding formal parameter.
Parent node	A directory or variation header that contains a node is said to be a parent of that node. (See also true parent and foster parent).
Pragma	A pragma is an instruction to the compiler, and may be language defined or implementation defined.
Primary machine	When a program is running on a distributed target, the primary machine is the machine on which the declaring unit of the remote task resides.
Private type	A private type is a type whose structure and set of values are clearly defined, but not known to the user of the type. A private type is known only by its discriminants and by the set of operations defined for it. A private type and its applicable operations are defined in the visible part of a package. Assignment and comparison for equality or inequality are also defined for private types, unless the private type is marked as limited.
Program	A collection of one or more compilation units which have all been compiled relative to each other with one of the subprograms designated to be the main subprogram. (This term can also refer to the source code for a collection of compilation units where the intention is that they will be compiled relative to each other and that there is an intended main program.)
Program library	The compilation units of a program are said to belong to a program library. The program library establishes the name scope for names mentioned in WITH clauses. It is the set of all units which have been compiled relative to each other.
Program library Container	A Container which has been produced by a tool and entered into the program library.
Qualified expression	A qualified expression is an expression qualified by the name of a type or subtype. It can be used to state the type or subtype of an expression, for example for an

	overloaded literal.
Range	A range is a contiguous set of values of a scalar type. A range is specified by giving the lower and upper bounds for the values.
Range constraint	See constraint.
Rendezvous	A rendezvous is the interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task.
Representation specification	Representation specifications specify the mapping between data types and features of the underlying machine that execute a program. In some cases, the specifications completely specify the mapping, in other cases the specifications provide criteria for choosing a mapping.
Remote machine	When a program is running on a distributed target, the remote machine is the machine on which the separated task is running. The remainder of the program is on the primary machine.
Revision	An instance of a file; each revision supersedes all previous ones.
Revision set	The collected revisions of a file viewed as a single set of files.
Runtime nucleus	The runtime nucleus is a portion of the runtime support library that is linked separately from the user programs.
Runtime stack	A data structure, associated with a task object, containing an activation record for each currently-active block or subprogram in the task object.
Scalar types	A scalar type is a type whose values have no components. Scalar types comprise discrete types (that is, enumeration and integer types) and real types.
Scope	The scope of a declaration is the region of text over which the declaration has an effect.

Selected component	See component.
Serial number	A short name by which a node may be unambiguously identified. At the time of creation, each node is assigned a unique serial number. Serial numbers are never recycled. (See also locator.)
Snapshot container	An image of the current Container (including temporary nodes) together with a list of accessible Program Library Containers. A snapshot Container can be used for the restart of a tool from the point of the snapshot or for input to the Display Tool.
Software configuration	See configuration.
Static expression	A static expression is one whose value does not depend on any dynamically computed values of variables.
Subprograms	A subprogram is an executable program unit, possibly with parameters for communication between the subprogram and its point of call. A subprogram declaration specifies the name of the subprogram and its parameters; a subprogram body specifies its execution. A subprogram may be a procedure, which performs an action, or a function, which returns a result.
Subtype	A subtype of a type is obtained from the type by constraining the set of possible values of the type. The operations over a subtype are the same as those of the type from which the subtype is obtained.
Task	A task is a program unit that may operate in parallel with other program units. A task specification establishes the name of the task and the names and parameters of its entries; a task body defines its execution. A task type is a specification that permits the subsequent declaration of any number of similar tasks.
True child	An offspring of a directory (or variation header) acquired by a "make" operation (as opposed to "share") is said to be a true child of that directory. (See also true parent, foster parent, and foster child.)
True parent	A directory (or variation header) that

	acquires an offspring by a "make" operation (as opposed to "share") is said to be the true parent of that offspring. (See also true child, foster parent, and foster child.)
Type	A type characterizes a set of values and a set of operations applicable to those values and a set of operations applicable to those values. A type definition is a language construct introducing a type. A type declaration associates a name with a type introduced by a type definition.
Usable container	Whenever an ALS tool produces a container it is marked as usable or unusable. A usable container can be used as input to a link, to a compile as context or as export, as well as input to the Display Tool CPCI. An unusable container can be used only as input to the Display Tool CPCI.
Use clause	A use clause opens the visibility to declarations given in the visible part of a package.
Variable	A variable is an object that is not constant (that is, the reserved word constant does not appear in its object declaration, and the object is not a component of a constant array or constant record).
Variant	A variant part of a record specifies alternative record components, depending on a discriminant of the record. Each value of the discriminant establishes a particular alternative of the variant part.
Variation	An element in a variation set.
Variation set	A set of incarnations of an object, each element co-exists with the others. (See also revision set.)
Visibility	At a given point in a program text, the declaration of an entity with a certain identifier is said to be visible if the entity is an acceptable meaning for an occurrence at that point of the identifier.

Table 6-2

LIST OF ACRONYMS

ACRONYM	MEANING
AFE	Assembler Front-End
ALS	Ada Language System
CCT	Configuration Control Tools
CDM	Container Data Manager
CHD	Current HELP Directory
CI	Configuration Item
CL	Command Language
CLP	Command Language Processor
CM	Configuration Management
CPC	Computer Program Component
CPCI	Computer Program Configuration Item
CPT&E	Computer Programming Test and Evaluation
CUT	Compilation Units Table
CWD	Current Working Directory
DBM	Database Manager
DT	Display Tools
ECP	Engineering Change Proposal
FA	File Administrator
FQT	Formal Qualification Testing
IL	Intermediate Language
KAPSE	Kernel Ada Programming Support Environment
MI	Machine-Independent section of the ALS compiler
MTF	Machine Text Formatter

Table 6-2 (cont.)

ACRONYM	MEANING
PL	Project Library
PLM	Program Library Manager
PQT	Preliminary Qualification Testing
PSECT	Program section
RO	Read-only
RSL	Runtime Support Library
RW	Read-write
SCN	Specification Change Notice
SOW	Statement Of Work
SDSS	Software Development Support System
WBS	Work Breakdown Structure

PREFACE
to
APPENDIXES

Descriptions of the functions of the Ada Language System included in the following appendixes shall be considered as requirements for the performance, development, and design of the system. In particular, the word "will" in the appendixes shall be the same as "shall" as defined in Par. 3.2.3.6 of MIL-STD-490 (2.1).

APPENDIX 10

10. ADA LANGUAGE SYSTEM IMPLEMENTATION DEPENDENCIES

In accordance with the requirements of Appendix F of the Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 17 February 1983 (2.1) descriptions of the Ada language implementation dependencies for each target environment in 3.1.1.2 are provided on the following pages.

This appendix contains:

a. The Ada Language for the VAX/VMS Target, and

b. The Ada Language for the MCF Target.

10.1 The Ada Language For The VAX/VMS Target.

The source language accepted by the compiler is Ada, as described in the Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 17 February 1983 ("Ada Reference Manual") (2.1).

The Ada definition permits certain implementation dependencies. Each Ada implementation is required to supply a complete description of its dependencies, to be thought of as Appendix F to the Ada Reference Manual. This section is that description for the VAX/VMS target.

10.1.1 Pragmas.

10.1.1.1 Pragma Definition.

The following are implementation-defined pragmas:

pragma PAGE;

> This is a listing-control pragma. If the source text to be listed contains any PAGE pragmas, the line on which the token PAGE appears is placed at the first line available on a new page in the source listing.

pragma TITLE (arg);

> This is a listing control pragma. It specifies a CHARACTER string that is to appear on the second line of each page of every listing produced for a compilation unit. At most one such pragma may appear for any compilation unit, and it must be the first lexical unit in the compilaton unit (comments excepted). The argument is a CHARACTER string.

The following notes specify the language-required definitions of the predefined pragmas. Unmentioned pragmas require no notes. (See Appendix B of the Ada Reference Manual.)

pragma INTERFACE (arg,arg);

> No INTERFACE pragmas are recognized.

pragma MEMORY_SIZE (arg)

> The MEMORY_SIZE pragma is ignored, other than to verify that the value of the argument is in the range 0..(2**30)-1.

pragma OPTIMIZE (arg)

> This pragma is effective only when the "OPTIMIZE" option has been given to the compiler, as described in 3.7.1.1.1.3 of this specification. The pragma is ignored when applied to inline

subprograms. The argument is either TIME or SPACE.

pragma PRIORITY (arg)

The PRIORITY argument is an integer static expression value in the range 1..15. The pragma has no effect in a location other than a task (type) specification or outermost declarative part of a subprogram. If the pragma appears in the declarative part of a subprogram, it has no effect unless that subprogram is designated as the "main" subprogram at link time.

pragma STORAGE_UNIT (arg)

If the argument is a value other than 8, a diagnostic of severity WARNING is generated. (See Appendix 80 for a complete summary of all diagnostic messages.) Otherwise, no action is taken.

pragma SUPPRESS (arg[,arg])

SUPPRESS pragmas which mention the following CHECK names have no effect:

```
DIVISION_CHECK
OVERFLOW_CHECK
```

These checks cannot be suppressed.

pragma SYSTEM (arg)

The SYSTEM argument is a value of the enumeration type SYSTEM.SYSTEM_NAME. The purpose of this pragma is to assert that the Ada compilation unit is specially designed to execute only on certain target environments.

The value VAX780_VMS means that the unit is designed to run on VAX under the VMS operating system; VAX780 means VAX under any or no operating system. For other values, a diagnostic of the severity WARNING is generated. (See Appendix 80 for a complete summary of all diagnostic messages).

10.1.1.2 Scope of Pragmas.

CONTROLLED	Applies only to the access type named in its argumeni
INLINE	Applies only to the subprograms named in its arguments. If the argument is an overloaded subprogram name, the INLINE pragma applies to all definitions of that subprogram name which appear in the same declarative part as the INLINE pragma.
LIST	In effect until the next LIST pragma in the source or included text, or if none, the end of the compilation unit
MEMORY_SIZE	Applies to the entire Program Library in which it appears. Multiple, conflicting occurrences in a Program Library are erroneous.
OPTIMIZE	Applies to the entire compilation unit in which it appears
PACK	Applies only to the array or record named in its argumeni
PAGE	No scope
PRIORITY	Applies only to the task specification or main subprogram in which it appears
STORAGE_UNIT	Applies to the entire compilation unit before which it appears
SUPPRESS	Applies to the block or body that contains the declarative part in which the pragma appears
SYSTEM	Applies to the entire compilation unit before which it appears
TITLE	No scope

10.1.2 Attributes.

There is one implementation-defined attribute in addition to the predefined attributes found in Appendix A of the Ada Reference Manual.

X'DISP

A value of type DISPLACEMENT which corresponds to the displacement that is used to address the first storage unit occupied by a data object X at a static offset within an implemented activation record. The type DISPLACEMENT is defined as:

```
type DISPLACEMENT is range -(2**(4*STORAGE_UNIT -1))..(2**(4*STORAGE_UNIT -1) -1);
```

This attribute differs from the ADDRESS attribute in that ADDRESS supplies the absolute address while DISP supplies the displacement relative to some base value (such as a stack frame pointer). It is the user's responsibility to determine the base value relevant to the attribute. The runtime environment is described in Section 20.1.9.

The following notes augment the language-required definitions of the predefined attributes found in Appendix A of the Ada Reference Manual.

T'MACHINE_ROUNDS	is false.
T'MACHINE_RADIX	is 2.
T'MACHINE_MANTISSA	if the size of the base type of T is 32, MACHINE_MANTISSA is 24. Otherwise it is 56.
T'MACHINE_EMAX	is 127.
T'MACHINE_EMIN	is -127.
T'MACHINE_OVERFLOWS	is true.

10.1.3 Predefined Language Environment.

The Package STANDARD contains the following definitions in addition to those specified in Appendix C of the Ada Reference Manual:

```
        type INTEGER is range -2**15..2**15-1;
        type LONG_INTEGER is range -2**31..2**31-1;
        --SHORT_INTEGER is not defined

        type FLOAT is digits 7 range
      -(2#0.1111_1111_1111_1111_1111_1111#E127)
      ..(2#0.1111_1111_1111_1111_1111_1111#E127);
        type LONG_FLOAT is digits 9 range
-(2#0.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E127)
..(2#0.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E127
        --SHORT_FLOAT is not defined

        subtype PRIORITY is INTEGER range 1..15;
        type DURATION is delta 2#.0000001# range -16777216.0 .. 16777215.99;
                    --Delta 1/128

    Package SYSTEM is

        type SYSTEM_NAME is (VAX780_VMS, VAX780, MCF);
        NAME: constant SYSTEM_NAME := VAX780_VMS;
        STORAGE_UNIT: constant := 8;
        MEMORY_SIZE: constant := (2**30)-1;
        MIN_INT: constant := -(2**31);
        MAX_INT: constant := (2**31)-1;
        UNRESOLVED_REFERENCE: exception; --see Appendix 30 of this
                                         --specification
        end SYSTEM;
```

10.1.4 Representations and Declaration Restrictions.

Representation specifications are described in Chapter 13 of the Ada Reference Manual. Declarations are described in Chapter 3 of that manual.

10.1.4.1 Integer Types.

Integer types are specified with constraints of the form

```
range L..R
```

where

```
R ≤ SYSTEM.MAX_INT
L ≥ SYSTEM.MIN_INT
```

Length specifications of the form:

```
for T'SIZE use N;
```

may specify integer values N such that

```
N = 32 or
N in 2..16,
```

and such that

```
R ≤ 2**(N-1)-1
and L ≥ -2**(N-1)
```

or else such that

```
R ≤ (2**N)-1
L ≥ 0
and 0 < N 15
```

When no length specification is provided, a SIZE of 16 is used when

```
R ≤ 2**15-1
and  L ≥ -2**15
```

Otherwise SIZE is 32.

10.1.4.2 Floating Types.

Floating types are specified with constraints of the form:

```
digits D
```

where
D is an integer value in 1..9.

Length specifications of the form
```
for T'SIZE use N;
```
may specify integer values N = 32 when D ≤ 7,
or N = 64 when D ≤ 9.

Where no length specification is provided, a size of 32 is used when D ≤ 7, 64 when D is 8..9.

10.1.4.3 Fixed Types.

Fixed types are specified with constraints of the form

```
delta D range L..R
```

where

$$\underset{\text{actual delta}}{\max\,(\text{abs}(R),\ \text{abs}(L))} \leq 2^{**}31-1$$

The actual delta defaults to the largest integral power of 2 less than or equal to the specified delta D. (This implies that fixed values are stored right-aligned.) For specifications of the form:

```
for T'ACTUAL_DELTA use X;
```

X must be specified as an integral power of 2 such that
X ≤ D

Length specifications of the form

```
for T'SIZE use N;
```

are permitted only when N = 32.
All fixed values have SIZE = 32.

10.1.4.4 Enumeration Types.

The default SIZE for an enumeration type is 8, unless there are more than 256 values. In the latter case the default size is 16. The size for the predefined type BOOLEAN is one. A length specification of the form

```
for T'SIZE use N;
```

is permitted for N in 1..16.

Enumeration types may have up to 2**SIZE different values.

An enumeration type representation of the form

```
for T use aggregate;
```

may specify codes in the range 0..(2**SIZE)-1

10.1.4.5 Access Types.

No representation specification is permitted for access types. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

10.1.4.6 Arrays and Records.

SIZE specifications of the form

```
for T'SIZE use N;
```

are not permitted for arrays and records. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

The PACK pragma may be used to minimize wasted space between components of arrays and records. The pragma causes the type representation to be chosen such that storage space requirements are minimized at the possible expense of data access time and code space. The PACK pragma may not be used in conjunction with a representation specification for the same type. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

A record type representation specification may be used to describe the allocation of components in a record. Bits are numbered 0..7 from the right. (Bit 8 starts at the right of the next higher-numbered byte.) Each location specification must allow at least X bits of range, where X is the SIZE of the type of the component being allocated. If X $\geq$ 32, the range

must be of the form 0..X-1, thereby guaranteeing byte alignment. If X < 32, unaligned allocation is legal, with the range being any correct-length subrange of 0..31. Components that are arrays, records, tasks, or access variables may not be allocated to specified locations. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

The alignment clause of the form

```
at mod N
```

may specify alignments of 1 (byte), 2 (word), 4 (longword), or 8 (quadword).

10.1.4.7 Other Length Specifications.

Length Specifications are described in Section 13.2 of the Ada Reference Manual.

T'STORAGE_SIZE for access type T - This length specification is ignored.

T'STORAGE_SIZE for task type T - Specifies the number of bytes to be allocated for the runtime stack of each task object of type T.

10.1.5 System Generated Names.

There are no system generated names.

10.1.6 Address Specifications.

Refer to Par. 13.5 of the Ada Reference Manual for a discussion of Address Specifications. Address specifications are not permitted.

10.1.7 Unchecked Conversions.

Refer to Par. 13.10.2 of the Ada Reference Manual for a description of UNCHECKED_CONVERSION.

A program is erroneous if it performs UNCHECKED_CONVERSION when the source and target have different size.

10.1.8 Input/Output.

Refer to Chapter 14 of the Ada Reference Manual for a discussion of Input/Output.

The type COUNT that appears in Par. 14.2 and 14.3 is defined as follows:

```
type COUNT is range 0..LONG_INTEGER'LAST;
```

10.1.8.1 Naming External Files:.

An external file name can be any valid path name, as defined in Appendix 50 of this specification.

10.1.8.2 The FORM Specification for External Files.

The "FORM" string parameter to the CREATE and OPEN procedures in packages DIRECT_IO, SEQUENTIAL_IO and TEXT_IO is saved as long as the file is open to allow the user to retrieve its value via calls to the FORM procedure, but it is not used in any way by these packages.

10.1.8.3 File Processing.

Processing allowed on ALS nodes is determined by the access controls (see Appendix 50) set by the owner of the file and by the physical characteristics of the underlying device. The following restrictions apply:

- A user may open a file as an IN_FILE only if that user has read access to the node. A user may open a file as an OUT_FILE only if that user has write access to the node. Finally, a user may open a file as an INOUT_FILE only if that user has read and write access to the node.

- No positioning operations are allowed on files associated with a printer or terminal. USE_ERROR is raised if this is violated.

- Writing a record on a file associated with a tape adds the record to the file such that the record just written becomes the last record of the file.

10.1.8.4 Text Input/Output.

At the beginning of program execution, the STANDARD_INPUT file and the STANDARD_OUTPUT file are open, and associated with the ALS-supported standard input and output files.

A program is erroneous if concurrently executing tasks attempt to perform overlapping GET and/or PUT operations on the same terminal.

Because of the physical nature of DecWriters and Video terminals, the semantics of text layout as specified in Ada Reference Manual Par. 14.3.2 (especially the concepts of current column number and current line) cannot be guaranteed when GET operations are interleaved with PUT operations. Programs which rely on the semantics of text layout under those circumstances are erroneous. For such devices, the function LINE raises USE_ERROR.

The standard output file has default line length of 120. SET_LINE_LENGTH can be used to set a length no longer than the 120 maximum. (An attempt to set a longer length raises USE_ERROR).

DecWriter and Video input is inherently a variable-line-length operation; SET_LINE_LENGTH raises USE_ERROR when applied, with a value other than zero, to the standard input file.

10.1.8.5 Low Level Input-Output.

The subroutines SEND_CONTROL and RECEIVE_CONTROL do not exist.

10.1.8.6 Hardware Interrupts.

The runtime support library for the VAX/VMS target does not handle hardware interrupts. All hardware interrupts are handled by the VMS operating system.

10.1.9 Character Set.

Ada compilations may be expressed using the following characters, in addition to the basic character set:

- The lower case letters
- ! $? ` @ [\ ] ^ ' { } ~

The following transliterations are not permitted (see Par. 2.10 of the Ada Reference Manual):

- Exclamation mark for vertical bar,
- Colon for sharp, and
- Percent for double-quote.

10.1.10 Machine Code Insertions.

The Ada language definition permits machine code insertions as described in Section 13.8 of the Ada Reference Manual. This section describes the implementation specific details for writing machine code insertions as provided by the predefined package MACHINE_CODE.

10.1.10.1 Machine Features.

This section describes specific machine language features which are needed to write code statements. These machine features include the DISP and ADDRESS attributes and the address mode specifiers. The address mode specifiers make it possible to describe both the address mode and register number of any operand as a single value by mapping these values directly onto the first byte of each operand. The following is an enumeration of

all mode specifiers:

```
--
--  The first 64 are the short literal modes.
--  These mode specifiers signify (short literal mode, value)
--  combinations.  The values are in the range 0 to 63.
--

                    L0,         L1,         L2,         L3,
                    L4,         L5,         L6,         L7,
                    L8,         L9,         L10,        L11,
                    L12,        L13,        L14,        L15,

                    L16,        L17,        L18,        L19,
                    L20,        L21,        L22,        L23,
                    L24,        L25,        L26,        L27,
                    L28,        L29,        L30,        L31,

                    L32,        L33,        L34,        L35,
                    L36,        L37,        L38,        L39,
                    L40,        L41,        L42,        L43,
                    L44,        L45,        L46,        L47,

                    L48,        L49,        L50,        L51,
                    L52,        L53,        L54,        L55,
                    L56,        L57,        L58,        L59,
                    L60,        L61,        L62,        L63,
--
--  Next are the (index mode, register) combinations.
--
                    X_R0,       X_R1,       X_R2,       X_R3,
                    X_R4,       X_R5,       X_R6,       X_R7,
                    X_R8,       X_R9,       X_R10,      X_R11,
                    X_AP,       X_FP,       X_SP,       X_PC,
--
--  The following are the (register mode, register) combinations.
--
                    R0,         R1,         R2,         R3,
                    R4,         R5,         R6,         R7,
                    R8,         R9,         R10,        R11,
                    AP,         FP,         SP,         PC,
--
--  The following are the (indirect register mode, register) combinations.
--
                    IR0,        IR1,        IR2,        IR3,
                    IR4,        IR5,        IR6,        IR7,
                    IR8,        IR9,        IR10,       IR11,
                    IAP,        IFP,        ISP,        IPC,
--
--  Next are the (autodecrement register mode, register) combinations.
```

```
--
                    DEC_R0,     DEC_R1,     DEC_R2,     DEC_R3,
                    DEC_R4,     DEC_R5,     DEC_R6,     DEC_R7,
                    DEC_R8,     DEC_R9,     DEC_R10,    DEC_R11,
                    DEC_AP,     DEC_FP,     DEC_SP,     DEC_PC,
--
--  Next are the (autoincrement register mode, register) combinations.
--  IMD (immediate mode) is autoincrement mode using the PC.
--
                    R0_INC,     R1_INC,     R2_INC,     R3_INC,
                    R4_INC,     R5_INC,     R6_INC,     R7_INC,
                    R8_INC,     R9_INC,     R10_INC,    R11_INC,
                    AP_INC,     FP_INC,     SP_INC,     IMD,
--
--  The following are the (autoincrement deferred mode, register)
--  combinations.  A (absolute address mode) is autoincrement
--  deferred using the PC.
--
                    IR0_INC,    IR1_INC,    IR2_INC,    IR3_INC,
                    IR4_INC,    IR5_INC,    IR6_INC,    IR7_INC,
                    IR8_INC,    IR9_INC,    IR10_INC,   IR11_INC,
                    IAP_INC,    IFP_INC,    ISP_INC,    A,
--
--  The following are the (byte displacement mode, register)
--  combinations.  B_PC is byte relative mode for the program counter.
--
                    B_R0,       B_R1,       B_R2,       B_R3,
                    B_R4,       B_R5,       B_R6,       B_R7,
                    B_R8,       B_R9,       B_R10,      B_R11,
                    B_AP,       B_FP,       B_SP,       B_PC,
--
--  Next are the (byte displacement deferred mode, register)
--  combinations.  IB_PC is byte relative deferred mode
--  for the program counter.
--
                    IB_R0,      IB_R1,      IB_R2,      IB_R3,
                    IB_R4,      IB_R5,      IB_R6,      IB_R7,
                    IB_R8,      IB_R9,      IB_R10,     IB_R11,
                    IB_AP,      IB_FP,      IB_SP,      IB_PC,
--
--  The following are the (word displacement mode, register)
--  combinations.  W_PC is word relative mode for
--  the program counter.
--
                    W_R0,       W_R1,       W_R2,       W_R3,
                    W_R4,       W_R5,       W_R6,       W_R7,
                    W_R8,       W_R9,       W_R10,      W_R11,
                    W_AP,       W_FP,       W_SP,       W_PC,
--
--  The following are the (word displacement deferred mode, register)
--  combinations.  IW_PC is word relative deferred mode for the
--  program counter.
--
```

```
                    IW_R0,      IW_R1,      IW_R2,      IW_R3,
                    IW_R4,      IW_R5,      IW_R6,      IW_R7,
                    IW_R8,      IW_R9,      IW_R10,     IW_R11,
                    IW_AP,      IW_FP,      IW_SP,      IW_PC,
--
-- Next are the (longword displacement mode, register) combinations.
-- L_PC is longword relative mode.
--
                    L_R0,       L_R1,       L_R2,       L_R3,
                    L_R4,       L_R5,       L_R6,       L_R7,
                    L_R8,       L_R9,       L_R10,      L_R11,
                    L_AP,       L_FP,       L_SP,       L_PC,
--
-- The following are the (longword displacement deferred mode,
-- register) combinations.  IL_PC is longword relative deferred mode.
--
                    IL_R0,      IL_R1,      IL_R2,      IL_R3,
                    IL_R4,      IL_R5,      IL_R6,      IL_R7,
                    IL_R8,      IL_R9,      IL_R10,     IL_R11,
                    IL_AP,      IL_FP,      IL_SP,      IL_PC);
```

10.1.10.2 Restrictions on the ADDRESS and DISP Attributes.

The following restriction applies to the use of the ADDRESS and DISP attributes:

> All displacements and addresses (i.e., branch destinations, program counter addressing mode displacements, etc.) must be static expressions. Since neither the ADDRESS nor the DISP attributes return static values, they may not be used in code statements.

10.1.10.3 Restrictions on Assembler Constructs.

The machine code insertion capability provided by the package MACHINE_CODE does not allow for all assembler constructs. These unsupported constructs are:

- The VAX/VMS Assembler's capability to compute the length of immediate and literal data is not replicated in MACHINE_CODE. This means the user cannot supply a value without specifying the length of that value. This disallows the assembler operand general formats:

```
D(R)
G
G^G
```

```
#cons
#cons[Rx]
D(R)[Rx]
G[Rx]
G^location[Rx]
@D(R)[Rx]
@G[Rx]
@D(R)
@G
```

where D and G are byte, word or long_word values. Operands must contain address mode specifiers which explicitly define the length of any immediate or literal values of that operand.

- The radix of the assembler notation is decimal. To express a hexidecimal literal, the notation 16#literal# should be used instead of ^X.

- To construct an octword, quadword, g_float or h_float number, it is important for the user to remember that the component fields of the records which make up the long numeric types are signed. This means that the user must take care to be assured that the values for these components, although signed, are interpreted correctly by the architecture.

- Edit instruction streams must be constructed through the use of the VAX data statements described in Section 10.1.11.3.

- Compatibility mode instruction streams must be constructed through the use of the VAX data statements described in Section 10.1.11.3.

- No error messages are generated if the PC is used as the register for operands taking a single register, if the SP or PC are used for operands taking two registers, or if the AP, FP, SP, or PC is used for operands taking four registers.

- No error message is generated if the PC is used in register deferred or autodecrement mode.

- If any register other than the PC is used as both the simple_operand and as the index_reg for an operand (see Section 10.1.11.1.1.2 for definitions of simple_operand and idex_reg), no error message is generated. An example of this case is the VAX/VMS Assembler operand (7)[7].

- Generic opcode selection is not supported. This means the opcode which reflects the specified number of operands must be used. For example, for 2 operand word addition, ADDW2 must be used, not just ADDW.

- The PC is not supplied as a default if no register is specified in an operand. The user must supply the mode specifier which is mapped onto the PC. Examples are IMD, A, B_PC, W_PC, etc.

10.1.11 Machine Instructions and Data.

This section describes the syntactic details for writing code statements (machine code insertions) as provided for the VAX by the pre-defined package MACHINE_CODE. The format for writing code statements is detailed, as are descriptions of the values to be supplied in the code statements. Each value is described by the named association for that value, and is defined in the order in which it must appear in positional notation. The programmer should refer to the VAX-11 Architecture Handbook (2.2) along with this section to insure that the machine instructions are correct from an architectural viewpoint.

To insure a proper interface between Ada and machine code insertions, the user must be aware of the calling conventions used by the Ada compiler, described in Appendix 20.

10.1.11.1 Vax Instructions.

The general format for VAX code statements where the opcode is a one byte opcode is

```
VAX1'(OP => opcode {,"opcode"_1 => operand
                    {,"opcode"_2 => operand
                      {,"opcode"_3 => operand
                        {,"opcode"_4 => operand
                          {,"opcode"_5 => operand
                            {,"opcode"_6 => operand}}}}}});
```

The general format for VAX code statements where the opcode is a two byte opcode is

```
VAX2'(OP => opcode2 {,"opcode2"_1 => operand
                      {,"opcode2"_2 => operand
                        {,"opcode2"_3 => operand
                          {,"opcode2"_4 => operand
                            {,"opcode2"_5 => operand
                              {,"opcode2"_6 => operand}}}}}});
```

where "opcode"_n and "opcode2"_n is the result of the concatenation of the VAX opcode, an underscore, and the position of the operand in the VAX instruction. The VAX1 and VAX2 code statements always require an opcode and may include from 1 to 6 operands. The opcode mnemonics are precisely the same as described in the previously referenced VAX-11 Architecture Handbook.

10.1.11.1.1 VAX Operands.

The VAX address modes divide the operands into six general categories, namely:

10.1.11.1.1.1 Short Literal Operands.

The assembler format for short literal operands is

```
S^#cons
```

where cons is an integer constant with an range from 0 to 63 (decimal).

The code statement format for short literal operands is

```
(OP => short_lit)
```

where short_lit is one of the enumerated values, range L0 to L63, of the address mode specifiers in Section 10.1.10.1.

The following are examples of how some VAX/VMS Assembler short literals would be expressed in code statements.

```
S^#7   becomes  (OP => L7)
S^#33  becomes  (OP => L33)
S^#60  becomes  (OP => L60)
```

(For explanations of named and unnamed component association, see Section 4.3 of the Ada Language Reference Manual.)

10.1.11.1.1.2 Indexed Operands.

The VAX/VMS Assembler format for the indexed operands is, in general

```
simple_operand[Rx]
```

where a simple_operand is an operand of any address mode except register, literal, or index.

The general code statement format for indexed operands is

```
(index_reg, simple_operand) or
    (OP => index_reg, OPND => simple_operand)
```

where index_reg is one of the enumerated address mode specifiers, range X_R0 to X_SP, from Section 10.1.10.1. Simple_operand is an operand of any address mode except register, literal, or index.

For example, the following indexed assembler operands,

```
(a)   (R8)[R7]          becomes   (X_R7, (OP => IR7))

(b)   (R8)+[R7]         becomes   (X_R7, (OP => R8_INC))

(c)   I^#600[R4]        becomes   (X_R4, (IMD,600))

(d)   -(R4)[R3]         becomes   (X_R3, (OP => DEC_R4))

(e)   B^4(R9)[R3]       becomes   (X_R3, (B_R9,4))

(f)   W^800(R8)[R5]     becomes   (X_R5, (W_R8,800))

(g)   L^34000(R8)[R4]   becomes   (X_R4, (L_R8,34000))

(h)   B^10[R9]          becomes   (X_R9, (B_PC,10))

(i)   W^130[R2]         becomes   (X_R2, (W_PC,130))

(j)   L^35000[R6]       becomes   (X_R6, (L_PC,35000))

(k)   @(R3)+[R5]        becomes   (X_R5, (OP => IR3_INC))

(l)   @#1432[R5]        becomes   (X_R5, (A,1432))

(m)   @B^4(R9)[R3]      becomes   (X_R3, IB_R9,4))

(n)   @W^8(R8)[R5]      becomes   (X_R5, (IW_R8,8))

(o)   @L^2(R8)[R4]      becomes   (X_R4, (IL_R8,2))

(p)   @B^3[R1]          becomes   (X_R1, (IB_PC,3))

(q)   @W^150[R2]        becomes   (X_R2, (IW_PC,150))

(r)   @L^100000[R3]     becomes   (X_R3, (IL_PC,100000))
```

would be expressed in named notation as:

```
(a)   (OP => X_R7, OPND => (OP => IR7))

(b)   (OP => X_R7, OPND => (OP => R8_INC))

(c)   (OP => X_R4, OPND => (OP => IMD, W_IMD => 600))
```

```
(d)   (OP => X_R3, OPND => (OP => DEC_R4))

(e)   (OP => X_R3, OPND => (OP => B_R9, BYTE_DISP => 4))

(f)   (OP => X_R5, OPND => (OP => W_R8, WORD_DISP => 800)

(g)   (OP => X_R4, OPND => (OP => L_R8, LONG_WORD_DISP => 34000))

(h)   (OP => X_R9, OPND => (OP => B_PC, BYTE_DISP => 10))

(i)   (OP => X_R2, OPND => (OP => W_PC, WORD_DISP => 130))

(j)   (OP => X_R6, OPND => (OP => L_PC, LONG_WORD_DISP => 35000))

(k)   (OP => X_R5, OPND => (OP => IR3_INC))

(l)   (OP => X_R5, OPND => (OP => A, ADDR => 1432))

(m)   (OP => X_R3, OPND => (OP => IB_R9, BYTE_DISP => 4))

(n)   (OP => X_R5, OPND => (OP => IW_R8, WORD_DISP => 8))

(o)   (OP => X_R4, OPND => (OP => IL_R8, LONG_WORD_DISP => 2))

(p)   (OP => X_R1, OPND => (OP => IB_PC, B_BISP => 3))

(q)   (OP => X_R2, OPND => (OP => IW_PC, WORD_DISP => 150))

(r)   (OP => X_R3, OPND => (OP => IL_PC, LONG_WORD_DISP => 100000))
```

10.1.11.1.1.3 Register Operands.

The VAX/VMS Assembler formats for register operands are

```
Rn        -- Register mode
(Rn)      -- Register deferred mode
-(Rn)     -- autodecrement mode
(Rn)+     -- Autoincrement mode
@(Rn)+    -- Autoincrement deferred mode
```

where Rn represents a register numbered from 0 to 15.

The general code statement format for register operands is

```
'OP => regmode__value)
```

where regmode_value represents one of the enumerated address mode specifier range R0 to PC, from Section 10.1.10.1.

The following are examples of how VAX/VMS Assembler register mode operands

would be written as code statements.

```
R7          becomes    (OP => R7)
(R8)        becomes    (OP => IR8)
-(R9)       becomes    (OP => DEC_R9)
(R1)+       becomes    (OP => R1_INC)
@(R3)+      becomes    (OP => IR3_INC)
```

10.1.11.1.1.4 Byte Displacement Operands.

The VAX/VMS Assembler syntax for the byte displacement operands is

```
B^d(Rn)        -- Byte displacement mode
@B^d(Rn)       -- Byte displacement deferred mode
```

where d is the displacement added to the contents of register Rn. If no register is specified, the program counter is assumed. The code statement general format for the byte displacement and byte displacement deferred modes is

```
(byte_disp_spec, value) or (OP => byte_disp_spec, BYTE_DISP => value)
```

where byte_disp_spec is one of the enumerated address mode specifiers, range B_R0 to B_PC for byte displacement or IB_R0 to IB_PC for byte displacement deferred, from Section 10.1.10.1. Value is in the range -128 to 127.

The following are examples of how VAX/VMS Assembler byte displacement operands would be written in code statements.

```
B^4(R5)     becomes  (B_R5, 4)     or (OP => B_R5,  BYTE_DISP => 4)
B^200(R5)   becomes  (B_R5, 200)   or (OP => B_R5,  BYTE_DISP => 200)
B^33        becomes  (B_PC, 33)    or (OP => B_PC,  BYTE_DISP => 33)
@B^4(R5)    becomes  (IB_R5, 4)    or (OP => IB_R5, BYTE_DISP => 4)
@B^200(R5)  becomes  (IB_R5, 200)  or (OP => IB_R5, BYTE_DISP => 200)
@B^33       becomes  (IB_PC, 33)   or (OP => IB_PC, BYTE_DISP => 33)
```

10.1.11.1.1.5 Word Displacement Operands.

The VAX/VMS Assembler syntax for the word displacement operands are

```
W^d(Rn)        -- Word displacement
@W^d(Rn)       -- Word displacement deferred
```

where d is the displacement to be added to the contents of register Rn. If no register is specified, the program counter is assumed. In code statements, word displacement operands are represented in general as

```
(word_disp_spec, value) or (OP => word_disp_spec, WORD_DISP => value)
```

where word_disp_spec is one of the enumerated address mode specifiers, range W_R0 to W_PC for word displacement mode or IW_R0 to IW_PC for word displacement deferred mode, from Section 10.1.10.1. Value is in the range -2**15 to 2**15 -1.

The following are examples of how VAX/VMS Assembler word displacement operands would be written in code statements.

```
W^10(R5)     becomes   (W_R5, 10)   or (OP => W_R5,  WORD_DISP => 10)
W^20         becomes   (W_PC, 20)   or (OP => W_PC,  WORD_DISP => 20)
@W^128(R7)   becomes   (W_R7, 128)  or (OP => IW_R7, WORD_DISP => 128)
@W^324       becomes   (W_PC, 324)  or (OP => IW_PC, WORD_DISP => 324)
```

10.1.11.1.1.6 Longword Displacement Operands.

The VAX/VMS Assembler general formats for the longword displacement operands is

```
L^d(Rn)       -- Long_word displacement
@L^d(Rn)      -- Long_word displacement deferred
```

where d is the displacement to be added to the register represented by Rn. Longword displacement operands are represented in code statements by the general format

```
(lword_disp_spec, value) or
                    (OP => lword_disp_spec, LONG_WORD_DISP => value)
```

where lword_disp_spec is one of the enumerated address mode specifiers, range L_R0 to L_PC for long word displacement mode or IL_R0 to IL_PC for longword displacement deferred mode, from Section 10.1.10.1. Value is in the range -2**31 to 2**31 - 1.

The following are examples of how VAX/VMS Assembler long_word displacement operands would be written in code statements.

```
L^1000(R7)   becomes  (L_R7, 1000)  or
                                (OP => L_R7, LONG_WORD_DISP => 1000)
L^25000      becomes  (L_PC, 25000) or
                                (OP => L_PC, LONG_WORD_DISP => 25000)
@L^1000(R9)  becomes  (IL_R9, 1000) or
                                (OP => IL_R9, LONG_WORD_DISP => 1000)
@L^3500      becomes  (IL_PC, 3500) or
                                (OP => IL_PC, LONG_WORD_DISP => 3500)
```

10.1.11.2 The CASE Statement.

The VAX case statements (mnemonics CASEB, CASEW, and CASEL) have the following general symbolic form

```
                                opcode selector.rx, base.rx, limit.rx,
                                      displ[0].bw, .. , displ[limit].bw
```

where x is dependent upon the opcode as to whether the operand is of type BYTE, WORD, or LONG_WORD. Displ[0].bw, .. , displ[limit].bw is a list of displacements to which to branch. Case statements would be written as code statements as

```
VAX1'(OP => case_opcode, "case_opcode"_1 => operand,
                          "case_opcode"_2 => operand,
                           "case_opcode"_3 => case_operand)
```

where case_opcode is one of CASEB, CASEW, or CASEL. The type of operand and case operand are as indicated in the opcode (BYTE, WORD, or LONG_WORD). A case_operand is a special case operand of the form

```
case_operand => (case_limit_address_mode, (case_enum))
```

or

```
case_operand => (LIMIT => case_limit_address_mode,
                 (CASES => case_enum))
```

if case_limit_address_mode is one of the short literal address specifiers. If case_limit_address_mode is the mode specifier IMD, the the case_operand takes the form

```
case_operand => (IMD, (case_limit, (case_enum)))
```

or

```
case_operand => (LIMIT => IMD, CASE_LIST =>
                 (LIMIT => case_limit, (CASES => case_enum)))
```

where case_operand is one of BYTE_CASE_OPERAND, WORD_CASE_OPERAND, or LONG_WORD_CASE_OPERAND. The case_limit_address_mode is one of the short literal mode specifiers or the mode specifier IMD. Case_enum is a list of branch addresses. The branch addresses must be of type WORD. The case_limit is a value of the type indicated by the case_opcode.

Some examples of case statements written as code statements are:

```
<<START>>    VAX1'(CASEB, (OP => R3), (IMD, 5), (IMD,
                  (2,(15,30,45))));  -- Case statement using
                                     -- immediate mode.

 <<S2>>      VAX1'(CASEW, (OP => (W_PC, 10)), (IMD, 100), (L2,
                  (10,20,30)));   -- Case statement using short
```

-- literal mode.

10.1.11.3 VAX Data.

Constant values such as absolute addresses or displacements may be entered into the code stream with any of these nine statements:

```
BYTE_VALUE'(byte)
WORD_VALUE'(word)
LONG_WORD_VALUE'(long_word)
QUADWORD_VALUE'(quadword)
OCTAWORD_VALUE'(octaword)
FLOAT_VALUE'(float)
LONG_FLOAT_VALUE'(long_float)
G_FLOAT_VALUE'(g_float)
H_FLOAT_VALUE'(h_float)
```

10.1.12 System Defined Exceptions

In additional to the exceptions defined in the Ada Language Reference Manual, this implementation pre-defines the following exceptions :

Name	Significance
UNRESOLVED_REFERENCE	Attempted call to a routine not linked into the user's image (See Section 30.1.2)
SYSTEM_ERROR	Serious error detected in underlying operating system.

10.2 Blank.

10.3 Blank.

10.4 Blank.

10.5 Blank.

10.6 The Ada Language For The MCF Target.

The source language accepted by the compiler is Ada, as described in the Military Standard, Ada Programming Language, ANSI/MIL-STD-1815A-1983, 17 February 1983 ("Ada Reference Manual") (2.1).

The Ada definition permits certain implementation dependencies. Each Ada implementation is required to supply a complete description of its dependencies, to be thought of as Appendix F to the Ada Reference Manual. This section is that description for the MCF target.

10.6.1 Pragmas.

10.6.1.1 Pragma Definition.

The following are implementation-defined pragmas:

pragma PAGE;

> This is a listing-control pragma. If the source text to be reformatted contains any PAGE pragmas, the line on which the token PAGE appears is placed at the first line available on a new page in the source listing.

pragma TITLE (arg);

> This is a listing control pragma. It specifies a CHARACTER string that is to appear on the second line of each page of every listing produced for a compilation unit. At most one such pragma may appear for any compilation unit, and it must be the first lexical unit in the compilaton unit (comments excepted). The argument is a CHARACTER string.

The following notes specify the language-required definitions of the predefined pragmas. Unmentioned pragmas require no notes. (See Appendix B of the Ada Reference Manual.)

pragma INTERFACE (arg,arg);

> No INTERFACE pragmas are recognized.

pragma MEMORY_SIZE (arg)

> The MEMORY_SIZE pragma is ignored, other than to verify that the value of the argument is in the range 0..(2**32)-1.

pragma OPTIMIZE (arg)

> This pragma is effective only when the "OPTIMIZE" option has been given to the compiler, as described in 3.7.1.1.1.3. The pragma is ignored when applied to inline subprograms. The

argument is either TIME or SPACE.

pragma PRIORITY (arg)

The PRIORITY argument is an integer static expression value in the range 1..15. The pragma has no effect in a location other than a task (type) specification or outermost declarative part of a subprogram. If the pragma appears in the declarative part of a subprogram, it has no effect unless that subprogram is designated as the "main" subprogram at link time.

pragma STORAGE_UNIT (arg)

If the argument is a value other than 8, a diagnostic of severity WARNING is generated. (See Appendix 80 for a complete summary of all diagnostic messages.) Otherwise, no action is taken.

pragma SUPPRESS (arg[,arg])

SUPPRESS pragmas which mention the following CHECK names have no effect:

```
DIVISION_CHECK
OVERFLOW_CHECK
```

These checks cannot be suppressed.

pragma SYSTEM (arg)

The SYSTEM argument is a value of the enumeration type SYSTEM.SYSTEM_NAME. The purpose of this pragma is to assert that the Ada compilation unit is specially designed to execute only on certain target environments.

The value MCF means that the unit is designed to run on a machine with Nebula architecture. For other values, a diagnostic of the severity WARNING is generated. (See Appendix 80 for a complete summary of all diagnostic messages).

10.6.1.2 Scope of Pragmas.

CONTROLLED	Applies only to the access type named in its argument
INLINE	Applies only to the subprograms named in its arguments. If the argument is an overloaded subprogram name, the INLINE pragma applies to all definitions of that subprogram name which appear in the same declarative part as the INLINE pragma.
LIST	In effect until the next LIST pragma in the source or included text, or if none, the end of the compilation unit
MEMORY_SIZE	Applies to the entire Program Library in which it appears. Multiple, conflicting occurrences in a Program Library are erroneous.
OPTIMIZE	Applies to the entire compilation unit in which it appears
PACK	Applies only to the array or record named in its argumeni
PAGE	No scope
PRIORITY	Applies only to the task specification or main subprogram in which it appears
STORAGE_UNIT	Applies to the entire compilation unit before which it appears
SUPPRESS	Applies to the block or body that contains the declarative part in which the pragma appears
SYSTEM	Applies to the entire compilation unit before which it appears
TITLE	No scope

10.6.2 Attributes.

There is one implementation-defined attribute in addition to the predefined attributes found in Appendix A of the Ada Reference Manual.

X'DISP

A value of type DISPLACEMENT which corresponds to the displacement that is used to address the first storage unit occupied by a data object X at a static offset within an implemented activation record. The type DISPLACEMENT is defined as:

```
type DISPLACEMENT is new LONG_INTEGER;
```

This attribute differs from the ADDRESS attribute in that ADDRESS supplies the absolute address while DISP supplies the displacement relative to some base value (such as a stack frame pointer), in the runtime enviromnent. It is the user's responsibility to determine the base value relevant to the attribute. The runtime environment is described in Appendix <TBD>.

The following notes augment the language-required definitions of the predefined attributes found in Appendix A of the Ada Reference Manual.

T'MACHINE_ROUNDS is true.

T'MACHINE_RADIX is 2.

T'MACHINE_MANTISSA If the size of the base type of T is 32, T'MACHINE_MANTISSA is 24. Otherwise it is 53.

T'MACHINE_EMAX If the size of the base type of T is 32, T'MACHINE_EMAX is 127. Otherwise it is 1023.

T'MACHINE_EMIN If the size of the base type of T is 32, T'MACHINE_EMIN is -126. Otherwise it is -1022.

T'MACHINE_OVERFLOWS is true.

The Nebula architecture allows software manipulation of the status register which controls rounding and response to overflow. The ALS for the MCF will establish these values at the beginning of a program execution, and will assume them to be in effect throughout a program. It is the user's responsibility to refrain from using code statements or assembly language to change these values in the status register.

10.6.3 Predefined Language Environment.

The Package STANDARD contains the following definitions in addition to those specified in Appendix C of the Ada Reference Manual:

```
type SHORT_INTEGER is range -2**7.. 2**7-1;
type INTEGER is range -2**15..2**15-1;
type LONG_INTEGER is range -2**31..2**31-1;

type FLOAT is digits 7 range
-(2#1.111_1111_1111_1111_1111_1111#E127)
..(2#1.111_1111_1111_1111_1111_1111#E127);
type LONG_FLOAT is digits 15 range
-(2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1023)
..(2#1.1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111_1111#E1023);
--SHORT_FLOAT is not defined

subtype PRIORITY is SHORT_INTEGER range 1..15;
type DURATION is delta 0.01 range -2.0**24 .. 2.0**24-0.01;
                --actual delta is 1/128

Package SYSTEM is

type SYSTEM_NAME is (VAX780_VMS, VAX780, MCF);
NAME: constant SYSTEM_NAME := MCF;
STORAGE_UNIT: constant := 8;
MEMORY_SIZE: constant := (2**32);
MIN_INT: constant := -(2**31);
MAX_INT: constant := (2**31)-1;
UNRESOLVED_REFERENCE: exception; --see Appendix 30 of this specification
end SYSTEM;
```

10.6.4 Representations and Declaration Restrictions.

Representation specifications are described in Chapter 13 of the Ada Reference Manual. Declarations are described in Chapter 3 of that manual.

10.6.4.1 Integer Types.

Integer types are specified with constraints of the form

```
range L..R
```

where

```
R ≤ SYSTEM.MAX_INT
L ≥ SYSTEM.MIN_INT
```

Length specifications of the form:

```
for T'SIZE use N;
```

may specify integer values N such that

```
N = 32 or
N in 2..16,
```

and such that

```
R ≤ 2**(N-1)-1
and  L ≥ -2**(N-1)
```

or else such that

```
R ≤ (2**N)-1
L ≥ 0
and  0 < N ≤ 15
```

When no length specification is provided, the smallest SIZE of 8, 16, or 32 is used such that

```
R ≤ 2**(SIZE-1)-1
and  L ≥ -2**(SIZE-1).
```

10.6.4.2 Floating Types.

Floating types are specified with constraints of the form:

```
digits D
```

where

D is an integer value in 1..15.

Length specifications of the form

```
for T'SIZE use N;
```

may specify integer values N = 32 when D ≤ 7,

or N = 64 when D $\leq$ 15.

Where no length specification is provided, a size of 32 is used when D $\leq$ 7, 64 when D is 8..15.

10.6.4.3 Fixed Types.

Fixed types are specified with constraints of the form

```
delta D range L..R
```

where

$$\frac{\max(\text{abs}(R), \text{abs}(L))}{\text{actual delta}} \leq 2^{**}31-1$$

The actual delta defaults to the largest integral power of 2 less than or equal to the specified delta D. (This implies that fixed values are stored right-aligned.) For specifications of the form:

```
for T'ACTUAL_DELTA use X;
```

X must be specified as an integral power of 2 such that
X $\leq$ D

Length specifications of the form

```
for T'SIZE use N;
```

are permitted only when N = 32.
All fixed values have SIZE = 32.

10.6.4.4 Enumeration Types.

The default SIZE for an enumeration type is 8, unless there are more than 256 values. In the latter case the default size is 16. The size for the predefined type BOOLEAN is one. A length specification of the form

```
for T'SIZE use N;
```

is permitted for N in 1..16.

Enumeration types may have up to 2**SIZE different values.

An enumeration type representation of the form

```
for T use aggregate;
```

may specify codes in the range 0..(2**SIZE)-1

10.6.4.5 Access Types.

No representation specification is permitted for access types. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

10.6.4.6 Arrays and Records.

SIZE specifications of the form

```
for T'SIZE use N;
```

are not permitted for arrays and records. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

The PACK pragma may be used to minimize wasted space between components of arrays and records. The pragma causes the type representation to be chosen such that storage space requirements are minimized at the possible expense of data access time and code space. The PACK pragma may not be used in conjunction with a representation specification for the same type. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

A record type representation specification may be used to describe the allocation of components in a record. Bits are numbered 0..7 from the left. (Bit 8 starts at the left of the next higher-numbered byte.) Each location specification must allow at least X bits of range, where X is the SIZE of the type of the component being allocated. If $X \geq 32$, the range must be of the form 0..X-1, thereby guaranteeing byte alignment. If $X < 32$, unaligned allocation is legal, with the range being any correct-length subrange of 0..31. Components that are arrays, records, tasks, or access variables may not be allocated to specified locations. If a specification of this form is entered, a diagnostic message of severity ERROR is generated. (See Appendix 80 for a complete list of all diagnostic messages.)

The alignment clause of the form

```
at mod N
```

may specify alignments of 1 (byte), 2 (half-word), 4 (word), or 8 (double-word).

10.6.4.7 Other Length Specifications.

Length Specifications are described in Section 13.2 of the Ada Reference Manual.

T'STORAGE_SIZE for access type T - This length specification is ignored.

T'STORAGE_SIZE for task type T - Specifies the number of bytes to be allocated for the runtime stack of each task object of type T.

10.6.5 System Generated Names.

There are no system generated names.

10.6.6 Address Specifications.

Refer to Par. 13.5 of the Ada Reference Manual for a discussion of address specifications.

For a task entry an address specification describes the address in the interrupt vector table in which the address of the designated entry is to be placed. The address specification allows the entry to be associated with an interrupt.

For scalar objects, or records without discriminants, address specifications may be used to designate a physical address in I/O space in which the object is to be permanently allocated. Such an address specification can only appear in the declarative part of a library package body, and may only specify an object declared in that body.

Address specifications may not designate subprograms.

Address specifications may be included at user risk. Since the ALS uses the exception interruption, and I/O space facilities of the machine, it is the user's responsibility to ensure that address specifications do not interfere with normal program execution.

10.6.7 Unchecked Conversions.

Refer to Par. 13.10.2 of the Ada Reference Manual for a description of UNCHECKED_CONVERSION.

A program is erroneous if it performs UNCHECKED_CONVERSION when the source and target have different size.

10.6.8 Input/Output.

Refer to Chapter 14 of the Ada Reference Manual for a discussion of Input/Output.

The type FILE_INDEX that appears in Par. 14.2 is defined as follows:

```
type FILE_INDEX is range 0..LONG_INTEGER'LAST;
```

10.6.8.1 Naming External Files:

The only valid external file names are "disk", "tape:...", "console_in" and "console_out". Use of any other external file name raises NAME_ERROR. The external files "console_in" and "console_out" are reserved for the text I/O standard files, and therefore cannot be used in an OPEN. The others, "disk" and "tape:...", are pre-created and can be opened or closed, but not created or deleted. Attempts to create or delete any file raises NAME_ERROR. Both "disk" and "tape:..." can be opened as IN_FILE, OUT_FILE, or INOUT_FILE. The ellipses in the tape file represent a user-specified string, identifying to the operator the reel that is to be mounted on the tape drive.

10.6.8.2 File Processing. <TBD>

10.6.8.3 Text Input/Output. <TBD>

10.6.8.4 Low Level Input-Output. <TBD>

10.6.8.5 Hardware Interrupts.

If the programmer has equated an I/O related hardware interrupt to a task entry (by means of an address specification), then the runtime support library converts such an interrupt into a conditional entry call. I/O related hardware interrupts which have not been equated to a task entry are ignored.

10.6.9 Character Set.

Ada compilations may be expressed using the following characters, in addition to the basic character set:

1. The lower case letters
2. ! $? @ [\ ] ^ ' { } ~

The following transliterations are not permitted (see Par. 2.10 of the Ada Reference Manual):

1. Exclamation mark for vertical bar,

2. Colon for sharp, and

3. Percent for double-quote.

10.6.10 Machine Code Insertions.

The Ada language definition permits machine code insertions as defined by Paragraph 13.8 in the Ada Reference Manual. This paragraph describes the specific details for writing machine code insertions as provided by the ALS-supplied package MCF_INSTRUCTIONS.

10.6.10.1 Machine Features.

This paragraph describes specific machine language features, such as registers, operand specifiers, etc., that are needed to code machine code statements. In addition to the attribute DISP, the ALS-supplied package MCF_INSTRUCTIONS will supply the following constants for use in code statements:

R1..R15	register operand values
P1..P7	short parameter operand values
EXT_P	specifier constant for extended parameter mode
GEN_P	specifier constant for general parameter mode
UNSC_IND	specifier constant for unscaled index mode
SC_IND	specifier constant for scaled index mode
B_IND(B..D,0..15)	specifier constants for byte indexed mode
W_IND(B..D,0..15)	specifier constants for word indexed mode
LIT(B..D)	specifier constants for literal mode
ABS(B..D)	specifier constants for absolute mode
IND(B..D)	specifier constants for indirect register mode

The use of these constants is demonstrated in Section 10.6.11.1.1.

10.6.10.2 Restrictions on ADDRESS and DISP Attributes.

The following restrictions apply to the use of the ADDRESS and DISP ATTRIBUTES:

a) ADDRESS - this attribute can only be safely applied to the names of statically allocated objects and subprograms.

b) DISP - this attribute must be used in conjunction with the correct base address, such as a stack frame pointer, that corresponds to the unit in which it was elaborated. To obtain this, the user must have detailed knowledge of the object code conventions and runtime environment of the Ada system (described in <TBD>). In particular, the user is responsible for insuring that the value supplied by DISP meets the range constraints of the displacement or address field of the machine code instruction.

10.6.11 Machine Instructions.

This section describes the syntactic details for writing code statements in machine code insertions as defined by the package specification MCF_INSTRUCTIONS. The format for writing code statements is defined along with the descriptions of the values to be supplied in the code statements. Each value is described by the named association for that value and is defined in the order that it must appear if positional association is used. The programmer should always refer to the Nebula Instruction Set Architecture Standard (2.1) along with this section to insure that the machine instructions are syntactically correct.

To insure a proper interface between high-level Ada and machine code insertions, the user must be aware of the calling conventions employed by the Ada compiler as described in <TBD>.

The value of any expression used in a code statement must be statically determinable.

10.6.11.1 MCF Instructions.

The format for MCF Instructions, except CALL instructions, is

```
MCF'(CODE=> opcode {, operand});
```

The MCF code statement always requires an opcode, and may include 0 to 4 operands (see section 10.6.11.1.1). The opcodes are the same mnemonics described in the Nebula architecture (previously referenced), with the exception of substitutions to differentiate different opcodes given identical mnemonics in the Architecture, or to avoid Ada reserved words. The exceptions follow:

Nebula codes	ALS MCF substitutions	
ADD =>	ADD2,ADD3	-- number of operands
SUB =>	SUB2,SUB3	
MUL =>	MUL2,MUL3	
DIV =>	DIV2,DIV3	
NEG =>	NEG1,NEG2	
NOT =>	NOT1,NOT2	
AND =>	AND1,AND2	
OR =>	OR1,OR2	
ADDF=>	ADDF2,ADDF3	
SUBF=>	SUBF2,SUBF3	
MULF=>	MULF2,MULF3	
DIVF=>	DIVF2,DIVF3	
NEGF=>	NEGF1,NEGF2	
BR =>	BR_B,BR_H	-- size of displacement
BEQL=>	BEQL_B,BEQL_H	
BNEQ=>	BNEQ_B,BNEQ_H	
BLEQ=>	BLEQ_B,BLEQ_H	

```
BLSS=>          BLSS_B,BLSS_H
BGEQ=>          BGEQ_B,BGEQ_H
BGTR=>          BGTR_B, BGTR_H
BCS =>          BCS_B,BCS_H
BCC =>          BCC_B,BCC_H
BTS =>          BTS_B,BTS_H
BTC =>          BTC_B,BTC_H
LOOP=>          LOOP_MCF          -- reserved words
MOD =>          MOD_MCF
REM =>          REM_MCF
XOR =>          XOR_MCF
RANGE =>        RANGE_MCF
RAISE =>        RAISE_MCF
CASE  =>        CASE_MCF
```

Procedure CALL instructions have the following format:

```
CALL_<n>'(PRIV=> level, OPND1 => operand
        [,PARAMS => (operand {, operand})]);
```

where:

a. <n> is to be the integer literal of the number of parameters listed, in the range 0...255, to be appended by the programmer to form an identifier, and

b. The level will be NORM, UNPRIV, or SUPV.

The first operand will resolve to the address of the procedure being called (or the SVC vector index), and the first parameter operand (if present) may be either the number of parameters to follow (parameter 0) or the first actual parameter, as specified in the conventions of the Nebula Instruction Set Architecture.

10.6.11.1.1 MCF Operands.

Except as noted in Sections 10.6.11.1.1.13 - 10.6.11.1.1.15, an MCF operand field will be one of the 12 operand addressing modes. In named aggregate notation, they are to be designated by the names OPND1...OPND4, for normal operands. CALL instruction parameters are given as an array aggregate. For example, the instruction

```
MOV    %2,@%3^B
```

would be represented as

```
MCF'(CODE => MOV, OPND1 => (SPEC => R2),
    OPND2 => (SPEC => IND(B,3)));
```

```
or MCF'(MOV,(SPEC => R2),(SPEC => IND(B,3)));
```

Branch instructions, the CASE_MCF instruction, and the WINDOW instruction include special operand fields, named and formatted as outlined in Sections 10.6.11.1.1.13 - 10.6.11.1.1.15. Note that Ada semantic rules require that aggregates of a single component must be given in named notation.

10.6.11.1.1.1 Short Literal Operands.

The format for short literal operands is

```
(SPEC => numeric value)
```

where numeric value is a value of type U_BYTE in the range 0...31.

For example, the assembler literal value, #15, in the assembler syntax would be expressed in the following manner:

```
(SPEC => 15)
```

10.6.11.1.1.2 Register Operands.

The format for register operands is

```
(SPEC => register constant)
```

where register constant is one of the package-supplied constants R1..R15.

For example, the assembler register field, %4, in the assembler syntax would be expressed in the following manner:

```
(SPEC => R4)
```

10.6.11.1.1.3 Short Parameter Operands.

The format for short parameter operands is

```
(SPEC => parameter number)
```

where parameter number is one of the package-supplied constants P1..P7.

For example, the assembler parameter operand, ?5, in the assembler syntax would be expressed in the following manner:

```
(SPEC => P5)
```

A consequence of the nature of parameter operands and of the fact that code statements are inserted in line is that parameter operands will be resolved by the hardware as referring to parameters of the most immediately encompassing Ada program unit on the active context stack.

10.6.11.1.1.4 Extended Parameter Operands.

The format for extended parameter operands is

```
(SPEC => EXT_P, PARM => parameter number)
```

where:

a. EXT_P is a package-supplied field constant, and

b. Parameter number is a value of type U_BYTE in the range 0..255.

For example, the assembler parameter operand, ?9, in the assembler syntax would be expressed in the following manner:

```
(SPEC => EXT_P, PARM => 9) or
   (EXT_P,9)
```

A consequence of the nature of parameter operands and of the fact that code statements are inserted in line is that parameter operands will be resolved by the hardware as referring to parameters of the most immediately emcompassing Ada program unit on the active context stack.

10.6.11.1.1.5 Literal Operands.

The format for literal operands is

```
(SPEC => LIT(SIZE), name => value
```

where:

a. Size designates the size of the value by the identifiers B, H, W, or D,

b. Name is one of B_LIT, H_LIT, W_LIT, or D_LIT, appropriate to the size, and

c. Value is of type SHORT_INTEGER, INTEGER, or LONG_INTEGER, for size B, H, and W, respectively. The value of a double-word literal must be specified as a record aggregate of type DOUBLE, which has two components, UPPER and LOWER, each of type LONG_INTEGER.

For example, the assembler literal operands, # 100, and # -1000, in the assembler syntax could be expressed in the following manner:

```
(SPEC => LIT(B), B_LIT => 100) or (LIT(B), 100)
(SPEC => LIT(W), W_LIT => -1000) or (LIT(W), -1000)
```

Alternately:

```
(SPEC => LIT(D), D_LIT => (UPPER => -1, LOWER => -1000))
 or (LIT(D), (-1,-1000))
```

10.6.11.1.1.6 Absolute Address Operands.

The format for absolute address operands is

```
(SPEC => ABS(SIZE), ADDR => ADDRESS)
```

where:

a. Size is B, H, W, or D, depending on the size of the referent of the address, and

b. Address is a virtual memory address, transposed into a signed integer value of type LONG_INTEGER.

For example, the assembler absolute operand, @(502)w, in the assembler syntax would be expressed in the following manner:

```
(SPEC => ABS(W), ADDR => 502) or (ABS(W), 502)
```

10.6.11.1.1.7 Indirect Register Operands.

The format for indirect register operands is

```
(SPEC => IND(size, register number))
```

where:

a. Size is B, H, W, or D, to indicate the size of the referent of the operand, and

b. Register number is a value of type INTEGER in the range 1..15.

For example, the assembler indirect register field, @%3^W, in the assembler syntax would be expressed in the following manner:

```
(SPEC => IND(W,3))
```

10.6.11.1.1.8 Byte Indexed Operands.

The format for byte indexed operands is

(SPEC => B_IND (size, register), B_DISP => offset)

where:

a. Size is B, H, W, or D, to indicate the size of the referent,

b. Register is a value of type INTEGER in the range 0..15, to indicate the register containing the address, and

c. Offset is a value of type SHORT_INTEGER.

For example, the assembler byte indexed field, 23(%2)W, in the assembler syntax would be expressed in the following manner:

(SPEC => B_IND(W,2), B_DISP => 23) or (B_IND(W,2),23)

10.6.11.1.1.9 Word Indexed Operands.

The format for word indexed operands is

(SPEC => W_IND(size, register), W_DISP => offset)

where:

a. Size is B, H, W, D, to indicate the size of the referent,

b. Register is a value of type INTEGER in the range 0..15, to indicate the register containing the address, and

c. Offset is a value of type LONG_INTEGER.

For example, the assembler word indexed field, -1024(%2)B, in the assembler syntax would be expressed in the following manner:

(SPEC => W_IND(B,2),W_DISP => -1024) or (W_IND(B,2), -1024)

10.6.11.1.1.10 General Parameter Operands.

The format for genera' parameter operands is

(SPEC => GEN_P, G_P_SUB => simple operand)

where:

a. GEN_P is a package-supplied constant operand specifier, and

b. Simple operand is any operand except a scaled-index, unscaled-index or another general parameter operand.

For example, the assembler operand, ?(%3), in the assembler syntax would be expressed in the following manner:

```
(SPEC => GEN_P,G_P_SUB => (SPEC => R3)) or
   (GEN_P, (SPEC =>R3))
```

A consequence of the nature of parameter operands and of the fact that code statements are inserted in line is that parameter operands will be resolved by the hardware as referring to parameters of the most immediately encompassing Ada program unit on the active context stack.

10.6.11.1.1.11 Unscaled Index Operands.

The format for unscaled index operands is

```
(SPEC => UNSC_IND,S_O_1 =>simple operand,S_O_2=>simple
operand
```

where:

a. UNSC_IND is a package-supplied constant operand specifier, and

b. The simple operands are any operands except scaled-index, unscaled-index, or general parameter operands.

For example, the assembler unscaled index operands, @%3(%2)W, in the assembler syntax would be expressed in the following manner:

```
(SPEC=>INSC_IND,S_O_1=>(SPEC=>R2),S_O_2=>(SPEC=>
IND(W,3)))
or (UNSC_IND,(SPEC=>R2),(SPEC=>IND(W,3)))
```

10.6.11.1.1.12 Scaled Index Operands.

The format for scaled index operands is

```
(SPEC=>SC_IND,S_O_1=>simple operand,S_O_2=>simple
operand)
```

where:

a. SC_IND is a package-supplied constant operand specifier, and

b. The simple operands are any operands except scaled-index, unscaled-index, or general parameter operands.

For example, the asembler scaled index operand, @%3[%2]W, in the assembler syntax would be expressed in the following manner:

```
(SPEC=>SC_IND,S_O_1=>(SPEC=>R2),S_O_2=>(SPEC=>
IND(W,3)))
or (SC_IND,(SPEC=>R2),(SPEC=>IND(W,3)))
```

10.6.11.1.1.13 Displacements.

The formats for displacements are

```
B_D_1st => signed byte value
H_D_1st => signed half-word value
H_D_3rd => signed half-word value
H_D_4th => signed half-word value
```

where:

a. B_D_1st and H_D_1st are of type SHORT_INTEGER or INTEGER, respectively, and will appear as the first (and only) operands of normal branch instructions,

b. H_D_3rd is of type INTEGER, and will appear after two other operands in the instructions IBLEQ, IBLSS, DBGEQ, DBGTR, and

c. H_D_4th is of type INTEGER, and will appear after the first three operands of the LOOP_MCF instruction.

Displacements are signed integer values designating address offsets from the program counter. Unlike the operand addressing modes, they are not defined as record types, and therefore are not to be placed in parentheses. For example, the assembler instruction

```
BR      -10
```

would be represented as

```
MCF '(CODE => BR_B, B_D_1st => -10)
or MCF '(BR_B, -10)
```

10.6.11.1.1.14 CASE_MCF Jump Table.

The format for a CASE_MCF jump table is

```
JUMPS => (NUM => table size, TABLE => (displacement list))
```

Where:

a. Table size is the number of displacements in the table, a value of type INTEGER in the range 0.. 2**15-1, and

b. Displacement list is a sequence of values of type INTEGER, separated by commas.

The jump table appears after the first two operands of the CASE_MCF instruction. For example, the assembler instruction

```
CASE    %2,#6,4,1000,1004,1008,1012
```

would be represented as

```
MCF' (CODE => CASE_MCF, OP1 => (SPEC => R2), OP2 => (SPEC => 6),
     JUMPS => (NUM => 4, TABLE => (1000,1004,1008,1012)))
```

or

```
MCF' (CASE_MCF, (SPEC => R2), (SPEC =>6), (4,(1000,1004,1008,1012)))
```

10.6.11.1.1.15 WINDOW Instruction Information.

The format for the information passed by the WINDOW instruction to the micromachine is

```
INFO => value
```

where value is of type U_BYTE.

For example, the assembler instruction

```
WINDOW ^XFF
```

would be represented as

```
MCF'(CODE => WINDOW, INFO => 16#FF#)
or MCF'(WINDOW,16#FF#)
```

APPENDIX 20

20. ADA LANGUAGE SYSTEM ASSEMBLERS

The Ada Language System includes the following assemblers:

a. ALS VAX-11/780 Assembler, and

b. ALS MCF Assembler.

Descriptions of the assembly language for each assembler are provided on the following pages.

20.1 ALS VAX Assembly Language.

RESTRICTIVE LEGEND

Material in this chapter has been taken with permission from the VAX-11 MACRO Language Reference Manual published by the Digital Equipment Corporation, Maynard, Mass. Permission for the use of the material has been given for this specific purpose only. Digital Equipment Corporation retains full rights under its copyright of the material. Accordingly, copies of this document shall not be made by any organization or individual outside of the U.S. Government without the prior written permission of the Digital Equipment Corporation.

This section describes the ALS VAX assembly language, including the syntax of assembly code statements and directives, assembler output, and operation of the assembler. Readers should already be familiar with assembly language programming and the VAX-11 instruction set.

The assembly language is intended to be used for writing small subprograms in order to access capabilities of the target machine that cannot be reached from Ada. It is not intended for any large scale development of software in assembly language. Accordingly, the ALS VAX-11/780 Assembler does not contain all of the functions provided by the assembler that may normally be supplied with the VAX-11/780. A comparison of the assembly language accepted by the ALS VAX-11/780 Assembler and that accepted by the assembler normally supplied with the VAX-11/780 is provided in Section 20.1.8.

The ALS VAX-11/780 Assembler translates source programs into object (i.e., binary) code and produces a listing file sent to the predefined internal standard output file and a Container placed in the environment database. The ALS VAX-11/780 Linker then combines this object information from several assemblies, Ada compilations, and other links to produce an executable program.

An assembly language program consists of a series of source statements representing an Ada library subprogram body or subunit subprogram body. In addition to the body written in assembly language, a subprogram specification, written in Ada, must also exist and must be compiled prior to assembling the body if the assembly subprogram is a library subprogram. If the assembly program is a subunit subprogram, then the parent body must be compiled prior to assembling the assembly language body.

The Ada subprogram specification must not include an INTERFACE pragma. The assembly language subprogram body will be called with the same linkage conventions used when calling subprograms written in Ada; it is the assembly language programmer's responsibility to insure that the subprogram body follows these conventions. The Ada entry/exit sequences, and parameter and function-value-return conventions are described in Appendix 20. (No checks are performed to make sure the subprogram body matches the subprogram specification, except to see that the subprogram names are identical.)

The following conventions are observed throughout this chapter:

- Brackets ([]) indicate that the enclosed argument is optional.
- Uppercase words and letters, used in formats, indicate that the word or letter should be typed exactly as shown.
- Lowercase words and letters, used in formats, indicate that a word or value of the user's choice is to be substituted.
- Curly Braces ({ }) indicate that the enclosed argument(s) may appear zero or more times.

20.1.1 Source Statement Format.

A source program consists of a sequence of source statements. The assembler interprets and processes the statements one by one, generating object code or performing a specific assembly-time process. Only one statement can appear on a line, and a statement cannot extend onto more than one line. A source line can be up to 80 characters long.

Source statements can consist of up to four fields:

- Label field -- allows the programmer to symbolically define a location in a program.
- Operator field -- specifies the action to be performed by the statement; this field can be an instruction opcode or an assembler directive.
- Operand field -- contains the instruction operand(s) or the assembler directive argument(s).
- Comment field -- contains a comment that explains the meaning of the statement; this field does not affect program execution.

The label field, the operator/operand fields combination, and the comment field are all optional. If the operator field is not present the operand field must also be not present. The operand field must conform to the format of the instruction specified by the operator field.

The label field, if present, must begin in Column 1; the operator field, if present, must begin in Column 2 or after. See Section 20.1.7 for a complete summary of the syntax of an assembly language statement.

Blank lines, although legal, have no significance in the source program.

The following sections describe each of the statement fields in detail.

20.1.1.1 Label Field.

A label is a user-defined symbol that identifies a location in the program. The symbol is assigned a value equal to the location counter at the location in the program section in which the label occurs. The user-defined symbol name can be up to 15 characters long and can contain any alphanumeric character. Section 20.1.2.3 describes the rules for forming user-defined symbol names in more detail.

If a statement contains a label, the label must start in the first column of the line.

A label is said to be defined when it appears in a label field. Once a label is defined, it cannot be redefined during the source program. If a label is defined more than once, the assembler displays an error message when the label is defined the second and subsequent times. A space, tab, semicolon, or end of line terminates the label field.

20.1.1.2 Operator Field.

The operator field specifies the action to be performed by the statement. This field can contain either an instruction or an assembler directive.

When the operator is an instruction opcode, the assembler generates the binary code for that instruction; the instruction set and mnemonics are described in the VAX-11/780 Architecture Handbook (2.2). When the operator is a directive, the assembler performs certain control actions or processing operations during source program assembly; the assembler directives are described in Section 20.1.4.

A space, tab, semicolon or end of line terminates the operator field.

20.1.1.3 Operand Field.

The operand field can contain operands for instructions or arguments for assembler directives.

Operands for instructions specify the locations in memory or the registers that are used by the machine operation including the addressing mode. Section 20.1.3 describes the VAX-11 addressing modes. The operand field for a specific instruction must contain the number of operands required by that instruction. See the VAX-11/780 Architecture Handbook (2.2) for a description of the instructions and their operands.

Arguments for a directive must meet the format requirements of the directive. Section 20.1.4 describes the directives and the format of their arguments.

If two or more operands are specified, they must be separated by commas. The operand field is terminated by a space, tab, semicolon, or end of line.

20.1.1.4 Comment Field.

The comment field contains text that explains the meaning of the statement. Comments do not affect assembly processing or program execution.

The comment field must be preceded by a semicolon and is terminated by the end of the line. The comment field can contain any ASCII character. A comment can appear on a line by itself, even beginning in Column 1.

20.1.2 Components of Source Statements.

This section describes the components of assembly language statements: the character set, numbers, symbols, and expressions.

20.1.2.1 Character Set.

The following characters can be used in assembly language statements:

- Both upper-case and lower-case letters (A through Z, a through z) are accepted; however, the assembler considers lower and upper case as equivalent representations of the same character set. E.g., "movw" is the same as "MOVW".
- The digits 0 through 9.
- The special characters listed in Table 20-1.

Table 20-1

RPECIAL CHARACTERS USED IN ASSEMBLER STATEMENTS

Character	Character Name	Function
	Tab	Field separator
	Space	Field separator
#	Number sign	Immediate addressing mode indicator
@	At sign	Deferred addressing mode indicator
,	Comma	Operand separator
;	Semicolon	Comment field indicator
+	Plus sign	Autoincrement addressing mode indicator and arithmetic addition operator
-	Minus sign	Autodecrement addressing mode indicator and arithmetic subtraction operator
^	Circumflex	Radix specification and operand size delimiter
_	Underscore	Break character, word_isolator
:	Colon	Optional label terminator
()	Parentheses	Register deferred
[]	Square brackets	Index addressing mode

20.1.2.2 Numbers.

Numbers can only be integers and can be used in any operand expression. (Section 20.1.2.4 describes expressions.)

Format

[r]n

where:

r is an optional radix specification. ^X indicates the following number is hexadecimal; ^O indicates octal; if the radix is not specified, it is decimal.

n is a string of alphanumeric characters that are legal for the specified radix. Numbers must be representable in 32 bits: decimal integers are limited to approximately 9 digits, hexadecimal to 8, and octal to approximately 11 digitsM

The following numbers represent the same value: 16, ^X10, ^020.

20.1.2.3 Symbols.

Two types of symbols are used in assembly language programs: permanent symbols and user-defined symbols.

20.1.2.3.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Section 20.1.7), the assembler directives (see Section 20.1.4), and the register names. The permanent symbols are predefined. The instruction mnemonics and assembler directives can also be user-defined (i.e., appear in a label field), in which case the symbol can represent either its pre-defined or user-defined meaning, depending on whether it appears in an operator or an operand context. The register mnemonics cannot be user defined.

The sixteen general registers of the VAX-11/780 processor can be expressed in a source program only as follows:

Register Name	Processor Register
R0-R11	General register 0 through general register 11
R12 or AP	General register 12 or argument pointer. If R12 is used as an argument pointer, the name AP is recommended; if R12 is used as a general register, the name R12 is recommended.
FP	Frame pointer (register 13)
SP	Stack pointer (register 14)
PC	Program counter (register 15)

20.1.2.3.2 User-Defined Symbols.

With the exception of the name on the SUBPROGRAM directive, user-defined symbols are only local, appearing in the label field somewhere in the assembly code subprogram and accessible only from within that subprogram.

The following rules govern the creation of user-defined symbols:

- User-defined symbols can only be composed of the 36 alphanumeric characters and the underscore. Any other character terminates the symbol.
- The first character of a symbol must be a letter.
- Two underscores must not be adjacent and the symbol must not begin or end with an underscore.
- The symbol must be no more than 15 characters long and must be unique.

20.1.2.4 Expressions.

There are four kinds of expressions that can be used in an assembly language program:

- number
- symbol
- symbol+number
- symbol-number

If the symbol is relocatable (i.e., defined by appearing as a label other than on an EQU directive), the expression will be relocatable; if absolute (defined by an EQU directive), the expression will be absolute.

Expressions can be used only in the displacement, literal, or immediate field of operands.

20.1.3 Addressing Modes.

This paragraph summarizes the VAX-11 addressing modes and contains examples of assembly language statements that use these addressing modes. The VAX-11/780 Architecture Handbook describes the addressing modes in detail.

Table 20-2 summarizes the addressing modes. Table 20-3 gives the values of the exponent and fraction portions of the 6-bit short literals that can be used in floating point instructions with literal mode addressing. Table 20-4 shows the syntax for all indexed addressing modes. The following pages include examples of each of the addressing modes.

Register mode

```
CLRB    R0          ; CLEAR LOWEST BYTE OF R0.
CLRA    R1          ; CLEAR R1 AND
TSTW    R10         ; TEST LOWER WORD OF R10
INCL    R4          ; ADD 1 TO R3
```

Register deferred mode

```
MOVAL   LDATA,R3    ; MOVE ADDRESS OF LDATA TO R3
CMPL    (R3),R0     ; COMPARE VALUE AT LDATA TO R0
MOVL    (SP),R1     ; COPY TOP ITEM OF STACK INTO R1
```

Autoincrement mode

```
MOVAL   TABLE,R1    ; GET ADDRESS OF TABLE
CLRO    (R1)+       ; CLEAR FIRST AND SECOND LONGWORDR
CLRL    (R1)+       ; AND THIRD LONGWORD IN TABLE
                    ; LEAVE R1 POINTING TO TABLE +12
```

Autoincrement deferred mode

```
MOVAL   PNTLIS,R2   ; GET ADDRESS OF POINTER LIST
CLRQ    @(R2)+      ; CLEAR QUADWORD POINTED TO BY
                    ; FIRST ABSOLUTE ADDRESS IN PNTLIS
                    ; THEN ADD 4 TO R2
CLRB    @(R2)+      ; CLEAR BYTE POINTED TO BY SECOND
                    ; ABSOLUTE ADDRESS IN PNTLIS
                    ; THEN ADD 4 TO R2
```

Autodecrement mode

```
CLRQ    -(R1)       ; SUBTRACT 8 FROM R1 AND ZERO THE
                    ; QUADWORD WHOSE ADDRESS IS THEN
                    ; IN R1
MOVZBL  R3,-(SP)    ; PUSH THE ZERO-EXTENDED LOW BYTE
                    ; OF R3 ONTO THE STACK AS A LONGWORD
```

Displacement mode

```
MOVAB   KEYWORDS,R3     ; GET ADDRESS OF KEYWORDS
MOVB    B^IO(R3),R4     ; GET BYTE WHOSE ADDRESS IS
                        ; IO PLUS ADDRESS OF KEYWORDS
                        ; THE DISPLACEMENT IS STORED
                        ; AS A BYTE
```

Displacement deferred mode

```
        MOVAL   ARRPOINT,R6         ; GET ADDRESS OF ARRAY OF POINTERS
        CLRL    @B^16(R6)           ; CLEAR LONGWORD POINTED TO BY
                                    ; LONGWORD WHOSE ADDRESS IS 16
                                    ; PLUS THE ADDRESS OF ARRPOINT
                                    ; THE DISPLACEMENT IS STORED AS A BYTE
```

Literal mode

```
        MOVAL   S^#1,R0             ; R0 IS SET TO 1; THE 1 IS STORED
                                    ; IN THE INSTRUCTION AS A SHORT
                                    ; LITERAL
        MOVF    S^#^023,R6          ; R6 IS SET TO THE FLOATING
                                    ; POINT VALUE 2.75; IT IS STORED
                                    ; IN THE FLOATING POINT SHORT
                                    ; LITERAL FORM
```

Relative mode

```
        CMPL    W^DATA+4,R10    ; COMPARE R10 WITH LONGWORD AT
                                ; ADDRESS DATA+4; THE ASSEMBLER
                                ; USES A WORD DISPLACEMENT
```

Relative defered mode

```
        INCB    @L^COUNTS+4     ; INCREMENT BYTE POINTED TO BY
                                ; LONGWORD AT COUNTS+4; ASSEMBLER
                                ; USES A LONGWORD DISPLACEMENT
```

Absolute mode

```
        CLRL    @#^X1100        ; CLEAR THE CONTENTS OF LOCATION 1100(HEX)
        CLRB    @#ACCNT         ; CLEAR THE CONTENTS OF LOCATION
                                ; ACCNT; THE ADDRESS IS STORED
                                ; ABSOLUTELY, NOT AS A DISPLACEMENT
```

Immediate mode

```
        ADDL2   I^#5,R0         ; THE 5 IS STORED IN A LONGWORD
                                ; BECAUSE THE I^ FORCES THE
                                ; ASSEMBLER TO USE IMMEDIATE MODE
```

Register deferred index mode

```
        MOVAB   BLIST,R9        ; GET ADDRESS OF BLIST
        MOVL    S^#20,R1        ; SET UP INDEX REGISTER
        CLRB    (R9)[R1]        ; CLEAR BYTE WHOSE ADDRESS
                                ; IS THE ADDRESS OF BLIST
                                ; PLUS 20*1
        CLRQ    (R9)[R1]        ; CLEAR QUADWORD WHOSE
                                ; ADDRESS IS THE ADDRESS
                                ; OF BLIST PLUS 20*8
```

Autoincrement index mode

```
        CLRW    (R9)+[R1]       ; CLEAR WORD WHOSE ADDRESS
                                ; IS ADDRESS OF BLIST PLUS
                                ; 20*2; R9 NOW CONTAINS
                                ; ADDRESS OF BLIST+2
```

Autoincrement deferred index mode

```
        MOVAL   POINT,R8        ; GET ADDRESS OF POINT
        MOVL    S^#30,R2        ; SET UP INDEX REGISTER
        CLRW    @(R8)+[R2]      ; CLEAR WORD WHOSE ADDRESS
                                ; IS 30*2 PLUS THE ADDRESS
                                ; STORED IN POINT; R8 NOW
                                ; CONTAINS 4 PLUS ADDRESS OF POINT
```

Displacement deferred index mode

```
        MOVAL   ADDARR,R9       ; GET ADDRESS OF ADDRESS ARRAY
        MOVL    I^#100,R1       ; SET UP INDEX REGISTER
        TSTF    @40(R9)[R1]     ; TEST FLOATING POINT VALUE
                                ; WHOSE ADDRESS IS 100*4 PLUS
                                ; THE ADDRESS STORED AT (ADDARR+40)
```

Branch mode

```
        ADDL3   (R1)+,R0,TOTAL  ; TOTAL VALUES AND SET CONDITION
                                ; CODES
        BLEQ    LABEL1          ; BRANCH TO LABEL1 IF RESULT IS
                                ; LESS THAN OR EQUAL TO 0
        BRW     LABEL           ; BRANCH UNCONDITIONALLY TO LABEL
```

Table 20-2

ADDRESSING MODES

Type	Addressing Mode	Format*	Hexa-decimal Value	Description	Indexable*
General Register	Register	Rn	5	Register contains the operand	No
	Register Deferred	(Rn)	6	Register contains the address of the operand	Yes
	Autoincrement	(Rn)+	8	Register contains the address of the operand; the processor increments the register contents by the size of the operand data type	Yes
	Autoincrement Deferred	@(Rn)+	9	Register contains the address of the operand address. the processor increments the register contents by 4	Yes
	Autodecrement	-(Rn)	7	The processor decrements the register contents by the size of the operand data type. the register then contains the address of the operand	Yes
	Displacement	B^dis(Rn) W^dis(Rn) L^dis(Rn)	A C E	The sum of the contents of the register and the displacement is the address of the operand; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Displacement Deferred	@B^dis(Rn) @W^dis(Rn) @L^dis(Rn)	B D F	The sum of the contents of the register and the displacement is the address of the operand address; B^, W^, and L^ indicate byte, word, and long-word displacement, respectively	Yes
	Literal	S^literal	0-3	The literal specified is the operand; the literal is stored as a short literal	No
Index	Index	base-mode[Rx]	4	The base-mode specifies the base address and the register specifies the index. the sum of the base address and the product of the contents of Rx and the size of the operand data type is the address of the operand. base-mode can be any addressing mode except register, immediate, literal, index, or branch	No
Branch	Branch	address		The address specified is the operand; this address is stored as a displacement to PC. branch mode can only be used with the branch instructions	No

*Key on following page

Type	Addressing Mode	Format*	Hexa-decimal Value	Description	Indexable?
Program Counter	Relative	B^address W^address L^address	A C E	The address specified is the address of the operand; the address specified is stored as a displacement from PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Relative Deferred	@B^address @W^address @L^address	B D F	The address specified is the address of the operand address; the address specified is stored as a displacement from PC; B^, W^, and L^ indicate byte, word, and longword displacement, respectively	Yes
	Absolute	@#address	9	The address specified is the address of the operand; the address specified is stored as an absolute virtual address (not as a displacement)	Yes
	Immediate	I^#literal	8	The literal specified is the operand; the literal is stored as a byte, word, longword, or quadword	No

* Key:

Rn
Any general register R0 through R12, AP (the same as R12), FP, or SP.

Rx
Any general register R0 through R12, AP, FP, or SP. Rx cannot be the same as the Rn specified in the base-mode for certain base modes.

disp
An expression specifying a displacement.

address
An expression specifying an address.

literal
An expression
*

Table 20-2 (Continued)

ADDRESSING MODES

Table 20-3

FLOATING POINT SHORT LITERALS

Exponent \ Fraction	0	1	2	3	4	5	6	7
0	0.5	0.5625	0.625	0.6875	0.75	0.8125	0.875	0.9375
1	1.0	1.125	1.25	1.375	1.5	1.625	1.75	1.875
2	2.0	2.25	2.5	2.75	3.0	3.25	3.5	3.75
3	4.0	4.5	5.0	5.5	6.0	6.5	7.0	7.5
4	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0
5	16.0	18.0	20.0	22.0	24.0	26.0	28.0	30.0
6	32.0	36.0	40.0	44.0	48.0	52.0	56.0	60.0
7	64.0	72.0	80.0	88.0	96.0	104.0	112.0	120.0

Example:

```
        MOVAL   S^#1,R0         ; R0 IS SET TO 1; THE 1 IS STORED
                                ; IN THE INSTRUCTION AS A SHORT
                                ; LITERAL
        MOVF    S^#^023,R6      ; R6 IS SET TO THE FLOATING
                                ; POINT VALUE 2.75; IT IS STORED
                                ; IN THE FLOATING POINT SHORT
                                ; LITERAL FORM
```

Table 20-4

INDEX MODE ADDRESSING

Mode	Format*
Register Deferred Index	(Rn)[Rx]
Autoincrement Index	(Rn)+[Rx]
Autoincrement Deferred Index	@(Rn)+[Rx]
Autodecrement Index	-(Rn)[Rx]
Displacement Index	dis(Rn)[Rx]
Displacement Deferred Index	@dis(Rn)[Rx]
Relative Index	address[Rx]
Relative Deferred Index	@address[Rx]
Absolute Index	@#address[Rx]

* Key:

Rn
: Any general register R0 through R12, AP, FP, or SP.

Rx
: Any general register R0 through R12, AP, FP, or SP. Rx cannot be the same register as Rn in the autoincrement index, autoincrement deferred index, and decrement index addressing modes.

dis
: An expression specifying a displacement.

address
: An expression specifying an address.

20.1.4 Assembler Directives.

Table 20-5 lists the assembler directives recognized by the ALS VAX-11/780 Assembler. This paragraph describes these directives in detail.

Table 20-5

SUMMARY OF ASSEMBLER DIRECTIVES

.BLKB	Space reservation/location control
.BYTE	Data definition
.WORD	Data definition
.LONG	Data definition
.END	Subprogram termination
.EQU	Symbol definition
.PSECT	Subprogram sectioning
.SUBPROGRAM	Subprogram initiation
.SEPARATE	Subunit initiation
.EXTREF	External reference definition

20.1.4.1 .BLKB.

The .BLKB directive reserves space in the current program section by advancing the program counter by the value of the operand expression.

Format:

```
[label]     .BLKB     expression
```

where:

Label is an optional label that will be assigned the value of the program counter before allocation of the space specified by the directive.

Expression is an expression as described in Section 20.1.2.4 that must evaluate to an absolute value. The program counter will be advanced by the number of bytes specified by the value of the number.

20.1.4.2 .BYTE, .WORD, .LONG.

These directives allocate one, two, or four bytes, respectively, at the current location and initialize the contents of that location to the value of the operand expression. The expression can evaluate to either a relocatable value or an absolute value.

Format:

```
[label]  .BYTE    expression
[label]  .WORD    expression
[label]  .LONG    expression
```

where:

Label is an optional label.

Expression is an expression described in Section 20.1.2.4 whose value will become the contents of the current location in the program section. The line will be flagged with an error if the value of the expression is absolute and too large for the amount of space specified. An error will be generated by the linker if the value is relocatable and found to be too large for the specified space.

20.1.4.3 .END.

The .END directive indicates the end of the assembly code compilation unit. It generates no code and is intended only to give the appearance of completeness to a compilation unit. Any lines after the .END directive will be flagged with an error, and will not be otherwise assembled.

Format:

```
.END
```

20.1.4.4 .EQU.

The .EQU directive defines the symbol in the label field and sets it equal to the value of the expression operand. The expression must evaluate to an absolute value (not relocatable). If the expression contains a symbol, that symbol must have been previously defined (in an .EQU directive occurring before this one).

Format:

```
label   .EQU expression
```

where:

Label is the symbol being defined.
Expression is an expression as described in Section 20.1.2.4 that must evauluate to an absolute value.

20.1.4.5 .PSECT.

The .PSECT directive separates the assembly code into program sections having different functions. There can be up to three program sections:

1. Executable code - represents instructions to be executed - data is not intended to be stored in the executable PSECT;
2. Read/write data - not intended to contain executable code; and
3. Read-only data - not intended to contain executable code, or data which is the target of a store.

The .PSECT directive defines the beginning of a portion of assembly code, continuing up to the next .PSECT or .END directive. An assembly may have no more than one PSECT of each of the three types and must have at least an executable PSECT. All code and data of the subprogram must follow a PSECT directive of one of the three types.

The body of the subprogram text always starts at the beginning of the executable PSECT.

Format:

```
.PSECT storage
```

where storage is one of the following storage area identifiers:

EXECUTABLE - the executable code of the subprogram body.

RWSTATIC - read/write data whose values will be maintained throughout the execution of the whole program, i.e., maintained across calls to this subprogram body.

ROSTATIC - read-only data maintained as in RWSTATIC.

20.1.4.6 .SUBPROGRAM.

The .SUBPROGRAM directive must appear exactly once in the assembly and must appear as the first statement after any .EXTREF directives unless this subprogram is a subunit, in which case the .SUBPROGRAM directive follows the .SEPARATE and .EXTREF directives. It supplies the name of the subprogram and must match the name in the Ada subprogram specification or procedure stub of the parent if this assembler body is a subunit subprogram.

Format:

.SUBPROGRAM name

where:

Name is the subprogram name and is constructed according to the rules for symbols (see Section 20.1.2.3).

20.1.4.7 .SEPARATE.

This directive specifies that the assembly program is to be a subunit subprogram of an Ada library unit package body. The assembly program will take on the same context as the parent package body. The package body which specifies this subprogram as a separate unit must be compiled before the assembly program.

Format:

.SEPARATE (package name)

where:

The package name is the name of an Ada library unit package of which the assembly program is to be a subunit. The package name is constructed according to the rules for symbols (see Section 20.1.2.3).

20.1.4.8 .EXTREF.

This directive facilitates the referencing of variables and subprograms declared in either library package specifications or library subprograms.

Format:

Label .EXTREF package_name.name

where:

1. The label, package_name and name are constructed according to the rules for symbols (see Section 20.1.2.3)

2. The label spelling is used in the symbol portion of an assembly instruction operand. The external reference will then be to a variable or subprogram specified by a package_name.name

20.1.5 Assembler Output.

This paragraph describes the outputs from an execution of the ALS VAX-11/780 Assembler: the source listing, error message, and machine text information.

20.1.5.1 Machine Text.

Machine text is what has been commonly referred to as "object module" or "relocatable binary" in other systems. In the Ada system there is no separate object module representation for the translator-generated executable code. Instead, translators, like the Ada VAX-11/780 Assembler, put their generated code into a Container in a program library.

20.1.5.2 Listing

At the option of the user a listing consisting of the source code side by side with the assembled machine text in hexadecimal can be produced by the assembler (see Section 20.1.6). The source line number and the hexadecimal location relative to the start of the PSECT are also listed. The total number of error messages is displayed at the end of the listing. The listing is produced in the standard output file.

Figure 20-1 shows some sample lines of the assembly listing. Note that the contents field in the listing is read right to left, i.e., the rightmost byte corresponds to the address given in the location field; the next byte to the left corresponds to that address plus one. Figure 20-2 shows the Ada library subprogram specification corresponding to the subprogram body in Figure 20-1.

```
a_spec_asm  PROGRAM LIBRARY=asmlib ASM VAX11/780 ASSEMBLER Version 0.4 ASSEMBLER    16-MAY-1983 14:37
BYTESTREAM (HEXADECIMAL:
READ FROM RIGHT TO LEFT) ADDRESS STMT #   LABEL   OPCODE            OPERANDS
-------------------------------------------------------------------------------

                                1           .SUBPROGRAM x
                                2           .PSECT      executable
                                3             ADDB3     x,y
                            *** ASMVAX E  10126    too few operands for this instruction
                                4             RSB
                                5           .PSECT      rwstatic
                                6   x       .BYTE       3
                                7   y       .LONG       3
                                8           .END
```

Figure 20-1. Sample Assembly Listing

```
Procedure   x;
```

Figure 20-2. Matching Ada Subprogram Specification

20.1.5.3 Diagnostic Messages.

Diagnostic messages are produced for all syntactic errors detected by the assembler. The message(s) for any particular line of source appears in the listing immediately following the source line. If no listing is requested, diagnostic messages appear in the message output file. The message number and text will be the same as the corresponding diagnostic generated for a listing. An example of a diagnostic message sent to the user through the message output file is as follows:

```
***ERROR ASMVAX 53403 AT SOURCE line Number 1 Subprogram Directive
Required
```

Messages indicating diagnostics in the assembler command are sent to the user via the message output file. Assembler command diagnostics are always sent to the user via message output regardless of the source option specification. An example of an Assembler Command Diagnostic Message is as follows:

```
***ERROR ASMVAX 56101 Command Diagnostic User Specified Source File
not found
```

If diagnostics of severity ERROR or FATAL are produced, a useable Container is not produced.

The diagnostic messages produced by the ALS VAX-11/780 Assembler are summarized in Appendix 80.

20.1.5.4 Summary Message.

At the completion of the assembly process, a summary message is sent to the user via the message output file. This summary message indicates the completion of the assembly process and the number of diagnostic conditions detected for each severity level. An example of a summary message is as follows:

```
ASMVAX Processing Complete
Number of Diagnostics Generated:
     2 of Severity level WARNING
     3 of Severity level ERROR
     0 of Severity level SYSTEM
     0 of Severity level FATAL
```

20.1.6 Invoking the Assembler.

The assembler is invoked with the command:

```
ASMVAX (source, prog_lib [,OPT=>option_list])
```

where:

source — the name of the source file

prog_lib — the name of the program library into which the source is assembled. The name of the Container produced in the program library is the name on the SUBPROGRAM directive.

option_list — [NO_]SOURCE specifies whether to produce a source listing or not. The default is SOURCE.

[NO_]CONTAINER_GENERATION specifies whether a Container is to be produced if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. If NO_CONTAINER_GENERATION is in effect, listings cannot be regenerated using the Display Tools CPCI. The default is CONTAINER_GENERATION.

20.1.7 Assembly Language Syntax.

This paragraph gives the formal syntax of the ALS VAX assembly language. The notation used here is a modified form of Backus-Naur Form (BNF). Angle brackets, "<" and ">", are used to enclose a syntactic unit which is defined to the left of a "::=". The symbol "::=" is read as "is defined as"; the vertical bar, "|", is read as "or"; anything enclosed in square brackets, "[" and "]", is optional; the curly braces "{ }" indicate that the enclosed unit can appear zero or more times; and two adjacent units (possibly on separate lines) indicate that the first must be followed by the second without intervening spaces.

For example, the following alternative for <executable instruction>:

```
| ADD<BWLFD>2 <space><r>,<m>
```

indicates that one of the operations an <executable instruction> can be is "ADD" immediately followed by one of the letters "B", "W", "L", "F", or "D" (that is the definition of <BWLFD> further down the page) followed by a "2", some number of spaces, a read (<r>) operand, a comma, and a modify (<m>) operand. Looking up the definitions of all the syntactic units in this example shows that

```
ADDW2 S^#59,(R10)+
```

is a valid <executable instruction>.

The full syntax of the assembly language follows.

```
<assembly code subprogram> ::=
     { [<space>] <comment> <eol> }
        <program_heading>
     { [<space>] <comment> <eol> }
        <program_section>
       [<program_section>]
       [<program_section>]
        <END_directive>
     { [<space>] <comment> <eol> }

<program_heading>       ::= <subprogram_directive>
                          ¦ <subunit_heading>
                            { [<space>] <comment> <eol> }
                             <subprogram_directive>

<subunit_heading>       ::=
     <space> .SEPARATE <space> ( <package_name> ) [<space>] [<comment>] <eol>
     { { [<space>] <comment> <eol> } <external_directive> }

<subprogram_directive> ::= {{[<space>] <comment> <col>} <external_directives>}
     <space> .SUBPROGRAM <space> <label> [<space>] [<comment>] <eol>

<external_directive>    ::=
     <line_symbol> <space> .EXTREF <space> <external_name>
              [<space>] [<comment>] <eol>

<external_name>         ::= <label>. <label> {.<label>}

<package_name>          ::= <label> -- where the label spelling is a
                                    -- library unit package name

<program_section>       ::=  <PSECT_directive>
                             <assembly_code_line>   -- there must exist
                           { <assembly_code_line> } -- at least one

<PSECT_directive>       ::=
     <space> .PSECT <space> <PSECT_attribute> [<space>] [<comment>] <eol>
<PSECT_attribute>       ::= ROSTATIC ¦ RWSTATIC ¦ EXECUTABLE

<assembly_code_line>    ::=
     [<line_symbol>] <space> <instruction> [<space>] [<comment>] <eol>
   ¦ <line_symbol> <space> .EQU <abs_expression> [<space>] [<comment>] <eol>
   ¦ [<space>] <comment> <eol>

<line_symbol>           ::= <label> [:] -- starting in column 1 of
                                        -- the source line.
<instruction>           ::=   <directive>
                            ¦ <executable_instruction>
```

```
<directive>              ::=   .BLKB <space> <abs_expression>
                             | .BYTE <space> <relocatable_expression>
                             | .WORD <space> <relocatable_expression>
                             | .LONG <space> <relocatable_expression>

<END_directive>          ::=
     <space> .END [<space>] [<comment>] <eol>

<executable_instruction> ::=
     ACBB    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH BYTE
   | ACBD    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH D_FLOAT
   | ACBF    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH F_FLOAT
   | ACBG    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH G_FLOAT
   | ACBH    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH H_FLOAT
   | ACBL    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH LONG
   | ACBW    <space> <r>,<r>,<m>,<b>          ADD COMPARE BRANCH WORD
   | ADAWI   <space> <r>,<m>                  ADD ALIGNED WORD INTERLOCKED
   | ADDB2   <space> <r>,<m>                  ADD BYTE 2 OPERAND
   | ADDD2   <space> <r>,<m>                  ADD D_FLOAT 2 OPERAND
   | ADDF2   <space> <r>,<m>                  ADD F_FLOAT 2 OPERAND
   | ADDG2   <space> <r>,<m>                  ADD G_FLOAT 2 OPERAND
   | ADDH2   <space> <r>,<m>                  ADD H_FLOAT 2 OPERAND
   | ADDL2   <space> <r>,<m>                  ADD LONG 2 OPERAND
   | ADDW2   <space> <r>,<m>                  ADD WORD 2 OPERAND
   | ADDB3   <space> <r>,<r>,<m>              ADD BYTE 3 OPERAND
   | ADDD3   <space> <r>,<r>,<m>              ADD D_FLOAT 3 OPERAND
   | ADDF3   <space> <r>,<r>,<m>              ADD F_FLOAT 3 OPERAND
   | ADDG3   <space> <r>,<r>,<m>              ADD G_FLOAT 3 OPERAND
   | ADDH3   <space> <r>,<r>,<m>              ADD H_FLOAT 3 OPERAND
   | ADDL3   <space> <r>,<r>,<m>              ADD LONG 3 OPERAND
   | ADDW3   <space> <r>,<r>,<m>              ADD WORD 3 OPERAND
   | ADDP4   <space> <r>,<a>,<r>,<a>          ADD PACKED 4 OPERNAD
   | ADDP6   <space> <r>,<a>,<r>,<a>,<r>,<a>  ADD PACKED 6 OPERAND
   | ADWC    <space> <r>,<m>                  ADD WITH CARRY
   | AOBLEQ  <space> <r>,<m>,<b>              ADD ONE BRANCH LESS EQUAL
   | AOBLSS  <space> <r>,<m>,<b>              ADD ONE BRANCH ON LESS
   | ASHL    <space> <r>,<r>,<m>              ARITH SHIFT LONG
   | ASHP    <space> <r>,<r>,<a>,<r>,<r>,<a>  ARITH SHIFT ROUND PACKED
   | ASHQ    <space> <r>,<r>,<m>              ARITH SHIFT QUADWORD
   | BBC     <space> <r>,<a>,<b>              BRANCH ON BIT CLEAR
   | BBCC    <space> <r>,<a>,<b>              BRANCH ON BIT CLEAR AND CLEAR
   | BBCCI   <space> <r>,<a>,<b>              BRANCH ON BIT CLEAR CLEAR INTER
   | BBCS    <space> <r>,<a>,<b>              BRANCH ON BIT CLEAR AND SET
   | BBS     <space> <r>,<a>,<b>              BRANCH ON BIT SET
   | BBSC    <space> <r>,<a>,<b>              BRANCH ON BIT SET AND CLEAR
   | BBSS    <space> <r>,<a>,<b>              BRANCH ON BIT SET AND SET
   | BBSSI   <space> <r>,<a>,<b>              BRANCH ON BIT SET SET INTERLOCK
   | BCC     <space> <b>                      BRANCH ON CARRY CLEAR
   | BCS     <space> <b>                      BRANCH ON CARRY SET
   | BEQL    <space> <b>                      BRANCH ON EQUAL
   | BEQLU   <space> <b>                      BRANCH ON EQUAL (JNSIGNED)
   | BGEQ    <space> <b>                      BRANCH ON GREATER EQUAL
   | BGEQU   <space> <b>                      BRANCH ON GREATER EQUAL (UNSIGN)
```

BGTR	<space> <b>	BRANCH ON GREATER
BGTRU	<space> <b>	BRANGH ON GREATER (UNSIGNED)
BICB2	<space> <r>,<m>	BIT CLEAR BYTE 2 OPERAND
BICL2	<space> <r>,<m>	BIT CLEAR LONG 2 OPERAND
BICW2	<space> <r>,<m>	BIT CLEAR WORD 2 OPERAND
BICB3	<space> <r>,<r>,<m>	BIT CLEAR BYTE 3 OPERAND
BICL3	<space> <r>,<r>,<m>	BIT CLEAR LONG 3 OPERAND
BICW3	<space> <r>,<r>,<m>	BIT CLEAR WORD 3 OPERAND
BICPSW	<space> <r>	BIT CLEAR PSW
BISB2	<space> <r>,<m>	BIT SET BYTE 2 OPERAND
BISL2	<space> <r>,<m>	BIT SET LONG 2 OPERAND
BISW2	<space> <r>,<m>	BIT SET WORD 2 OPERAND
BISB3	<space> <r>,<r>,<m>	BIT SET BYTE 3 OPERAND
BISL3	<space> <r>,<r>,<m>	BIT SET LONG 3 OPERAND
BISW3	<space> <r>,<r>,<m>	BIT SET WORD 3 OPERAND
BISPSW	<space> <r>	BIT SET PSW
BITB	<space> <r>,<r>	BIT TEST BYTE
BITL	<space> <r>,<r>	BIT TEST LONGWORD
BITW	<space> <r>,<r>	BIT TEST WORD
BLBC	<space> <r>,<b>	BRANCH ON LOW BIT CLEAR
BLBS	<space> <r>,<b>	BRANCH ON LOW BIT SET
BLEQ	<space> <b>	BRANCH ON LESS EQUAL
BLEQU	<space> <b>	BRANCH ON LESS EQUAL (UNSIGN)
BLSS	<space> <b>	BRANCH ON LESS
BLSSU	<space> <b>	BRANCH ON LESS (UNSIGNED)
BNEQ	<space> <b>	BRANCH ON NOT EQUAL
BNEQU	<space> <b>	BRANCH ON NOT EQUAL (UNSIGNED)
BPT		BREAK POINT FAULT
BRB	<space> <b>	BRANCH WITH BYTE DISP
BRW	<space> <c>	BRANCH WITH WORD
BSBB	<space> <b>	BRANCH TO SUBROUTINE WITH BYTE
BSBW	<space> <c>	BRANCH TO SUBROUTINE WITH WORD
BUGL	<space> <r>	BUGCHECK LONGWORD
BUGW	<space> <r>	BUGCHECK WORD
BVC	<space> <b>	BRANCH ON OVERFLOW CLEAR
BVS	<space> <b>	BRANCH ON OVERFLOW SET
CALLG	<space> <a>,<a>	CALL WITH GENERAL ARGUMENTS
CALLS	<space> <r>,<a>	CALL WITH STACK
CASEB	<space> <r>,<r>,<r>	CASE BYTE
CASEL	<space> <r>,<r>,<r>	CASE LONG
CASEW	<space> <r>,<r>,<r>	CASE WORD
CHME	<space> <r>	CHANGE MODE TO EXECUTIVE
CHMK	<space> <r>	CHANGE MODE TO KERNAL
CHMS	<space> <r>	CHANGE MODE TO SUPERVISOR
CHMU	<space> <r>	CHANGE MODE TO USER
CLRB	<space> <m>	CLEAR BYTE
CLRD	<space> <m>	CLEAR D_FLOATING
CLRF	<space> <m>	CLEAR F_FLOAT
CLRG	<space> <m>	CLEAR G_FLOAT
CLRH	<space> <m>	CLEAR H_FLOAT
CLRL	<space> <m>	CLEAR LONG
CLRO	<space> <m>	CLEAR OCTALWORD
CLRQ	<space> <m>	CLEAR QUADWORD

```
| CLRW    <space> <m>                      CLEAR WORD
| CMPB    <space> <r>,<r>                  COMPARE BYTE
| CMPD    <space> <r>,<r>                  COMPARE D_FLOATING
| CMPF    <space> <r>,<r>                  COMPARE F_FLOAT
| CMPG    <space> <r>,<r>                  COMPARE G_FLOAT
| CMPH    <space> <r>,<r>                  COMPARE H_FLOAT
| CMPL    <space> <r>,<r>                  COMPARE LONG
| CMPW    <space> <r>,<r>                  COMPARE WORD
| CMPC3   <space> <r>,<a>,<a>              COMPARE CHARACTER 3 OPERAND
| CMPC5   <space> <r>,<a>,<r>,<r>,<a>      COMPARE CHARACTER 5 OPERAND
| CMPP3   <space> <r>,<a>,<a>              COMPARE PACKED 3 OPERAND
| CMPP4   <space> <r>,<a>,<r>,<a>          COMPARE PACKED 5 OPERAND
| CMPV    <space> <r>,<r>,<v>,<r>          COMPARE FIELD
| CMPZV   <space> <r>,<r>,<v>,<r>          COMPARE ZERO EXTENDED FIELD
| CRC     <space> <a>,<r>,<r>,<a>          CALC CYCLIC REDUND CHECK
| CVTBD   <space> <r>,<m>                  CONVERT BYTE TO D_FLOAT
| CVTBF   <space> <r>,<m>                  CONVERT BYTE TO F_FLOAT
| CVTBG   <space> <r>,<m>                  CONVERT BYTE TO G_FLOAT
| CVTBH   <space> <r>,<m>                  CONVERT BYTE TO H_FLOAT
| CVTBL   <space> <r>,<m>                  CONVERT BYTE TO LONG
| CVTBW   <space> <r>,<m>                  CONVERT BYTE TO WORD
| CVTDB   <space> <r>,<m>                  CONVERT D_FLOAT TO BYTE
| CVTDF   <space> <r>,<m>                  CONVERT D_FLOAT TO F_FLOAT
| CVTDH   <space> <r>,<m>                  CONVERT D_FLOAT TO H_FLOAT
| CVTDL   <space> <r>,<m>                  CONVERT D_FLOAT TO LONG
| CVTDW   <space> <r>,<m>                  CONVERT D_FLOAT TO WORD
| CVTFB   <space> <r>,<m>                  CONVERT F_FLOAT TO BYTE
| CVTFD   <space> <r>,<m>                  CONVERT F_FLOAT TO D_FLOAT
| CVTFG   <space> <r>,<m>                  CONVERT F_FLOAT TO G_FLOAT
| CVTFH   <space> <r>,<m>                  CONVERT F_FLOAT TO H_FLOAT
| CVTFL   <space> <r>,<m>                  CONVERT F_FLOAT TO LONG
| CVTFW   <space> <r>,<m>                  CONVERT F_FLOAT TO WORD
| CVTGB   <space> <r>,<m>                  CONVERT G_FLOAT TO BYTE
| CVTGH   <space> <r>,<m>                  CONVERT G_FLOAT TO H_FLOAT
| CVTGL   <space> <r>,<m>                  CONVERT G_FLOAT TO LONGWORD
| CVTGW   <space> <r>,<m>                  CONVERT G_FLOAT TO WORD
| CVTHB   <space> <r>,<m>                  CONVERT H_FLOAT TO BYTE
| CVTHD   <space> <r>,<m>                  CONVERT H_FLOAT TO D_FLOAT
| CVTHF   <space> <r>,<m>                  CONVERT H_FLOAT TO F_FLOAT
| CVTHG   <space> <r>,<m>                  CONVERT H_FLOAT TO G_FLOAT
| CVTHL   <space> <r>,<m>                  CONVERT H_FLOAT TO LONGWORD
| CVTHW   <space> <r>,<m>                  CONVERT H_FLOAT TO WORD
| CVTLB   <space> <r>,<m>                  CONVERT LONG TO BYTE
| CVTLD   <space> <r>,<m>                  CONVERT LONG TO D_FLOAT
| CVTLF   <space> <r>,<m>                  CONVERT LONG TO F_FLOAT
| CVTLG   <space> <r>,<m>                  CONVERT LONG TO G_FLOAT
| CVTLH   <space> <r>,<m>                  CONVERT LONG TO H_FLOAT
| CVTLW   <space> <r>,<m>                  CONVERT LONG TO WORD
| CVTLP   <space> <r>,<r>,<a>              CONVERT LONG TO PACKED
| CVTPL   <space> <r>,<a>,<m>              CONVERT PACKED TO LONG
| CVTPS   <space> <r>,<a>,<r>,<a>          CONVERT PACKED TO LEAD SEP NUM
| CVTPT   <space> <r>,<a>,<a>,<r>,<a>      CONVERT PACKED TO TRAIL SEP NUM
| CVTRDL  <space> <r>,<m>                  CONVERT ROUNDED D_FLOAT TO LONG
```

```
| CVTRFL  <space> <r>,<m>                  CONVERT ROUNDED F_FLOAT TO LONG
| CVTRGL  <space> <r>,<m>                  CONVERT ROUNDED G_FLOAT TO LONG
| CVTRHL  <space> <r>,<m>                  CONVERT ROUNDED H_FLOAT TO LONG
| CVTSP   <space> <r>,<a>,<r>,<a>          CONVERT LEAD SEP NUM TO PACKED
| CVTTP   <space> <r>,<a>,<a>,<r>,<a>      CONVERT TRAIL SEP NUM TO PACKED
| CVTWB   <space> <r>,<m>                  CONVERT WORD TO BYTE
| CVTWD   <space> <r>,<m>                  CONVERT WORD TO D_FLOATING
| CVTWF   <space> <r>,<m>                  CONVERT WORD TO F_FLOATING
| CVTWG   <space> <r>,<m>                  CONVERT WORD TO G_FLOATING
| CVTWH   <space> <r>,<m>                  CONVERT WORD TO H_FLOATING
| CVTWL   <space> <r>,<m>                  CONVERT WORD TO LONG
| DECB    <space> <m>                      DECREMENT BYTE
| DECL    <space> <m>                      DECREMENT LONG
| DECW    <space> <m>                      DECREMENT WORD
| DIVB2   <space> <r>,<m>                  DIVIDE BYTE 2 OPERAND
| DIVD2   <space> <r>,<m>                  DIVIDE D_FLOAT 2 OPERAND
| DIVF2   <space> <r>,<m>                  DIVIDE F_FLOAT 2 OPERAND
| DIVG2   <space> <r>,<m>                  DIVIDE G_FLOAT 2 OPERAND
| DIVH2   <space> <r>,<m>                  DIVIDE H_FLOAT 2 OPERAND
| DIVL2   <space> <r>,<m>                  DIVIDE LONG 2 OPERAND
| DIVW2   <space> <r>,<m>                  DIVIDE WORD 2 OPERAND
| DIVB3   <space> <r>,<r>,<m>              DIVIDE BYTE 3 OPERAND
| DIVD3   <space> <r>,<r>,<m>              DIVIDE D_FLOAT 3 OPERAND
| DIVF3   <space> <r>,<r>,<m>              DIVIDE F_FLOAT 3 OPERAND
| DIVG3   <space> <r>,<r>,<m>              DIVIDE G_FLOAT 3 OPERAND
| DIVH3   <space> <r>,<r>,<m>              DIVIDE H_FLOAT 3 OPERAND
| DIVL3   <space> <r>,<r>,<m>              DIVIDE LONG 3 OPERAND
| DIVW3   <space> <r>,<r>,<m>              DIVIDE WORD 3 OPERAND
| DIVP    <space> <r>,<a>,<r>,<a>,<r>,<a>  DIVIDE PACKED
| EDITPC  <space> <r>,<a>,<a>,<a>          EDIT PACKED TO CHARACTER
| EDIV    <space> <r>,<r>,<m>,<m>          EXTENDED DIVIDE
| EMODD   <space> <r>,<r>,<r>,<m>,<m>      EXTENDED MODULUS D_FLOATING
| EMODF   <space> <r>,<r>,<r>,<m>,<m>      EXTENDED MODULUS F_FLOAT
| EMODG   <space> <r>,<r>,<r>,<m>,<m>      EXTENDED MODULUS G_FLOAT
| EMODH   <space> <r>,<r>,<r>,<m>,<m>      EXTENDED MODULUS H_FLOAT
| EMUL    <space> <r>,<r>,<r>,<m>          EXTENDED MULTIPLY
| EXTV    <space> <r>,<r>,<v>,<m>          EXTRACT FIELD
| EXTZV   <space> <r>,<r>,<v>,<m>          EXTRACT ZERO EXTENDED FIELD
| FFC     <space> <r>,<r>,<v>,<m>          FIND FIRST CLEAR BIT
| FFS     <space> <r>,<r>,<v>,<m>          FIND FIRST SET BIT
| HALT                                     HALT
| INCB    <space> <m>                      INCREMENT BYTE
| INCL    <space> <m>                      INCREMENT LONG
| INCW    <space> <m>                      INCREMENT WORD
| INDEX   <space> <r>,<r>,<r>,<r>,<r>,<m> COMPUTE INDEX
| INSQHI  <space> <a>,<a>                  INSERT INTO Q HEAD INTERLOCKED
| INSQTI  <space> <a>,<a>                  INSERT INTO Q TAIL INTERLOCKED
| INSQUE  <space> <a>,<a>                  INSERT INTO Q
| INSV    <space> <r>,<r>,<r>,<a>          INSERT FIELD
| JMP     <space> <a>                      JUMP
| JSB     <space> <a>                      JUMP TO SUBROUTINE
| LDPCTX                                   LOAD PROCESS CONTEXT
| LOCC    <space> <r>,<r>,<a>              LOCATE CHARACTER
```

```
| MATCHC   <space> <r>,<a>,<r>,<a>          MATCH CHARACTERS
| MCOMB    <space> <r>,<m>                  MOVE COMPLEMENTED BYTE
| MCOML    <space> <r>,<m>                  MOVE COMPLEMENTED LONG
| MCOMW    <space> <r>,<m>                  MOVE COMPLEMENTED WORD
| MFPR     <space> <r>,<r>                  MOVE FROM PRIVILEGE REGISTER
| MNEGB    <space> <r>,<m>                  MOVE NEGATED BYTE
| MNEGD    <space> <r>,<m>                  MOVE NEGATED D_FLOATING
| MNEGF    <space> <r>,<m>                  MOVE NEGATED F_FLOAT
| MNEGG    <space> <r>,<m>                  MOVE NEGATED G_FLOAT
| MNEGH    <space> <r>,<m>                  MOVE NEGATED H_FLOAT
| MNEGL    <space> <r>,<m>                  MOVE NEGATED LONG
| MNEGW    <space> <r>,<m>                  MOVE NEGATED WORD
| MOVAB    <space> <a>,<m>                  MOVE ADDRESS OF BYTE
| MOVAD    <space> <a>,<m>                  MOVE ADDRESS OF D_FLOATING
| MOVAF    <space> <a>,<m>                  MOVE ADDRESS OF F_FLOAT
| MOVAG    <space> <a>,<m>                  MOVE ADDRESS OF G_FLOAT
| MOVAH    <space> <a>,<m>                  MOVE ADDRESS OF H_FLOAT
| MOVAL    <space> <a>,<m>                  MOVE ADDRESS OF LONG
| MOVAO    <space> <a>,<m>                  MOVE ADDRESS OF OCTALWORD
| MOVAQ    <space> <a>,<m>                  MOVE ADDRESS OF QUADWORD
| MOVAW    <space> <a>,<m>                  MOVE ADDRESS OF WORD
| MOVB     <space> <r>,<m>                  MOVE BYTE
| MOVD     <space> <r>,<m>                  MOVE D_FLOAT
| MOVF     <space> <r>,<m>                  MOVE F_FLOAT
| MOVG     <space> <r>,<m>                  MOVE G_FLOAT
| MOVH     <space> <r>,<m>                  MOVE H_FLOAT
| MOVL     <space> <r>,<m>                  MOVE LONG
| MOVO     <space> <r>,<m>                  MOVE OCTALWORD
| MOVQ     <space> <r>,<m>                  MOVE QUADWORD
| MOVW     <space> <r>,<m>                  MOVE WORD
| MOVC3    <space> <r>,<a>,<a>              MOVE CHARACTER 3 OPERAND
| MOVC5    <space> <r>,<a>,<r>,<r>,<a>      MOVE CHARACTER 5 OPERAND
| MOVP     <space> <r>,<a>,<a>              MOVE PACKED
| MOVPSL   <space> <m>                      MOVE PROCESSOR STATUS LONG
| MOVTC    <space> <r>,<a>,<r>,<a>,<r>,<a>  MOVE TRANSLATED CHARACTERS
| MOVTUC   <space> <r>,<a>,<r>,<a>,<r>,<a>  MOVE TRANSLATED UNTIL CHAR
| MOVZBL   <space> <r>,<m>                  MOVE ZERO EXTENDED BYTE LONG
| MOVZBW   <space> <r>,<m>                  MOVE ZERO EXTENDED BYTE WORD
| MOVZWL   <space> <r>,<m>                  MOVE ZERO EXTENDED WORD LONG
| MTPR     <space> <r>,<m>                  MOVE TO PRIVILEGE REGISTER
| MULB2    <space> <r>,<m>                  MULTIPLY BYTE 2 OPERAND
| MULD2    <space> <r>,<m>                  MULTIPLY D_FLOAT 2 OPERAND
| MULF2    <space> <r>,<m>                  MULTIPLY F_FLOAT 2 OPERAND
| MULG2    <space> <r>,<m>                  MULTIPLY G_FLOAT 2 OPERAND
| MULH2    <space> <r>,<m>                  MULTIPLY H_FLOAT 2 OPERAND
| MULL2    <space> <r>,<m>                  MULTIPLY LONG 2 OPERAND
| MULW2    <space> <r>,<m>                  MULTIPLY WORD 2 OPERAND
| MULB3    <space> <r>,<r>,<m>              MULTIPLY BYTE 3 OPERAND
| MULD3    <space> <r>,<r>,<m>              MULTIPLY D_FLOAT 3 OPERAND
| MULF3    <space> <r>,<r>,<m>              MULTIPLY F_FLOAT 3 OPERAND
| MULG3    <space> <r>,<r>,<m>              MULTIPLY G_FLOAT 3 OPERAND
| MULH3    <space> <r>,<r>,<m>              MULTIPLY H_FLOAT 3 OPERAND
| MULL3    <space> <r>,<r>,<m>              MULTIPLY LONG 3 OPERAND
```

```
| MULW3   <space> <r>,<r>,<m>                 MULTIPLY WORD 3 OPERAND
| MULP    <space> <r>,<a>,<r>,<a>,<r>,<a>     MULTIPLY PACKED
| NOP                                         NO OPERATION
| POLYD   <space> <r>,<r>,<a>                 EVALUATE POLYNOMIAL D_FLOAT
| POLYF   <space> <r>,<r>,<a>                 EVALUATE POLYNOMIAL F_FLOAT
| POLYG   <space> <r>,<r>,<a>                 EVALUATE POLYNOMIAL G_FLOAT
| POLYH   <space> <r>,<r>,<a>                 EVALUATE POLYNOMIAL H_FLOAT
| POPR    <space> <r>                         POP REGISTERS
| PROBER  <space> <r>,<r>,<a>                 PROBE READ ACCESS
| PROBEW  <space> <r>,<r>,<a>                 PROBE WRITE ACCESS
| PUSHAB  <space> <a>                         PUSH ADDRESS OF BYTE
| PUSHAD  <space> <a>                         PUSH ADDRESS OF D_FLOAT
| PUSHAF  <space> <a>                         PUSH ADDRESS OF F_FLOAT
| PUSHAG  <space> <a>                         PUSH ADDRESS OF G_FLOAT
| PUSHAH  <space> <a>                         PUSH ADDRESS OF H_FLOAT
| PUSHAL  <space> <a>                         PUSH ADDRESS OF LONG
| PUSHAO  <space> <a>                         PUSH ADDRESS OF OCTALWORD
| PUSHAQ  <space> <a>                         PUSH ADDRESS OF QUADWORD
| PUSHAW  <space> <a>                         PUSH ADDRESS OF WORD
| PUSHL   <space> <r>                         PUSH LONGWORD
| PUSHR   <space> <r>                         PUSH REGISTERS
| REI                                         RETURN FROM EXCEPTION OR INTERUPT
| REMQHI  <space> <a>,<m>                     REMOVE FROM Q HEAD INTERLOCKED
| REMQTI  <space> <a>,<m>                     REMOVE FROM Q TAIL INTERLOCKED
| REMQUE  <space> <a>,<m>                     REMOVE FROM Q
| RET     <space>                             RETURN FROM CALLED PROCEDURE
| ROTL    <space> <r>,<r>,<m>                 ROTATE LONGWORD
| RSB                                         RETURN FROM SUBROUTINE
| SBWC    <space> <r>,<m>                     SUBTRACT WITH CARRY
| SCANC   <space> <r>,<a>,<a>,<r>             SCAN FOR CHARACTER
| SKPC    <space> <r>,<r>,<a>                 SKIP CHARACTER
| SOBGEQ  <space> <m>,<b>                     SUB 1 BRANCH ON GREATER EQUAL
| SOBGTR  <space> <m>,<b>                     SUB 1 BRANCH ON GREATER
| SPANC   <space> <r>,<a>,<a>,<r>             SPAN CHARACTERS
| SUBB2   <space> <r>,<m>                     SUBTRACT BYTE 2 OPERAND
| SUBD2   <space> <r>,<m>                     SUBTRACT D_FLOAT 2 OPERAND
| SUBF2   <space> <r>,<m>                     SUBTRACT F_FLOAT 2 OPERAND
| SUBG2   <space> <r>,<m>                     SUBTRACT G_FLOAT 2 OPERAND
| SUBH2   <space> <r>,<m>                     SUBTRACT H_FLOAT 2 OPERAND
| SUBL2   <space> <r>,<m>                     SUBTRACT LONG 2 OPERAND
| SUBW2   <space> <r>,<m>                     SUBTRACT WORD 2 OPERAND
| SUBB3   <space> <r>,<r>,<m>                 SUBTRACT BYTE 3 OPERAND
| SUBD3   <space> <r>,<r>,<m>                 SUBTRACT D_FLOAT 3 OPERAND
| SUBF3   <space> <r>,<r>,<m>                 SUBTRACT F_FLOAT 3 OPERAND
| SUBG3   <space> <r>,<r>,<m>                 SUBTRACT G_FLOAT 3 OPERAND
| SUBH3   <space> <r>,<r>,<m>                 SUBTRACT H_FLOAT 3 OPERAND
| SUBL3   <space> <r>,<r>,<m>                 SUBTRACT LONG 3 OPERAND
| SUBW3   <space> <r>,<r>,<m>                 SUBTRACT WORD 3 OPERAND
| SUBP4   <space> <r>,<a>,<r>,<a>             SUBTRACT PACKED 4 OPERAND
| SUBP6   <space> <r>,<a>,<r>,<a>,<r>,<a>     SUBTRACT PACKED 6 OPERAND
| SVPCTX                                      SAVE PROCESS CONTEXT
| TSTB    <space> <r>                         TEST BYTE
| TSTD    <space> <r>                         TEST D_FLOAT
```

```
    | TSTF    <space> <r>                    TEST F_FLOAT
    | TSTG    <space> <r>                    TEST G_FLOAT
    | TSTH    <space> <r>                    TEST H_FLOAT
    | TSTL    <space> <r>                    TEST LONG
    | TSTW    <space> <r>                    TEST WORD
    | XFC                                    EXTENDED FUNCTION CALL
    | XORB2   <space> <r>,<m>                EXCLUSIVE OR BYTE 2 OPERAND
    | XORL2   <space> <r>,<m>                EXCLUSIVE OR LONG 2 OPERAND
    | XORW2   <space> <r>,<m>                EXCLUSIVE OR WORD 2 OPERAND
    | XORB3   <space> <r>,<r>,<m>            EXCLUSIVE OR BYTE 3 OPERAND
    | XORL3   <space> <r>,<r>,<m>            EXCLUSIVE OR LONG 3 OPERAND
    | XORW3   <space> <r>,<r>,<m>            EXCLUSIVE OR WORD 3 OPERAND

<m>           ::=   <memory_operand>
                  | <register_operand>

<a>           ::=   <memory_operand>

<v>           ::=   <memory_operand>
                  | <register_operand>
                  | <immediate_operand>

<r>           ::=   <memory_operand>
                  | <register_operand>
                  | <literal_operand>
                  | <immediate_operand>

<b>           ::=   <label> -- within 127 of current instruction
<c>           ::=   <label> -- within 32K of current instruction

<memory_operand>    ::=
     <base_operand> [<left_bracket> <general_register><right_bracket>]

<literal_operand>   ::= S^# <literal_value>

<immediate_operand> ::= I^# <numeric_value>

<base_operand>  ::=    ( <register_designator> )          -- reg deferred
                     | -( <register_designator> )         -- autodec
                     |  ( <register_designator> )+        -- autoinc
                     | @( <register_designator> )+        -- autoinc def
                     | B^ <byte_value> ( <register_designator> )
                                                          -- byte disp
                     | W^ <word_value> ( <register_designator> )
                                                          -- word disp
                     | L^ <long_value> ( <register_designator> )
                                                          -- long disp
                     | @B^ <byte_value> ( <register_designator> )
                                                          -- byte disp def
                     | @W^ <word_value> ( <register_designator> )
                                                          -- word disp def
                     | @L^ <long_value> ( <register_designator> )
                                                          -- long disp def
```

```
                    | @ # <relocatable_expression>        -- absolute
                    |  B^ <byte_relative_value>          -- byte relative
                    |  W^ <word_relative_value>          -- word relative
                    |  L^ <long_relative_value>          -- long relative
                    | @B^ <byte_relative_value>          -- byte rel def
                    | @W^ <word_relative_value>          -- word rel def
                    | @L^ <long_relative_value>          -- long rel def

<register_operand>    ::= <general_register>
<register_designator> ::= <general_register> | PC
<general_register>    ::=  R0 | R1 | R2 | R3
                         | R4 | R5 | R6 | R7
                         | R8 | R9 | R10| R11
                         | AP | FP | SP
                         | r0 | r1 | r2 | r3
                         | r4 | r5 | r6 | r7
                         | r8 | r9 | r10| r11
                         | ap | fp | sp

<literal_value>    ::= <abs_expression>  -- in the range 0..63

<byte_value>       ::= <abs_expression>  -- of 8 bits
<word_value>       ::= <abs_expression>  -- of 16 bits
<long_value>       ::= <abs_expression>  -- of 32 bits

<byte_relative>    ::= <psect_relative_expression>  -- of 8 bits
<word_relative>    ::= <psect_relative_expression>  -- of 16 bits
<long_relative>    ::= <psect_relative_expression>  -- of 32 bits

<psect_relative_expression> ::= <label>
                              | <label> + <numeric_value>
                              | <label> - <numeric_value>
                            -- where label is defined in the current psect

<relocatable_expression>    ::= <symbol>+<numeric_value>
                              | <symbol>-<numeric_value>
                              | <symbol>
                              | <case_expression>  -- used exclusively for
                                                  -- case statement displace-
                                                  -- ments

<case_expression>           ::= <symbol>-<symbol>

<abs_expression>            ::= <numeric_value>
                              | <symbolic_constant>
                              | <symbolic_constant>+<numeric_value>
                              | <symbolic_constant>-<numeric_value>

<symbol>                    ::= <label>

<symbolic_constant>         ::= <label> -- assigned by an equ directive
```

```
<numeric_value>          ::= [=]<digit> { <digit> }
                           | ^X<hex_digit> { <hex_digit> }
                           | ^O<octal_digit> { <octal_digit> }

<space>           ::= <separator_char> { <separator_char> }
<separator_char>  ::= space character | horizontal tab character
<label>           ::= <letter> { [_] <alphanumeric> }
<left_bracket>    ::= [
<right_bracket>   ::= ]
<eol>             ::= end of line character { end of line character }
<alphanumeric>    ::= <letter> | <digit>
<letter>          ::= 'A'..'Z' | 'a'..'z'
<digit>           ::= '0'..'9'
<hex_digit>       ::= '0'..'9' | 'a'..'f' | 'A'..'F'
<octal_digit>     ::= '0'..'7'
<comment>         ::= ; { <character> }
<character>       ::= Any ASCII character except null, del, or eol
```

20.1.8 Assembly Language Comparison

The differences between the ALS VAX-11/780 Assembly Language and the Digital Equipment Corporation (DEC) VAX-11 Macro Language are generally a result of the different intended purposes of the two languages. The DEC VAX-11 Macro Language provides features necessary for efficient, large assembly language program development and maintenance. The ALS VAX-11/780 Assembly Language is intended for use in writing small routines to allow Ada programs direct access to machine-level operations.

The ALS VAX-11/780 Assembly Language will allow access to all instructions and hardware features accessible through the DEC VAX-11 Macro Language. The specific differences between the two assembly languages are described in Tables 20-6, 20-7, and 20-8. For more details on the VAX-11/780 Macro language see the VAX-11 Macro Language Reference Manual published by the Digital Equipment Corporation (2.2).

Table 20-6

FEATURES IN THE DEC VAX-11 MACRO LANGUAGE THAT ARE NOT INCLUDED IN THE ALS VAX-11/780 ASSEMBLER

Feature	Comment
Listing Pseudo Ops	There are no listing options such as nlist or cref, as in DEC VAX-11 Macro.
Macros	There is no Macro capability in the ALS VAX-11/780 Assembler.
General Directives	The only assembler directives that are included in the ALS VAX-11/780 Assembler are: BLKB, BYTE, WORD, LONG, and EQU. (PSECT is described below.)
Literals	Only short literals are supported.
Symbol Table	There is no symbol table printed in the listing.
Complex Expressions	The ALS VAX-11/780 Assembler only supports addition or subtraction of one symbol and a constant in arithmetic expressions.
Psects	Only three program sections are allowed.
Global Symbols	Global symbols cannot be defined.
Current Location Count	There is no access to the current location counter as with the '.' symbol in the DEC VAX-11 Macro.
External Program References	No external program references are allowed in the ALS VAX-11/780 Assembler.
Generic Instructions	The precise opcode must be specified to the ALS VAX-11/780 Assembler.
General Addressing	The precise operand must be specified to the ALS VAX-11/780 Assembler.
Psect Relative Addressing	Only variables and labels in the current psect are accessible in the Psect Relative Addressing mode.

Table 20-7

FEATURES THAT ARE DIFFERENT IN THE ALS VAX-11/780
ASSEMBLER AND THE DEC VAX-11 MACRO LANGUAGE

Feature	Comment
Use of Colon Character	The colon character is used to delimit instruction labels in the DEC VAX-11 Macro language. Colon, the label delimiter is optional in the ALS VAX-11/780 Assembly language.
Line Length	The VAX-11 Macro language allows 80 characters per line and a continuation character. The ALS VAX-11/780 Assembler allows 132 characters per line and no line continuation character.
EQU instead of '='	The VAX-11 Macro language uses '=' to assign symbol values. This directive is implemented with the 'EQU' string in the ALS VAX-11/780 Assembler.

Table 20-8

FEATURES IN THE ALS VAX-11/780 ASSEMBLER
THAT ARE NOT IN THE DEC VAX-11 MACRO LANGUAGE

Feature	Comment
.SUBPROGRAM Directive	This directive is not applicable in the VAX-11 Macro Language.
.SEPARATE Directive	This directive is not applicable in the VAX-11 Macro Language.
.EXTREF Directive	This directive is not applicable in the VAX-11 Macro Language.

20.1.9 Runtime Conventions for Assembly Language Routines

Assembly language routines may be written as either Ada subprogram bodies for which there is an Ada specification or as subunit bodies for which there is an Ada stub. As such, they may be invoked from an Ada subprogram or invoke an Ada subprogram as long as they are written in a manner that is consistent with Ada conventions for calling subprograms and addressing arguments, external data, and external subprograms.

20.1.9.1 Organization into Program Sections

Assembly language code must be organized into programs section (PSECTs) for compatibility with code produced by an Ada compiler. The three types of PSECTs that are used for assembly code are the executable, read-write, and read-only which contain, respectively, executable code, data variables, and constants. An assembly language subprogram or subunit may have no more than one segment of each type. Elaboration code PSECTs, which are produced by Ada compilers, may not be used for assembly code. Data variables may also be placed on the stack as long as the appropriate conventions for the layout of stack frames are observed. Variables for reentrant routines must be placed on the stack rather than in a read-write PSECT.

20.1.9.2 Register Use Conventions

The VAX-11/780 has sixteen 32-bit registers. They are used as follows:

R15 is the program counter (PC) (reserved register),
R14 is the stack pointer (SP) (reserved register),
R13 is the frame pointer (FP) (reserved register),
R12 is the argument pointer (AP) (reserved register), and
R0-R11 are general purpose registers.

20.1.9.3 Starting Point of Executable Code

The starting point of executable code must contain an appropriate entry mask so that the CALLS or CALLG instruction used to invoke the assembly language routine will save the necessary registers. The first executable instruction must follow the entry mask. (See the VAX-11 Architecture Handbook for a detailed treatment of the entry mask format and functions.)

The entry mask must be preceeded by a .SUBPROGRAM and an executable .PSECT directive in the source text for the assembly language routine, in that order. If the routine is a separate subprogram body of an Ada package body, the .SUBPROGRAM directive must be proceeded by a .SEPARATE directive and any required .EXTREF directives. If present, .EXTREF directives always follow the .SEPARATE directive.

20.1.9.4 External References

External references may be to any data declared in the specification of any package used in the linked program. They may also refer to subprograms declared in the visible part of any package specification or to library subprograms. They may not refer to data declared in a package or subprogram body, even if the stub for the routine is located in the same package body or subprogram body. An .EXTREF directive is required for each external reference.

20.1.9.5 Calls

20.1.9.5.1 From Ada Subprograms

As indicated above, the starting point of the executable code is the entry mask required by the CALLS instruction. The CALLS instruction adds the saved registers, the mask, the PSW, and the exception handler table pointer to the stack. The called assembly language routine must then invoke the runtime support library routine RSLSTACK.ADA_STACK to add information to the stack needed to properly handle Ada exceptions. Thus, a JSB instruction to the routine RSLSTACK.ADA_STACK is normally the first executable instruction following the entry mask. An .EXTREF directive is required to address the routine RSLSTACK.ADA_STACK. Once the JSB instruction has been executed, the routine may add any required local stack variables to the stack.

The layout of the stack frame is shown in Figure 20-3.

The return from the assembly routine is accomplished with a RET instruction. The space occupied by the stack frame for the routine is automatically released.

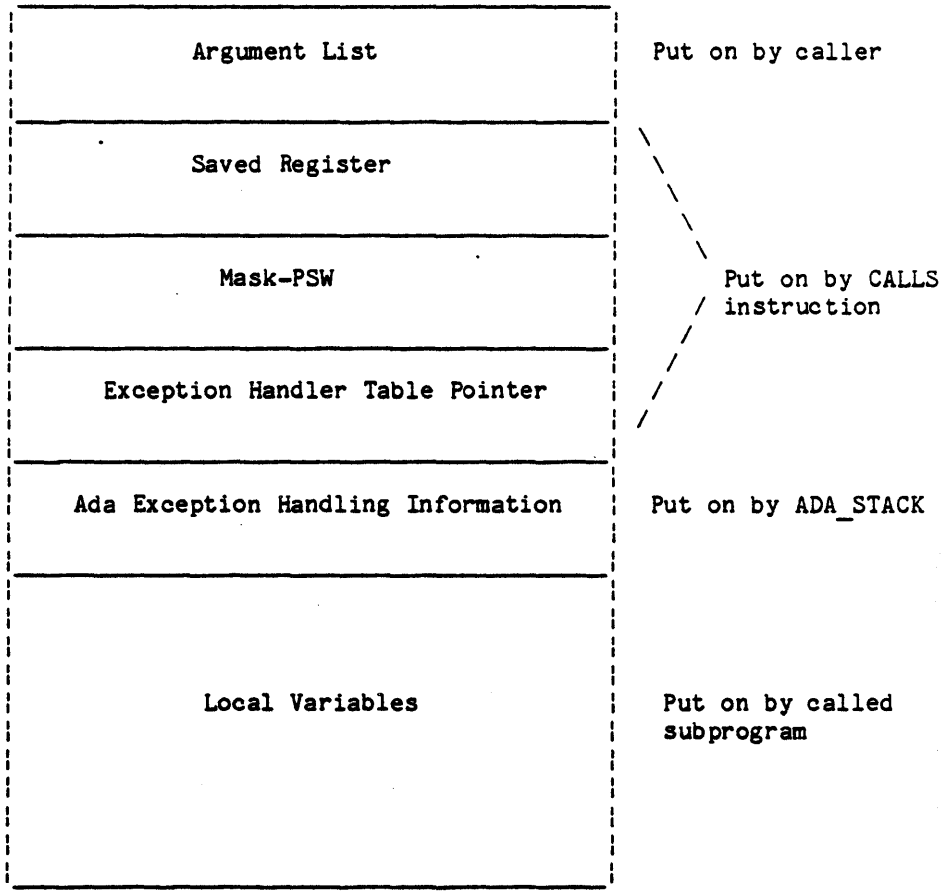


Figure 20-3. Format of the Subprogram Stack Frame

20.1.9.5.2 To Ada Subprograms

The calling sequence used to call an external subprogram is as follows:

Push the arguments on the stack.

Execute a CALLS instruction.

On return, all OUT arguments can be referenced from the argument list and all registers are restored.

20.1.9.6 Arguments

20.1.9.6.1 Scalar Arguments

All scalar arguments are passed by value in a 32-bit longword for all but double float; double float (64-bit) arguments are passed in two longwords in normal DEC D-float format. Numeric values are right justified and sign extended. Values without sign, such as characters, are zero filled.

20.1.9.6.2 Access Arguments

All access arguments are passed by value in a 32-bit longword. If an access value is defined as pointing to an array of an unconstrained type, the access value points to an access type descriptor (ATD). The first component of the ATD is the pointer to the storage for the unconstrained object (on the Heap). The second component is a pointer to the array index descriptor (see array arguments below).

If the formal parameter is an access type pointing to any unconstrained composite type, then an additional longword value is pushed on the stack containing a boolean value indicating whether the actual argument is constrained. This is for constraint checking in the called subprogram. In any event, the actual argument will be a pointer to the ATD in the case of arrays.

20.1.9.6.3 Task Arguments

All task arguments are 32-bit references(pointers) to a record which is the task object.

20.1.9.6.4 Array Arguments

All array arguments are passed by reference using from one to three longwords depending on the type of the formal parameter type according to the following scheme:

a. For unpacked constrained array types, one longword is used for the reference address to the array storage.

b. For unpacked unconstrained array types, two longwords are used: the first longword holds the reference address to array storage and the second longword holds the address of the runtime array index descriptor.

c. For packed constrained array types, two longwords are used; the first longword holds the address of the first unit of the array storage and the second longword holds the bit offset for the start of the array in the first storage unit.

d. For packed unconstrained array types, three longwords are used; the first longword holds the address of the first unit of array storage, the second longword holds the bit offset for the start of the array in the first storage unit, and the third longword holds the address of the runtime array index descriptor.

For these descriptions, "first" means closest to the argument pointer.

The format of the runtime array index descriptor depends on the number of dimensions of the array. It has the following Ada description:

```
type index record is
   record
      lower_bound,
      upper_bound,
      span        : long integer;
   end record;

type index_record_array is
   array ( integer <> ) of index records;

generic
   n : long integer;

type array_desc is
   record
      size,
      virtual_offset : long_integer;
      index_desc     : index_record_array ( 0 .. n-1 );
   end record;

type one_dimension_desc is new array_desc ( 1 )
type two_dimension_desc is new array_desc ( 2 )
```

.
.
.

20.1.9.6.5 Record Arguments

Record arguments are passed by reference to the base of the record.

20.1.9.7 Example of an Assembly Language Subprogram

The sample program below has been designed to illustrate some of the conventions described above. It obtains an arbitrary value from another subprogram that is written in Ada, multiplies it by one of its input arguments, subtracts "100" and returns the result. All local variables are kept on the stack.

```
            .SEPARATE    (PACKAGE_A)
STACK       .EXTREF      RSLSTACK.ADA_STACK
PROGY       .EXTREF      PACKAGE_B.SUBPROG_Y
            .SUBPROGRAM  SUBPROG_X
            .PSECT       EXECUTABLE
            .WORD        ^X0000             ; entry register save mask
            JSB          @#STACK            ; add Ada stack frame
                                            ; information
            PUSHL        ZERO               ; push the constant "0" on the
                                            ; stack
            CALLS        #^S0,@#PROGY       ; call an Ada subprogram
            MOVL         (SP)+,R6           ; create a local variable on the
                                            ; stack
            MULL2        ^B4(AP),R6         ; multiply returned value by an
                                            ; input argument
            SUBL3        HUNDRED,R6,^B4(AP) ; subtract 100 and store as out
                                            ; argument
            RET
            .PSECT       ROSTATIC
ZERO        .LONG        0                  ; define constant "0"
HUNDRED     .BYTE        100                ; define constant of "100"
            .END
```

20.2 Blank.

20.3 Blank.

20.4 Blank.

The removal of Sections 20.2, 20.3 and 20.4 included the following tables and figures:

Tables 20-9 through 20-26,

Figures 20-4 through 20-9.

20.5 ALS MCF Assembler Language.

RESTRICTIVE LEGEND

Material in this chapter has been taken with permission from the VAX-11 MACRO Language Reference Manual published by the Digital Equipment Corporation, Maynard, Mass. Permission for the use of the material has been given for this specific purpose only. Digital Equipment Corporation retains full rights under its copyright of the material. Accordingly, copies of this document shall not be made by any organization or individual outside of the U.S. Government without the prior written permission of the Digital Equipment Corporation.

This section describes the ALS MCF assembly language, including the syntax of assembly code statements and directives, assembler output, and operation of the assembler. Readers should already be familiar with assembly language programming and the Nebula instruction set.

The assembly language is intended to be used for writing small subprograms in order to access capabilities of the target machine that cannot be reached from Ada. It is not intended for any large scale development of software in assembly language. Accordingly, the ALS MCF Assembler does not contain all of the functions provided by the assembler that may normally be supplied with the MCF. A comparison of the assembly language accepted by the ALS MCF Assembler and that specified by the Nebula Instruction Set Architecture (2.1) and The Nebula Assembler (2.2) is provided in Section 20.5.8.

The ALS MCF Assembler translates source programs into object (i.e., binary) code and produces a listing file sent to the predefined internal standard output file and a Container placed in the environment database. The ALS MCF Linker then combines this object information from several assemblies, Ada compilations, and other links to produce an executable program.

An assembly language program consists of a series of source statements representing an Ada library subprogram body or subunit subprogram body. In addition to the body written in assembly language, a subprogram specification, written in Ada, must also exist and must be compiled prior to assembling the body.

The Ada subprogram specification must not include an INTERFACE pragma. The assembly language subprogram body will be called with the same linkage conventions used when calling subprograms written in Ada; it is the assembly language programmer's responsibility to insure that the subprogram body follows these conventions. The Ada entry/exit sequences, and parameter and function-value-return conventions are described in <TBD>. (No checks are performed to make sure the subprogram body matches the subprogram specification, except to see that the subprogram names are identical.)

The following conventions are observed throughout this chapter:

- Brackets ([]) indicate that the enclosed argument is optional.
- Uppercase words and letters, used in formats, indicate that the word or letter should be typed exactly as shown.
- Lowercase words and letters, used in formats, indicate that a word or value of the user's choice is to be substituted.
- Curly braces ({ }) indicate that the enclosed argument(s) may appear zero or more times.

20.5.1 Source Statement Format.

A source program consists of a sequence of source statements. The assembler interprets and processes the statements one by one, generating object code or performing a specific assembly-time process. Only one statement can appear on a line, and a statement cannot extend onto more than one line. A source line can be up to 80 characters long.

Source statements can consist of up to four fields:

- Label field -- allows the programmer to symbolically define a location in a program.
- Operator field -- specifies the action to be performed by the statement; this field can be an instruction opcode or an assembler directive.
- Operand field -- contains the instruction operand(s) or the assembler directive argument(s).
- Comment field -- contains a comment that explains the meaning of the statement; this field does not affect program execution.

The label field, the operator/operand fields combination, and the comment field are all optional. If the operator field is not present the operand field must also be not present. The operand field must conform to the format of the instruction specified by the operator field.

The label field, if present, must begin in Column 1; the operator field, if present, must begin in Column 2 or after. See Section 20.5.7 for a complete summary of the syntax of an assembly language statement.

Blank lines, although legal, have no significance in the source program.

The following sections describe each of the statement fields in detail.

20.5.1.1 Label Field.

A label is a user-defined symbol that identifies a location in the program. The symbol is assigned a value equal to the location counter at the location in the program section in which the label occurs. The user-defined symbol name can be up to 15 characters long and can contain any alphanumeric character. Section 20.5.2.3 describes the rules for forming user-defined symbol names in more detail.

If a statement contains a label, the label must start in the first column of the line.

A label is said to be defined when it appears in a label field. Once a label is defined, it cannot be redefined during the source program. If a label is defined more than once, the assembler displays an error message when the label is defined the second and subsequent times. A space, tab, semicolon, or end of line terminates the label field.

20.5.1.2 Operator Field.

The operator field specifies the action to be performed by the statement. This field can contain either an instruction or an assembler directive.

When the operator is an instruction opcode, the assembler generates the binary code for that instruction; the instruction set and mnemonics are described in the Nebula Instruction Set Architecture (2.1). When the operator is a directive, the assembler performs certain control actions or processing operations during source program assembly; the assembler directives are described in Section 20.5.4.

A space, tab, semicolon or end of line terminates the operator field.

20.5.1.3 Operand Field.

The operand field can contain operands for instructions or arguments for assembler directives.

Operands for instructions specify the locations in memory or the registers that are used by the machine operation including the addressing mode. Section 20.5.3 describes the MCF addressing modes. The operand field for a specific instruction must contain the number of operands required by that instruction. See the Nebula Instruction Set Architecture (2.1). for a description of the instructions and their operands.

Arguments for a directive must meet the format requirements of the directive. Section 20.5.4 describes the directives and the format of their arguments.

If two or more operands are specified, they must be separated by commas. The operand field is terminated by a space, tab, semicolon, or end of line.

20.5.1.4 Comment Field.

The comment field contains text that explains the meaning of the statement. Comments do not affect assembly processing or program execution.

The comment field must be preceded by a semicolon and is terminated by the end of the line. The comment field can contain any ASCII character. A comment can appear on a line by itself, even beginning in Column 1.

20.5.2 Components of Source Statements.

This section describes the components of assembly language statements: the character set, numbers, symbols, and expressions.

20.5.2.1 Character Set.

The following characters can be used in assembly language statements:

- Both upper-case and lower-case letters (A through Z, a through z) are accepted; however, the assembler considers lower and upper case as equivalent representations of the same character set. E.g., "mov" is the same as "MOV".
- The digits 0 through 9.
- The special characters listed in Table 20-27.

The sixteen general registers of the MCF processor can be expressed in a source program only as follows:

Register Name	Processor Register
%0-%15	General register 0 through general register 15

20.5.2.3.2 User-Defined Symbols.

With the exception of the name on the SUBPROGRAM directive, user-defined symbols are only local, appearing in the label field somewhere in the assembly code subprogram and accessible only from within that subprogram.

The following rules govern the creation of user-defined symbols:

- User-defined symbols can only be composed of the 36 alphanumeric characters and the underscore. Any other character terminates the symbol.
- The first character of a symbol must be a letter.
- Two underscores must not be adjacent and the symbol must not begin or end with an underscore.
- The symbol must be no more than 15 characters long and must be unique.

20.5.2.4 Expressions.

There are five kinds of expressions that can be used in an assembly language program:

- number
- symbol
- symbol+number
- symbol-number
- symbol-symbol

If the symbol is relocatable (i.e., defined by appearing as a label other than on an "=" directive), the expression will be relocatable; if absolute (defined by an "=" directive), the expression will be absolute.

20.5.3 Addressing Modes.

This paragraph summarizes the Nebula addressing modes and contains examples of assembly language statements that use these addressing modes. The Nebula Instruction Set Architecture (2.1) describes the addressing modes in detail.

The following pages include examples of each of the addressing modes.

Note: ^B(byte), ^H(halfword), ^W(word), and ^D(doubleword) qualify the size of the operand. The size (width) is used when the operand is evaluated and, in the case of scaled index, used to determine the scaling factor.

Register mode

```
        mov     %3,%4           ; Register 3 => Register 4
```

Short Literal Mode

```
        mov     #1,%4           ; Literal 1 => Register 4
                                ; Literal must be in the range 0..31
```

Literal Mode

```
        mov     #^XFF,%4        ; Literal hexidecimal "FF" =>
                                ; Register 4
```

Indirect Register Mode

```
        mov     @%3^W,%4        ; The word at the address
                                ; within Register 3 =>
                                ; Register 4
```

Register indexed Mode

```
        mov     2(%8)^B,%5      ; The byte at the address
                                ; derived from adding 2 to the contents
                                ; of Register 8 => Register 5
```

Absolute Mode

```
        mov     @(9)^W,%5       ; The word at location 9 in memory =>
                                ; Register 5

        mov     @(place)^H,%5   ; Halfword at location symbolized by
                                ; place => Register 5

        mov     @(place+4)^W,%5 ; Word at location symbolized by the
                                ; expression place+4 => Register 5
```

Short Parameter Mode

```
        mov     ?4,%5           ; Parameter 4 => Register 5
                                ; Parameter number must be in the
                                ; range 1..7
```

Extended Parameter Mode

```
        mov     ?250,%5         ; Parameter 250 => Register 5
                                ; Parameter number must be in the
                                ; range 0..255
```

General Parameter Mode

```
        mov     ?(%8),%6        ; Parameter whose number is contained
                                ; in Register 8 => Register 6

        mov     ?(@%3^B),%6     ; Parameter whose number is contained
                                ; in the byte pointed to by Register 3 =>
                                ; Register 6

        mov     ?(4(%5)^B),%6   ; Parameter whose number is contained
                                ; in the byte pointed to by the result
                                ; of the addition of 4 to the contents
                                ; of Register 5 => Register 6

        mov     ?(place^W),%6   ; Parameter whose number is contained
                                ; in the word at the address symbolized
                                ; by the symbol place => Register 6

        mov     ?(?5),%6        ; Parameter whose number is contained
                                ; in parameter number 5 =>
                                ; Register 6
```

Unscaled Index Mode

Operands of this mode take the form:

Base (Index) width

Where the base is any non-compound instruction mode that yields an address and the index is any other non-compound mode. (Compound modes consist of Scaled Index modes, Unscaled Index modes, and General Parameter modes.)

```
        mov     @%4(%5)^B,%6    ; The byte at the address calculated
                                ; by the addition of the values of
                                ; Register 4 and Register 5 =>
                                ; Register 6

        mov     @%4(@%3^H)^W,%6 ; The word at the address calculated
                                ; by the addition of the halfword
                                ; pointed to by Register 3 and the
                                ; value of Register 4 => Register 6

        mov     @%4(place^H)^W,%6 ; The word at the address calculated
                                  ; by the addition of the halfword
                                  ; located at the memory location
```

```
                    ; symbolically represented by the
                    ; symbol place and the contents of
                    ; Register 4 => Register 6
```

Scaled Index Mode

Operands of this mode take the form:

Base [Index] width

Where the base is any non-compound mode that yields an address and the index is any other non-compound mode.

The address of the Scaled index operand is found by evaluating the index specifier as a signed integer operand, multiplying it by the scale factor, and adding the result to the base specifier. The scale factor is 1, 2, 4, or 8, depending on the number of bytes in the operand specified by the base specifier size field (^B, ^H, ^W, ^D).

The form of the instruction is exactly the same as the Unscaled Index mode if the parenthesis are replaced by square brackets.

```
mov     @%4[@%3^H]^W,%6 ; The word at the address calculated
                        ; by the addition of the halfword pointed
                        ; to by Register 3 scaled by 4 (multiplied
                        ; by 4) and the value of Register 4 =>
                        ; Register 6
```

20.5.4 Assembler Directives.

Table 20-28 lists the assembler directives recognized by the ALS MCF Assembler. This paragraph describes these directives in detail.

Table 20-28

SUMMARY OF ASSEMBLER DIRECTIVES

.BLKB	Space reservation/location control
.BYTE	Data definition
.HWORD	Data definition
.WORD	Data definition
.DWORD	Data definition
.END	Subprogram termination
"="	Symbol definition
.PARM	Parameter line specifier
.SECT	Subprogram sectioning
.ENTRY	Entry point specifier

.SUBPROGRAM	Subprogram initiation
.SEPARATE	Subunit initiation
.EXTREF	External reference definition

20.5.4.1 .BLKB.

The BLKB directive reserves space in the current program section by advancing the program counter by the value of the operand expression.

Format:

```
[label]     .BLKB     expression
```

where:

Label is an optional label that will be assigned the value of the program counter before allocation of the space specified by the directive.

Expression is an expression as described in Section 20.5.2.4 that must evaluate to an absolute value. The program counter will be advanced by the number of bytes specified by the value of the number.

20.5.4.2 .BYTE, .HWORD, .WORD, .DWORD.

These directives allocate one, two, four, or eight bytes, respectively, at the current location and initialize the contents of that location to the value of the operand expression. The expression can evaluate to either a relocatable value or an absolute value.

Format:

```
[label]  .BYTE     expression
[label]  .HWORD    expression
[label]  .WORD     expression
[label]  .DWORD    expression
```

where:

Label is an optional label.

Expression is an expression described in Section 20.5.2.4 whose value will become the contents of the current location in the program section. The line will be flagged with an error if the value of the expression is absolute and too large for the amount of space specified. An error will be generated by the linker if the value is relocatable and found to be too large for the

specified space.

20.5.4.3 .END.

The END directive indicates the end of the assembly code compilation unit. It generates no code and is intended only to give the appearance of completeness to a compilation unit. Any lines after the END directive will be flagged with an error, and will not be otherwise assembled.

Format:

```
.END
```

20.5.4.4 "=".

The "=" directive defines the symbol in the label field and sets it equal to the value of the expression operand. The expression must evaluate to an absolute value (not relocatable). If the expression contains a symbol, that symbol must have been previously defined (in an "=" directive occurring before this one).

Format:

```
label    = expression
```

where:

Label is the symbol being defined.
Expression is an expression as described in Section 20.5.2.4 that must evauluate to an absolute value.

20.5.4.5 .SECT.

The SECT directive separates the assembly code into program sections (psects) having different functions. There can be up to three program sections:

1. Executable code - represents instructions to be executed - data is not intended to be stored in the executable psect;
2. Read/write data - not intended to contain executable code; and
3. Read-only data - not intended to contain executable code. or data which is the target of a store.

The SECT directive defines the beginning of a portion of assembly code, continuing up to the next SECT or END directive. An assembly may have no more than one psect of each of the three types and must have at least an executable psect. All code and data of the subprogram must follow a SECT directive of one of the three types.

The body of the subprogram text always starts at the beginning of the executable psect.

Format:

```
.SECT   storage
```

where storage is one of the following storage area identifiers:

CODE - the executable code of the subprogram body.

DATA - read/write data whose values will be maintained throughout the execution of the whole program, i.e., maintained across calls to this subprogram body.

READ - read-only data maintained as in DATA.

20.5.4.6 .SUBPROGRAM.

The SUBPROGRAM directive must appear exactly once in the assembly and must appear as the first statement unless this subprogram is a subunit in which case the .SUBPROGRAM directive follows the .SEPARATE and .EXTREF directives. It supplies the name of the subprogram and must match the name in the Ada subprogram specification.

Format:

```
.SUBPROGRAM    name
```

where:

Name is the subprogram name and is constructed according to the rules for symbols (see Section 20.5.2.3).

20.5.4.7 .ENTRY.

The .ENTRY directive establishes the entry conditions for a subprogram.

Format:

```
label     .ENTRY     [<arg>] { , <arg> }
```

where:

Label is the name of the entry point,
arg is one or more of the following list, separated by commas:

REG=<unsigned int>	— number of registers
NP= <unsigned int>	— number of parameters
USER	— Clear the UDLE bit
SUPV	— Set the UDLE bit
NOAE	— Clear the enable Arithmetic Exception bit
EAE	— Enable Arithmetic Exception

20.5.4.8 .PARM.

This directive immediately follows an SVC, CALL or CALLU mnemonic and defines the parameters to be passed to the subprogram indicated by the preceeding call statement. Zero to 255 PARM directives may follow a call statement.

Form:

```
.PARM <operand>{,<operand>}
```

where:

<operand> is any operand whose addressing mode was cited in Section 8.5.3.

20.5.4.9 .SEPARATE.

This directive specifies that the assembly program is to be a subunit subprogram of an Ada library unit package body. The assembly program will take on the same context as the parent package body. The package body which specifies this subprogram as a separate unit must be compiled before the assembly program.

Format:

.SEPARATE (package name)

where:

The package name is the name of an Ada library unit package of which the assembly program is to be a subunit. The package name is constructed according to the rules for symbols (see Section 20.5.2.3).

20.5.4.10 .EXTREF.

This directive facilitates the referencing of variables declared globally in Ada package specifications. The ".EXTREF" directive may only appear in an assembly program that is a subunit (i.e., a program with the ".SEPARATE" directive).

Format:

Label .EXTREF package_name.name

where:

1. The label, package_name and name are constructed according to the rules for symbols (see Section 20.5.2.3)

2. The label spelling is used in the symbol portion of an assembly instruction operand. The external reference will then be to a variable or subprogram specified by a package_name.name.

20.5.5 Assembler Output.

This paragraph describes the outputs from an execution of the ALS MCF Assembler: the source listing, error message, and machine text information.

20.5.5.1 Machine Text.

Machine text is what has been commonly referred to as "object module" or "relocatable binary" in other systems. In the Ada system there is no separate object module representation for the translator-generated executable code. Instead, translators, like the ALS MCF Assembler, put their generated code into a Container in a program library.

20.5.5.2 Listing.

At the option of the user a listing consisting of the source code side by side with the assembled machine text in hexadecimal can be produced by the assembler (see Section 20.5.6). The source line number and the hexadecimal location relative to the start of the psect are also listed. The total number of error messages is displayed at the end of the listing. The listing is produced in the standard output file.

Figure 20-9 shows some sample lines of the assembly listing. Note that the contents field in the listing is read left to right, i.e., the leftmost byte corresponds to the address given in the location field; the next byte to the right corresponds to that address plus one. Figure 20-10 shows the Ada library subprogram specification corresponding to the subprogram body in Figure 20-9.

The listing is sent to the standard output file.

```
Contents          Location    Line    Source.

                  00000000:     1             .SUBPGROGRAM Sample
                  00000000:     2             .SECT        CODE
0006              00000000:     3     sub1:   .ENTRY       REG=6,NP=0
                  00000002:     4     ; This subprogram is a sample for
                  00000002:     5     ; the A-spec
                  00000002:     6     ;
45 24             00000002:     7             CLR          %4
40 22 23          00000004:     8     here:   MOV          %2,%3
67                00000007:     9             RET
30 3A22E3 3A25E3  00000008:    10             ADD @%3[%2]^W,@%3[%5]^W
                  0000000F:    11             .SECT        READ
30FF              0000000F:    12     var:    .BYTE        ^X30,^XFF
                  00000011:    13             .END
```

Figure 20-9 Sample Assembly Listing

```
PROCEDURE    SAMPLE;
```

Figure 20-10 Matching Ada Subprogram Specification

20.5.5.3 Diagnostic Messages.

Diagnostic messages are produced for all syntactic errors detected by the assembler. The message(s) for any particular line of source appears in the listing immediately following the source line. If no listing is requested, diagnostic messages appear in the message output file. The message number and text will be the same as the corresponding diagnostic generated for a listing. An example of a diagnostic message sent to the user through the message output file is as follows:

```
***ERROR ASMMCF 53403 AT SOURCE line Number  1  Subprogram  Directive
Required
```

Messages indicating diagnostics in the assembler command are sent to the user via the message output file. Assembler command diagnostics are always sent to the user via message output regardless of the source option specification. An example of an Assembler Command Diagnostic Message is as follows:

```
***ERROR ASMMCF 56101 Command Diagnostic User Specified  Source  File
not found
```

If diagnostics of severity ERROR or FATAL are produced, a useable Container is not produced.

The diagnostic messages produced by the ALS MCF Assembler are summarized in Appendix 80.

20.5.5.4 Summary Message.

At the completion of the assembly process, a summary message is sent to the user via the message output file. This summary message indicates the completion of the assembly process and the number of diagnostic conditions detected for each severity level. An example of a summary message is as follows:

```
ASMMCF Processing Complete
Number of Diagnostics Generated:
     2 of Severity level WARNING
     3 of Severity level ERROR
     0 of Severity level SYSTEM
     0 of Severity level FATAL
```

20.5.6 Invoking the Assembler.

The assembler is invoked with the command:

```
ASMMCF (source, prog_lib [,OPT=>option_list])
```

where:

source	the name of the source file
prog_lib	the name of the program library into which the source is assembled. The name of the Container produced in the program library is the name on the SUBPROGRAM directive.
option_list	[NO_]SOURCE specifies whether to produce a source listing or not. The default is SOURCE.
	[NO_]CONTAINER_GENERATION specifies whether a Container is to be produced if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. If NO_CONTAINER_GENERATION is in effect, listings cannot be regenerated using the Display Tools CPCI. The default is CONTAINER_GENERATION.

20.5.7 Assembly Language Syntax.

This paragraph gives the formal syntax of the ALS MCF assembly language. The notation used here is a modified form of Backus-Naur Form (BNF). Angle brackets, "<" and ">", to enclose a syntactic unit which is defined to the left of a "::=". The symbol "::=" is read as "is defined as"; the vertical bar, "|", is read as "or"; anything enclosed in square brackets, "[" and "]", is optional; the curly braces "{ }" indicate that the enclosed unit can appear zero or more times; and two adjacent units (possibly on separate lines) indicate that the first must be followed by the second without intervening spaces.

For example, the following alternative for

```
<label>:

<letter> {<more label>}

<more label> ::= [_] <alpha numeric>
```

indicates that label consists of at least one letter with optional alpha numeric characters following. Note that the underscore is positioned such that it may not appear at the beginning or end of a label.

The full syntax of the assembly language follows:

```
<assembly code subprogram>  ::=
              {[<space>] <comment> <eol>}
              <program heading>
              {[<space>] <comment> <eol>}
              <program section>      -- one executable psect is required
             {<program section>}     -- only one of each psect type is
                                     -- allowed
              <END directive>{<comment>}
<program heading>           ::=  <subprogram directive>
                                 | <subunit heading>
                                 <subprogram directive>
<program section>           ::=
              <SECT directive>
              {<assembly code line>}
<subunit heading>           ::=
   <space>.SEPARATE <space> (<package name>) [<space>]
          [<comment>] <eol>
   {<external directive>}
<external directive         ::=
   <label><space>.EXTREF
      <space><external name>
      [<space>][<comment>]
               <eol>
<SUBPROGRAM directive>      ::=
   <space>.SUBPROGRAM <space><label>{[<space>] [<comment>] <eol>}
<END directive>             ::=  <space> .END[<space>] [<comment>] <eol>
<SECT directive>           ::=  <space> .SECT <space><SECT attribute>
                                {[<space>] [<comment>] <eol>}
<assembly code line>        ::=  <regular code line>
                                 | <line symbol><symboled line>
<regular code line>         ::=  [<instruction>]
                                {[<space>] [<comment>] <eol>}
                                | <space> .PARM <space> <oper>
                                {,<oper>}[<space>][<comment>] <eol>
<symboled line>             ::= <regular code line>
                                 | <equ line>
                                 | <entry>
<equ line>                  ::= <space> = <abs expression> [<space>]
                                [<comment>] <eol>
<entry>                     ::= <space> .ENTRY [<space><arg> { , <arg> }]
                               [<space>].[<comment>] <eol>
<line symbol>               ::= <label>:
                                -- starting in column 1 of source line
<comment>                   ::=  ;{<character>}
<instruction>               ::=  <space> <executable instruction>
                                 |  <space> <directive>
<external name>             ::= <package name>.<label>
<package name>              ::= <label>
<arg>        ::=  NP= <unsigned int>     -- number of parms
                  | REG= <unsigned int>   -- number of regs
                  | EAE                   -- enable arith excepts
```

```
                | NOAE                    -- no arith excepts
                | USER                    -- exceptions by user
                | SUPV                    -- Exceptions to supv

<directive>  ::= .BLKB <space><abs expression>
                | .BYTE <space><relocatable expression>
                | .HWORD <space><relocatable expression>
                | .WORD <space><relocatable expression>
                | .DWORD <space><relocatable expression>

<SECT attribute>  ::= READ | DATA | CODE
<space>  ::= <space character> {<space character>}
<space character>  ::= blank | tab
<executable instruction>  ::=
          ABS           <space> <oper> , <oper>
        | ABSF          <space> <oper> , <oper>
        | ADD     <space> <oper> , <oper> [ , <oper>]
        | ADDC    <space> <oper> , <oper> , <oper>
        | ADDF    <space> <oper> , <oper> [ , <oper>]
        | ADDU    <space> <oper> , <oper> , <oper>
        | AND     <space> <oper> , <oper> [ , <oper>]
        | BCC     <space> <label>
        | BCS     <space> <label>
        | BEQL    <space> <label>
        | BGEQ    <space> <label>
        | BGTR    <space> <label>
        | BLEQ    <space> <label>
        | BLSS    <space> <label>
        | BNEQ    <space> <label>
        | BR      <space> <label>
        | BREAK
        | BTC     <space> <label>
        | BTS     <space> <label>
        | CALL    <space> <addr> [, <unsigned int>]
        | CALLU   <space> <addr> [, <unsigned int>]
        | CASE    <space> <oper> ,<oper>, <unsigned int>
        | CLR     <space> <oper>
        | CLRBIT  <space> <oper> , <addr>
        | CLRF    <space> <oper>
        | CMP     <space> <oper> , <oper>
        | CMPBK   <space> <oper> , <addr> , <addr>
        | CMPF    <space> <oper> , <oper>
        | CMPS    <space> <oper> , <oper> , <oper>
        | CMPU    <space> <oper> , <oper>
        | CMPWB   <space> <oper> , <oper> , <oper>
        | COB     <space> <oper> , <oper>
        | DBGEQ   <space> <oper> , <oper> , <label>
        | DBGTR   <space> <oper> , <oper> , <label>
        | DEC     <space> <oper>
        | DIV     <space> <oper> , <oper> [ , <oper>]
        | DIVF    <space> <oper> , <oper> [ , <oper>]
        | DIVFIX  <space> <oper> , <oper> , <oper> , <oper>
        | DIVU    <space> <oper> , <oper> , <oper>
```

```
| ECODE   <space> <oper>
| EDIV    <space> <oper> , <oper> , <oper> , <oper>
| EMUL    <space> <oper> , <oper> , <oper>
| EQL     <space> <oper>
| ERET    <space> <oper>
| ERP     <space> <oper>
| EXCEPT  <space> <addr>
| EXCH    <space> <oper> , <oper>
| FIX     <space> <oper> , <oper>
| FLOAT   <space> <oper> , <oper>
| GEQ     <space> <oper>
| GTR     <space> <oper>
| IBLEQ   <space> <oper> , <oper> , <label>
| IBLSS   <space> <oper> , <oper> , <label>
| INC     <space> <oper>
| INC2    <space> <oper>
| INC4    <space> <oper>
| INC8    <space> <oper>
| INVBIT  <space> <oper> , <addr>
| JSR     <space> <addr>
| JUMP    <space> <label>
| LBF     <space> <oper> , <oper> , <addr> , <oper>
| LBFS    <space> <oper> , <oper> , <addr> , <oper>
| LEQ     <space> <oper>
| LOOP    <space> <oper> , <oper> , <oper> , <label>
| LPSW    <space> <oper>
| LSH     <space> <oper> , <oper> , <oper>
| LSS     <space> <oper>
| LTASK   <space> <addr>
| MAP     <space> <oper> , <oper> , <oper> , <oper>
| MOD     <space> <oper> , <oper> , <oper>
| MOV     <space> <oper> , <oper>
| MOVA    <space> <addr> , <oper>
| MOVBK   <space> <oper> , <addr> , <addr>
| MOVF    <space> <oper> , <oper>
| MOVL    <space> <oper> , <oper>
| MOVM    <space> <oper> , <oper> , <addr>
| MOVTR   <space> <addr> , <oper> , <addr> , <addr>
| MUL     <space> <oper> , <oper> [ , <oper>]
| MULF    <space> <oper> , <oper> [ , <oper>]
| MULFIX  <space> <oper> , <oper> , <oper> , <oper>
| MULU    <space> <oper> , <oper> , <oper>
| NEG     <space> <oper> [ , <oper>]
| NEGF    <space> <oper> [ , <oper>]
| NEQ     <space> <oper>
| NOP
| NOT     <space> <oper> [ , <oper>]
| OR      <space> <oper> , <oper> [ , <oper>]
| PCHECK  <space> <oper>
| PINIT   <space> <addr> , <oper>
| POP     <space> <oper>
| PRAISE  <space> <oper> , <oper>
| PSTART  <space> <oper>
```

```
| PUSH    <space> <oper>
| RAISE   <space> <oper>
| RANGE   <space> <oper> , <oper> , <oper>
| REM     <space> <oper> , <oper> , <oper>
| REMF    <space> <oper> , <oper> , <oper>
| REPENT  <space> <oper> , <oper> , <oper>
| RESET
| RET
| RNDI    <space> <oper> , <oper>
| ROT     <space> <oper> , <oper> , <oper>
| RSR
| SBF     <space> <oper> , <oper> , <oper> , <addr>
| SCALE   <space> <oper> , <oper> , <oper>
| SCANB   <space> <addr> , <addr> , <oper> , <oper>
| SCHECK  <space> <oper>
| SETBIT  <space> <oper> , <addr>
| SETCC   <space> <oper>
| SETSEG  <space> <addr> , <addr>
| SIZE    <space> <oper> , <oper>
| SPSW    <space> <oper>
| SQRTF   <space> <oper> , <oper>
| STASK   <space> <addr>
| STOREH  <space> <oper>
| SUB     <space> <oper> , <oper> [ , <oper>]
| SUBC    <space> <oper> , <oper> , <oper>
| SUBF    <space> <oper> , <oper> [ , <oper>]
| SUBU    <space> <oper> , <oper> , <oper>
| SVC     <space> <oper>  [ , <unsigned int>]
| TEST    <space> <oper>
| TINIT   <space> <addr> , <oper>
| TRAISE  <space> <oper> , <oper>
| TSTART  <space> <oper>
| TSTBIT  <space> <oper> , <addr>
| WAIT
| WINDOW  <space> <literal value>
| XOR     <space> <oper> , <oper> , <oper>
<addr>              ::= <unscaled index>
                      | <scaled index>
                      | <memory operand>
                      | <general parm>
                      | <parameter>
<oper>              ::= <scaled index>
                      | <unscaled index>
                      | <general parm>
                      | <simple>
                      | <memory operand>
<general parm>      ::= ? (<memory operand>)
                      | ? (<simple>)
<scaled index>      ::= <memory operand> <left bracket> <simple>
                        <right bracket> <width>
                      | <memory operand> <left bracket>
                        <memory operand>
                        <right bracket> <width>
```

```
<unscaled index>     ::= <memory operand> ( <simple> ) <width>
                       | <memory operand> ( <memory operand> )
                         <width>
<simple>             ::= <general register>
                       | <literal value>
                       | <parameter>
<memory operand>     ::= <indirect reg>
                       | <register index>
                       | <absolute>
<register index>:::=  <numeric value> (<general register>)<width>
<indirect reg>       ::=@<general register><width>
<general register>   ::=    %1  | %2  | %3
                       |    %4  | %5  | %6  | %7
                       |    %8  | %9  | %10 | %11
                       |    %12 | %13 | %14 | %15
<absolute>           ::= @ ( <abs expression> ) <width>
                       |     <simple expression> <width>
<literal value>      ::=  #<abs expression>
<parameter>          ::=  ? <unsigned int>
<width>              ::=  ^B | ^H | ^W | ^D
<simple expression>  ::=  <label>
                       |  <label>+<unsigned int>
                       |  <label>-<unsigned int>
<relocatable expression> ::= <numeric value>
                           | <symbol>
                           | <symbol>+<unsigned int>
                           | <symbol>-<unsigned int>
                           | <symbol>-<symbol>
<abs expression>     ::=  <numeric value>
                       |  <symbolic constant>
                       |  <symbolic constant>+<numeric value>
                       |  <symbolic constant>-<numeric value>
<symbol>             ::=  <label>
                       |  <symbolic constant>
<symbolic constant>  ::=  <label>     -- assigned by an equ line
<unsigned int>       ::=  <digit> {<digit>}
<numeric value>      ::=  [-]<digit>{<digit>}
                       |  ^X<hex digit> {<hex digit>}
                       |  ^O<octal digit> {<octal digit>}
<label>              ::=  <letter> {<more label>}
<more label>         ::=  [_] <alphanumeric>
<left bracket>       ::= [
<right bracket>      ::= ]
<eol>::= end of line character {end of line character}
<letter>             ::= 'A'..'Z'
                       | 'a'..'z'
<digit>              ::= '0'..'9'
<alphanumeric>       ::= <letter>
                         <digit>
<hex digit>          ::= <digit>
                       | 'A'..'F'
<octal digit>        ::= '0'..'7'
```

20.5.8 Assembly Language Comparison.

The differences between the ALS MCF Assembly Language and the Nebula Assembly Language are generally a result of the different intended purposes of the two languages. The Nebula Assembly Language provides features necessary for efficient, large assembly language program development and maintenance. The ALS MCF Assembly Language is intended for use in writing small routines to allow Ada programs direct access to machine-level operations.

The ALS MCF Assembly Language will allow access to all instructions and hardware features accessible through the Nebula Assembly Language. The specific differences between the two assembly languages are described in Tables 20-29, 20-30, and 20-31. For more details on the Nebula Assembly Language see The Nebula Assembler (2.2).

Table 20-29

FEATURES IN THE NEBULA ASSEMBLY LANGUAGE THAT ARE NOT INCLUDED IN THE ALS MCF ASSEMBLER

Feature	Comment
Type Propagation	No type propagation is performed by the ALS MCF Assembler.
ASCII character or Floating Point Constants	ASCII and Floating point constants are not supported.
Type Specifications	Type specifications are not permitted on operands.
.GLOBAL	The .GLOBAL directive is not supported.
.FLOAT, .DOUBLE	Floating point directives are not supported.
.ASCII, .ASCIZ	Character constant directives are not supported.
.BLKH, .BLKW, BLKDW, .BLKF, .BLKD	These storage directives are not supported.
.ALIGN	The .ALIGN directive is not supported.
.RADIX	The .RADIX directive is not supported.
.EXTERN	The .EXTERN directive is not supported.
Access of the location counter	No access to the location counter is allowed.
Explicit Radix Specifiers	No explicit radix specifiers for binary or decimal are provided in the ALS MCF Assembler.

Table 20-30

FEATURES THAT ARE RESTRICTED IN THE ALS MCF ASSEMBLER AS COMPARED TO THE NEBULA ASSEMBLER

Feature	Comment
Size Propagation	No size propagation and size checking is done in the ALS MCF Assembler and therefore no type specifiers may be used. Only the size specifiers for BYTE, HALFWORD, WORD, and DOUBLE WORD (^B, ^H, ^W, ^D) are recognized.
Default Size	There will be no default size.
Literals	Only constants may appear in literals.
.END	No expression defining the starting address may be specified.
.SECT	The only legal operands to this pseudo-op are CODE, DATA, and READ.
Label Length	Labels may only consist of 15 characters in the ALS MCF Assembler. The colon terminating the label is optional.
Expressions	Expressions may consist of only a constant or a label plus or minus a constant.
Symbols	Maximum length of a symbol is 15 characters. Symbols may not begin or end with a special character.
=	The "=" may only equate a label to an expression.

Table 20-30 (cont.)

FEATURES THAT ARE RESTRICTED IN THE ALS MCF ASSEMBLER AS COMPARED TO THE NEBULA ASSEMBLER

Feature	Comment
CASE	The CASE mnemonic has 1 or more HWORD directives following it. The number of HWORD expected by the Assembler is specified as the third operand of the instruction.
CALL, CALLU, SVC	The CALL, CALLU, and SVC instructions have as their second operand a number specifying how many parameters will appear in the following PARM directives following the instruction.

Table 20-31

FEATURES IN THE ALS MCF ASSEMBLER
THAT ARE NOT FOUND IN THE NEBULA ASSEMBLER

Feature	Comment
.SEPARATE Directive	This directive is not applicable in the Nebula Assembler.
.EXTREF Directive	This directive is not applicable in the Nebula Assembler.

APPENDIX 30

30. ADA LANGUAGE SYSTEM LINKERS

The Ada Language System includes the following linkers:

a. ALS VAX-11/780 Linker, and

b. ALS MCF Linker.

Descriptions of these linkers are provided on the following pages.

30.1 Using The ALS VAX-11/780 Linker.

This section describes how to use the VAX-11/780 Linker in the Ada Language System (ALS) to combine compilation units for execution on the VAX-11/780 target environment.

The purposes of linking are:

a. To combine many separately translated units into a single unit, matching references to externally-defined names with their proper definition;

b. To include the text of system runtime routines into the new linked unit (Calls to system runtime routines are automatically inserted by the compiler into the translated machine text; the routines are predefined and need not be provided by the user.);

c. To designate one subprogram as the "main subprogram", i.e., the one that receives initial control when the program begins execution;

d. To allocate storage space to all of the user-supplied code and data, and to the necessary system runtime routines;

e. To check all units for compliance with the Ada language rules regarding the order of compilation (for example, to ensure that a library unit is compiled before any of its subunits);

f. To decide in what order library units are to be elaborated, and to ensure that a legal order of elaboration exists, following the rules of the Ada language; and

g. To produce listings that show the units that have been linked together, and describe how storage has been allocated.

When invoking the linker, the user must designate the Containers that are to be linked together. These Containers may have been generated by the compiler, the ALS VAX-11/780 Assembler, or may have been created by a previous invocation of the ALS VAX-11/780 Linker (see Section 30.1.2). All of the Containers must represent compilation units intended for execution on the VAX-11/780 target environment, and must reside in the same program library.

If the user wants to modify an already linked Container, he may do so by naming the Container to be modified and the new Containers to be used as replacements in the UNITLIST. The replacement Containers should be named first because the linker always uses the first definition of any compilation unit.

The user may also designate the library subprogram that is to be the "main" subprogram, i.e., the subprogram that initially receives control at execution time, or may designate that there is no main program because the output of the link is not intended for execution.

The linker will produce as output a Container. If the Containers designated by the user contain all the elements that constitute a complete program, then a complete program is produced. If some elements have been omitted, the output is an incomplete program (see Section 30.1.2). The Container produced by the linker can be supplied to the exporter (via the environment database), so that it can be executed on a VAX-11/780 target environment, or it can be used as input to a future link.

30.1.1 Invoking the Linker.

To invoke the ALS VAX-11/780 target linker the user must supply a command of the following form to the ALS Command Language Processor:

LNKVAX main_name prog_lib output_container [UNITLIST=>file_name] [OPT=>option_list]

The following paragraphs describe the linker arguments. Those arguments enclosed in brackets ('[]') are optional.

a. main_name: Either the name of the main subprogram or the keyword "NULL". If NULL is used then there is to be no main subprogram. In this case, the output Container is not eligible for exportation and execution. (It may be used as input to a future link as described in Section 30.1.2.)

   If a main subprogram is designated then it must be a library function or a library procedure. The IN string parameter, if present, and return string, if needed, must be of type STANDARD.STRING and must be unconstrained. If the main subprogram is a function, the return value is reflected in the RSTRING substitutor. (The return value from the main function must be a CHARACTER string.) Normally, the Containers that will be linked are the ones in the program library whose compilation units are referenced directly or indirectly by the WITH and SEPARATE clauses of the main subprogram. (See Section 30.1.1.1 for the method of altering this procedure.)

b. prog_lib: Name of the program library which contains all of the Containers to be linked and in which the new output Container will be placed.

c. output_Container: Name for the new linked output Container. This name must follow the Ada rules for a compilation unit. It must be distinct from names of library units and linked containers already in the program library.

d. UNITLIST filename: Normally, the main subprogram name provides a starting point for the linker search process that finds the Containers to be linked. This argument is used to provide additional starting points, or to enumerate specifically the units to be linked. The file name designated here must be the pathname of a database file containing ASCII text. If the main_name is null a UNITLIST is required.

   The file should contain a list of units in the program library that are to be linked one unit per line. If the search option is off, exactly the units listed will be linked; otherwise, these units and all units referenced by the designated units will be linked.

e. option_list: The form of the option list is:

(option_name {,option_name})

The parentheses may be omitted if there is only one option. The options are discussed in detail in Section 30.1.1.1.

30.1.1.1 Options.

This section describes the options that may be specified to the linker. Each option may be specified as shown, or may be preceded by the three characters NO_, which indicate the opposite option. (For example, SYMBOLS and NO_SYMBOLS.) It is an error to specify both an option and its opposite.

SYMBOLS	Provide a Symbol Definition Listing, if a Container is produced. Default: NO_SYMBOLS
LOCAL_SYMBOLS	If a Symbol Definition Listing is produced, include names local to library package bodies as well as externally visible names. Default: NO_LOCAL_SYMBOLS, means to include only names that are externally visible.
UNITS	Provide a Units Listing if a container is produced. Default: UNITS.
SEARCH	Linker will automatically follow the WITH and SEPARATE clauses to find all of the units in the program library that are referenced from the designated starting point units. All of the referenced units will be linked into the output Container. If NO_SEARCH is specified, only the starting point Containers and the runtime routines referenced by the Containers are included in the linked output. The SEARCH procedure will take the first occurence of a unit. Default: SEARCH.

30.1.1.2 Units Listing.

The Units Listing is a list of all the compilation units that are represented in the linked Container. For each compilation unit, the following information is provided:

a. Name.

b. Size, in bytes, including all code and statically-allocated data.

c. Compilation date.

d. The tool (compiler, assembler, linker) that created the Container containing this unit.

e. Unit Status: L for Library Unit, S for subunit.

f. Diagnostic Status: R for recompilation advised, O for out-of-date revision.

If the unit has status R, a recompilation advisory message has been issued by the compiler, but the unit has not been recompiled (see Section 30.1.4). If the unit has status O, it is not the most recent revision of that unit.

The UNITS option controls the production of Units Listings. An example of a Units Listing is shown in Figure 30-1.

```
MYPROG      PROGRAM_LIBRARY=DEMO_PROG      LNKVAX      ALS VAX 11/780 LINKER V 1.0      UNITS LIST      MAY 27, 1981      PAGE 1

UNIT NAME          STORAGE UNITS    CREATION DATE            CREATOR              STATUS

MAIN.SPEC                  0        SEP 16, 1980 9:15        ADA V0.1               L
MAIN.BODY                 38        SEP 17, 1980 4:00        ADA V0.1               L
MAIN.SUB1                 50        SEP 19, 1980 6:12        ADA V0.1               S
FAST_HASH.SPEC             0        SEP 17, 1980 2:36        ADA V0.1               L
FAST_HASH.BODY            16        SEP 18, 1980 3:45        ADA ASSEMBLER V1.0     L
PACK1.SPEC                18        SEP 16, 1980 11:40       ADA V0.1               L
PACK1.BODY                68        SEP 19, 1980 9:30        ADA V0.1               L
PACK1.SUB1                24        SEP 18, 1980 12:03       ADA V0.1               SR
```

Figure 30-1. Units Listing

30.1.1.3 Symbol Definition Listing.

The Symbol Definition Listing describes the allocation of objects and compilation units. For the VAX-11/780 target, the listing is divided into four parts showing the allocation of:

a. Elaboration code for library units,

b. Executable code for bodies,

c. Static read/write data, and

d. Static read-only data.

An example Symbol Definition Listing is shown in Figure 30-2. The option, SYMBOL, indicates whether a Symbol Definition Listing should be produced.

The first two parts of the listing show the allocation of compilation units. For each library unit or subunit, the following information is provided:

a. The compilation unit name;

b. The location of the instructions representing the body and elaboration of the unit (hexadecimal origin and length in storage units);

c. The location of entrances to externally visible subprograms (hexadecimal address); and

d. If the LOCAL_SYMBOLS option is specified, the location of the subprogram entrances for subprograms local to package bodies (hexadecimal address).

The last two parts of the listing show the allocation of data in library units. If the LOCAL_SYMBOLS option is provided, data declared in the package body is included. For each object, the following information is provided:

a. The object name, and

b. The location of the data (hexadecimal origin and length).

```
MYPROG      PROGRAM_LIBRARY=DEMO_PROG    LNKVAX     ALS VAX 11/780 LINKER V 1.0     SYMBOL DEFINITION     MAY 27, 1981     PAGE 1

    ELABORATION INSTRUCTIONS                 START       LENGTH

        PACK1.BODY                           00000000    0000000A

    BODY INSTRUCTIONS

        MAIN.SPEC                            00000000    00000000
        MAIN.BODY                            00000000    00000026
        PACK1.SPEC                           00000026    00000000
        PACK1.BODY                           00000026    00000040
        PACK1.ACCESS_GLOBAL                  00000030    00000036
       *PACK1.LOC                             00000025    0000000A
        FAST_HASH.SPEC                       00000065    00000000
        FAST_HASH.BODY                       00000066    00000010
        PACK1.SUB1                           00000076    00000018
        MAIN.SUB1                            0000008E    00000032

    READ/WRITE DATA

     PACK1.GLOBAL                            00000000    00000008
    *PACK1.TEMP                              00000008    00000004

    READ/ONLY DATA

     PACK1.SIZ                               00000000    00000004
     PACK1.MAX                               00000004    00000006

*Indicates a local subprogram or data item.
```

Figure 30-2. Symbol Definition Listing

30.1.1.4 Link Summary Listing.

A Link Summary Listing is produced each time the linker is invoked. The listing includes:

a. Whether an output Container is produced.

b. The total size of instruction (hexadecimal bytes),

c. The total size of statically-allocated data (hexadecimal bytes),

d. The number of compilation units linked,

e. The name and hexadecimal address of the entrance to the main subprogram, and

f. A summary of diagnostic messages.

An example of a Link Summary Listing is shown in Figure 30-3.

```
MYPROG      PROGRAM_LIBRARY=DEMO_PROG      LNKVAX      ALS VAX 11/780 LINKER V 1.0      LINK SUMMARY      MAY 27, 1981      PAGE 1


A LINKED CONTAINER NAMED MY_PROG HAS BEEN PRODUCED

INSTRUCTIONS    192    BYTES
DATA             22    BYTES

NUMBER OF COMPILATION UNITS:  8
      MAIN SUBPROGRAM NAME:   MAIN
                ENTRY POINT:  00000000

DIAGNOSTIC SUMMARY

   1) PACK1.SUB1 WAS NOT COMPILED AFTER PACK1.BODY
```

Figure 30-3. Link Summary Listing

30.1.2 Preparing Incomplete Programs.

It is possible to use the linker to prepare incomplete programs for execution and test. An "incomplete program" consists of one or more Ada compilation units which together do not constitute a complete Ada program because:

a. Some library subprogram or library packages bodies are omitted; and/or

b. Some separate subprogram or package bodies are omitted.

Incomplete programs may be created and then exported for execution on the target environment. Through the use of this facility, it is possible for programmers to test portions of large programs before all the units of the complete program are written, or to test portions of programs, then to integrate the tested portions for further testing.

An omitted subprogram body or package body may be stubbed, using the STUBGEN tool (see Appendix 70). STUBGEN creates a text file which is a legal Ada package body or subprogram body, in conformance with the declaration or specification. This stub may be compiled, producing a Container in the program library for the omitted body. The Container may be linked, exported, and executed in the normal manner.

When a stubbed subprogram is called, it returns legal values for all OUT and INOUT parameters, and in the case of functions, it produces a legal return value. Optionally, the stub prints the name of the subprogram which was called.

It is possible that a body might not be available in the program library, even in stub form. It is expected that an incomplete program will not attempt to call subprograms whose bodies are not present. A call of this type will raise the runtime exception, SYSTEM.UNRESOLVED_REFERENCE, which will abort execution if the incomplete program does not contain an appropriate exception handler.

If a unit body referenced by another part of the program cannot be found in the program library the linker will warn the user of each missing subprogram body or package body (see Section 30.1.5) and will create a Container representing an incomplete program. This Container may be exported and executed (see Appendix 40).

A Container may be resubmitted as input to a future link. For example, when a programmer has tested part of his program and wishes to add more units, or when many programmers have tested parts of programs and wish to integrate the pieces for further testing, it is possible to input to the linker one or more linker-created Containers, possibly along with Containers created by the compiler or assembler. In this case, the linker-created Container represents a short-hand notation for all of the compilation units of which it is composed, indicating that all these units are to be included in the link. (Note that the original Containers

representing the compilation units may need to be accessed by the linker.)

The following example illustrates the use of the linker to create incomplete programs:

a. Assume that subprogram ENTER has two separate subunit procedures CHECK and MODIFY. The application is a database update; CHECK verifies the data and MODIFY updates the database. ENTER is the driver that calls CHECK, then MODIFY. The subprogram ENTER and CHECK are to be tested. When they are working, MODIFY will be added and the whole program will be tested.

b. The user will compile ENTER, then CHECK.

   Alternative 1: He will generate a stub for MODIFY, then compile it. He will link the three Containers together, export, and test them. When they are working, the real MODIFY is compiled, producing a new revision of the Container. The three Containers, including the most recent MODIFY, are linked, exported, and tested.

   Alternative 2: He will link ENTER and CHECK into an incomplete program named TEST1. During this link, the linker will warn the user of the omission of MODIFY. TEST1 is exported and tested; an exception will be raised when MODIFY is called. When TEST1 is working, MODIFY is compiled, and TEST1 and MODIFY are linked to form a complete program.

30.1.3 Allocation of Storage.

The VAX-11/780 Linker allocates storage in four groups, each group starting on a virtual-memory page boundary. The allocation is designed to use VAX-11/780 hardware facilities to provide maximum protection against modification of instructions and read-only data.

The four groups are:

a. Executable code that elaborates library units;

b. Executable code that represents the bodies of subprograms and packages;

c. Read-write static data, such as that declared in library units; and

d. Read-only static data, including literals.

Because of the large virtual address space of the VAX-11/780, there is no requirement for an overlaying capability, or for allocating instructions to a particular location in physical or virtual space.

30.1.4 Blank.

30.1.5 Diagnostics.

The diagnostic messages produced by the ALS VAX-11/780 Linker are summarized in Appendix 80.

30.2 Blank.

30.3 Blank.

30.4 Blank.

Removal of Sections 30.2, 30.3 and 30.4 included Figures 30-4 through 30-21.

30.5 Using The ALS MCF Linker.

This section describes how to use the MCF Linker in the Ada Language System (ALS) to combine compilation units for execution on the MCF Target Environment.

The purposes of linking are:

a. To combine many separately translated units into a single unit, matching references to externally-defined names with their proper definition;

b. To include the text of system runtime routines into the new linked unit (Calls to system runtime routines are automatically inserted by the compiler into the translated machine text; the routines are predefined and need not be provided by the user.);

c. To designate one subprogram as the "main subprogram", i.e., the one that receives initial control when the program begins execution;

d. To allocate storage space to all of the user-supplied code and data, and to the necessary system runtime routines;

e. To check all units for compliance with the Ada language rules regarding the order of compilation (for example, to ensure that a library unit is compiled before any of its subunits);

f. To decide in what order library units are to be elaborated, and to ensure that a legal order of elaboration exists, following the rules of the Ada language; and

g. To produce listings that show the units that have been linked together, and describe how storage has been allocated.

When invoking the linker, the user must designate the Containers that are to be linked together. These Containers may have been generated by the compiler, the ALS MCF Assembler, or may have been created by a previous invocation of the ALS MCF Linker (see Section 30.5.2). All of the Containers must represent compilation units intended for execution on the MCF Target environment, and must reside in the same program library.

If the user wants to modify an already linked Container, he may do so by naming the Container to be modified and the new Containers to be used as replacements in the UNITLIST. The replacement Containers should be named first because the linker always uses the first definition of any compilation unit.

The user may also designate the library subprogram that is to be the "main" subprogram, i.e., the subprogram that initially receives control at execution time, or may designate that there is no main program because the output of the link is not intended for execution.

The linker will produce as output a Container. If the Containers designated by the user contain all the elements that constitute a complete program, then a complete program is produced. If some elements have been omitted, the output is an incomplete program (see Section 30.5.2). The Container produced by the linker can be supplied to the exporter (via the environment database), so that it can be executed on a MCF Target environment, or it can be used as input to a future link.

30.5.1 Invoking the Linker.

To invoke the ALS MCF Linker the user must supply a command of the following form to the ALS Command Language Processor:

```
LNKMCF main_name prog_lib output_container [UNITLIST=>file_name]
[OPT=>option_list]
```

The following paragraphs describe the linker arguments. Those arguments enclosed in brackets ('[]') are optional.

a. main_name: Either the name of the main subprogram or the keyword "NULL". If NULL is used then there is to be no main subprogram. In this case, the output Container is not eligible for exportation and execution. (It may be used as input to a future link as described in Section 30.5.2.)

If a main subprogram is designated then it must be a library procedure. The IN string parameter, if present, and return string, if needed, must be of type Standard.String and must be unconstrained. Normally, the Containers that will be linked are the ones in the program library whose compilation units are referenced directly or indirectly by the WITH and SEPARATE clauses of the main subprogram. (See Section 30.5.1.1. for the method of altering this procedure.)

b. prog_lib: Name of the program library which contains all of the Containers to be linked and in which the new output Container will be placed.

c. output_Container: Name for the new linked output Container. This name must follow the Ada rules for a compilation unit. It must be distinct from names of library units and linked containers already in the program library.

d. UNITLIST filename: Normally, the main subprogram name provides a starting point for the linker search process that finds the Containers to be linked. This argument is used to provide additional starting points, or to enumerate specifically the units to be linked. The file name designated here must be the pathname of a database file containing ASCII text. If the main_name is null a UNITLIST is required.

The file should contain a list of units in the program library that are to be linked one unit per line. If the search option is off, exactly the units listed will be linked; otherwise, these units and all units referenced by the designated units will be linked.

e. option_list: The form of the option list is:

(option_name {,option_name})

The parentheses may be omitted if there is only one option. The options are discussed in detail in Section 30.5.1.1.

30.5.1.1 Options.

This section describes the options that may be specified to the linker. Each option may be specified as shown, or may be preceded by the three characters NO_, which indicate the opposite option. (For example, SYMBOLS and NO_SYMBOLS.) It is an error to specify both an option and its opposite.

SYMBOLS — Provide a Symbol Definition Listing, if a Container is produced. Default: NO_SYMBOLS

LOCAL_SYMBOLS — If a Symbol Definition Listing is produced, include names local to library package bodies as well as externally visible names. Default: NO_LOCAL_SYMBOLS, means to include only names that are externally visible.

UNITS — Provide a Units Listing if a container is produced. Default: UNITS.

SEARCH — Linker will automatically follow the WITH and SEPARATE clauses to find all of the units in the program library that are referenced from the designated starting point units. All of the referenced units will be linked into the output Container. If NO_SEARCH is specified, only the starting point Containers and the runtime routines referenced by the Containers are included in the linked output. The SEARCH procedure will take the first occurence of a unit. Default: SEARCH.

30.5.1.2 Units Listing.

The Units Listing is a list of all the compilation units that are represented in the linked Container. For each compilation unit, the following information is provided:

a. Name.

b. Size, in bytes, including all code and statically-allocated data.

c. Compilation date.

d. The tool (compiler, assembler, linker) that created the Container containing this unit.

e. Unit Status: L for Library Unit, S for subunit.

f. Diagnostic Status: R for recompilation advised, O for out-of-date revision.

If the unit has status R, a recompilation advisory message has been issued by the compiler, but the unit has not been recompiled (see Section 30.5.4). If the unit has status O, it is not the most recent revision of that unit.

The UNITS option controls the production of Units Listings. An example of a Units Listing is shown in Figure 30-22.

```
MYPROG      PROGRAM_LIBRARY=DEMO_PROG      LNKMCF      A_S MCF LINKER V 1.0      UNITS LIST      MAY 27, 1981      PAGE 1

UNIT NAME              SUBPAGE UNITS    CREATION DATE              CREATOR                 STATUS

MAIN.SPEC                     0         SEP 16, 1980 9:15          ADA V3.1                L
MAIN.BODY                    38         SEP 17, 1980 4:00          ADA V0.1                L
MAIN.SUB1                    50         SEP 19, 1980 6:12          ADA V0.1                S
FAST_HASH.SPEC                0         SEP 17, 1980 2:36          ADA V0.1                L
FAST_HASH.BODY               16         SEP 18, 1980 3:45          ADA ASSEMBLER V1.0      L
PACK1.SPEC                   18         SEP 16, 1980 11:40         ADA V3.1                L
PACK1.BODY                   68         SEP 19, 1980 9:30          ADA V0.1                L
PACK1.SUB1                   24         SEP 18, 1980 12:03         ADA V3.1                SR
```

Figure 30-22. Units Listing

30.5.1.3 Symbol Definition Listing.

The Symbol Definition Listing describes the allocation of objects and compilation units. For the MCF Target, the listing is divided into four parts showing the allocation of:

a. Elaboration code for library units,

b. Executable code for bodies,

c. Static read/write data, and

d. Static read-only data.

An example Symbol Definition Listing is shown in Figure 30-23. The option, SYMBOL, indicates whether a Symbol Definition Listing should be produced.

The first two parts of the listing show the allocation of compilation units. For each library unit or subunit, the following informat ion is provided:

a. The compilation unit name;

b. The location of the instructions representing the body and elaboration of the unit (hexadecimal origin and length in storage units);

c. The location of entrances to externally visible subprograms (hexadecimal address); and

d. If the LOCAL_SYMBOLS option is specified, the location of the subprogram entrances for subprograms local to package bodies (hexadecimal address).

The last two parts of the listing show the allocation of data in library units. If the LOCAL_SYMBOLS option is provided, data declared in the package body is included. For each object, the following information is provided:

a. The object name, and

b. The location of the data (hexadecimal origin and length).

```
MYPROG      PROGRAM_LIBRARY=DEMO_PROG      LNKMCF      ALS MCF LINKER V 1.0      SYMBOL DEFINITION      MAY 27, 1981      PAGE 1

                                                 START          LENGTH

    ELABORATION INSTRUCTIONS

        PACK1.BODY                               00000000       0000000A

    BODY INSTRUCTIONS

        MAIN.SPEC                                00000000       00000000
        MAIN.BODY                                00000000       00000026
        PACK1.SPEC                               00000026       00000000
        PACK1.BODY                               00000026       00000040
        PACK1.ACCESS_GLOBAL                      00000030       00000036
       *PACK1.LOC                                00000026       0000000A
        FAST_HASH.SPEC                           00000066       00000000
        FAST_HASH.BODY                           00000066       00000010
        PACK1.SUB1                               00000076       00000018
        MAIN.SUB1                                0000008E       00000032

    READ/WRITE DATA

     PACK1.GLOBAL                                00000000       00000008
    *PACK1.TEMP                                  00000008       00000004

    READ/ONLY DATA

     PACK1.SIZ                                   00000000       00000004
     PACK1.MAX                                   00000004       00000006

*Indicates a local subprogram or data item.
```

Figure 30-23. Symbol Definition Listing

30.5.1.4 Link Summary Listing.

A Link Summary Listing is produced each time the linker is invoked. The listing includes:

a. Whether an output Container is produced.

b. The total size of instruction (hexadecimal bytes),

c. The total size of statically-allocated data (hexadecimal bytes),

d. The number of compilation units linked,

e. The name and hexadecimal address of the entrance to the main subprogram, and

f. A summary of diagnostic messages.

An example of a Link Summary Listing is shown in Figure 30-24.

```
MYPROG        PROGRAM_LIBRARY=DEMO_PROG        LNKMCF        ALS MCF LINKER V 1.0        LINK SUMMARY        MAY 27, 1981        PAGE 1


A LINKED CONTAINER NAMED MY_PROG HAS BEEN PRODUCED

INSTRUCTIONS    192    BYTES
DATA             22    BYTES

NUMBER OF COMPILATION UNITS:  8
       MAIN SUBPROGRAM NAME:  MAIN
                 ENTRY POINT:  00000000

DIAGNOSTIC SUMMARY

   18 PACK1.SUB1 WAS NOT COMPILED AFTER PACK1.BODY
```

Figure 30-24. Link Summary Listing

30.5.2 Preparing Incomplete Programs.

It is possible to use the linker to prepare incomplete programs for execution and test. An "incomplete program" consists of one or more Ada compilation units which together do not constitute a complete Ada program because:

a. Some library subprogram or library packages bodies are omitted; and/or

b. Some separate subprogram or package bodies are omitted.

Incomplete programs may be created and then exported for execution on the target environment. Through the use of this facility, it is possible for programmers to test portions of large programs before all the units of the complete program are written, or to test portions of programs, then to integrate the tested portions for further testing.

An omitted subprogram body or package body may be stubbed, using the STUBGEN tool (see Appendix 70). STUBGEN creates a text file which is a legal Ada package body or subprogram body, in conformance with the declaration or specification. This stub may be compiled, producing a Container in the program library for the omitted body. The Container may be linked, exported, and executed in the normal manner.

When a stubbed subprogram is called, it returns legal values for all OUT and INOUT parameters, and in the case of functions, it produces a legal return value. Optionally, the stub prints the name of the subprogram which was called.

It is possible that a body might not be available in the program library, even in stub form. It is expected that an incomplete program will not attempt to call subprograms whose bodies are not present. A call of this type will raise the runtime exception, SYSTEM.UNRESOLVED_REFERENCE, which will abort execution if the incomplete program does not contain an appropriate exception handler.

If a unit body referenced by another part of the program cannot be found in the program library the linker will warn the user of each missing subprogram body or package body (see Section 30.5.5) and will create a Container representing an incomplete program. This Container may be exported and executed (see Appendix 40).

A Container may be resubmitted as input to a future link. For example, when a programmer has tested part of his program and wishes to add more units, or when many programmers have tested parts of programs and wish to integrate the pieces for further testing, it is possible to input to the linker one or more linker-created Containers, possibly along with Containers created by the compiler or assembler. In this case, the linker-created Container represents a short-hand notation for all of the compilation units of which it is composed, indicating that all these units are to be included in the link. (Note that the original Containers

representing the compilation units may need to be accessed by the linker.)

The following example illustrates the use of the linker to create incomplete programs:

a. Assume that subprogram ENTER has two separate subunit procedures CHECK and MODIFY. The application is a database update; CHECK verifies the data and MODIFY updates the database. ENTER is the driver that calls CHECK, then MODIFY. The subprogram ENTER and CHECK are to be tested. When they are working, MODIFY will be added and the whole program will be tested.

b. The user will compile ENTER, then CHECK.

   Alternative 1: He will generate a stub for MODIFY, then compile it. He will link the three Containers together, export, and test them. When they are working, the real MODIFY is compiled, producing a new revision of the Container. The three Containers, including the most recent MODIFY, are linked, exported, and tested.

   Alternative 2: He will link ENTER and CHECK into an incomplete program named TEST1. During this link, the linker will warn the user of the omission of MODIFY. TEST1 is exported and tested; an exception will be raised when MODIFY is called. When TEST1 is working, MODIFY is compiled, and TEST1 and MODIFY are linked to form a complete program.

30.5.3 Allocation of Storage.

The ALS MCF Linker allocates storage in four groups. The allocation is designed to use MCF hardware facilities to provide maximum protection against modification of instructions and read-only data.

The four groups are:

a. Executable code that elaborates library units;

b. Executable code that represents the bodies of subprograms and packages;

c. Read-write static data, such as that declared in library units; and

d. Read-only static data, including literals.

Because of the large virtual address space of the MCF, there is no requirement for an overlaying capability, or for allocating instructions to a particular location in physical or virtual space.

30.5.4 Blank.

30.5.5 Diagnostics.

The diagnostic messages produced by the ALS MCF Linker are summarized in Appendix 80.

APPENDIX 40

40. EXPORTING, LOADING, AND EXECUTING PROGRAMS IN THE ADA LANGUAGE SYSTEM

Descriptions of the procedures for exporting, loading, and executing programs for the VAX-11/780 VAX/VMS and MCF Target Environments are provided on the following pages.

The Ada Language System includes the following loaders:

a. ALS MCF Loader.

40.1 Exporting, Loading, Executing Programs On The VAX/VMS.

This section describes how to bring programs into execution on the VAX/VMS Target Environment. The Ada program is compiled and linked on the host ALS system, producing a Container containing the executable representation of the program. Execution of the program from the Container is performed as described below.

40.1.1 Exporting.

The Container created by the Linker must be exported to an ALS file before it can be executed. This Container may represent a complete or incomplete program as described in Section 30.1.

If a Container is to be exported, a main subprogram must have been designated to the Linker. (See Section 30.1) The Exporter performs the following actions:

a. Performs relocation of the user program and converts it to VAX/VMS load module format, and

b. Writes the load module into the designated ALS or VMS file.

The command to invoke the Exporter for the VAX/VMS Target Environment is:

```
EXPVMS (name, prog_lib, output_medium [, OPT => option_list])
```

where:

name	is the name of the Container to be exported;
prog_lib	is the name of the program library that contains the Container to be exported; and
output_medium	is the name of the ALS or VMS file where the load module is to be written.

option_list

DEBUG	Directs the Exporter to activate a Debugger Kernel in the program image to allow debugging. Default is: NO_DEBUG.
FREQUENCY	Activate the frequency kernel to monitor execution frequency. When NO_FREQUENCY is specified, or is in effect by default, no execution frequency is monitored, regardless of compile options. Default: NO_FREQUENCY.

STAT — Activate the statistical kernel to monitor execution timing. When NO_STAT is specified, or is in effect by default, no execution timing is monitored.
Default: NO_STAT.

The Exporter does not produce any listings except for diagnostic messages, as appropriate. The output consists of the load module in a designated file and any diagnostic messages, summarized in Appendix 80, as appropriate.

40.1.2 Loading and Executing.

After compiling, linking, and exporting an Ada program, the program can be executed under the ALS. The command to execute the program is:

<file_name> <arguments>

The <file name> is the file produced by the Exporter. The <arguments> portion is passed to the Ada Program as a character string input parameter. (See Appendix 90 for details of parameters and return codes for an Ada program running under the ALS.)

If the Ada program has been exported with the STAT option in effect, then before the target program begins execution, prompts will be issued on .STDOUT for <interval> and <statistical_data_file>, explained below. Values for these parameters are read from .STDIN.

<interval> is the mean rate at which the locus of control (i.e., the value of the program counter) is sampled and recorded. The interval is specified in milliseconds as a decimal integer without exponent, in accordance with the Ada syntax for unsigned integer literals. The interval represents wall-clock time. The elapsed process CPU time will be some fraction of this interval, varying with the load on the target computer. If not specified, a default interval of 10 milliseconds will be used.

<statistical_data_file> is the name of the file that is to contain the time data. If the statistical_data_file already exists, and is not frozen and not unmodifiable, the new data will be appended to the old data. Otherwise, it is created. If the null string is given no data are collected.

If the Ada program has been exported with the STAT option, and the null string has been provided in response to the prompt for the statistical_data_file, data are not collected and a warning diagnostic is issued on the MSGOUT file.

If the Ada program has been compiled and exported with the FREQUENCY option, the name of the <frequency_datafile> will be requested by a prompt on .STDOUT. The name is then read from .STDIN.

<frequency_data_file> is the name of the file that contains the frequency data. If the frequency_data_file already exists, and is not frozen and not unmodifiable, the new data are appended to the file. Otherwise, a new file is created. If the null string is given, no data are collected.

If the Ada program has been exported with the FREQUENCY option, and the null string has been provided in response to the prompt for the frequency_data_file, frequency data are not collected and a warning diagnostic is issued on the MSGOUT file.

Statistical and frequency analysis may be done at the same time. It should be noted that frequency analysis alters the time characteristics of the Ada program. Moreover, the presence of frequency monitors in the compiled code (even if frequency monitoring is not turned on) may reduce the amount of optimization that would otherwise be done by the compiler.

40.1.3 Termination of Execution.

Execution of the Ada program can terminate by the normal completion of the main subprogram, by the occurrence of an unhandled exception, or by a manual interruption by the operator/user.

Upon normal completion of the main subprogram, control is returned to the user in the command language from which the program was invoked.

When an exception is raised, the runtime nucleus attempts to propagate the exception as defined by the language. If there is no active handler for the exception, execution of the program is terminated and a message generated in the message_output file. The message is of the form:

```
Exception  <e_name>  raised, exception was
 RAISED BY:  <e_unit>  <e_subprogram>  (statement <e_statement>)
 CALLED BY:  <c_unit>  <c_subprogram>  (statement <c_statement>)
      .
      .
      .
 CALLED BY: <c_unit>  <c_subprogram>  (statement <c_statment>)
End of Traceback.
```

where:

<e_name> is the exception name (either pre-defined or user-defined).

<e_unit> is the name of the compilation unit (library unit or

subunit) in which the exception occurred.

<e_statement> is the statement number of the compilation unit in which the exception occurred.

<e_subprogram> is the subprogram, if any, containing the statement (The first and last lines of the message will not show a subprogram if the exception occurred during elaboration).

The list of "c_units", "c_statements", and "c_subprograms" following the initial line of the message constitute a retrace of the subprogram calls; the first is the caller of the executing procedure, the last is the main subprogram.

For example, the message

```
Exception  CONSTRAINT_ERROR  raised, exception was
 RAISED BY:  XYZ  C  (statement 350)
 CALLED BY:  XYZ  B  (statement 240)
 CALLED BY:  A    A  (statement 130)
End of Traceback.
```

could be generated by Subprogram C attempting to store an out-of-range value into a variable at Statement Number 350. Subprogram A is the main program and it called Subprogram B with a call at Statement 130; B then called C from Statement 240. The three subprograms are in compilation units A, XYZ, and XYZ, respectively.

In the presence of optimization, the statement number given in the traceback may not exactly match the statement number in the program listing. The construct causing the exception, however, will lie in close proximity to the statement identified in the traceback.

Units not containing traceback information (i.e., compiled with the option NO_TRACEBACK see 3.7.1.1.1.3) will contain the output line,

CALLED BY: <UNIT WITHOUT TRACEBACK TABLES> (address <c_address>)

<c_address> is the address of the call statement instead of the normal line with traceback information.

The termination of a task, as opposed to a program, due to an unhandled exception does not cause a message to be generated since this is defined within the language as a normal termination.

When executing under VMS, manual interruption can be effected by using the Break-in facility described in Appendix 110.

40.2 Blank.

40.3 Blank.

40.4 Blank.

40.5 Blank.

Removal of Sections 40.2, 40.3, 40.4, and 40.5 included Figures 40-1, 40-2, and 40-3.

40.6 Exporting, Loading, Executing Programs On The MCF.

This section is <TBD>.

APPENDIX 50

50. THE ENVIRONMENT DATABASE

The environment database provides the user with a complete file system. The services provided are very similar to those of many modern file systems. In addition to storage, the services include structuring, access control, sharing, and support for configuration management.

The environment database is a self-contained entity. Users are not required to have any knowledge of the VAX/VMS file system.

50.1 Nodes

The objects in the database are called nodes. There are three basic types of nodes: files, directories, and variation headers. Files and directories are described here. Variation headers are described in Section 50.5.

Nodes are created by the "mkfile", "mkdir" and "mkvar" tools, and deleted by the "delnode" tool. Many other tools are available for manipulating nodes. Appendix 70 includes a complete description of all the ALS tools.

All nodes have properties called attributes and associations. Attributes are named properties with character string values. Associations are named properties with values that are collections of "pointers" to other nodes. Most attributes and associations are given to nodes as "side effects" of the tools that process a node. Users may also examine and manipulate attributes and associations with the simple tools, such as "lstattr", "lstass", "chattr", "chass", "addref", and "delref".

Attributes enable the database to contain character string information about a node. Associations enable the database to contain arbitrary networks of relationships among nodes. Each association is a set of pointers emanating from the node possessing the association, such that there is one pointer to each node referenced in the association. Associations are primarily created by tools that need to maintain information about relationships between nodes. Associations and attributes are further described in Section 50.7.

In addition to attributes and associations, file nodes contain a data portion that may be read and written by Ada programs via the standard Ada I/O packages, Input_Output and Text_IO. As is common practice, the data portion of file nodes does not have a type. Ada programs interpret the data as a sequence of values of some type when an Ada (internal) file is associated with an ALS (external) file.

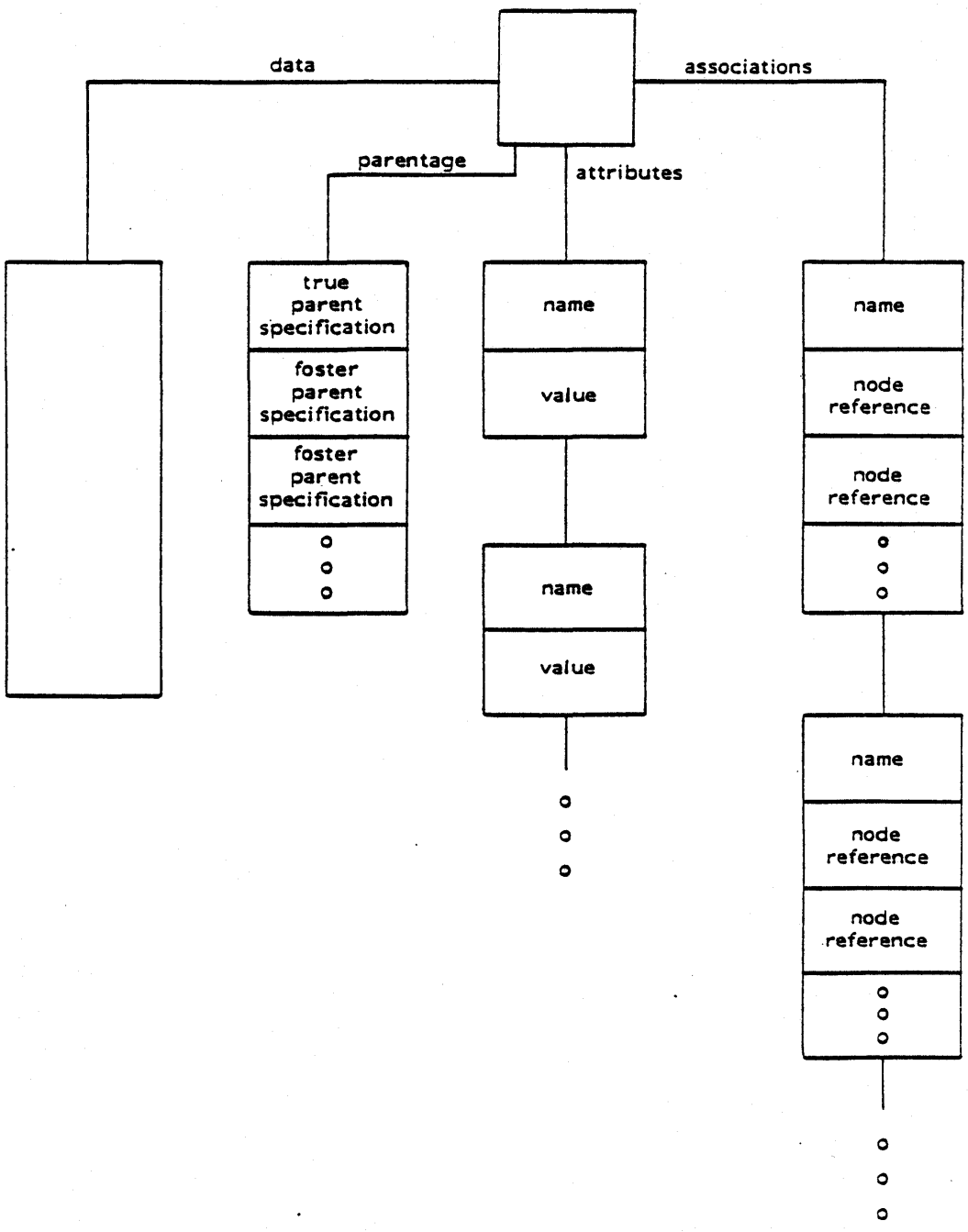


Figure 50-1. File Node Structure

Figure 50-1 shows the logical structure of a file node. The "true parent" part of the "parentage" information is described in Section 50.3. The "foster parent" part of the "parentage" information is described in Section 50.10.

Directory nodes are used to name and group other nodes. When a node is created, it is created "within" a directory. In addition to attributes and associations, each directory contains a specification of the nodes grouped in it. These nodes are called the "offspring" of the directory. An empty directory is one with no offspring.

Figure 50-2 shows the logical structure of a directory node. The "true parent" part of the "parentage" information is described in Section 50.3. The "foster parent" part of the "parentage" information is described in Section 50.10.

The database is organized in a directed acyclic graph (DAG). Variation headers and directories are nodes in this DAG. Files are leaves in this structure. (Leaves may also be empty directories or empty variation headers. An empty directory is a directory with no offspring.) Every node in the database appears in this DAG. The DAG is further described in Section 50.3.

50.2 File Revisions

The environment database supports the tracking of changes made to a file through time. This tracking is accomplished by the use of file revisions.

The revisions are a linearly ordered set of numbered files. When a file is created it is automatically assigned a revision number of one. Subsequent revisions automatically receive revision numbers two, three, etc. There is no fixed limit on the number of revisions of a file. Users may create a new revision of a file with the "revise" and other tools.

The most recent revision of a file is special; it is considered to supersede all prior revisions. Use of a file name automatically selects the most recent revision. For example, if alpha is a file with five revisions, the command

```
lst (alpha)
```

will list the data portion of the fifth revision of alpha.

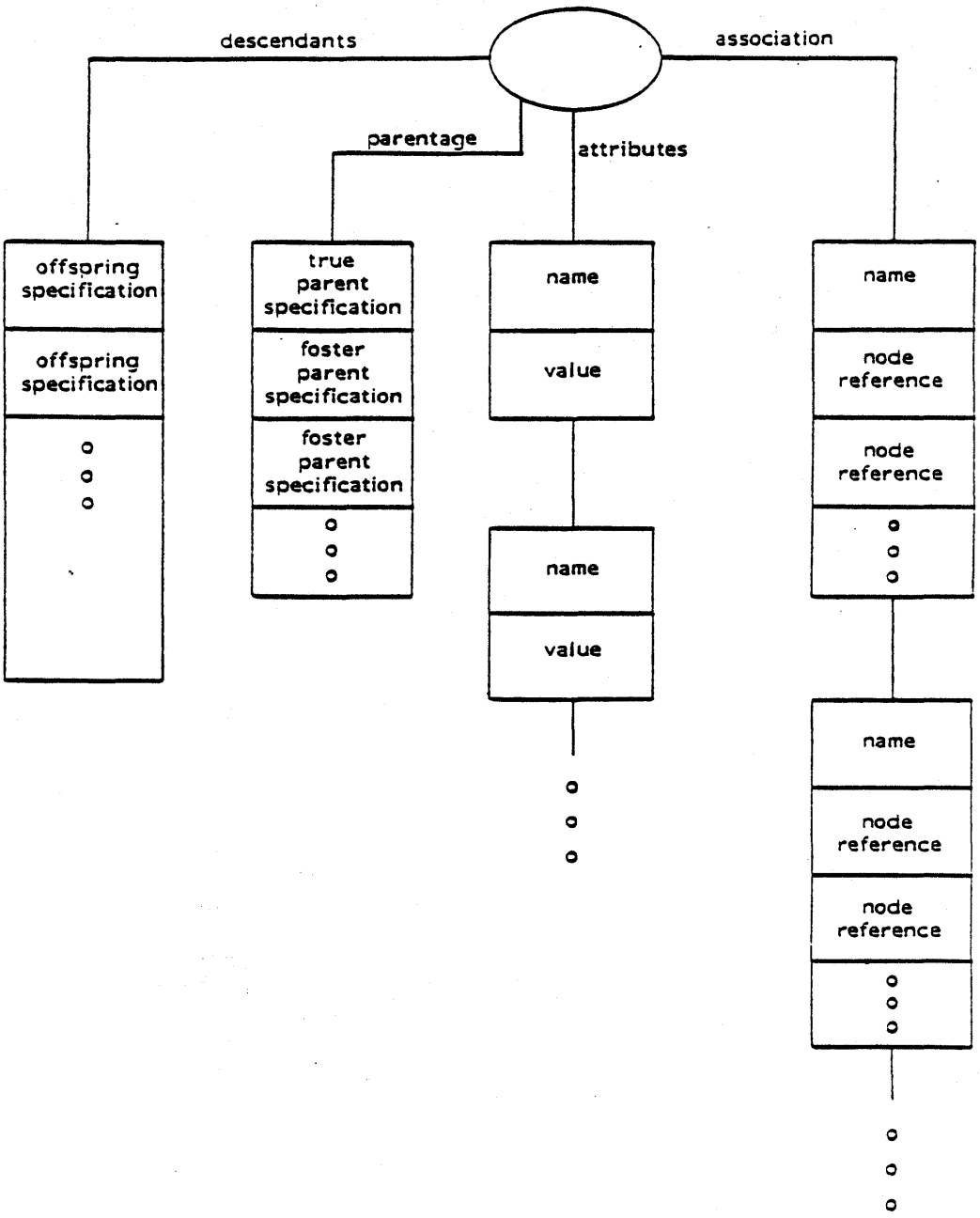


Figure 50-2. Directory Node Structure

Any particular revision may be selected by adding an integer index to the name of the file. For example, the command

```
lst (alpha(3))
```

will list the data portion of the third revision of alpha. If the user selects a revision that does not exist the "lst" tool will produce a diagnostic.

When a new revision of a file is created, ALS automatically freezes the previous revision. The most recent revision of a file may also be permanently frozen by the "freeze" tool. Revisions that have become frozen by the creation of a subsequent revision, or by the freeze tool, are said to be frozen. Such revisions may be never be unfrozen even if the subsequent revision is later deleted. Frozen revisions may be deleted.

In addition to freezing, the data portion of a file revision may not be changed if the derivation_count attribute (see Section 50.8) is greater than zero. Such revisions are said to be unmodifiable. Since the derivation count may decrease to zero, revisions which are unmodifiable, but not frozen, may become modifiable again. Unmodifiable revisions may not be deleted. These rules are summarized in Figure 50-3a.

All of the revisions of a file are collectively called a revision set. Only the most recent revision of a set may ever be modified. The data parts of all earlier revisions are permanently frozen and protected from modification by the ALS. The attributes and associations of frozen or unmodifiable revisions may, however, be modified. Since frozen revisions can be deleted but not modified, the user can rely on the permanence of the contents of any existing revision set, and yet have the convenience of being able to delete unused revisions.

Unless specifically stated otherwise, ALS tools will overwrite the latest revision if it is not frozen, and automatically create a new revision if the latest is frozen or unmodifiable.

The most recent frozen revision of a file may be selected by using the index "+". For example, the name alpha(+) selects the most recent revision of a file alpha if it is frozen; otherwise it selects the previous revision.

The entire revision set can be referenced by using the index "*". For example , the name alpha(*) refers to all revisions of the revision set alpha. The command

```
delnode alpha(*)
```

deletes every revision of alpha.

File Deletion and Modification File is:	Modifiable (derivation_count = D)	Unmodifiable (derivation_count ≠ D)
Frozen	D	–
Not Frozen	M,D	–

M = text portion of file may be modified in-place

D = file may be deleted

Figure 50-3a. File Rule Summary

50.3 Directory Hierarchy

As mentioned above, directories are organized in a DAG. The nesting of directories may be arbitrarily deep (subject to the limitations of the total length of a path name, described in 50.12). The root of the structure is an ALS-created directory called the Root. The nodes grouped in a directory are called its offspring.

The directory in which a node is created is called the node's true parent. Every node except the Root has exactly one true parent directory. Parentage is described in Section 50.10.

A directory contains the identifiers that name its offspring. These identifiers conform to the rules for Ada identifiers, a sequence of alphanumeric characters, the first of which must be alphabetic, possibly containing isolated, embedded, underscore characters. As in Ada, there is no distinction between upper and lower case characters. Unlike Ada, these identifiers are limited to a maximum length of 20 characters. The names of all offspring of a particular directory must be distinct, but offspring of different directories may have the same name.

Figure 50-3b shows a directory hierarchy for a simple mathematics package. Directories are drawn as ellipses, files are drawn as squares. The offspring of a directory are drawn below the directory, connected to it by lines. The name of each offspring labels the line. File revisions are drawn in a "stack" of squares, the revision number appears within the square.

50.4 Path Name Basics

The hierarchy of directories provides the naming structure for all nodes in the database. Nodes are selected by entering their path names, the sequence of directories that must be followed to find the node. A simple path name consists of one or more node identifiers separated by periods, and possibly starting with a period. Some simple path names are a.b.c, .e.f.g and x. Path names may not contain embedded blanks. (More complex forms of path names are described in Section 50.5 and Section 50.12).

Finding a node involves using the identifiers of its path name to trace a path through the directory hierarchy. The final node in the path is the named node. The final node may be a file, directory, or variation header.

Absolute path names start with a period; the search for the node begins at the Root. For example, the name .math_pac.source.sin means that the Root has an offspring named math_pac, that is a directory with an offspring named source, that is a directory with an offspring named sin, that is the named node.

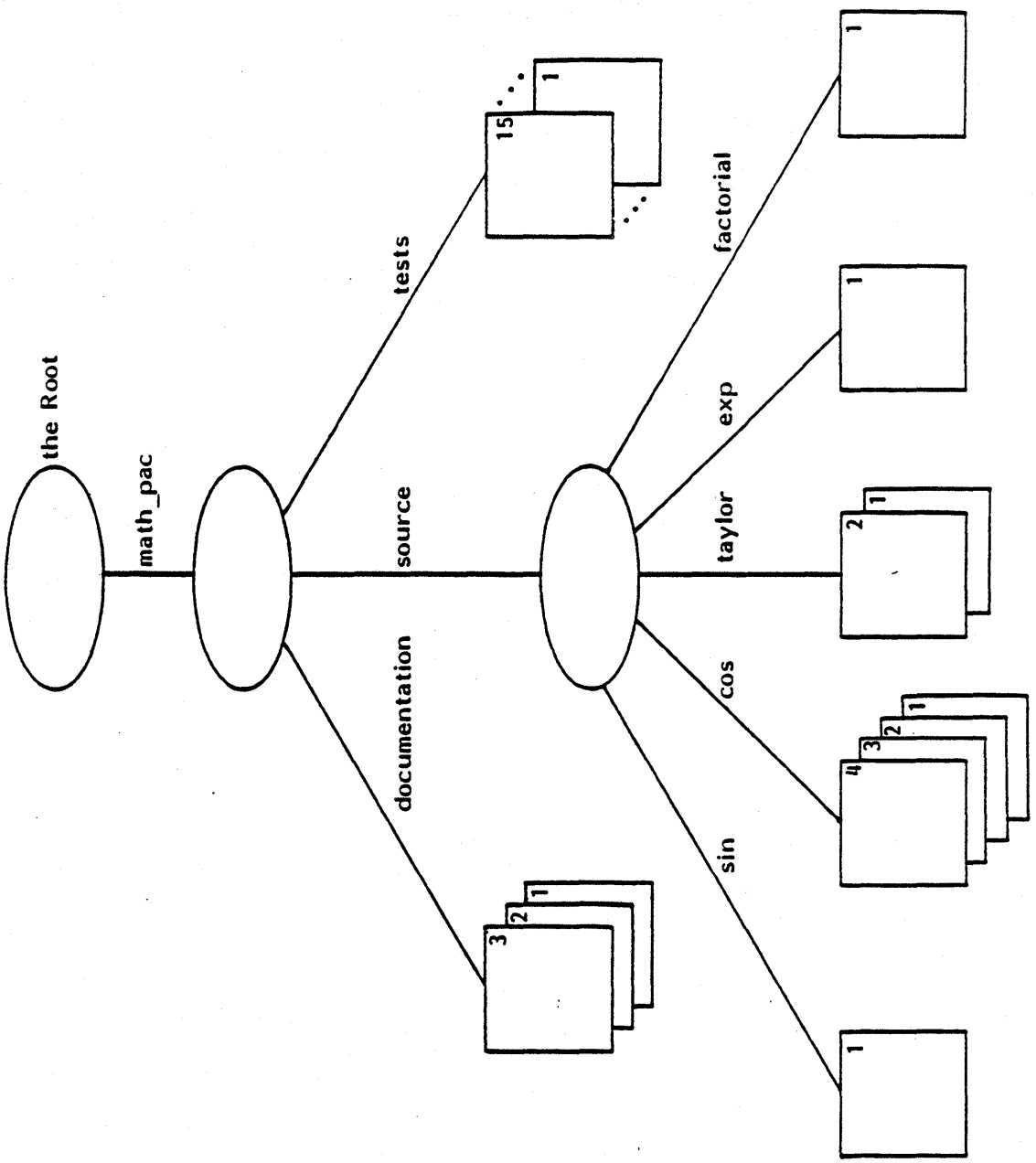


Figure 50-3b. Directory Hierarchy Example

Using absolute path names all of the time can be a burden when dealing with the lower levels of the directory hierarchy. To ease this burden, every user has a directory called his "current working directory" (CWD). Path names that do not begin with a period are relative to the CWD. For example, the name, source.taylor, means that the CWD has an offspring named source, that is a directory with an offspring named taylor, that is the named node. As a trivial example, the name "tests" names the node by that name that is an offspring of the CWD.

Figure 50-4 shows path names for the hierarchy of Figure 50-3. Both absolute and relative path names are shown with the "math_pac" directory assumed to be the CWD.

Every user interacting with the ALS is automatically assigned a default directory as his CWD at the time of log-in to the ALS. (Logging-in to the ALS is fully described in Appendix 110.) The user may use his default CWD, or use the "chwdir" tool to change his CWD to another directory at any time during his ALS session. Each time the user logs on to the ALS he will start with his default CWD even though he may have previously logged off with another directory as his CWD. (A user's default directory may be changed by a system administrator.)

50.5 Variation Sets

The environment database supports collections of related objects that do not supersede one another, but that are "equal" alternatives. One example is a collection of different bodies implementing the same Ada package specification. These collections are called variation sets; their elements are called variations. A variation set is an unordered set of named nodes, with a header node called the variation set header.

Variation sets are created by the "mkvar" tool, which creates an empty variation set. The elements in a set may be files, directories, or other variation sets. Elements of a single variation set may be a mixture of different types of nodes; all the elements do not have to be the same type of node. Elements are named by the user when created.

Variation sets have a variation header node, the third basic kind of node (the other two are files and directories). A variation header node is similar to a directory, it is created by the mkvar tool. The elements of a set, called variations, may be thought of as the offspring of the header node. The header node is the true parent of each variation created in the set. Directories may contain variation sets as offspring and variation sets may have directories as offspring.

Figure 50-5 shows the logical structure of a variation header node. The "foster parent" part of the "parentage" information is described in Section 50.10.

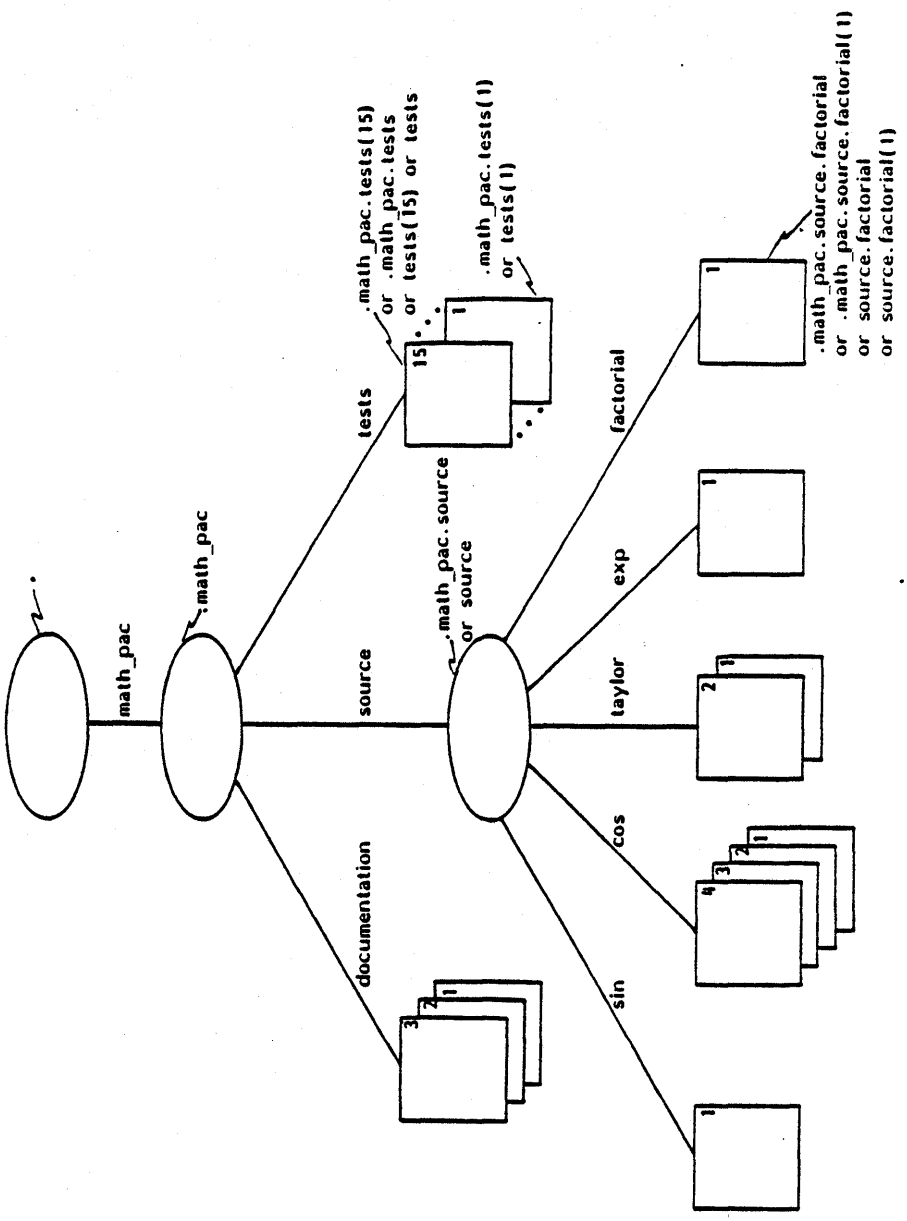


Figure 50-4. Path Name Example

Variations are selected by adding the variation name as an index to the set name. For example, the path name, source(integer) names the variation of source whose name is integer (where source is a variation header which is an offspring of the CWD).

Figure 50-6 shows variation sets using the math package of Figures 50-3 and 50-4. The source files have two variations, one for integer hardware and one for floating point hardware. The floating point variation is further subdivided into long and short variations. The variation header node is drawn as a hexagon. Some of the nodes are labeled with their path names.

Variations are added to a set by the normal node creation tools "mkfile", "mkdir", "mkvar" and "share". For example, the command

```
mkfile (source(decimal))
```

adds a variation named decimal (not shown in Figure 50-6) to the set source. (The share tool is described in Section 50.10.)

A variation may also be selected by attributes. This can be performed by giving an arbitrarily long sequence of attribute name and value pairs as an index (subject to limitations on total path name length, described in 50.12). For example, the path name, source(mode=>flt_pt,size=>long), names the variation of source for which the mode attribute has the value "flt_pt" and the size attribute has the value "long". It is an error if no variation has these attribute values, or if more than one does. Selection by attribute may not be used to create, to share, or to rename nodes.

Figure 50-7 shows an alternative organization for the example of Figure 50-6. A single variation set whose element names are just "placeholders" is used. Attribute selection is used to name the elements.

If an element of a variation set is a file with revisions, the name of that element is the name of the file and the previously described rules for revision selection apply. For example, the command

```
lst (source(integer).sin)
```

causes the latest revision of the sin file in the integer variation of the source to be listed. Similarly,

```
lst (source(integer).sin(3))
```

causes the third revision of the sin file variation to be listed.

Nested variation sets behave analogously. For example, "source(flt_pt)(long)" names the long variation of the set flt_pt that is a variation of the set source. Since variation sets may be nested arbitrarily deeply, an arbitrarily long sequence of variation indices may appear. These indices may be a mixture of "name" selection and "attribute" selection in any order.

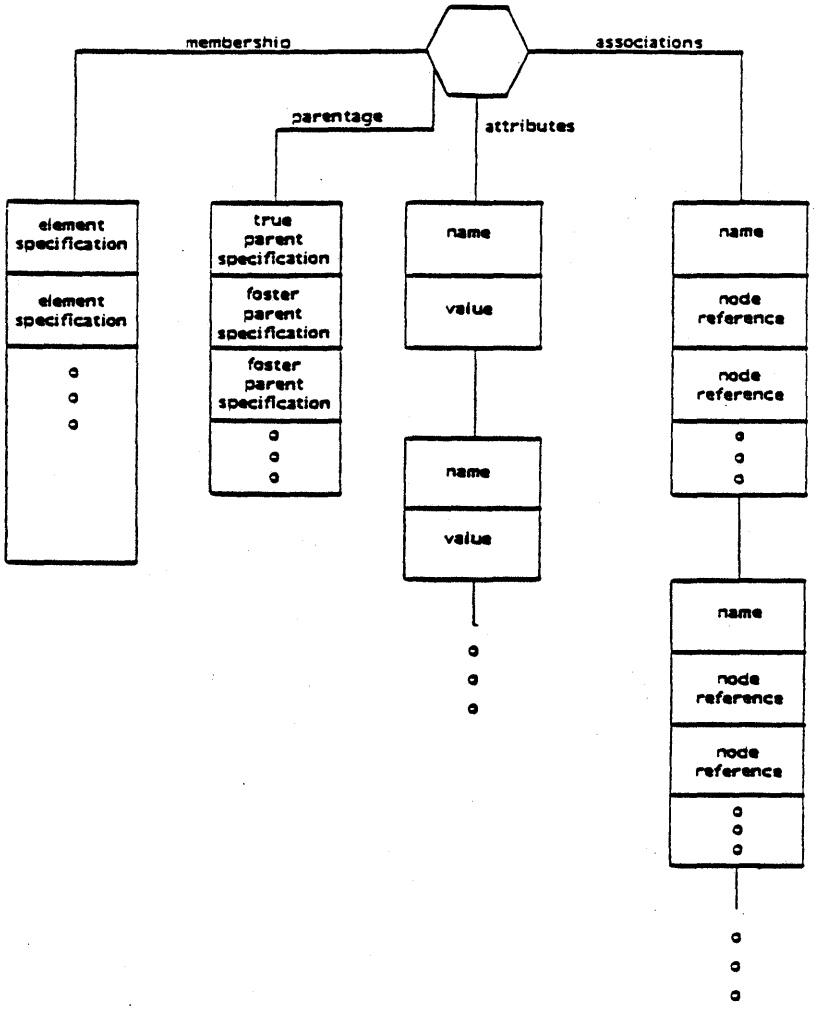


Figure 50-5. Variation Header Node Structure

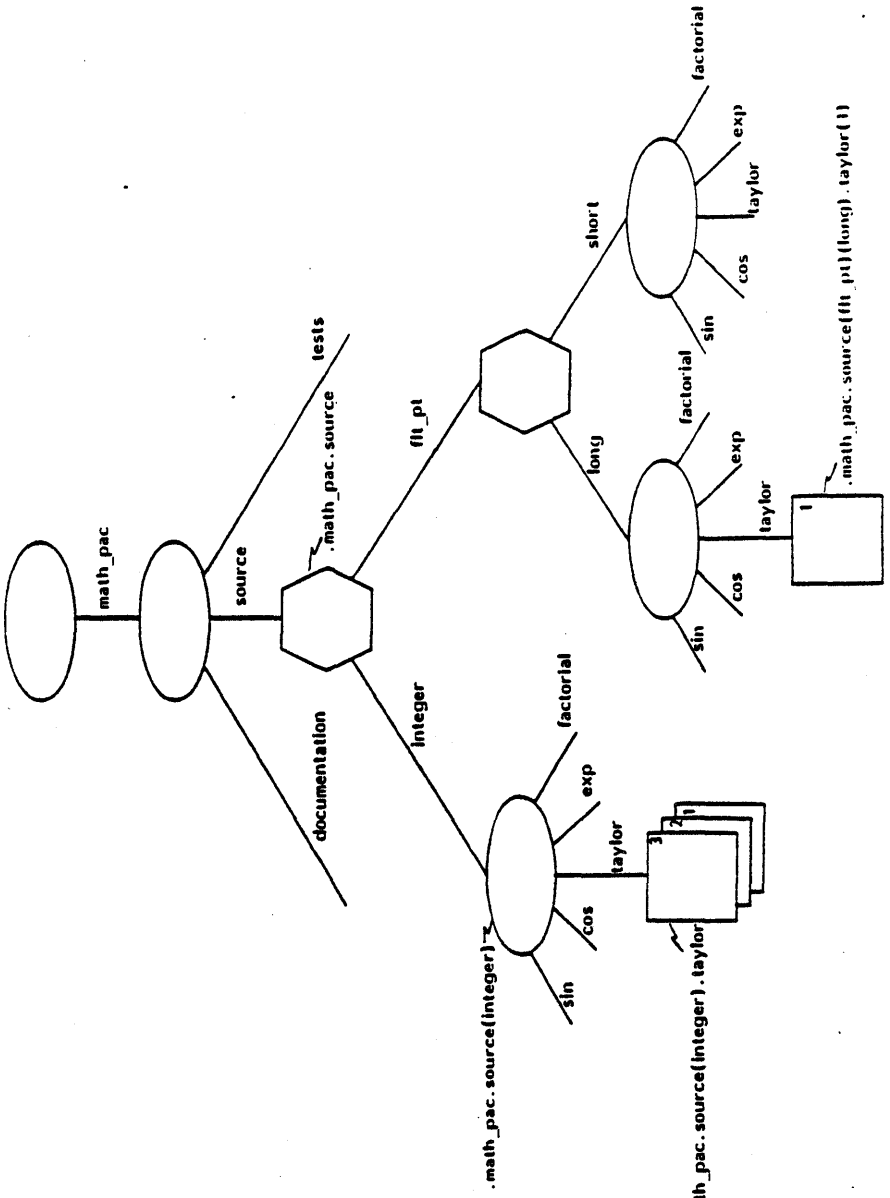


Figure 50-6. Variation Set Example with "Name" Selection

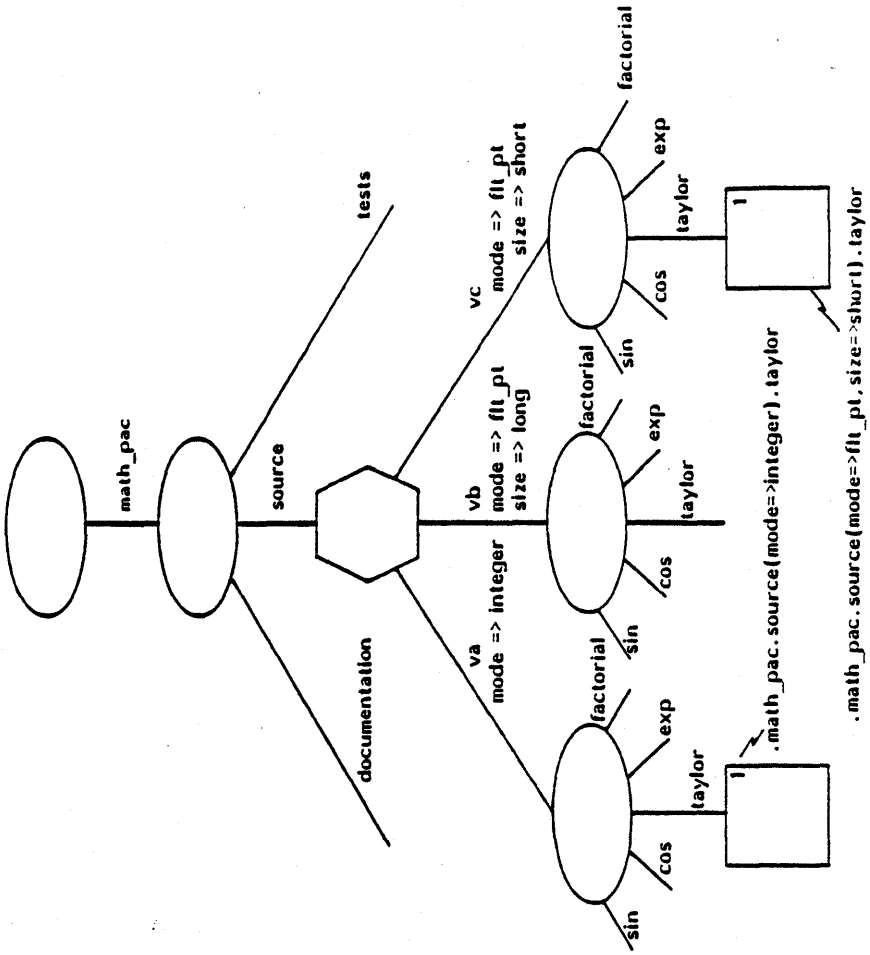


Figure 50-7. Variation Set Example with "Attribute" Selection

A variation set may (but is not required to) have one variation specified as the default; the default_variation attribute of the variation header node gives its name. The default may be selected with an empty index. For example, source() selects the default variation of the variation set source. The "chattr" tool may be used to define a default. For example, the command

```
chattr (source, default_variation=> integer)
```

defines integer as the default variation. (Note that the default must be explicitly chosen, the name source by itself selects the header node, not the default variation.)

In a contiguous nest of variation sets, the empty index for default selection may only appear once as the last (rightmost) variation index. A default index causes selection of the default variation all the way to the bottom of the variation set nesting. It is an error if any of these "lower" sets do not have a default variation. To select a file at the bottom of the nest, the default index may be followed by a revision index.

In selecting the default variation, the user is allowing his choice of a variation to be decided by the person who set up the default. In this case, the user should not care which variation is the default. Furthermore, he should not care if the default is changed. In particular, the way he names the default should not change if the default changes.

For example, consider the hierarchy shown in Figure 50-6. If the ".math_pac.source" variation header specifies the "integer" variation as the default, the name

```
.math_pac.source().taylor
```

is the same as

```
.math_pac.source(integer).taylor
```

Now suppose the default is changed to the "flt_pt" variation and that the "long" variation is its default. Then the name

```
.math_pac.source().taylor
```

is the same as

```
.math_pac.source(flt_pt)(long).taylor.
```

The user is not required to, and cannot, write

```
.math_pac.source()().taylor
```

in this latter case.

50.6 Access Control

The ALS identifies each user by a user name consisting of a sequence of one or more identifiers separated by periods. These identifiers follow the syntax for Ada identifiers but are limited to a 20 character maximum length. These user names are used in the control of access to ALS database nodes.

Every node has a set of standard attributes specifying the access controls for that node. The values of most of these attributes are lists of names of users granted that means of access. The via attribute is different in that it is a list of tool access names (described below). Each user name and tool access name must be followed by a slash (/). (The attribute value may be the empty string.) The names of these attributes and their meanings are:

read	:	for files, the data portion may be read; for directories and variation set headers, the node's offspring names are visible.
append	:	for files, the data portion may be appended to; for directories and variation set headers, offspring (elements) may be added
write	:	for files, the data portion may be rewritten or appended to; for directories and variation set headers, offspring (elements) may be added or removed
attr_change	:	the values of attributes and associations may be altered
execute	:	for files, the data portion may be loaded and executed or interpreted by the command language processor; for directories and variation set headers, this attribute is meaningless
via	:	this node may only be accessed by specified tools (further described below)

The ALS checks the legality of all node accesses. To check if a request for access is legal, the node must first be found by tracing the path name. Read access must be granted for every directory and variation header along the path.

If the node is found, its access control attributes are examined. (If the request is to open the node for reading, the read attribute is checked, etc.) If the user is named in the appropriate access attribute the requested access is granted.

The access control attributes may be set by the "chattr" tool. For example, the commands

```
chattr (gyro_spec, read=>"gyro.smith/thruster.jones/")

chattr (gyro_spec, write=>"gyro.smith/")
```

change the read and write attributes for the node named gyro_spec. Read access is given to two users; user Smith on the gyro team, and user Jones on the thruster team. Write access is only given to user Smith on the gyro team.

It should be noted that access to examine the attributes and associations of a node is automatically granted to every user who has access to a pathname to the node. The data part of frozen file revisions may not be changed regardless of the access control attributes. A user id may have up to,but not more than, 20 characters. A user name (user id plus team name) may have up to, but not more than, 200 characters.

The rightmost identifier of the user name is called the user id and is specified at the time of user log-in to the ALS. The rest of the user name is called the team id; it may be null.

The team id is initialized automatically at log-in to the ALS. An initial team id is defined and maintained for each user by the system administrator. It may be changed during the ALS session by the "chteam" tool. The ALS maintains a file of legal user names, and "chteam" checks the validity of all requests to change the team id. The user names allow for a flexible organization of project members into a hierarchy of teams.

To simplify the specification of groups of users, a single asterisk (*) may be used to match zero or more characters in a user name. Only one asterisk is allowed in the entire name, i.e., *.controller.*/ is illegal. As an example the command,

```
chattr (switch.simul, execute=>"*.wong/switch.io.*/switch.qa.black/")
```

specifies who can execute the program in the file "switch.simul". Execute permission is given to user Wong no matter what team, he is on, to any user on the switch.io team, and to user Black on the switch.qa team. (Note that the "io" team specification could include members of "subteams". For example, any user on the switch.io.encoding team has execute permission.) Access may be granted to all users by specifying either "*/" or ".*/" for the appropriate access attribute value. The attribute value "/" denies access to all users. The attribute value "./" is not allowed.

When a node is created its access control attributes are automatically initialized. The following accesses are allowed:

read	:	any member of the creating user's team
append	:	only the creating user
write	:	only the creating user

attr_change : only the creating user

execute : any member of the creating user's team

via : the null string (no tools are named here)

For example, if the user's name is "thruster.roll.smith", the default values of the access attributes will be:

read : thruster.roll.*/

write : thruster.roll.smith/

append : thruster.roll.smith/

attr_change : thruster.roll.smith/

execute : thruster.roll.*/

via : (empty)

Tools may need access to nodes that should not be granted to ordinary users. To provide this capability, the file node containing the tool may be given an access_name attribute. The value of this attribute has the form of a user name.

If a request for access is made to a node with a non-null via attribute, the tool making the request must have an access_name attribute that matches the via attribute. Thus, for example, a program library may be protected from indiscriminant access by giving every node in it a via attribute of "pl_tool" and giving all tools that are allowed to manipulate program libraries an access_name attribute of "pl_tool". (Program libraries are described in Section 50.13.)

The access_name attribute may only be altered by the "chacc" tool. This tool sets the access_name to the invoking user's name, preventing a user from granting more power to a tool than he himself possesses.

50.7 Attribute and Association Details

Attributes and associations have several common characteristics. Each has a "name" and a "value". The names conform to the syntax of Ada identifiers with a restriction to a 20-character maximum length. The names are not "declared" or "registered" with the ALS; they may be freely chosen by tool writers and users. There is no fixed limit on the number of attributes or associations that a node may possess.

An attribute is removed from a node by giving the attribute the empty string as its value. Inquiry about the value of an attribute not possessed by a node is not an error; the empty string is returned as the

value.

Similarly, an association is removed from a node when all of the references in the association are deleted. Inquiry about the value of an association not possessed by a node is not an error; an indication is returned that the association contains no references.

The values of attributes and associations may be examined with the "lstattr" and "lstass" tools. Attributes and associations may be given to a node, modified, or removed with the "chattr", "chass", "addref", and "delref" tools.

The ALS does not require that all association references be to existing nodes. The references must simply be syntactically legal path names. Such "hanging" references can arise if the referenced node is destroyed or renamed. References may also be created in a "hanging" state. For example, tools may create an association before creating the referenced nodes.

All references in an association must have the form of path names for ALS nodes as described in Section 50.12.

The path names used as association references may be absolute path names or relative path names. Relative path names in associations are relative to the node possessing the association; they are not relative to the CWD. The "chass", "addref" and "delref" tools allow specification of both forms. See Appendix 70 for full details.

When examining the value of an association, the pathnames displayed show the "mode" of each reference. Absolute pathnames are displayed for absolute (non-relative) references. Relative references are displayed as relative pathnames, but note that these names should be interpreted relative to the node possessing the association; they should not be interpreted relative to the CWD.

In general, relative path names in associations are more compact than full path names. Furthermore, they allow a subtree of the directory hierarchy to possess self-contained associations that are immune to the renaming of directories outside of the subtree. (Node renaming is described in Section 50.11.)

The ALS and the standard tools utilize a set of standard attributes and associations. Some of these attributes and associations contain information crucial to the integrity of the database, and are protected by the ALS. Users may change some of these protected attributes and associations with special tools. For example, the "chacc" tool is used to change the access_name attribute. Others, such as the creation_date attribute, may not be changed by users. There are no fixed limits to the sizes of attributes or associations.

These attributes are maintained by the KAPSE to control the database: node_type, creation_date, derivation_text, derivation_count, no_access, read, write, append, execute, via, attr_change, access_name, revision,

default_variation, purpose, locator, availability and archive_volume.

Three assocations maintained by the KAPSE are: derived_from, logged_inputs, and other_inputs.

The following attributes and associations are controlled by the KAPSE and cannot be directly modified by the user: node_type, creation_date, derivation_count, revision, access_name, availability, archive_volume, locator, derived_from, logged_inputs, and other_inputs.

The node_type attribute indicates whether a node is a file, a directory or a variation header. The only legal values for it are "file", "dir" and "var". It is assigned by the ALS when a node is created and cannot be altered.

The creation_date attribute contains the date and time that the node was created. It cannot be altered. The attribute names "reference_date" and "change_date" are reserved for use by automatic backup and archiving tools.

The access control attributes: read, write, append, execute, attr_change, via, and the access_name attribute, are described in Section 50.6. They may be freely modified but must obey the syntax specified in 50.6.

The revision attribute contains the revision number of a file. It has the form of an Ada integer literal. It is assigned by the ALS when a file is created and cannot be altered.

The default_variation attribute contains the identifier naming the default variation of a variation set. Only variation header nodes should have this attribute. It may be freely changed by users.

The purpose attribute tells why the node exists. It should be a sentence documenting the purpose of the node. (As opposed to the category which is a short description of the node's contents.) The purpose attribute may be freely changed. It is not used by the ALS or the standard tools; it is intended to provide a standard place for self-documentation of a node.

The locator attribute is a unique, invariant identification of an ALS node. Locators persist for the life of a node. Locators of deleted nodes may be reused.

The availability and archive_volume attributes are special attributes used in archiving (See 50.16). The availability attribute may have the following values:

1. On_line is the normal value.

2. Off_line indicates that the entire node (except for the derivation_count, availability, and archive_volume attributes) has been rolled out and is not available. Only file nodes

which have been revised or explicitly frozen may be given this attribute value.

3. Other possible values are reserved for internal use.

The archive_volume attribute has as its value the name of the archive tape which contains a copy of the node. It assumes a value the first time the availability attribute assumes the off_line value.

Other attributes and associations maintained by the KAPSE not described above are detailed in the next section.

50.7.1 Unique Identifiers Attribute.

A Unique Identifier Attribute (UID) is a name consisting of three fields:

a. Organization Name - Five characters for the Validation Number assigned at the time of validation by the ALS System Manager of the ALS Configuration Control Board. These are followed by five characters identifying the organization to which this copy of the ALS is delivered.

b. ALS ID - Seven characters distinguishing each ALS database established within the organization.

c. Object Serial Number - Ten characters identifying each object originated in the database.

Every validated ALS System is assigned a unique Validation Number. The ALS System Manager of the ALS Configuration Control Board maintains a list of all assigned Validation Numbers. Each successfully validated ALS is assigned a new Validation Number.

A unique Organization Name is assigned to each validated ALS by the person in charge of distributing the validated systems. The first five characters are the Validation Number supplied by the ALS Configuration Control Board. The distributing organization supplies the second five characters. The distributing organization must maintain a list of all the assigned Organization Names and guarantee their uniqueness.

The ALS IDs are assigned by the local ALS Administrators of each organization. Within each organization there will, again, have to be an authority whose job it is to assign a unique ALS ID for every database that is established.

Object Serial Numbers are assigned automatically by the ALS. If just numeric digits are used, ten digits yields enough numbers to create 100 objects per second, 24 hours per day, 365 days per year for about 30 years. Using upper and lowercase alphabetics as well, expands that

capability manyfold. Serial numbers are independent of the system clock, and are, therefore, not affected if the clock is set incorrectly, or by local time changes. Whenever 50 serial numbers have been used, the last number is written in two files, the Change File used for incremental backup and a special file called UID.DAT in VMS. When the ACP is restarted after a crash or planned reboot, the serial number is read from UID.DAT. Fifty is added to the number and the assignment of serial numbers continues. In the event of a catastrophe, the last serial number could be obtained from the Change File by the ALS Administrator and used to re-establish UID.DAT. If both files are lost, the ALS would have to be restored with the incremental or full backup. This may have been done anyway if the Change File is lost.

Everytime a new file revision, directory, or variation header is created, it is assigned a new UID by incrementing the Object Serial Number. New UIDs are not created when restoring objects that previously existed, by using the RESTORE tool, or when importing objects from other databases using the RECEIVE tool. The rationale for restoration is obvious; the rationale for importation is not so obvious.

By preserving UIDs in inter-ALS transfers, we are preserving the capability of doing configuration management across ALS boundaries. When moving revisions from one ALS to another, both the original revision on the sending side and the new copy on the receiving side are frozen. So, by comparing UIDs, a configuration manager can be confident that the same revision exists on both sides of the boundary. We believe that this capability will be essential for use of the ALS in network situations. It is also necessary to simply verify the correct installation of configuration-managed software.

50.8 Derivations.

Any ALS file can, potentially, possess a derivation. A derivation is a combination of attributes and associations that document the circumstances under which a file was created and subsequently modified. The purpose of derivations is to document the differences among files, showing why files differ rather than the exact text of the differences that would be obtained from a file comparison. Although derivations can be used to recreate files, they are not intended for database backup, but rather for configuration management.

Files in the ALS database can only be created or modified during the execution of a tool which calls KAPSE services. The derivation is an accounting of the conditions under which the tool executed. In particular, the name of the tool, the parameters passed to the tool, and files opened and read by the tool are automatically recorded in the derivation. Since the creating or modifying tool may "know" that some of this information is not significant and that other information may be more significant, the tool can modify the derivation.

The use of derivations causes files to be locked into the ALS database so that they cannot be deleted until all files that evolved from those files have been deleted. In certain situations, this makes deletion of unwanted files very difficult or impossible. For this reason, the use of derivations is recommended only for baseline objects under strict configuration management.

A derivation consists of the attributes: derivation_text and derivation_count, and the associations: derived_from, logged_inputs, and other_inputs. Every file that has a derivation possesses all of these attributes and associations which together constitute the derivation. Except for derivation_text, these attributes and derivations are controlled by the KAPSE and cannot be directly modified by a tool. The functions of these are:

derivation_text

> The value of this attribute is ASCII text conveying the name of the tool or tools creating or modifying the file, the parameters passed to those tools, and annotations posted by those tools. This attribute may be altered by tools.

derivation_count

> The value of this attribute is the character representation of an integer which is the count of the number of other files that were derived from the file possessing this attribute. More specifically, it is the total number of files in the ALS database that have this file in their derived_from association. If the derivation_count is greater than zero, the file may not be deleted from the database. If referencing files are deleted, the derivation_count is appropriately decremented. All files may have a derivation_count, even though they might not have other derivation attributes and associations.

derived_from

> This is a special association containing the locators of files appearing in the logged_inputs association. No other association has references that are not valid pathnames. Locators are immune to renaming. References in this association cause the incrementation of the derivation_count of the named file.

logged_inputs

> This association lists the full pathnames of files that were read by the process that created the file possessing this association. The named files must have been opened and read prior to the time the created or modified file was closed and the citation must not have been explicitly suppressed. Note that if a referenced node is subsequently renamed, the association reference will no longer be correct. The

derived_from association will still, however, be valid.

other_inputs

This association lists the full pathnames of files that were open and read by the process that created the file possessing this association, but were not entered in the logged_inputs association because they were explicitly suppressed by the tool. References in other_inputs do not cause incrementation of the derivation_count of the named file.

The derivation mechanism of the KAPSE will not automatically create derivations which are circularly dependent. Files involved in such derivations are impossible to delete from the database by ordinary means. Circularly dependent derivations can arise from the use of files opened for both input and output. To avoid this, the KAPSE will not automatically include in the logged_inputs association files that have been opened for both input and output. Such files will be included, instead, in the other_inputs association. Such references can be explicitly moved from other_inputs to logged_inputs. However, care must be taken to avoid a circular derivation dependence. Since the notion of files which are used for input and subsequently modified during tool execution is counter to good configuration management, this practice is strongly discouraged in tools that may be used in parts of the database under configuration management.

50.9 Node Deletion.

Offspring are deleted from directories and variation headers by the "delnode" tool. To delete a node, the user must have write access for the directory (or variation header) from which the node is being deleted. When a node is deleted from its true parent it is destroyed; all space allocated to it is recycled.

It is an error to attempt to delete a node that has a non-zero derivation_count.

Either individual revisions or entire revisons sets may be deleted. The deletion of an entire revision set will fail if any revision in the set has a non-zero derivation_count; the deletion of an individual revision will fail if the revision has a non-zero derivation_count.

If an individual revision being deleted is the only existing revision in the revision set, the entire revision set will automatically be deleted. For example, if alpha(3) were the only remaining revision of alpha, the deletion of alpha(3) would have the same effect as the deletion of alpha(*).

To delete either an individual revision or an entire revision set, the user must have write access to the directory containing the revision set.

The "delnode" tool is used to delete both revision sets and revisions. For example, the command

```
delnode (alpha(3))
```

deletes the third revision of alpha. The command

```
delnode (alpha(*))
```

deletes all revisions of alpha.

The delete tool will also delete directories and variation headers. If the node has any offspring (elements), the tool will ask for confirmation. If a positive response is received, the entire subtree will be deleted.

To delete a subtree, the user need only have write access for the directory containing the root of the subtree in its offspring list; no access is required for any of the directories to be deleted.

To delete a subtree, each node is visited in a post-order traversal. When visited, an attempt is made to delete the node as a single node. This may fail due to any of the constraints mentioned above, excluding the need for write access to the parent directory.

50.10 Node Sharing

The environment database supports the sharing of nodes among directories and variation headers. (Throughout this paragraph the word "directory" is used for brevity. It should be taken to mean "directory and variation header".) The "share" tool is used to share a node.

A node may have two kinds of parents. The directory in which the node is first created is the node's true parent. Every node has exactly one true parent. Directories which share a node are foster parents of the node. There is no fixed limit to the number of foster parents that a node may possess.

Figure 50-8a shows a hierarchy with node sharing. A hierarchy like the one in Figure 50-6 is shown, but with the sin, cos, and taylor procedures the same for both forms of floating point hardware. The flt_pt (long) directory is the true parent of these three files, and the flt_pt (short) directory is a foster parent.

As mentioned in Section 50.3, every node contains information that identifies its true parent directory. Shared nodes also contain information specifying all foster parents of the node.

Since directories specify the names of their offspring, the same node may appear in different directories under different names. This allows the foster parents of a node to use their own local names for the shared node, if desired. For example, the foster parent may already have a node with the same name as the node to be shared, or the user may wish to denote that he is sharing the node. To illustrate, Project A may wish to share Project B's flight navigation package and refer to it by the name "proj_B_navpak". The command to do this would be:

```
share (.project_B.flight_navigation, .project_A.proj_B_navpak)
```

Sharing of both revision sets and directories is permitted. However, a directory may not have itself as a descendant. Only entire revisions sets may be shared, not individual revisions in a set.

The "delnode" tool is used to delete offspring from a directory. Deletion of a shared offspring from a foster parent directory does not destroy the shared node. A node is destroyed, however, when it is deleted from its true parent directory. It is illegal to delete a node from its true parent directory if it is shared by other directories.

A shared offspring, which is a directory, variation header, or revision set, may always be deleted since its deletion does not cause the node to be physically destroyed. The derivation_count, availability, or number of offspring of the shared node are not checked. The user need onlyhave write access to the foster parent from which the node is being deleted.

Individual revisions, however may be deleted only when the revision set to which they belong is not shared by other directories. The revision set must have a single parent. For example, if the revision set .A.B.C were shared by .X.Y it would be illegal to delete an individual revision of C. In other words, if the command

```
share (.A.B.C, .X.Y)
```

were entered, the revision set C would have two parent directories, the true parent .A.B and the foster parent .X . In this case, illustrated in Figure 50-8b(a), the revision set has two names, .A.B.C and the alias .X.Y.

It is legal to delete individual revisions if the revision set is only indirectly shared by virtue of the parent of the revision set or one of its ancestors being shared. For example, if the directory .A.B is shared by .X.Y , then the revision set .A.B.C is indirectly shared, and has the alias .X.Y.C . In other words , if the command

```
share(.A.B, .X.Y)
```

were entered, the revision set C would have one parent, .A.B . In this case, illustrated by Figure 50-8b(b), the revison set has two names .A.B.C and .X.Y.C. Here, individual revisions of C can be deleted since the revision set has a single parent, which happens to be shared.

Sharing is always resolved to a "real" node. For example, if alpha.beta.gamma is a file, a user may type

```
share (alpha.beta.gamma, delta.epsilon)
```

Then delta.epsilon is not a real node but is a "link" to alpha.beta.gamma. If another user then types

```
share (delta.epsilon, zeta)
```

zeta is a "link" to alpha.beta.gamma also. The effect is the same as typing

```
share (alpha.beta.gamma, zeta)
```

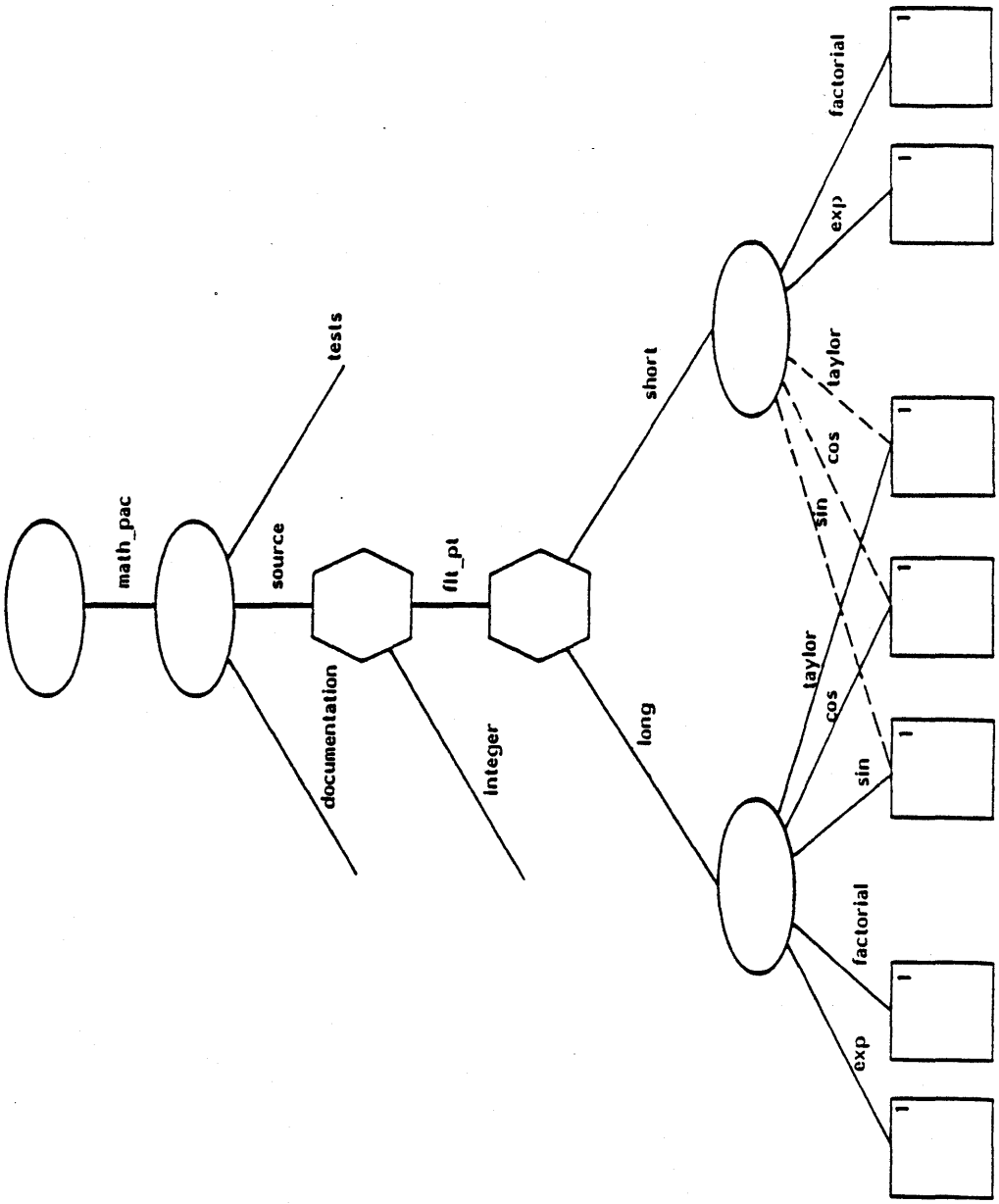
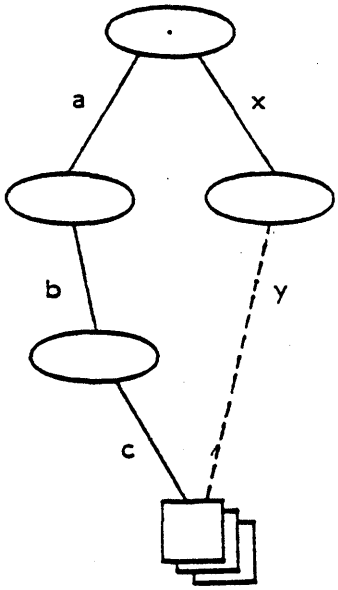
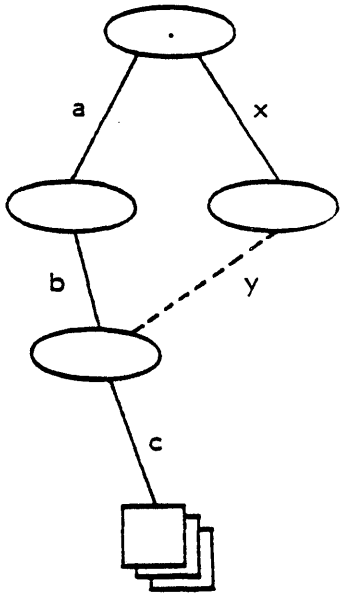


Figure 50-8a. Node Sharing Example



Individual revisions may not be deleted.

(a)



Individual revisions may be deleted.

(b)

Figure 50-8b. Revision Deletion with Sharing

50.11 Node Renaming

The offspring of directories may be freely renamed by the use of the "rename" tool. (Throughout this paragraph the word "directory" is used for brevity. It should be taken to mean "directory and variation header".) For example, the command

```
rename (alpha, beta)
```

renames node alpha in the CWD as beta.

Although a node cannot be destroyed if it is shared, it may be renamed in its true parent or a foster parent directory. The renaming has no effect on any directory except the one in which the node is renamed.

Association references pointing to a node will be "voided" if the node is renamed. As mentioned in Section 50.7, associations can contain references to non-existent nodes. One way in which such references arise is from a referenced node being renamed.

The rename command can also be used to move a node from one directory to another. For example,

```
rename (alpha.beta.gamma, .delta.epsilon)
```

moves the node named "gamma" from the directory "alpha.beta" under the CWD to the directory "delta" under the Root. The node's name within delta is "epsilon".

A node may be moved from its true parent directory or a foster parent directory. The move has no effect on any directories except for the two involved in the move.

Individual revisions can also be renamed. This, in effect, deletes the individual revision from one revision set and renames it to the latest revision in the new revision set. For example, the command

```
rename (alpha(3), beta)
```

moves the revision alpha(3) to the latest revision of beta. All attributes and associations of alpha(3) are moved together with its data portion to the target revision set, beta. The node that was alpha(3) becomes the latest revision in the revision set, beta. Standard rules apply for creating a new revision or overwriting the latest existing revision of the target revision set. It is illegal to specify a revision number on the name of the target revison set.

50.12 Path Name Details

Most of the syntax and semantics for path names has been presented in previous paragraphs of this appendix. This paragraph presents a complete collected syntax for names and introduces a few specialized forms.

As mentioned previously, if a node name begins with a period the search starts at the Root; otherwise it starts at the current working directory. A node name beginning with an exclamation point (!) searches upwards from the CWD. For example, if the CWD is .alpha.beta.delta, the name !.gamma is the same as .alpha.beta.gamma, !.!.gamma is the same as .alpha.gamma, etc.

A name of the form "<<VMS>>string" selects a file from the VAX/VMS file system. The content of the string is the VAX/VMS file name. This form of name may only be used in a context where the file is to be opened for reading or writing. For example, the command

```
lst ("<<VMS>>DB1:[EXAMPLES]CALENDAR.;1")
```

lists the contents of the VAX/VMS file DB1:[EXAMPLES]CALENDAR.;1 on the user's terminal. Note the upper and lower case are equivalent in VAX/VMS file names. The above command could also be written

```
lst ("<<vms>>db1:[examples]calendar.;1")
```

The syntax for path names is:

```
object_name ::= path_name
              | <<VMS>> character_string

path_name   ::= . [node_id {. node_id} [revision_index]]
              | { ! .} node_id {. node_id} [revision_index]
              |  ! {. ! }

node_id     ::= identifier {variation_index} [default_index]

variation_index ::= (identifier)
                  | (attr_spec{,attr_spec})

default_index ::= ()

revision_index ::= (integer) | (+) | (*)
```

attr_spec ::= identifier => attr_value

Identifiers, integers, and character strings are as defined by Par. 2.3, 2.4, and 2.6 of the Reference Manual for the Ada Programming Language (2.1). Upper-and lower-case characters are equivalent in identifiers and in the word <<VMS>>. They are also equivalent in the (*) or (+) revision index. Identifiers are limited to a 20-character maximum length. The entire path name is limited to a maximum length of 65,535 characters. (This includes the length of the CWD for relative path names.)

Embedded blanks are not allowed in path names. The grammar does not show it but adjacent periods are allowed and are treated as a single period. This is to facilitate the mechanical generation of pieces of path names.

A default index may be applied to a file or directory (in addition to a variation header). In these two cases it has no effect and can be ignored. For example, if alpha is the name of a file, "alpha()" is the same as "alpha", and "alpha()(3)" is the same as "alpha(3)". This feature facilitates mechanical generation of names in cases where variations may or may not be present and where the default would be wanted if a variation is present. The name generator may safely act as though variations are always present.

The meaning of the revision_index has been presented in Section 50.2 of this appendix; the meanings of variation_index, default_index and attr_spec in Section 50.5.

The "attr_value" is an optionally quoted sequence of characters. If the value is not enclosed in quotes it starts at the first non-blank character after the arrow (=>) and ends at the next blank, comma, or right parenthesis that is not inside nested parentheses.

50.13 Program libraries

A program library (PL) is a subtree in which Containers are stored. A PL holds the Containers comprising a single Ada program. (Program libraries are discussed briefly in Section 10.4 of the Ada Language Reference Manual, 2.1). The only objects in a PL are Containers produced by the compilers, assemblers, importers and linkers, and the internal directory nodes which provide links between the Containers. Ada source files are not stored in a PL.

The PL is used to support the separate compilation facility of Ada and to enforce Ada's order of compilation rules (see Section 10.3 of the Ada Language Reference Manual, 2.1). The compiler can do type checking across compilation units since it has access to all previously compiled units in the PL. It enforces the order of compilation rules by verifying that all required compilation units are in the PL.

A user may have several PL's, one for each Ada program. All of the Containers comprising one program must be stored in the same PL. One PL can be used to store Containers for more than one program, although such use violates the intent of program libraries.

Each PL has one target environment associated with it. Every Container within a PL must be targeted for the same target environment. Users are not allowed to place Containers intended for different targets into the PL. In the case where multiple targets share a common hardware architecture, it may be that a compiled Container can be targeted for more than one target environment. The pragma SYSTEM allows the user to direct the targets for which a Container can be used. (See Appendix 10 for a discussion of pragma SYSTEM). When a Container is suitable for more than one target, it can be inserted into a PL as long as one of the targets for which it is intended is the target of the PL. For example, a compiled Container created with a Pragma SYSTEM (VAX780) source line, could be inserted into a PL intended for either the VAX780_VMS target environment or the VAX780_SA target environment.

Each supported target enviroment has a System Program Library (SPL). SPL's hold compiled packages that are automatically acquired (see 50-13.2) when other program libraries are created. All SPL's contain the following packages defined in the Ada Reference Manual:

```
STANDARD                    SEQUENTIAL_IO
   ASCII                    DIRECT_IO
CALENDAR                    TEXT_IO
SYSTEM                      IO_EXCEPTIONS
MACHINE_CODE
UNCHECKED_DEALLOCATION
UNCHECKED_CONVERSION
```

SLP's for targets which are also hosts contain the following additional packages of the KAPSE:

```
AUX_IO                      PARM_LIST
BASIC_IO                    PROG_CALL
COM_DEF                     PROG_CONTROL
FILE_DERIV                  PROG_DEFS
HOST_ESCAPE                 PROG_STRINGS
ID_DEFS                     STANDARD_NAMES
KAPSE_COM                   STRING_DEFS
KAPSE_DEFS                  STR_CONVERT
MISC_DEFS                   STRING_UTIL
MISC_SERVICE                TAPE_IO
```

SPL's also contain packages not directly visible to the Ada programmer. These packages are used for run-time support and for lower level KAPSE support. SPL's are stored in the variation set .als_tools.program_library. There is one variation for each supported target. The SPL for the VMS host is named .als_tools.program_library(vax780_vms).

Typical tools which operate on a PL are the compilers, linkers, assemblers, importers, exporters, and the interactive program library manager tool, LIB. LIB allows the user to acquire,delete, examine and manipulate the Containers in a PL. LIB is discussed in Section 50.13.4. All of these tools operate with respect to a particular PL. They use a single PL from which to retrieve Container input and into which to place Container output.

50.13.1 PL structure

Each PL is a directory offspring of a user's directory. Users create PL's with the MKLIB subcommand of LIB. Variations of PL's are allowed, but the user must create the variation header before creating the first variant PL.

The internal structure of a PL is a tree structure which captures the logical heirarchy of the source compilation units. The root of the tree is a directory node of category "program_library". The ALS sets the via attribute of this node to restrict access to the PL. Users must use tools which operate on PLs to access the PL root node and all the nodes beneath it. The user controls other users' access to the PL with the no_access, read, write, append, attr_change, and execute access attributes of the root node. These attributes can be altered with the CHATTR subcommand of LIB.

Files in the PL tree are Containers. Subtrees are either library unit or subunit subtrees. A library unit subtree contains the specification and body Containers for the library unit plus a subunit subtree for each of the body's subunits. A subunit subtree contains the body Container of the subunit plus a subunit subtree for each of the subunit's body's subunits. Containers and subtrees in a PL are referred to by their Adanames, defined below.

The Adaname of a compiled Container is determined by its corresponding source unit name qualified by the keywords SPEC or BODY. The Adaname of a Container of a library unit specification is the Ada source unit name qualified by the keyword SPEC. For example, the Adaname of the Container for the specification of the library unit TOP is TOP.SPEC (Any combination of lower and upper case represents the same name.) The Adaname of the Container for a library unit body or subunit body is the Ada source unit name qualified by the keyword BODY. For example, the Adaname of the Container for the body of subunit A of the library unit TOP is TOP.A.BODY.

The Adaname of a linked Container is the outputname parameter given to the linker when the Container was created. This name cannot duplicate the name of a library unit already in the PL. Thus if the user specified "TOP_LNK" as the output_name, the Adaname of the resulting linked Container is TOP_LNK.

The Adaname of a library unit or subunit subtree is determined by the corresponding Ada source unit name qualified by the keyword ALL. For example, the Adaname of the subtree of the library unit TOP is TOP.ALL. The Adaname of the subunit subtree of subunit A of the library unit TOP is TOP.A.ALL. Similarly, the Adaname of an entire program library is the qualifier ALL with no library unit or subunit name; that is, the Adaname of every program library is ".ALL". ALL is the default whenever the Adaname does not include any of the keywords SPEC, BODY or ALL. Thus the Adaname TOP.A is the same subunit subtree as TOP.A.ALL and the Adaname TOP is the same library unit subtree as TOP.ALL.

The Adaname of the compiled Container for the predefined package STANDARD is special, since STANDARD is the only package which is not a library unit. The Adaname for STANDARD is STANDARD.PACKAGE. Since this Adaname is different from library unit Adanames, users can have a library unit named STANDARD without causing ambiguity. Its Containers would be called STANDARD.SPEC and STANDARD.BODY.

The Adaname of a Container can include a revision number. When a revision number is not included, the latest revision is assumed. Revision numbers are not allowed on Adanames which specify library unit or subunit subtrees.

50.13.1.1 Revisions of Containers

When a tool such as the compiler or linker produces a Container, this new Container either becomes the next revision of the Container, or overwrites the latest revision of the Container. A new revision of a Container is created only when the existing latest revision has been used to derive another Container. When the latest revision has not been used to derive any Container, it is replaced by the new Container. The Ada compilation order rules determine when Containers are used in derivations. For instance, a library unit's specification Container is used to derive the library unit's body Container and Containers of compilation units which mention that library unit in a WITH clause; and a body's Container is used to derive the Containers of its separately compiled subunits. Additionally all Containers used in a link are used to derive the linked Container and a shared Container is used to derive the Containers which share it (sharing is discussed in Section 50.13.2.) The following example demonstrates when revisions are created and replaced.

a. The specification of TOP is compiled for the first time.

   (The Container TOP.SPEC(1) is created.)

b. The body of TOP is compiled for the first time.

   (The Container TOP.BODY(1) is created.)

c. The specification of TOP is recompiled.

(A new revision of the Container top.spec is created. This is TOP.SPEC(2). ((TOP.SPEC(1) was used to derive TOP.BODY.))

d. The body of TOP is recompiled.

(The new Container for TOP.BODY replaces the first revision. This is still TOP.BODY(1). ((The original TOP.BODY(1) was not used in any derivation.))

e. Subunit A of TOP is compiled for the first time.

(The first revision of the Container TOP.A.BODY is created.)

f. The body of TOP is recompiled.

(A new revision of the Container TOP.BODY is created. This is TOP.BODY(2). (TOP.BODY(1) was used to derive TOP.A.BODY).

50.13.2 Sharing Containers

Containers may be shared across PL boundaries thereby sparing users time-consuming recompilations and relinks and saving storage space. For this discussion, the term "acquired Container" refers to the Container originally produced by a compilation or link. The term, "acquiring Container" refers to the Container produced by the sharing operation. The term "acquiring PL" refers to the PL which contains the acquiring Container.

Sharing in a PL is different from node-sharing in that sharing only involves the data portion of a Container. Even an acquiring Container exists as a separate object. It has its own attributes, associations and parent. The data portion of an acquiring Container, however, is empty. Whenever a reference is made to the data portion of the acquiring Container, the data is automatically retrieved from the acquired Container.

Containers are shared only upon the request of the user. The user accomplishes this with the ACQUIRE subcommand of LIB. Container sharing applies only to individual revisions, not to a revision set. The user either explicitly or implicitly shares a specific revision of a Container. (If he does not specify a revision on his ACQUIRE request, the latest revision is assumed.) Thus if a new revision of an acquired Container is created, the user must issue another ACQUIRE subcommand to acquire the new revision. Unless he does so, he will continue to share only the old revision of the Container. Sharing is always done between acquired and acquiring Containers. If the user acquires a Container that is itself an acquiring Container, that acquisition is drawn from the PL that has the acquired Container. The intermediate acquiring Container is not acquired.

A user can create his own revision of the acquiring Container by recompiling or relinking into his own PL. When he does this, the rules for creating new revisions of Containers apply. If the acquiring Container has been used in a derivation, the acquiring Container remains in the PL and the new revision becomes the next revision. If the acquiring Container has not been used in a derivation, the new revision of the Container replaces it and all the sharing links are deleted. In either case, the user's revision will be a complete Container, not an acquiring Container.

Sharing is permitted between Containers in any PL as long as the acquired Container was created for a target compatible with the acquring PL. Containers created for the same target as the acquiring PL are always compatible. Thus any Container in a PL for the VAX780_VMS target can be acquired by another PL for the VAX780_VMS target. Users can force Containers to be compatible with additional targets by using the SYSTEM pragma in the source unit. For example, if the user included the pragma SYSTEM VAX780 in a source unit, its compiled Container could be acquired by either a PL for the VAX780_VMS target or a PL for the VAX780_SA target.

The PL keeps track of sharing by the absolute pathnames of the acquired and acquiring Containers. The user must use caution whenever he changes the pathname of a PL, since doing so could cause errors when shared Containers are referenced.

50.13.3 Attributes and Associations

Containers within a PL have attributes and associations. These are maintained solely by the tools which operate on the PL. Users may examine the contents of attributes and associations with the LSTASS and LSTATTR subcommands of LIB, but they are not allowed to delete or change them. The pathnames referenced in associations or attributes are relative to the PL root node if they refer to Containers within the same PL and are absolute if they refer to Containers within other PLs.

Each Container in a PL has all of the standard attributes and associations possessed by other database nodes. (See Section 50.7 for a discussion of attributes and associations). Those intended for the exclusive use of a PL are the depends_on, referenced_by and acquiring_containers associations and the acquired_data and compatible_targets attributes.

A Container's depends_on association contains the names of all Containers which were required for its creation. The Containers appearing in this association are a subset of those appearing in the Container's derivation history. Containers for a body's specification, a library unit named in a WITH clause, and a body for a separately compiled subunit might all appear in a depends_on association. For example, this association in the Container TOP.BODY lists the names of all the library units or subunits which must be acquired before acquiring TOP.BODY.

A Container's referenced_by association contains the names of all compiled Containers which depend on that Container. This relationship is the inverse of the depends_on association. However, unlike the depends_on association, the referenced_by association names only compiled Containers. Linked, acquired, and other Containers are not included. For example, this association in the Container TOP.SPEC lists the library units and subunits which must be recompiled once TOP.SPEC is recompiled.

A Container's acquiring_containers association lists all the acquiring Containers in other PLs which share the data portion of the Container.

A Container's acquired_data attribute names the acquired Container in another PL whose data it is sharing. This attribute appears in all acquiring Containers.

A Container's compatible_targets attribute names each target for which the Container might be used. As explained in Section 50.13, compiled Containers can have multiple targets for which they are compatible.

Every node within a PL has a category attribute. A Container's category attribute always has the value "Container".

A Container's creator attribute describes the class of tools that created the Container. The creator attribute has values such as "compiler", "linker", "assembler",etc.

The PL root node also has all the standard associations and attributes. Of special interest is the target attribute, which names the intended target of all the Containers in the PL. A Container can be shared only if the value of the target attribute of the acquiring PL appears in the compatible_targets attribute of the acquired Container.

50.13.4 LIB

LIB is the interactive program library manager tool which allows the user to create, delete, examine and manipulate the contents of a PL. Its subcommands are MKLIB, ACQUIRE, DELETE, CHATTR, LST, LSTASS, LSTATTR, ARCHIVE and UNARCHIVE.

The subcommand MKLIB allows the user to create a new PL. When the PL is created, it automatically acquires the predefined package STANDARD, and the runtime support library for the target named by the user.

The subcommand ACQUIRE provides the PL sharing facility. It sets up links between the acquired Container and acquiring Container by wr ting the acquired_data attribute in the acquiring Container and appending to the acquiring_Containers association in the acquired Container. It also

adjusts other associations and attributes so that pathnames refer to the appropriate revisions in the acquiring PL. The user can acquire either Containers or subtrees. He must acquire Containers according to compilation order rules, so that Containers on which the acquired Container depends already reside in the acquiring PL. When a library unit or subunit subtree is acquired, the latest revision of every Container in the subtree is acquired. Acquiring a subtree is simply shorthand for acquiring each of the individual Containers. A subtree being acquired must be internally compatible; that is, Containers within it must depend on only the latest revision of other Containers within the same subtree. For instance, a library unit's body's Container must depend on the latest revision of the specification's Container. When this is not the case, it implies that the subtree is in the process of being recompiled and is currently not suitable for acquisition.

The DELETE subcommand allows the user to delete Containers from a PL. The user can delete subtrees, entire revision sets of Containers, or individual Container revisions. When a library unit or subunit subtree is deleted, all of the Containers within the subtree are deleted. Deletion of a Container is allowed only when its derivation_count is zero. The derivation_count is non-zero when the Container has been used to derive another Container.

The CHATTR subcommand allows the user to change the no_access, read, append, write, attr_change, and execute attributes of the PL root node, enabling users to control access to PLs.

The LST subcommand displays a directory of Adanames within a PL beginning at a point specified by the user.

The LSTASS subcommand displays the values of associations of Containers. Additionally, it can be used to list all of the associations held by a Container. The user can display the associations of a single Container or of all the Containers within a subtree.

The LSTATTR subcommand displays the values of attributes of Containers in a PL. Additionally it can be used to list all the attributes held by a Container. The user can display the information for a single Container or for all the Containers within a subtree.

The ARCHIVE subcommand sends a list of Containers to be archived to the ALS. (See Section 3.7.11 for a discussion of the File Administrator.) Only specific revisions of Containers can be archived.

The UNARCHIVE subcommand sends a list of Containers to be unarchived to the ALS. (See Section 3.7.11.) Only specific revisions of Containers can be unarchived.

50.14 HELP Database

The HELP database holds information on how to use the ALS as well as installation-specific subjects. Two tools, HELP and QHELP, help the user to extract information from the database. HELP is an interactive tool which allows the user to explore the database, providing easy access to extensive information. QHELP is a non-interactive tool which supplies quick access to a single piece of information.

The HELP database is designed so that users may easily extend it to include installation or project specific information. The system HELP database is stored as a subtree in the directory, .ALS_TOOLS. The root of the subtree is the directory HELP_DATA. The predefined global substitutor, CHD (current help directory), identifies the HELP database to be used by the HELP and QHELP tools. Its default value, citing the system HELP database, is ".ALS_TOOLS.HELP_DATA". If the user wishes to have his own HELP database, he creates a HELP subtree in his own directory and changes the value of CHD to his HELP directory node. A user's HELP database subtree should share all of the offspring of the system HELP_DATA directory so that a user has access to the system information.

The HELP_DATA database is a simple tree data structure (see Figure 50-9) where each subtree describes one subject. The root of each subtree has an arbitrary number of directory offspring and either one or two file offspring. These files are named "information" and "syntax". The "information" file contains information text and exists for all subtrees in the HELP database. The "syntax" file contains syntax information and exists only for those subtrees for which such information is relevant. Each offspring directory is a subtree describing a subtopic of the subject. There is no fixed limit on either the nesting depth of subtrees or on the number of subjects. When the user extends the HELP database, he normally adds new subject subtrees which are direct offspring of the HELP_DATA root.

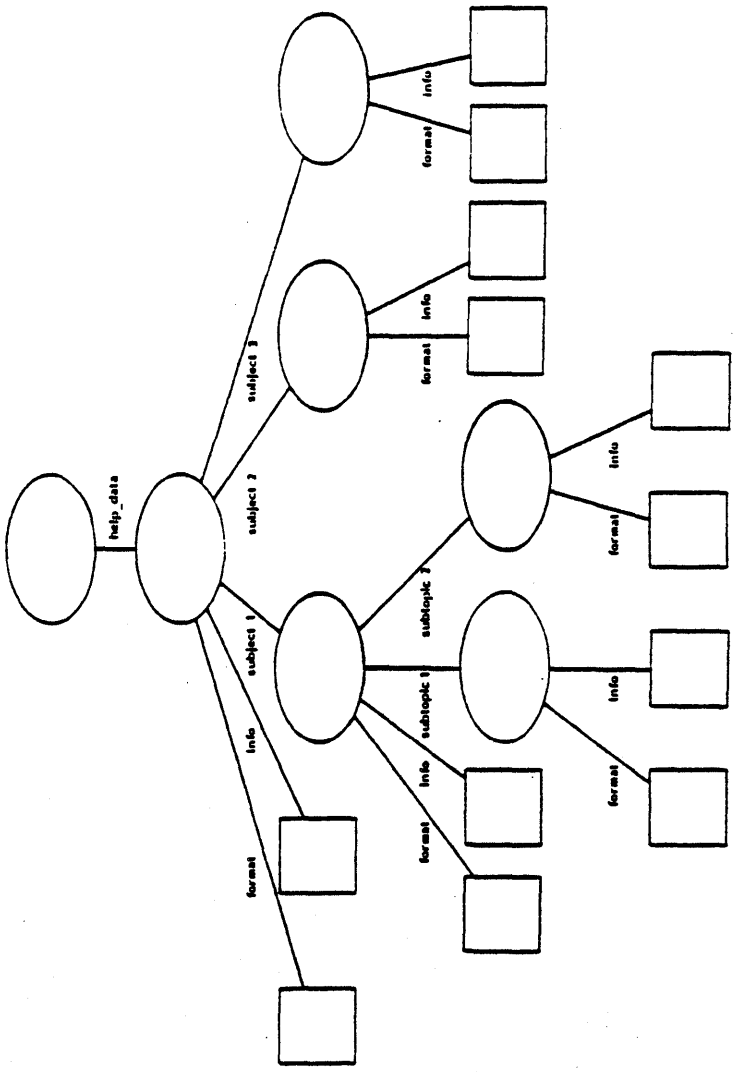


Figure 50-9. HELP_DATA Example

50.15 Subtree Transmission

The "transmit" and "receive" tools provide a method for communicating subtrees of the on-line portion of the environment database of a particular ALS host to magnetic tape, and then, using that information, to create identical subtrees within the environment database of the same or of a different ALS host.

The format of tapes written by the transmit tool and read by the receive tool complies with a host-independent subtree format definition, which permits porting of subtrees of ALS databases between ALS hosts. (Non-text files may not be portable to another host or to another revision of the ALS on the same host). These tapes may also be used for long-term storage of snapshots of the progress of a development project.

50.16 Archiving

Economic considerations require that the totality of a large ALS environment database not be entirely on line but be balanced between the on-line disks and a magnetic tape library associated with the host. Since the access time to information stored on tape is much larger than the access time to information on disk, the information chosen to be stored on tape is generally that which is less frequently used. The process by which certain parts of the database are saved on tape in such a way that they can be recalled to disk when they need to be accessed is called "archiving". The actual process of writing the contents of a node on tape and reclaiming the disk space used by that node is called "rollout"; its inverse is called "rollin".

Only file nodes which have been revised or explicitly frozen may be archived. When a specific file node is rolled out, its data, its associations, and its attributes (except for availability, archive_volume, and derivation_count) become inaccessible until that node is rolled back in, at which time the values of the data, associations and attributes are the same as before the rollout, with the possible exception of the archive_volume attribute, which may have been changed as a result of the rollout, and the derivation_count attribute, which may have been decremented as a result of the deletion of another node.

The choice of times at which rollout and rollin take place is a matter of installation policy; for example, rollout and rollin may be performed periodically. The "archive" and "unarchive" commands give users a mail facility with which they may send lists of file pathnames to system files which can be used by operational personnel as input to rollin and rollout operations.

50.17 Backup

There is a set of facilities within the ALS which provide a line of defense against loss of a portion of the environment database due to equipment failure or improper operation. This line of defense consists of copying portions of the database to tape, at times which depend on installation policy. It is possible to make a tape backup copy of the entire database (full backup), of specified subtrees of the database (tree backup), or only of nodes which have been changed (incremental backup) since the previous full or incremental backup. The recovery function can then rebuild the database (or a portion of it) from a full (or tree) backup plus incremental backups; or node-by-node reconstruction is possible in the case that damage is limited and recovery action needs to be directed to a specific set of nodes.

The backup and recovery tools are provided for operational support and are not available to users in general. It is important, however, that users be aware that operation of a host installation includes these lines of defense.

APPENDIX 60

60. ADA LANGUAGE SYSTEM COMMAND LANGUAGE.

The Command Language (CL) provides a single, uniform interface between the user and the tools in the Ada Language System (ALS). The command language is interpreted by the Command Language Processor (CLP). The CLP is used to invoke other tools as well as other instances of itself.

60.1 Structure of an ALS Session.

The command language used from the time a user enters the ALS to the time he leaves is called an ALS Session. A session is divided into a series of command streams. A command stream begins:

a. When the user enters the ALS from the VMS using the VMS command, ALS, and

b. Each time, a command procedure is invoked. (Note: A command procedure is an ALS command sequence stored in the environment database for subsequent invocation. Command sequence is defined in 60.3.)

A command stream ends:

c. When the user leaves the ALS via the EXITALS Break-In command,

d. When an end-of-file is encountered, or

e. When the RETURN command is interpreted.

The command stream established at the time the user enters the ALS is called the initial command stream. When the initial stream ends due to conditions (c), (d) or (e) above, the ALS session is terminated.

A command stream is composed of commands. Each command is composed of tokens. As the CLP reads the command stream, it identifies the token and command boundaries. If any token or command boundary is ambiguous, a syntax error will be diagnosed.

During the process of identifying tokens, the CLP also performs string substitution. This process is similar to the process commonly called macro substitution in assembler languages and some high level programming languages. The rules for defining and using string substitution are outlined below in greater detail.

The CLP also performs expression evaluation. Unlike programming languages, the operands in an expression may only be literal strings. However, since string substitution is performed before expression evaluation, substitution can be used much the same way that variables are used in a programming language. Substitution can also be used in ways that variables would not. A substituted string can, for example, be an entire command, an expression to be evaluated, an entire argument list, or even the the name of one or more strings to be substituted again. In using the CL, it is important to bear in mind the order in which the steps of command processing occur, specifically:

1. Token acquisition i.e., get the next token from the keyboard or other device, or from a database node, or from the internal stack of the CLP (loops). String substitution takes place as a by-product of token acquisition.

2. Parsing.

3. Expression evaluation. (This involves literal strings only.)

4. Execution.

Commonly used sequences of commands can be stored in database files and subsequently invoked as any other tool would be invoked. The invocation of command procedures is indistinguishable from the invocation of tools written in Ada.

Sections 60.6.1, 60.6.2, 60.6.4, and 60.7 describe how information is passed to and from command procedures and other tools. Each command procedure must be stored in a separate file. Command procedures may invoke other command procedures, even themselves, to an arbitrary depth.

When an error occurs, an error message is generated. Some of the conditions that result in error messages are described throughout this chapter. The procedure for handling errors is described in Section 60.11. A list of the diagnostic messages produced by the Command Language Processor is provided in Appendix 80 of this specification.

60.2 Basic Language Elements.

60.2.1 Character Set.

The character set for the command language is the same as defined in Par. 2.1 of the Reference Manual for the Ada Programming Language, July 1980, called the "Ada Reference Manual".

60.2.2 Lexical Units and Spacing Conventions.

The tokens recognized by the CLP are delimiters, reserved words, and character string literals. A delimiter is any of the following special characters in the basic character set:

& ( ) * + - / ; " : = < > # , . '

or any of the following compound symbols:

:= /= >= <= *= *< *> */= *<= *>= => **

Adjacent tokens are separated by spaces, delimiters, or by the end of a command. An identifier or character string literal must be separated in this way from an adjacent identifier or character string literal.

60.2.3 Identifiers.

Identifiers are used for substitutor names (see Section 60.2.9) and reserved words. Command language identifiers conform to the rules for identifiers specified in Par. 2.3 of the Ada Reference Manual, with the additional restriction that the length of an identifier may not exceed 20 characters.

60.2.4 Character String Literals.

Character string literals conform to the rules cited in Par. 2.6 of the Ada Reference Manual. For convenience, the Ada string bracket character (") may be omitted except in the following cases:

1. To treat a reserved word as a string, and

2. To include spaces, quotation marks, other delimiters, or unprintable control characters in a character string literal.

The value of a quoted literal is the string contained within the quotes. The term character string is used to denote this construct.

60.2.5 Integer String Literals.

In the CL, some character string literals are converted to integers during expression evaluation. The values of such character string literals are expected to conform to the Ada rules for integer literals given in Par. 2.4 of the Ada Reference Manual. The term integer string is used for this construct. Integers must lie in the range plus or minus 2,147,483,647, i.e., 32 bit signed arithmetic.

60.2.6 Boolean String Literals.

Some operators in the CL require Boolean strings. The Boolean strings are:

TRUE and FALSE

Boolean strings may be represented in any combination of corresponding upper and lower case characters. The term Boolean string is used for this construct.

60.2.7 Comments.

A comment starts with two hyphens and is terminated by the end of a command line. It has no effect on a command.

60.2.8 Reserved Words.

The following identifiers are reserved for use in the command language.

ADD AND CLP_OPT ELSE ELSIF END EXIT GLOBAL IF IN INLINE LOOP

MOD MSG NOT NOWAIT NULL OR OUT REM RETURN THEN WHEN WHILE XOR

Reserved words are recognized without regard to upper or lower case.

60.2.9 Substitutors.

In the command language, substitutor identifiers, or simply substitutors, are used to denote string substitution. Suppose that the substitutor name "short" has been associated with a string value e.g. "long_file_name" by use of the assign command (see Section 60.5.). When the substitutor name subsequently appears in a command, prefixed by a "#", the name is replaced by the value of the string for which it stands prior to any syntactical interpretation, expression evaluation, or execution. For example, given the following sequence of commands:

```
short := long_file_name
cpydata (#short, myfile)
```

The command that is executed is:

```
CPYDATA (long_file_name, myfile)
```

60.2.9.1 String Substitution Rules.

Substitution occurs as an integral part of token acquisition. Substitution is performed by the following algorithm:

1. The command is scanned left-to-right, one character at a time.

2. When a sharp sign (#) is encountered, the next character is examined. If it is another sharp sign, the pair of sharps is replaced by a single sharp and scanning continues at the character after the second sharp sign with no substitution performed.

3. If the character after the first sharp sign is a left parenthesis or letter string substitution is recognized. Any other character is a syntax error, causing the command to fail.

4. When string substitution is recognized, the substitutor identifier is isolated by scanning to the right until a delimiter is found. If substitution was introduced with a left parenthesis, any delimiter other than right parenthesis or sharp sign results in a syntax error and subsequent command failure. In substitution of parenthesized identifiers, the parentheses are absorbed, otherwise only the sharp sign and identifier are absorbed. If the identifier has been defined (for example, by an earlier assignment statement), then the appropriate string is substituted. If the identifier has not been defined, then the null string is substituted. The substituted string is then reexamined for possible further string substitution.

5. The substitutor identifier itself may require string substitution, in which case the inner substitution is performed first.

6. The string which is substituted is scanned starting with the first substituted character. Any substitutions required are performed. Since recursive string substitution does not terminate, the CLP detects this condition which subsequently causes command failure.

7. Recursive substitution is defined as occurring when the expanded value for a substitutor contains a substitution for itself.

8. The value of a substitutor may not be longer than 65,535 characters.

The substitution mechanism is activated for any token containing the flag character, #. String substitution occurs within quoted literals, but not within comments. Corresponding upper and lower case letters are equivalent in substitutor identifiers. Section 60.6.2 contains additional information about substitutors.

Error Conditions

The following conditions cause a syntax error to be recognized:

Unbalanced parentheses
Recursion
Identifier too long
Ill-formed identifier

Examples

```
-- Given:

     short :=  "long_file_name"

-- Then:

     #short         --  results in long_file_name
     this.#short    --  results in this.long_file_name
     that#short     --  results in thatlong_file_name
     #(short)that   --  results in long_file_namethat
     ##short        --  results in #short

-- Given:

     XYZ  :=  "##(YY)"
     YY   :=  "23"

-- Then:
```

```
ZZ#(XYZ)X      -- results in ZZ23X
               -- note that the value of XYZ is #(YY)
```

-- Equivalently, given:

```
XYZ  :=  YY
YY   :=  23
```

-- Then:

```
ZZ#(#(XYZ))X  -- results in ZZ23X
```

60.2.9.2 Sharing Substitutors.

There are three types of substitutors:

a. Global,

b. Local, and

c. Local inherited.

Global substitutors are visible to all command streams in which they have been declared in a GLOBAL command (see 60.11). The value of a global substitutor set in one command stream is available in other command streams that are in the same ALS session and that have been initiated after the value was set. There is no guarantee that the altered value of a global substitutor will be available in command streams that are initiated prior to the time the value is changed. Global substitutors should not be used to achieve asynchronous communication between command procedures. Any command procedure which depends upon the value of a global substitutor changing during the execution of that procedure is erroneous in the sense of the Ada Reference Manual.

Local substitutors are visible only in the command stream in which they are used. They are declared implicitly by their use as parameters, see 60.6.1, or by appearing on the left-hand-side of an ASSIGN command, see 60.5. The execution of a GLOBAL command will render invisible any local substitutor having the same name as a global substitutor. Any substitutor name that does not appear in an executed GLOBAL command is, by definition, local.

The value of some local substitutors can be inherited from the calling command stream. If the value of such a substitutor is changed, the new value is inherited by any tools (command procedures or programs) subsequently invoked by the command stream in which the change was made. The inherited local substitutors are described in the next section.

60.2.9.3 Predefined Substitutors.

For user convenience in controlling the operation of the CLP, there are a number of predefined substitutors. These are listed here with a short explanation and reference to additional information.

60.2.9.3.1 Predefined Global Substitutors.

The following is a predefined global substitutor. Its value is obtained automatically by the tools that use it, i.e., HELP and QHELP, see 70.3.

NAME	DESCRIPTION	INITIAL VALUE	REFERENCE
CHD	Current Help Directory	.ALS_TOOLS.HELP_DATA.	70.3

60.2.9.3.2 Predefined Local Substitutors For Parameter Passing

The following are predefined local substitutors used for obtaining parameters passed to tools.

NAME	DESCRIPTION	REFERENCE
ARGS	Number of arguments passed to the tool	60.6.2
NARGS	Number of named arguments passed to the tool	60.6.2
PARGS	Number of positional arguments passed to the tool	60.6.2
Pi	Value of the ith positional argument, where "i" is an integer string and 0 <= i <= PARGS	60.6.2
Ni	Name of the ith named argument, where i is an integer string and 1 = i <= NARGS	60.6.2

60.2.9.3.3 Predefined Local Substitutors For Obtaining Tool Execution Status.

Values of the following local predefined substitutors are set upon return from the execution of tools.

NAME	DESCRIPTION	REFERENCE
CSTATUS	Call status from the KAPSE	60.6.4
RSTRING	String generated and returned by the tool	60.6.4
RSTATUS	Status code from the tool	60.6.4, 60.6.7

If no tool command has been issued, CSTATUS has the value OK, and RSTATUS is 0.

60.2.9.3.4 Predefined Control Substitutors

The following substitutors aid in controlling the operation of the CLP. They are local inherited substitutors.

NAME	DESCRIPTION	INITIAL value	REFERENCE
CONFIRM	Switch to control generation of confirmation messages	On	60.14
CWD	Current working directory	(From the auth. access file)	70.3 (chwdir)
ERROR	Switch to control the CLP response to command failure	Stop	60.13
SEARCH	Tool search string	.ALS_TOOLS./CWD/	60.6.5
WARNING	Switch to supress WARNING messages on .MSGOUT	On	60.13
TIME	Switch to control the display of elapsed and CPU times for tool commands	Off	60.14

60.2.10 Expressions.

Expression evaluation in the command language is somewhat different than Ada expression evaluation. Since all operands are character literals, the operators determine the interpretation of the operand values. String operators are not overloaded with integer and Boolean operators.

Conversion is provided on an operator-by-operator basis to map character strings to and from representations for Booleans and integers. Conversion is performed on operands at the time the operator is executed. Unsuccessful conversion causes the command to fail.

60.2.11 Operators.

Operators are "executed" in the following way:

a) The operands are checked for compliance with legal values (e.g., if the operation is +, the operands must be integer string literals, as defined in Section 60.2.5). Command failure results from non-compliance.

b) The operation is performed, producing a string result. Any problem, such as overflow, is diagnosed, causing command failure. All intermediate results in expression evaluation must fall within the legal range for integer strings defined in Section 60.2.5.

The operators in the command language are predefined and divide into the following classes of increasing precedence:

logical operators	AND	OR	XOR			
relational operators	<	<=	>=	/=	=	
	>	*<	*<=	*>=	*/=	
	*=	*>				
adding operators	+	-	&			
unary operators	+	-	NOT			
multiplying operators	*	/	MOD	REM		
exponentiating operator	**					

Ada precedence rules are used. Specifically, operators of higher precedence are always applied first. For a sequence of operators of the same precedence level, the operators are applied in left to right textual order. Parentheses may be used to impose a specific order.

60.2.11.1 Logical Operators.

The logical operators require that the operands be Boolean strings. The result of a logical operation is also a Boolean string. The short circuit forms defined in Ada are not defined in the command language.

60.2.11.2 Relational Operators.

There are two subclasses of relational operators defined at the same precedence level. The subclasses are:

1. String relational operators, and

2. Integer and Boolean relational operators.

Operators with an asterisk "*" prefix are the string relational operators. Others are integer operators. The operators are:

<	integer and Boolean less than
<=	integer and Boolean less or equal to
>=	integer and Boolean greater than or equal to
/=	integer and Boolean not equal
=	integer and Boolean equal
>	integer and Boolean greater than
*<	string less than
*<=	string less than or equal to
*>=	string greater than or equal to
*/=	string not equal
*=	string equal
*>	string greater than

Strings are compared lexically according to the ASCII collating sequence. Integer comparisons conform to normal algebraic rules. No membership operators are defined. If Boolean strings are compared with string operators, upper-lower case equivalence will not be recognized. If the left and right-hand operands of the integer/Boolean operators are not both integer or both Boolean, an error will be diagnosed. FALSE is less than TRUE in Boolean relational expressions.

60.2.11.3 Adding Operators.

The + and - adding operators are integer operators, requiring integer string operands and yielding an integer string result. Ampersand (&) is the string catenation operator taking character string operands and yielding a character string result.

60.2.11.4 Unary Operators.

The + and - unary operators are integer operators, requiring an integer string operand and yielding an integer string result. NOT is a Boolean operator requiring a Boolean string operand and yielding a Boolean string result.

60.2.11.5 Multiplying Operators.

The *, /, MOD and REM operators are defined to operate on integer string operands and yield a similar result.

60.2.11.6 Exponentiation.

Exponentiation is an integer operator yielding an integer result. As in Ada, a negative exponent is not allowed.

60.2.11.7 Diagnostic Messages.

Diagnostic messages produced by the CLP are summarized in Appendix 80.

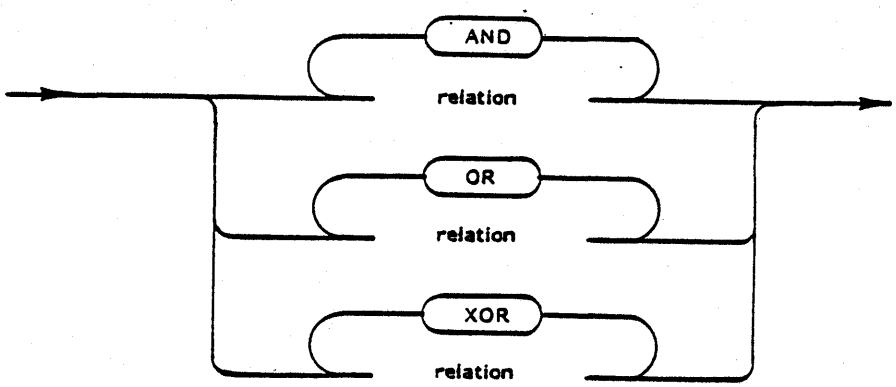
60.2.11.8 Expression Formation.

The rules for building command language conditions and expressions are very similar to the Ada language rules, and are summarized below.

condition

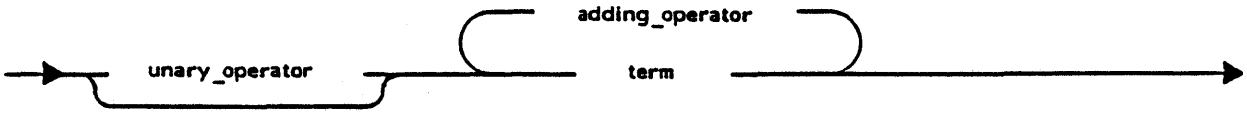
Note: A condition is just an expression that yields a Boolean result.

expression

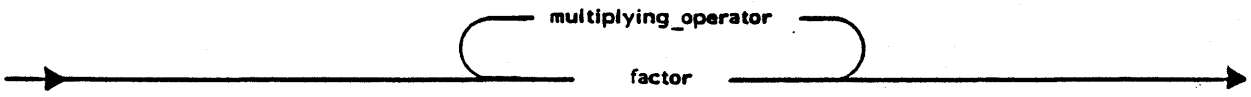


relation

simple_expression

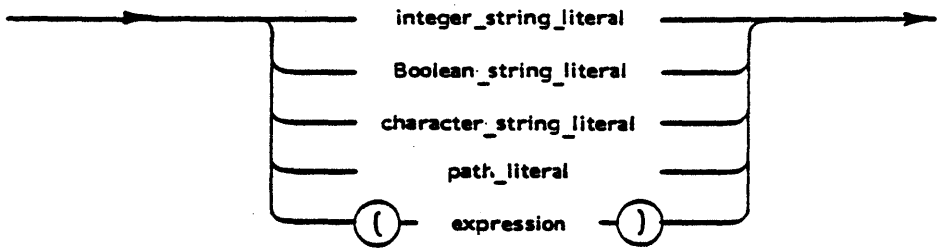


term



factor

primary



path_literal

The constructs not defined in the syntax charts are:

Construct	Paragraph
relational_operator	60.2.11.2
unary_operator	60.2.11.4
adding_operator	60.2.11.3
multiplying-operator	60.2.11.5
integer_string_literal	60.2.4
Boolean_string_literal	60.2.5
character_string_literal	60.2.6
path_name	50.11
attribute_name	50.7

The path_literal construct is used to obtain node attributes from the environment database. The node name and attribute name are specified, separated by an apostrophe. The CLP will then substitute the value of the attribute for the path_literal construct. Path_literals may only be used in expression contexts, i.e., in conditions, the right-hand side of the assignment command, and the return statement.

Examples

```
"2"  <  100        -- yields TRUE
"2"  < "100"       -- yields TRUE
 2   <  100        -- yields TRUE
 2  *<  100        -- yields FALSE, i.e. string compare
"2" *< "100"       -- yields FALSE

 1 + 2             -- yields  3
"1" + "2"          -- yields  3
"1" & "2"          -- yields  12
 1 & 2             -- yields  12
a:= "1 + 2"        -- a is 1 + 2
"True" ="true"     -- yields TRUE
"True"*="true"     -- yields FALSE
```

60.2.11.9 Line marks.

A line mark on the VAX/VMS host is defined as a Carriage Return.

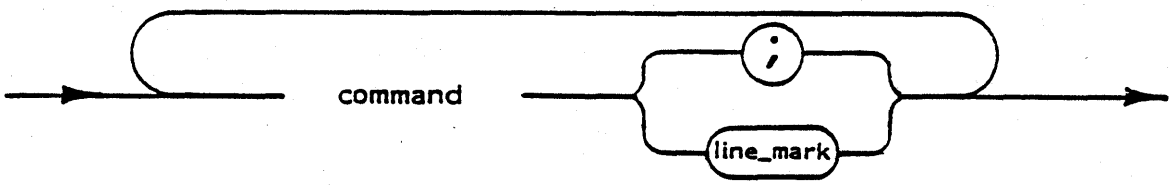
60.2.11.10 Line continuation.

Line continuation may be done by using the hyphen ("-"). If a hyphen is the last character on a line before the line mark then the line mark and the hyphen will be ignored. Lines may be continued at any place a space may appear in the command language, except that continuation may not occur within comments or quoted string literals.

Example

```
-- the following two commands are equivalent.

echo (this, is, an, -
  example);
echo (this, is, an, example);
```

60.3 Command sequence.



A command sequence is a sequence of commands, each terminated by a line_mark or semicolon (see Ada Reference Manual Par. 14.3.3). The notion of command sequence has special significance when the CLP is obtaining its commands from the user's keyboard. Whenever the CLP is waiting for the user to enter a command sequence, the CLP will first issue a prompt message to the user's terminal. The purpose of the prompt is to remind the user that a response is expected from him and to identify the tool that is expecting the response. The general format of a prompt is:

tool_id nest_level>

where:

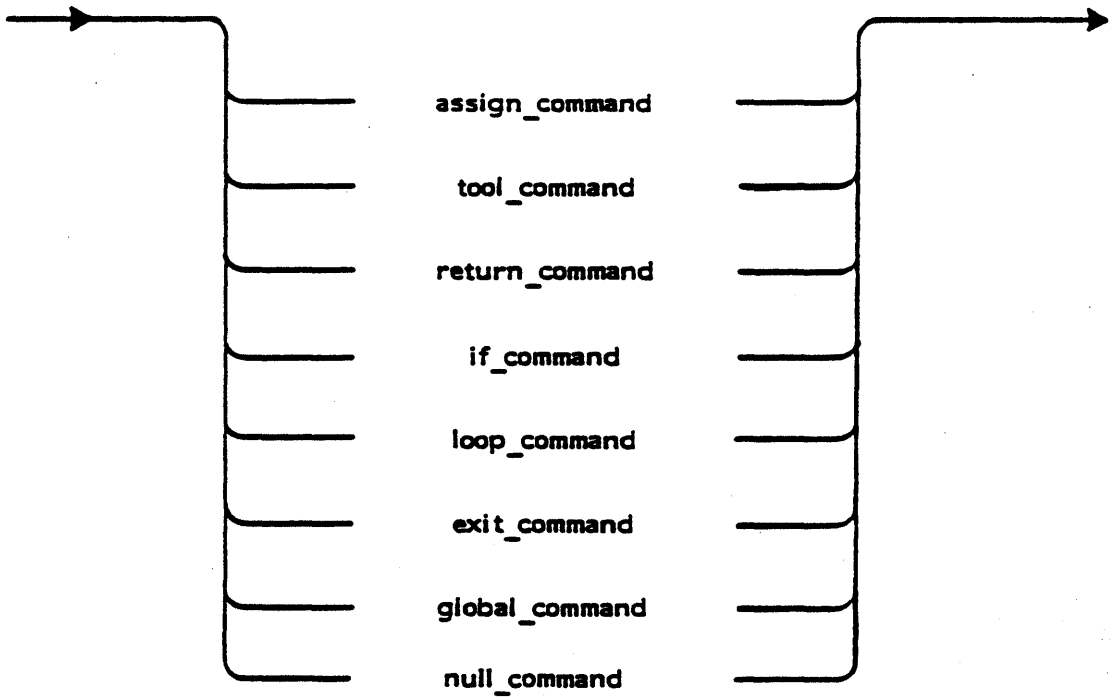
tool_id — is a short string identifying the tool expecting the response. The initial command stream will use a null string for the tool id.

nest_level — is an integer indicating the level of nesting within command language loops and conditionals. (Nested command files do not cause the nest_level to increase.) The first level of command blocks is the zeroth level, for which the nest_level is null.

Since tools are free to alter the prompt, adherence to this prompt protocol is optional. It will, however, be used within the standard tool set.

60.4 Command.

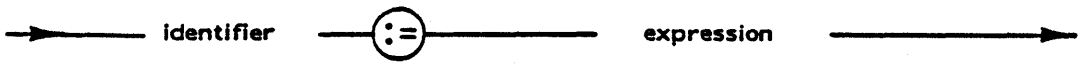
The general command syntax is:



The eight types of commands are described below.

60.5 Assign Command.

The syntax of the assign command is:



The assign command is used to create a substitutor, or to replace the current value of a substitutor with a new value specified by the right-hand-side expression.

The value of a substitutor may be altered by subsequent re-assignment. The value of the substitutor is the string resulting from the expression evaluation. If expression evaluation fails, the command fails and the substitutor value remains unaltered.

Example:
```
B := H
C := 1
A := #B + #C -- Since expression evaluation fails
             -- A is unchanged
C := #C + 2  -- same as C := 3
```

Inside the body of the command procedure ANALYZE:

```
#P0          -- yields "ANALYZE"
#P1          -- yields "text"
#N1          -- yields "OPT"
#OPT         -- yields "NO_LIST"
#P2          -- yields null string
```

Substitutors for named parameters are created when the command procedure is initiated and may not be re-assigned like other substitutors. The value of the ith named parameter can be obtained with the construct

```
#(#Ni)
```

Three predefined substitutors aid in referencing parameters:

ARGS giving the total number of arguments in the call of the command procedure,

PARGS giving the number of positional arguments specified in the call, and

NARGS giving the number of named arguments.

```
ARGS = PARGS + NARGS
```

To help build argument lists for nested tools, the notation

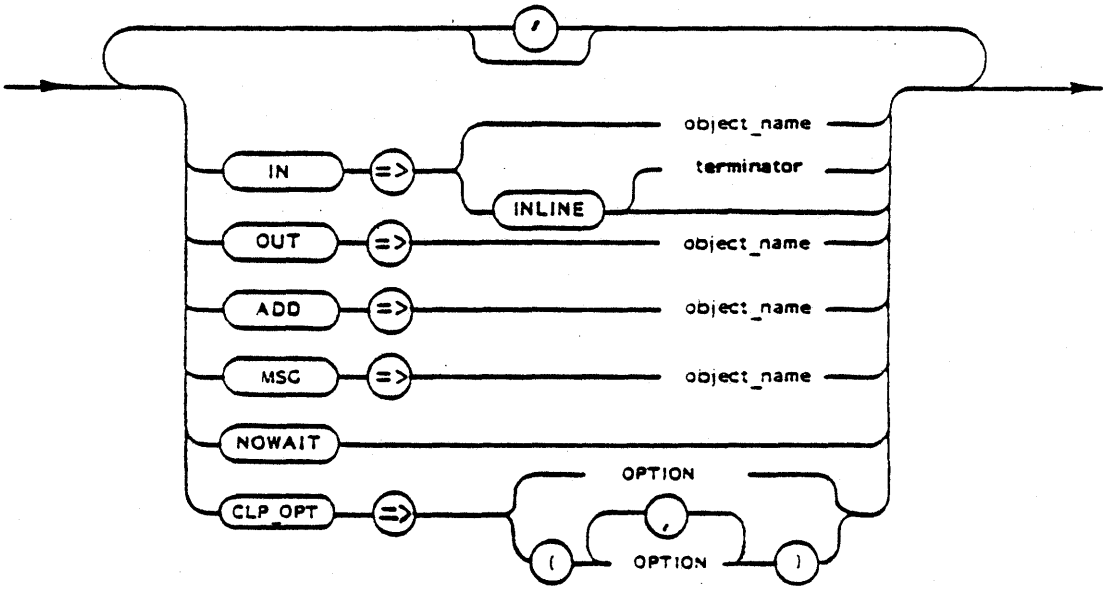
```
#(Pi..Nj)
```

can be used to denote the ith positional argument through jth named argument, where i and j are integer strings >= 0. In this notation, the ith positional argument through jth named arguments are substituted in the order they appeared in the call. Named arguments appear with the arrow notation; the list is not enclosed in parentheses; and commas and spaces appear as delimiters. If i > #PARGS or j > NARGS, the null string is substituted for all non-existent parameters an delimiters. If i or j are not integer strings, a conversion error occurs and the command fails. The notation #(Pi..Pj) and #(Ni..Nk) can likewise be denoted just the positional or just the named arguments, where i and j are integer strings, i <= j, i>= 0, i <= K, j<=#PARGS, 2nd K<=#NARGS. For example, inside the command procedure ANALYZE, invoked above

```
#ARGS          -- yields   2
#(P1..N1)      -- yields   text,'OPT=>NO_LIST
```

60.6.3 Control Part.

The syntax for the control part is:



When a tool is invoked, six predefined internal files are open and ready for use. These files are referred to as:

1. Master input,
2. Master output,
3. Standard input,
4. Standard output,
5. Message output, and
6. The Null file.

Standard input and output are defined by the Ada TEXT_IO package. The others are defined by the KAPSE. The null file is an infinite waste basket. Information written to it disappears. Attempts to read from the

null file return an end-of-file indication. The standard names for these files are:

1. .MSTRIN,
2. .MSTROUT,
3. .STDIN,
4. .STDOUT,
5. .MSGOUT, and
6. .NULL_FILE or .NF, respectively.

These names may be used anywhere a file name is expected; however, attempts to destroy, create, rename, or otherwise alter the character of these predefined files will result in diagnostics and command failure.

The control-part is used to specify:

1. From what file or device the tool reads its standard input,
2. To what file or device the tool writes its standard output,
3. To what file or device the tool writes messages, and
4. Whether standard output is to be rewritten or appended.

   (This type of control is called I/O redirection or simply, redirection.)

5. Background execution
6. Options to the CLP

The master input, master output, and null files may never be reassigned. All predefined files are inherited from the calling command procedure. This means that the predefined files are initially connected to whatever files or devices were assigned in the calling command stream, provided the tool command in the calling stream has no explicit redirections. In the presence of explicit redirection, the called procedure's standard files are connected as specified in the redirection.

In the initial command stream the assignments are:

1. Master input - keyboard of the user's terminal.

2. Master output - crt display or printer of the user's terminal.

3. Standard input - master input.

4. Standard output - master output.

5. Message output - master output.

Standard output may be redirected by using the following form of tool command:

```
command OUT=>result
```

This command is interpreted by the CLP as an output redirection to the file named result, in the current working directory. If result does not exist, the CLP causes it to be created. In order to append output to a file, the following notation may be used:

```
command ADD=>result
```

In this case, standard output from command is appended to the end of the file named result. The standard input for a command may be obtained from a file by using:

```
command IN=>source
```

In this example, the command reads standard input from the file named source. Text for standard input can also come directly from the command stream by using the construct:

```
command IN=>INLINE
     <text for standard input>
END_INLINE
```

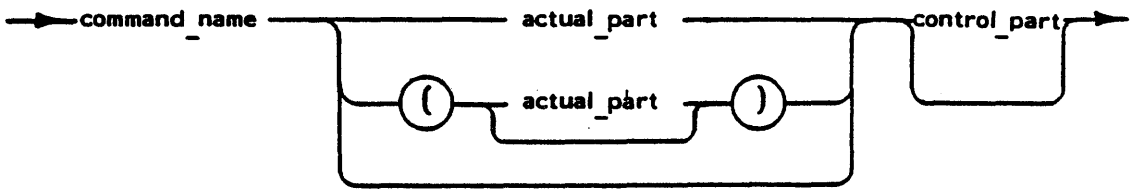
An alternate closing terminator for in-line input can be specified after INLINE. For example:

```
command IN=>INLINE ZAP OUT =>RESULT
     <text for standard input>
ZAP
```

In either case, the closing terminator must appear on a line by itself after the text. Alternate terminators are in effect only for the command in which they are specified. The inline text begins on the line following the command. Other control_part specifications may occur within the command after the INLINE specification. String substitution will be performed within the inline text. Moreover, the terminator for the inline input can result from substitution.

60.6 Tool Command.

The syntax of the tool command is:



A tool command is used to invoke an Ada program or command procedure. The specification of a tool command is given by its command_name optionally followed by an actual_part and/or a control_part in that order.

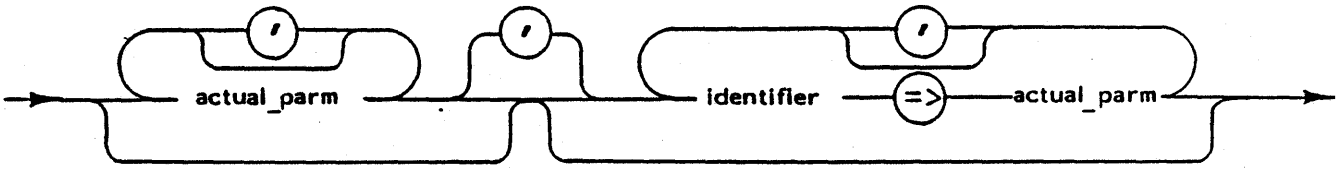
The actual_part is used to pass input parameters to the tool. The control_part is used to control input_output file assignment and background execution. If the optional parentheses are used to enclose the actual_part, a space must separate the command_name and the left parenthesis. The parentheses are mandatory if the first parameter in the actual part starts with a left parenthesis. In other words, if the actual_part begins with a left parenthesis, the actual_part must be enclosed in parentheses.

Examples

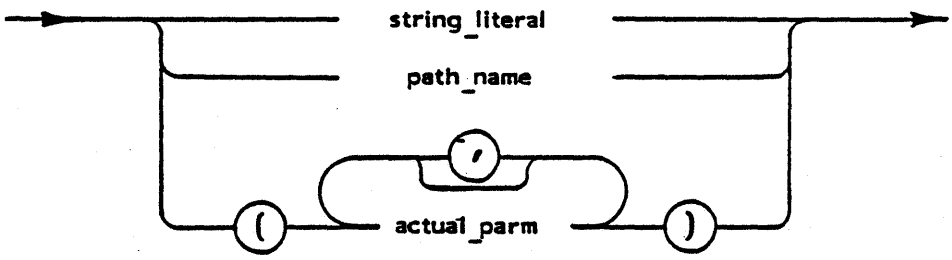
```
EDIT (myfile)     -- invoke editor to create an Ada program

ADA1602 (myfile, local_library, OPT=>(LIST_INCLUDE, XREF))
         OUT=>.LISTINGS
                  -- compile an Ada program with options
                  -- send output listings to file .LISTINGS

PRINT myfile NOWAIT
                  -- print myfile in background on the system printer
```

60.6.1 Passing In Parameters.

Actual_part syntax in CL is very similar to the Ada syntax for procedure calls. In the actual_part the user may specify both positional and keyword parameters. The syntax of the actual_part is:



actual_parm



Actual parameters are delimited by blanks and/or commas. If the parameter list is enclosed in parentheses, the right parenthesis is sufficient to delimit the last actual parameter. Actual parameters are character strings, path_names, or parenthesized lists of these. The arrow (=>) notation of Ada is used to specify named parameters. All positional parameters must be specified first and may not be intermingled with keyword parameters.

60.6.2 Referencing Parameters.

In a command procedure, parameters are numbered from the left. The string corresponding to a parameter is obtained via string substitution. The ith positional argument from the left is substituted for #Pi, where i is a natural number. The name of the jth named argument from the left is substituted for #Nj, where j is a natural number. If no ith positional parameter or no jth named parameter was given in the call, the empty string is substituted for #Pi or #Nj, respectively. The value of the named parameter is obtained by using the parameter name as a substitutor. #P0 becomes the command name as specified in the tool command, (not the absolute path name of the tool).

Only a single control_part specification of each type may appear in one command. Duplicate "IN=>" specifications, for example, will cause a control_part error and subsequent command failure.

To assign a file for message output, the construct:

```
command MSG=>collector
```

is used. Here, messages are appended to the file named collector. If such a node does not exist, it is created. Messages are always appended to the message file.

If a command procedure contains several commands which read from the standard input, and the standard input is assigned to a file, then each command obtains the standard input file at the beginning, irrespective of what part of the standard input was read by previous commands. In other words, the standard input is "rewound" between commands. If the standard input is connected to a device, then the behavior is device dependent, e.g. redirection from the keyboard would start with the next line entered, not the first line of the session.

For standard output, a called command procedure inherits the mode of redirection as well as the file or device. For example if CP is a command procedure, and the tool command:

```
CP   OUT=>file_out
```

is issued, then each tool within CP that does not have an explicit redirection, will overwrite file_out. If, on the other hand, the command:

```
CP   ADD=>file_out
```

is issued, then tools in CP will append to file_out. Tools in CP which explicitly overwrite the standard_output will overwrite the entire file_out, not just the part appended by CP.

The control part of the tool command provides a mechanism for the user to set options which produce output that enable the user to observe intermediate processing of commands by the command language processor. This output helps the user to debug command procedures. If the options are set for a tool that is not a command procedure then a warning diagnostic is generated and the options are ignored.

The options are:

READ_VERIFY - shows how each line looks as it is about to be processed.

SUB_VERIFY - shows how each command looks after substitution has been performed.

EXPR_VERIFY - shows how each command looks after expression evaluation has been performed.

LIST_GLOBALS - shows all global substitutors whenever the CLP encounters a global command.

LIST_PARAMETERS - shows the initial values of the parameter substitutors for the command procedure.

The output produced by these options is directed to message output.

The options can be invoked by using the following forms of the tool command:

```
command CLP_OPT=>EXPR_VERIFY

command CLP_OPT=>(READ_VERIFY, SUB_VERIFY)
```

60.6.4 Obtaining Returned Information.

After completion of a tool command, three predefined substitutors are set to indicate the result of tool execution, CSTATUS, RSTATUS, and RSTRING.

The value of CSTATUS is the completion status which is set by the KAPSE. CSTATUS is used to signal error conditions which are beyond the purview of the tool itself. The strings returned in CSTATUS are limited to the following predefined set.

Value	Meaning
OK	Tool invocation was successful, RSTATUS also indicates success
NO_SUCH_NODE	node specified by command_name could not be found
NODE_NOT_EXECUTABLE	specified node did not contain a command procedure or executable program
ACCESS_PROHIBITED	user does not have permission to execute the tool
UNHANDLED_EXCEPTION	the tool terminated because of an unhandled exception
BAD_ARG_LIST	bad arguments used to invoke tool.
PARM_LIST_MISMATCH	<currently undefined>
PROGRAM_ABORTED	the tool was aborted
TIME_LIMIT_EXCEEDED	the tool was terminated because

	the allotted time expired
SPACE_LIMIT_EXCEEDED	the tool was aborted because the mass memory allotment was exceeded
INSUFFICIENT_MEMORY	too little main memory space is available or allotted to support tool execution
CALL_FAILURE	Tool invocation was successful, RSTATUS indicates failure
NO_RETURN	Tool did not return using ALS conventions.
IO_ERROR	Tool caused an I/O error.
DUPLICATE_PROG	Tool has same program identifiers as a concurrently running tool.

The value of the RSTATUS substitutor is an integer used to signal error and other conditions within the purview of the tool. Values >=0 indicate successful command execution. Values <0 indicate command failure. The value of the RSTRING substitutor is a string also returned by the tool. See 60.7 for RSTATUS and RSTRING. If the tool does not set the values of these substitutors for the CLP, the default values of zero (0) for RSTATUS and blanks for RSTRING are used.

The values of CSTATUS, RSTATUS, and RSTRING are changed by the execution of tool commands. Between tool commands they may be used in expressions. These substitutors may not appear on the left side of an assignment command. The CSTATUS, RSTATUS, and RSTRING values are written to the message file at the completion of every tool command. (See 60.13.)

The CLP waits for tool commands to be completed before accepting the next command. However, if NOWAIT is specified, the CLP will accept the next command as soon as the tool has been started. The tool is then viewed as executing concurrently with the CLP. The CSTATUS value is available immediately after execution of a tool command with the NOWAIT option, along with the default values of RSTATUS (0) and RSTRING (blanks).

60.6.5 Search Rules.

When the CLP recognizes a tool command, it uses a set of search rules to find the database file containing the tool. These rules are:

1. If the command_name begins with a period denoting an absolute node name, the CLP attempts to determine if the node is executable. If the node cannot be executed, the command fails, and a message is sent to the user via MSGOUT.

2. If the command_name does not begin with a period, the CLP utilizes the predefined substitutor SEARCH. SEARCH contains a list of partial pathnames used to build a series of pathnames by appending the command_name to each partial pathname. These pathnames are then used in sequence to search for the node containing the command. For each partial pathname in SEARCH, a pathname consisting of the toolname appended to the partial pathname is constructed, and Steps 3 and 4 are performed.

3. The CLP then attempts to find the tool by using the constructed pathnames in order.

4. If the node is found, the CLP determines if the node is a program or a command file. If the node cannot be executed, the command fails and a message is sent to the user.

5. If the node is not found using the pathnames from SEARCH, the command fails and a message is sent to the user.

SEARCH is a string formed according to the following rules:

a) Partial pathnames are followed by a virgule (/).

b) The ending virgule (/) is mandatory.

c) A period is <u>not</u> automatically inserted when appending the command_name.

d) Two special pathnames are recognized, ROOTPATH and CWD, described below. These are recognized only if they occur as an entire partial pathname entry.

CWD causes the current working directory to be searched for the tool.

ROOTPATH causes the path from the parent of the current working directory to the root of the database to be searched for the tool. This is accomplished with the following algorithm:

1. The search is started from the parent node of the current working directory.

2. The names of all offspring nodes are scanned for a match with the command_name. (The tool name can itself be a relative pathname.)

3. If there is no offspring match, the search is continued at the next parent and so on.

4. After searching the offspring of the database root without a match, searching continues with the next pathname in SEARCH.

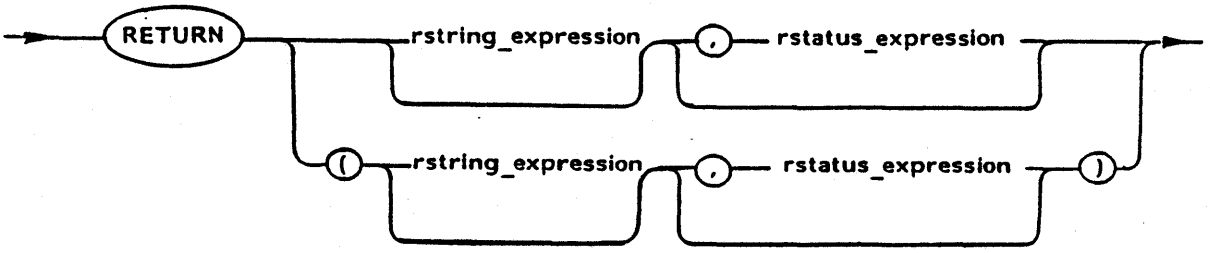
The SEARCH string is a substitutor and may be modified by the user. When not explicitly modified, SEARCH is .ALS_TOOLS./CWD/

Examples of SEARCH:

```
.Tools./CWD/.my_directory.my_tools./Rootpath/

          -- Searches the tools directory first,
          -- the current directory next,
          -- my_tools directory, and finally
          -- the rootpath

CWD/

          -- Searches only the current directory
```

60.7 RETURN COMMAND.

The syntax of the return command is:



The return command is used to terminate the execution of a command procedure and to send RSTRING and RSTATUS back to the calling procedure. The returned value may be referenced in the calling command procedure by using the predefined RSTRING substitutor. RSTATUS may be similarly referenced by using the predefined substitutor RSTATUS.

If the rstring_expression starts with a left parenthesis, then the clause containing the rstring_expression and the rstatus_expression (if one is present) must also be enclosed in parentheses.

The expressions are evaluated before control returns to the calling procedure. If both expressions are specified, they must be separated by a comma. If the rstring_expression is not specified, the null string is returned. If the rstatus_expression is not specified, 0 is returned. The rstatus_expression must evaluate to an integer string.

The value of RSTATUS is used to indicate success or failure of a tool. It may be set to any value within the allowed integer range. Zero and positive values indicate successful completion of the command. A negative RSTATUS indicates failure of the tool when the return to the caller is completed. The following table summarizes the RSTATUS protocol:

Value	Meaning
<= -2	Tool-defined tool failure
-1	Bad rstatus_expression
>= 0	Tool-defined successful tool completion

If a return is encountered in the initial command stream, the ALS is terminated and control returns to the host operating system.

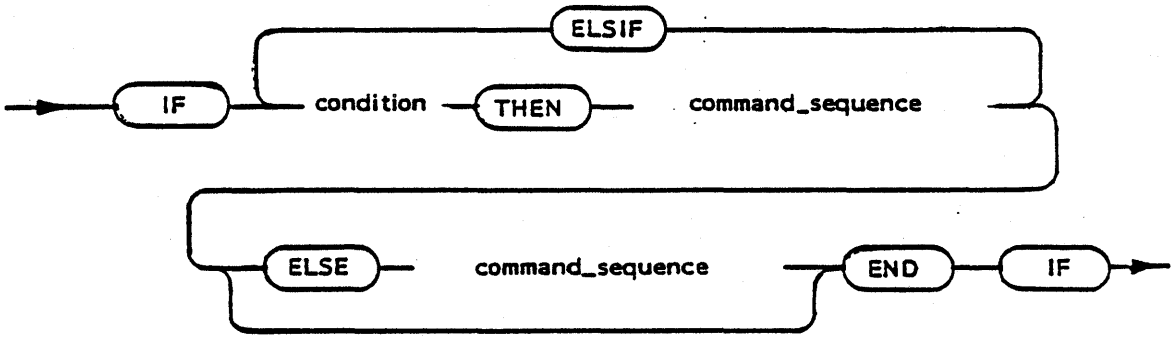
60.7.1 Error Conditions.

If an error is encountered in the evaluation of the rstring_expression, the null string is returned. If the rstatus_expression does not evaluate to a legal integer string, RSTATUS in the calling command stream will be set to -1.

Diagnostic messages produced by the CLP are summarized in Appendix 80.

60.8 IF Command.

The syntax of an IF command is:



The if command may be used to conditionally execute a sequence of one or more commands depending on the value of a Boolean expression. The expression specifying the condition must yield a Boolean result.

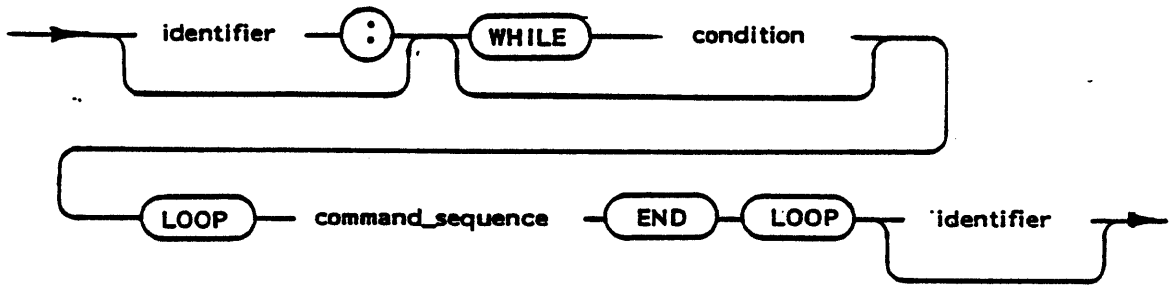
The condition specified after IF and any conditions specified after ELSIF are evaluated in succession until one evaluates to TRUE. A final ELSE is treated as ELSIF TRUE THEN. The corresponding sequence of commands is then executed. If none of the conditions evaluates to TRUE, then none of the sequences of commands is executed in the absence of an ELSE clause.

Example:

```
ADA1170 (sine, my_prog)
IF #RSTATUS >= 0 AND #CSTATUS *= OK  THEN    — Link if
   LNK1170 (myprog,main, new_mod)            — compile
                                             — succeeds
END IF;
IF #RSTATUS >= 0 AND #CSTATUS *= OK THEN
   ECHO (DONE)
ELSE
   ECHO (FAILED)
END IF
```

60.9 Loop Command.

The syntax of the loop command is:



A loop command may be used to specify the repeated execution of a sequence of commands zero or more times. If a loop identifier is used it must be given both at the beginning and at the end of the identified loop. (A loop identifier is used in conjunction with an exit command to escape from nested loops.)

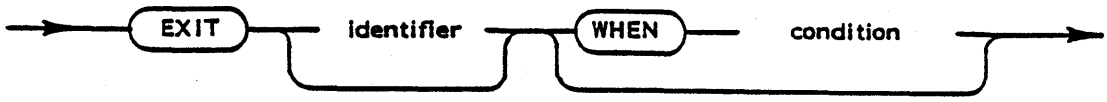
A loop command without a while clause specifies repeated execution of the basic loop. If an exit or RETURN command is not used to terminate such a loop, it will not be terminated.

In a loop command with a while clause, the condition is evaluated and tested before each execution of the basic loop. If the condition is TRUE, the sequence of commands within the basic loop is executed. If the condition is FALSE, the loop is terminated. Commands in the loop body undergo string substitution at each iteration.

The command sequence within a loop command is limited to 100 lines.

60.10 Exit Command.

The syntax of the exit command is:



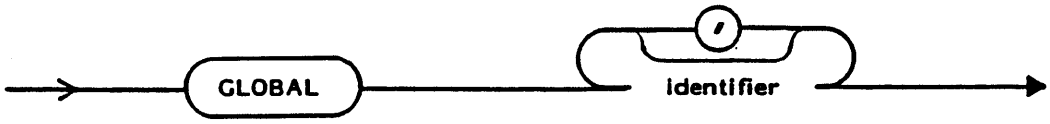
The exit command is used to terminate an enclosing loop. If a WHEN condition is specified, the exit will be taken only if the condition is true. If no condition is specified, the exit will always take place.

The loop exited is the innermost loop unless the name of an enclosing loop is specified. The named loop is exited together with any loops inside the named loop. A named exit command may only appear within the named loop.

An exit command may only appear within a loop and is meaningless otherwise. If an exit command appears outside a loop, a warning message is posted to the message file.

60.11 Global Command.

The syntax of the GLOBAL command is:



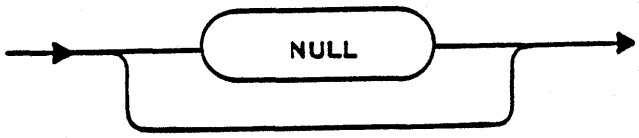
The GLOBAL command is used to declare one or more substitutors as global. A substitutor referenced in a global command does not become visible until the command is processed. Local substitutors by the same name are hidden. Predefined local substitutors and parameter substitutors (e.g., Pn) cannot be declared global.

Examples are:

```
GLOBAL xx,yy a  -- declares xx,yy and a as global substitutors

GLOBAL M        -- declares M to be a global substitutor
```

60.12 Null Command.

The syntax of the NULL command is:



The NULL command has no effect other than to pass to the next command.

60.13 Error Handling.

ALS commands either succeed or fail. Command failure or success is signalled by CSTATUS and RSTATUS. To succeed, both C and R status must indicate success; an adverse indication in either indicator will cause failure. Specifically, to succeed:

a) CSTATUS must be "OK", and

b) RSTATUS must be >= 0.

The predefined substitutor ERROR controls the behavior of command procedures in which failures occur. If ERROR has the value "continue", then processing of command procedures will continue even when some commands fail. If ERROR has any other value, a command failure will cause termination of the command procedure at that point and the failure of the command procedure itself will be signalled at the next higher level. This is accomplished by returning a CSTATUS of "PROCEDURE_FAILURE" and a null RSTRING. The RSTATUS returned will be 0 or the value resulting from the last return command encountered in the failing command procedure.

The predefined substitutor WARNING can be used to suppress warning messages issued on .MSGOUT. If WARNING has any value other than "on", warnings will be suppressed. Since the interpretation of WARNING is performed by the individual tools, the correct behavior of user-supplied tools cannot be ensured.

Diagnostic messages produced by the CLP are summarized in Appendix 80.

60.14 Command Completion Reporting.

The success or failure of all tool commands is reported on the MSGOUT file. Each tool command causes a message containing the CSTATUS and RSTATUS to be posted. If CONFIRM has any value other than "on" reporting of successful completion will be disabled.

The elapsed and CPU times used by a process are displayed by the CLP, if the TIME substitutor has the value "On". If this switch is On, the elapsed and CPU time used by tool commands is displayed on the MSGOUT file. The CPU time is the processor time used by the tool, including the CPU time expended by any child processors in behalf of the parent. (CPU time expended by ACP portions of the KAPSE are not included.) Times are not reported for tools executed with the NOWAIT option.

60.15 Command Language.

60.15.1 Notation.

The syntax of the command language is described using the following variant of Backus-Naur Form.

a) Lower case words, some containing embedded underscores, denote syntactic categories, for example

expression

b) Upper case words denote reserved words, for example

RSTRING

c) Square brackets enclose optional items, for example

RSTRING [expression]

d) Braces enclose a repeated item. The item may appear zero or more times.

e) A vertical bar separates alternative items. Some alternative items are grouped with parentheses. Such parentheses are meta-parentheses only; they are not part of the command language.

60.15.2 Grammar.

```
command_sequence ::= { command (;¦ line_mark) }

command ::= ¦ assign_command
            ¦ tool_command
            ¦ rstring_command
            ¦ if_command
            ¦ loop_command
            ¦ exit_command
            ¦ global_command
            ¦ null_command

assign_command ::= identifier := expression

tool_command ::= command_name [(actual_part)] control_part
                 ¦ command_name [actual_part] [control_part]

actual_part ::  [actual_parm_part] [named_parm_part]
```

```
                | actual_parm_part [,] named_parm_part

actual_parm_part ::= actual_parm {[,] actual_parm}

command_name ::= object_name

actual_parm ::= string

string ::= string_literal | path_name | (string {[,] string})

named_parm_part ::= parm_name=>actual_parm {[,] parm_name=>actual_parm}

parm_name ::= identifier

control_part ::= control_item {[,] control_item}

control_item ::= IN => object_name | INLINE [terminator]
                | OUT => object_name
                | ADD => object_name
                | MSG => object_name
                | NOWAIT
                | CLP_OPT => option_list

terminator ::= string_literal

option_list ::= option

                | (option {,option})

option ::= identifier

object_name ::= path_name
                | <<VMS>> string_literal

path_name ::= .[node_id {.node_id} [revision_index]]
                | {!.} node_id {.node_id} [revision_index]
                | !{.!}

node_id ::= identifier {variation_index} [default_index]

variation_index ::= (identifier)
            | (attribute_spec {,attribute_spec})

default_index ::= ()

revision_index ::= (integer) | (+) | (*)

attr_spec ::= attribute_name => attr_value

attr_value ::= string_literal

return_command ::= RETURN  [rstring_expression] [, rstatus_expression]
                 | RETURN  ([rstring_expression] [, rstatus_expression])
```

```
rstring_expression  ::= expression

rstatus_expression ::= expression

tool_sequence ::= {(tool_command ¦ NULL) (;¦ line_mark)}

if_command  ::=  IF condition THEN
                   command_sequence
                 {ELSIF condition THEN
                   command_sequence}
                 [ELSE
                   command_sequence]
                  END IF

loop_command  ::=  [loop_id:] [iterate_clause] basic_loop [loop_id]

loop_id ::= identifier

iterate_clause  ::=  WHILE condition

basic_loop  ::=  LOOP command_sequence END LOOP

exit_command  ::=  EXIT [loop_id] [WHEN condition]

global_command ::=  GLOBAL identifier {[,] identifier}

null_command ::=  [NULL]

condition ::= expression

expression ::= relation {AND relation}
             ¦ relation {OR relation}
             ¦ relation {XOR relation}

relation ::= simple_expression [relational_operator simple_expression]

simple_expression  ::=  [unary_operator] term {adding_operator term}

term  ::=  factor {multiplying_operator factor}

factor ::= primary [** primary]

primary  ::=  integer_string_literal
            ¦  Boolean_string_literal
            ¦  character_string_literal
            ¦  path_literal
            ¦  (expression)

relational_operator  ::=  =  ¦ /= ¦ < ¦ <= ¦ > ¦ >=
                        ¦ *= ¦ */= ¦ *< ¦ *<= ¦ *> ¦ *>=

adding_operator  ::=  + ¦ - ¦ &
```

```
unary_operator  ::=  + | - | NOT

multiplying_operator ::= * | / | MOD | REM

string_literal ::= character_string_literal

character_string_literal ::= "{character}" | [digit] identifier
                           | integer_string_literal

integer_string_literal  ::=  integer  [exponent]

integer  ::=  digit  {[underscore] digit}

exponent  ::=  E  integer

Boolean_string_literal  ::=  TRUE | FALSE

path_literal  ::=  path_name'attribute_name

attribute_name ::= identifier

identifier ::= letter {[underscore] letter_or_digit}

letter_or_digit  ::=  letter | digit

letter  ::=  upper_case_letter | lower_case_letter

character ::= letter | digit | delimiter | underscore

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

underscore ::= _

upper_case_letter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

lower_case_letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

delimiter ::= & | ( | ) | * | + | - | / | ; | " | : | = | < | > | # | ,
            | . | `

line_mark ::= <carriage_return>
```

APPENDIX 70

70. ADA LANGUAGE SYSTEM TOOLSET

This appendix describes the standard tools provided in the initial ALS tool set. These tools may be invoked through the command language processor or by an Ada program. Table 70-1 summarizes the tools.

70.1 Tool Description Format.

All tool descriptions in this appendix conform to the following format:

NAME:	Name and title of the tool
FUNCTION:	Brief description of the operation of the tool
FORMAT:	A prototype for the command (see below)
PARAMETER DESCRIPTION:	Description of the parameters named in the prototype
DISPOSITION:	A summary of how information is passed in and out of the tool (see below)
NOTES:	Conditions which may cause command failure and any other pertinent information
EXAMPLE(S):	One or more examples of the tool invocation

The meta-language used to express tool command formats is similar to the meta-language used in the Military Standard, Ada Programming Language, ANSI/MIL-STD-1815-1983, 17 February 1983 (2.1). Specifically:

a) Square brackets enclose optional items,
b) Braces enclose repeated items, and
c) A vertical bar separates alternative items.

Words that are to appear with exact spelling are shown in upper case, although they may be given in either upper or lower case in the actual command. For clarity, the command formats have parentheses enclosing the parameters, and commas separating them. The parentheses and commas may be omitted and spaces used to separate parameters. A space must also be used to separate the command name from the parameter list, even when parentheses are used.

The terms node and file, used in the format description, are defined as follows:

a. Node: any environment database node, i.e., a file, directory, or variation header. Any restrictions are stated in the parameter description or notes, e.g., in chwdir, a node must be a directory.

b. File: any environment database file node or any VAX/VMS file/device named with the "<<VMS>>" notation (described in Section 50.11).

In the disposition description, IN, OUT, MSG, RSTRING, and RSTATUS are used to denote standard input, standard output, message output file, the tool return string, and the RSTATUS, respectively.

70.2 Tool Set Protocols.

A number of protocols are used to promote uniformity in the tool set. These are enforced by tool design and usage.

70.2.1 Parameter Passing.

These protocols pertain to passing of tool parameters.

a. The parameters that must be given every time a tool is invoked are specified positionally. As such, these parameters appear first in the parameter list.

b. Inputs are specified before outputs. A corollary to this protocol is that where there is a clear from-to relationship between parameters, the from parameter is specified first, e.g., in a copy command.

c. For some commands, the last (rightmost) positional parameter is optional. There are no other optional positional parameters.

d. Named parameters are used to specify optional parameters and may be given in any order after the positional parameters.

e. A toggle is an on/off switch used to control tool execution. A toggle may be used, for example, to turn off a listing produced by a compiler. Toggles are represented as keywords and are grouped into a sublist following the named parameter OPT (for options). (OPT may be spelled in lower case.) The elements of the option sublist are grouped by enclosure in parentheses or quotes, and are separated by commas or spaces. If only one toggle is specified, the quotes or parentheses may be omitted.

"NO_" or "no_" preceding a toggle name inverts the sense of the toggle, e.g., LIST and NO_LIST.

f. Some tools may optionally accept a sublist where a single parameter would otherwise be specified. The use of this convention must be specifically mentioned in the tool description.

70.2.2 Disposition of Output.

This protocol pertains to the creation or overwriting of file revisions.

Whenever a tool is writing to an output file, and no revision number is specified, it obeys the following rules:

a If no revision set exists by the required pathname, the tool creates one and output goes to Revision 1.

b If a revision set by that pathname already exists, the highest revision is always overwritten unless it is frozen, or has a derivation count greater than zero. In this case, the tool creates the next revision and uses it for output. (Note that <<VMS>> files are never frozen, so are always overwritten.)

Table 70-1

TOOL SET SUMMARY

NAME	CPCI*	DESCRIPTION
ADAMCF		MCF compiler
ADAVAX		VAX 11/780 compiler
ADDREF	DBM	add references to an association
ARCHIVE	FA	archive a set of file revisions
ASMMCF		MCF assembler
ASMVAX		VAX 11/780 assembler
CHACC	DBM	change access_name attribute
CHASS	DBM	change an entire association
CHATTR	DBM	change attribute values
CHREF	DBM	change references in an association
CHTEAM	CLP	change team identification
CHWDIR	CLP	change working directory
CMPFILE	FA	compare data of two file nodes
CMPNODE	FA	compare nodes except for file data
CMPTEXT	FA	compare text files
CONCAT	CCT	concatenate text files
CPYALL	CCT	copy all of a file, data, attributes and associations
CPYDATA	CCT	copy only data portion of a file
DATE	CCT	get current date
DEBUGVMS		debug an Ada program
DELNODE	CCT	delete a node and all its offspring
DELREF	DBM	delete references from an association
ECHO	CLP	print command arguments
EDT	CCT	VAX-11 EDT text editor
EXPMCF		MCF exporter
EXPVMS		VAX/VMS exporter
FREEZE	DBM	freeze latest revision
GENLISTMCF	DT	MCF listing generator
GENLISTVAX	DT	VAX-11/780 listing generator

Table 70-1 (continued)

NAME	CPCI*	DESCRIPTION
HELP	CCT	get help information
LIB	CCT	program library manager
LNKMCF		MCF linker
LNKVAX		VAX-11/780 linker
LST	CCT	list contents or offspring of a node
LSTASS	DBM	list contents of an association
LSTATTR	DBM	list value of an attribute
MKDIR	DBM	make a directory
MKFILE	DBM	make a file
MKVAR	DBM	make a variation header
PRINT	CCT	print file on line printer
PROFILEVMS	DT	VAX/VMS statistical and frequency profile display
QHELP	CCT	supplies information from the HELP database
RECEIVE	FA	receive subtree written in interchange format
RENAME	DBM	rename node
REVISE	DBM	make new revision of a file
RUNOFF	CCT	DEC Standard Runoff Text Processor
SHARE	DBM	share a node
SHOW_SUBS	CLP	display substitutors
STUBGEN	CCT	generate a stub for a body
TCX	CCT	index generator for RUNOFF
TIME	CCT	get current time
TOC	CCT	table of contents generator for RUNOFF
TRANSMIT	FA	transmit subtree in interchange format
UNARCHIVE	FA	unarchive a set of file revisions

*NOTE: This column identifies those tools which are part of the Database Manager (DBM) CPCI, the Configuration Control Tools (CCT) CPCI, the Display Tools (DT) CPCI, the File Administrator (FA) CPCI, and the Command Language Processor (CLP).

70.3 Tool Descriptions.

Descriptions of the ALS tools are provided in alphabetical order on the following pages of this chapter.

ALS COMMAND DESCRIPTION ADAMCF

NAME: ADAMCF - Ada compiler for MCF Target

FUNCTION: Translate one or more Ada compilation units, generating code for the MCF target

FORMAT: ADAMCF (source,prog_lib[,NEW_SRC=>out_src][,OPT=>option_list])

PARAMETER DESCRIPTION:

source	Name of the file node or file nodes containing the source text to be compiled.
prog_lib	Name of the program library into which the Containers generated by this compilation will be placed.
out_src	Name of the file node that is to receive the reformatted source text. The NEW_SRC parameter will have no effect if the REFORMAT option is not in effect.
option_list	List of options that are in effect for this compilation.

Listing Control Options:

SOURCE	Produce a listing of the source text. Default: SOURCE
REFORMAT	Reformat the source, the result being reflected in the source listing, if present, and the out_src node, if specified. Default: REFORMAT
PRIVATE	If there is a source listing, text in the private part of a package specifier is to be listed in accordance with the selected Source or Reformat option, subject to requirements of LIST pragmas. Default: PRIVATE

NOTES — Include diagnostics of severity NOTE in the source listing, and in the diagnostic summary listing. Default: NO_NOTES

ATTRIBUTE — Produce a symbol attribute listing. Default: ATTRIBUTE

XREF — Produce a cross-reference listing. Default: NO_XREF

STATISTICS — Produce a statistics listing. Default: NO_STATISTICS

MACHINE — Produce a machine code listing if code is generated. Code is generated when CONTAINER_GENERATION is in effect and there are no diagnostics of severity ERROR, SYSTEM, or FATAL, and, if there are diagnostics of severity level WARNING, CODE_ON_WARNING is in effect. If a machine code listing is requested, and no code is generated, a diagnostic of severity NOTE is reported. Default: NO_MACHINE

DIAGNOSTICS — Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS

Other Options:

CODE_ON_WARNING — Generate code (and, if requested, a machine code listing) when there are diagnostics of severity level WARNING, provided there are no FATAL, SYSTEM, or ERROR diagnostics. NO_CODE_ON_WARNING means generate no code (and, if requested, no machine code listing) when there are diagnostics of severity level WARNING. Default: CODE_ON_WARNING

CONTAINER_GENERATION — Produce a Container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. If a Container is not produced because NO_CONTAINER_GENERATION is in effect, code is not generated (nor is a machine code listing, if requested). Default: CONTAINER_GENERATION

OPTIMIZE — Permit optimization in accordance with OPTIMIZE pragmas that appear in the text. When NO_OPTIMIZE is specified or is in effect by default, no optimization is performed, regardless of pragmas. When no optimize pragmas are included,

optimization tries to conserve code space.
Default: NO_OPTIMIZE

DISPOSITION:

IN	Source program if source parameter is .STDIN
OUT	Listings except for diagnostic summary
MSG	Diagnostic summary listing and other messages
RSTRING	Not used
RSTATUS	See Appendix 80
Container	Placed in specified program library

NOTES:

1. Five severity levels of diagnostic situations can arise. Refer to 3.7.1.3.4 for a complete description.
2. FAILS if source cannot be found
3. FAILS if program library cannot be found
4. FAILS if diagnostic severity level worse than WARNING occurs (or worse than ERROR if CODE_ON_WARNING is specified)
5. FAILS if unable to create a Container

EXAMPLE:

```
ADAMCF  (disk_drive, ground_sys, OPT=> xref)

-- compile disk_drive into the ground_sys program library

-- generate code for MCF target, produce cross-reference
-- listing
```

ALS COMMAND DESCRIPTION ADAVAX

NAME: ADAVAX - Ada compiler for VAX-11/780 target

FUNCTION: Translate one or more Ada compilation units, generating code for the VAX-11/780 target

FORMAT: ADAVAX (source,prog_lib[,NEW_SRC=>out_src][,OPT=>option_list])

PARAMETER DESCRIPTION:

source	Name of the file node or file nodes containing the source text to be compiled.
prog_lib	Name of the program library into which the Containers generated by this compilation will be placed.
out_src	Name of the file node that is to receive the reformatted source text. The NEW_SRC parameter will have no effect if the REFORMAT option is not in effect.
option_list	List of options that are in effect for this compilation.

Listing Control Options:

SOURCE	Produce a listing of the source text. Default: SOURCE
REFORMAT	Reformat the source, the result being reflected in the source listing, if present, and the out_src node, if specified. Default: REFORMAT
PRIVATE	If there is a source listing, text in the private part of a package specifier is to be listed in accordance with the selected Source or Reformat option, subject to requirements of LIST pragmas. Default: PRIVATE

NOTES — Include diagnostics of severity NOTE in the source listing, and in the diagnostic summary listing. Default: NO_NOTES

ATTRIBUTE — Produce a symbol attribute listing. Default: ATTRIBUTE

XREF — Produce a cross-reference listing. Default: NO_XREF

STATISTICS — Produce a statistics listing. Default: NO_STATISTICS.

MACHINE — Produce a machine code listing if code is generated. Code is generated when CONTAINER_GENERATION is in effect and there are no diagnostics of severity ERROR, SYSTEM, or FATAL, and, if there are diagnostics of severity level WARNING, CODE_ON_WARNING is in effect. If a machine code listing is requested, and no code is generated, a diagnostic of severity NOTE is reported. Default: NO_MACHINE

DIAGNOSTICS — Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS

Other Options:

CODE_ON_WARNING — Generate code (and, if requested, a machine code listing) when there are diagnostics of severity level WARNING, provided there are no FATAL, SYSTEM, or ERROR diagnostics. NO_CODE_ON_WARNING means generate no code (and, if requested, no machine code listing) when there are diagnostics of severity level WARNING. Default: CODE_ON_WARNING

CONTAINER_GENERATION — Produce a Container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. If a Container is not produced because NO_CONTAINER_GENERATION is in effect, code is not generated (nor is a machine code listing, if requested). Default: CONTAINER_GENERATION

FREQUENCY — Permit generation of code to monitor execution frequency at the basic block level. Default: NO_FREQUENCY.

OPTIMIZE — Permit optimization in accordance with OPTIMIZE

pragmas that appear in the text. When NO_OPTIMIZE is specified or is in effect by default, no optimization is performed, regardless of pragmas. When no optimize pragmas are included, optimization tries to conserve code space. Default: NO_OPTIMIZE

DISPOSITION:

IN	Source program if source parameter is .STDIN
OUT	Listings except for diagnostic summary
MSG	Diagnostic summary listing and other messages
RSTRING	Not used
RSTATUS	See Appendix 80
Container	Placed in specified program library

NOTES:

1. Five severity levels of diagnostic situations can arise. Refer to 3.7.1.3.4 for a complete description.
2. FAILS if source cannot be found
3. FAILS if program library cannot be found
4. FAILS if diagnostic severity level worse than WARNING occurs (or worse than ERROR if CODE_ON_WARNING is specified)
5. FAILS if unable to create a Container

EXAMPLE:

```
ADAVAX  (disk_drive, ground_sys, OPT=> xref)

-- compile disk_drive into the ground_sys program library

-- generate code for VAX 11/780 target, produce cross-reference
-- listing
```

ALS COMMAND DESCRIPTION ADDREF

NAME: ADDREF - Add association reference(s)

FUNCTION: Add reference(s) to an association of a node.

FORMAT: ADDREF (node,assocname=> value_list
|(pos_val_list))

PARAMETER DESCRIPTION:

node	Name of the node possessing the association
assocname	Name of the association receiving the new references. A new association is created if necessary.
value_list	List of references to be added to the association.
pos_val_list	List of (position,new_value) pairs which are interpreted in the following manner:
position	The position number of the reference to be added. Position must conform to the Ada rules for integer literals.
new_value	The reference to be placed at the specified position.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and errors
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if node does not exist
2. FAILS if the user does not have attr_change permission.
3. FAILS if an attempt is made to alter a KAPSE controlled association.

4. FAILS if a reference in value_list or pos_val_list is not a syntactically legal node name.

5. FAILS if a position in the pos_val_list is not a positive number.

6. When adding by value_lists:

   a) No use of pos_val_lists is allowed.

   b) The new references are appended after the references that are already part of the association.

7. When adding by pos_val_lists:

   a) No use of value_lists is allowed.

   b) The insertions are done in numerically ascending order. The position numbers refer to the final positions of the references and existing references are shifted to higher numbered positions as necessary to accomodate the new entries. Existing references are numbered from one in ascending order with no holes in the sequence.

   c) No holes are allowed in the occupied reference positions. If a specified position number lies beyond the first unoccupied position, then it is assigned to the first unoccupied position. In this case, a WARNING diagnostic is issued.

   d) If the pos_val_list contains multiple occurences of the same position, the rightmost pair is used, the others are discarded. In this case, a WARNING diagnostic is issued.

8. This tool does not differentiate between absolute and relative association references. The value of the current working directory does not implicitly prefix old_value or new_value parameters; nor does it prefix values of existing association references. (For input parameters, this can be achieved explicitly by using the #CWD substitutor.)

EXAMPLES:

```
ADDREF(signal_analyzer.doc, cross_ref=>(fft.doc, spectrum.doc))

-- appends the references "fft.doc" and "spectrum.doc" to the
-- "cross_ref" association of the "signal_analyzer.doc" node.
-- The association is created if it does not exist.
```

```
ADDREF(my_node,assoc_x=>(path1,street))

-- Creates the association "assoc_x" if it did not exist and appends
-- "path1" and "street" onto the association.

ADDREF(my_node,assoc_x=>((1,path1),(3,street)))

-- Insert "path1" and "street" into the association "assoc_x" in
-- positions 1 and 3.
```

ALS COMMAND DESCRIPTION ARCHIVE

NAME: ARCHIVE - Archive a set of file revisions

FUNCTION: Send a list of names of nodes to be archived to a protected file which will subsecquently be used as input to a rollout operation.

FORMAT: ARCHIVE ([NODE=>list_1][,FILE=>list_2][,OPT=>option_list])

PARAMETER DESCRIPTION:

list_1

list_2: The syntax of both list_1 and list_2 is a list of pathnames. At least one NODE or FILE parameter must be present. Execution of the ARCHIVE tool is a request by the user to the system operator(s) that the node(s) specified in the parameter list be rolled out. Each node specified for rollout whether by the NODE or FILE form, is a specific frozen revision of a file. The NODE form specifies directly, in the parameter, one or more nodes for rollout. The FILE form names one or more ALS text files each of which contains a list of pathnames for rollout, one per line.

The set of all nodes to be rolled out is appended to a system file of to-be-rolled-out pathnames which is intended to be subsequently referenced by an operator. In addition to transmitting the list, the archive tool checks that each node named is a frozen file revision whose availability attribute has the value on_line. Pathnames found to be invalid by this check are not sent to the to-be-rolled-out file; instead, appropriate diagnostic messages are written to the message file.

option_list

LIST Default: NO_LIST. LIST writes to standard output every pathname appended to the to-be-rolled-out file.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. For each node specified for rollout for which the user does not have attribute change access, the pathname is not written to the to-be-rolled-out file but is written to the message file along with a diagnostic message.

2. Displays a diagnostic if the nodes have already been archived.

EXAMPLE:

```
ARCHIVE  (FILE=>pre_config_c)

-- File pre_config_c is a file of pathnames, one per line, of frozen
-- file revisions to which the user has attribute-change access.
-- These names are appended to the to-be-rolled-out file.
```

ALS COMMAND DESCRIPTION ASMMCF

NAME: ASMMCF - ALS MCF Assembler

FUNCTION: Translates ALS MCF assembly code

FORMAT: ASMMCF (source,prog_lib[,OPT=>option_list])

PARAMETER DESCRIPTION:

source	Name of the source file
prog_lib	Name of the program library into which the source is to be assembled
option_list	
SOURCE	Produce a source listing or not. Default: SOURCE.
CONTAINER_GENERATION	Produce a Container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. Default: CONTAINER_GENERATION

DISPOSITION:

IN	If .STDIN is specified as the source parameter, the standard input is used to get the source.
OUT	Source listing and diagnostic message
MSG	Confirmation and command diagnostics (and diagnostic messages if NO_SOURCE is specified)
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if input source node does not exist
2. FAILS if the program library does not exist
3. FAILS if errors are found in the source text

EXAMPLE:

```
ASMMCF  (my_prog, my_plib)

-- gets the input source from the node named my_prog
-- in the current working directory and puts assembled
-- object code and source listing into the program
-- library named my_plib
```

ALS COMMAND DESCRIPTION ASMVAX

NAME: ASMVAX - VAX Assembler

FUNCTION: Translates ALS VAX-11/780 assembly code

FORMAT: ASMVAX (source,prog_lib[,OPT=>option_list])

PARAMETER DESCRIPTION:

source	Name of the source file
prog_lib	Name of the program library into which the source is to be assembled
option_list	
SOURCE	Produce a source listing or not. Default: SOURCE.
CONTAINER_GENERATION	Produce a Container if diagnostic severity permits. NO_CONTAINER_GENERATION means that no Container is to be produced, regardless of diagnostic severity. Default: CONTAINER_GENERATION

DISPOSITION:

IN	If .STDIN is specified as the source parameter, the standard input is used to get the source.
OUT	Source listing and diagnostic message
MSG	Confirmation and command diagnostics (and diagnostic messages if NO_SOURCE is specified)
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if input source node does not exist
2. FAILS if the program library does not exist
3. FAILS if errors are found in the source text

EXAMPLE:

```
ASMVAX  (my_prog, my_plib)

-- gets the input source from the node named my_prog
-- in the current working directory and puts assembled
-- object code and source listing into the program
-- library named my_plib
```

ALS COMMAND DESCRIPTION CHACC

NAME: CHACC - Change access_name attribute

FUNCTION: Change access name attribute of an executable program to the invoking user's name

FORMAT: CHACC (file_name)

PARAMETER DESCRIPTION:

file_name	Name of the file containing the program

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the file does not exist
2. FAILS if the user does not have ATTR_CHANGE access
3. FAILS if file does not have CATEGORY attribute with value of "EXECUTABLE".

EXAMPLE:

```
CHACC  (new_tool)

-- sets access_name attribute of "new_tool" to user name
-- of the issuer
```

ALS COMMAND DESCRIPTION CHASS

NAME: CHASS - Change the value of an entire association

FUNCTION: Replace all references of an association with new references.

FORMAT: CHASS (node,assocname=>value_list)

PARAMETER DESCRIPTION:

node	Name of the node possessing the association.
assocname	Name of the association whose references are to be changed.
value_list	List of references to replace all of the current references of the association. An empty list ("") causes the association to be entirely deleted.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if node does not exist.
2. FAILS if the user does not have attr_change permission.
3. FAILS if an attempt is made to alter a KAPSE controlled association.
4. FAILS if a reference in value_list is not a syntactically legal node name.
5. This tool does not differentiate between absolute and relative asociation references. The value of the current working directory does not implicitly prefix elements of value_list; nor does it prefix values of existing association references. (For input parameters, this can be achieved explicitly by using the #CWD substitutor.)

EXAMPLES:

```
    CHASS(netwk_comm.source,include_file=>(incl_a,incl_b))

    -- makes the "include_file" association of the node
    -- "netwk_comm.source" refer to just the nodes "incl_a"
    -- and "incl_b"

    CHASS (gyro_control,documentation=>"")

    -- deletes the "documentation" association from the "gyro_control"
    -- node
```

ALS COMMAND DESCRIPTION CHATTR

NAME: CHATTR - Change the value of an attribute

FUNCTION: Changes the value of a node attribute. A new attribute is created, or an old one deleted automatically as appropriate.

FORMAT: CHATTR(node,attrname=> new_value
|(old_substring,new_substring))

PARAMETER DESCRIPTION:

node	Name of the node possessing the attribute.
attrname	Name of the attribute whose value is to be changed.
new_value	The new value of the attribute. The string must be enclosed in quotes (") if it contains any delimiter characters.
old_substring	The substring to be changed in the attribute value. The string must be enclosed in quotes (") if the string contains any delimiter characters.
new_substring	The substring to substitute in place of the old_substring. If old_substring is the null string, then new_substring is appended to the end of the attribute. If new_substring is the null string, then the old_substring is just deleted from the attribute. The string must be enclosed in quotes (") if it contains any delimiter characters.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if node does not exist.

2. FAILS if the user does not have attr_change permission.

3. FAILS if an attempt is made to alter a KAPSE controlled attribute. This includes the access_name attribute.

4. If quotes (") are to be embedded in the attribute value, then they must be doubled and the string must be enclosed in quotes.

5. A new attribute is created if one does not already exist and its new value is not null; an attribute is deleted if its new value is null.

6. Each (old_substring,new_substring) pair substitutes for only one occurrence of the substring and does no substitution if the old_value cannot be found in the attribute. If old_value cannot be found, a WARNING diagnostic is issued.

EXAMPLES:

```
CHATTR(my_node,purpose=>"create a new attribute")

-- create a new attribute if it did not already exist and give it a
-- value.

CHATTR(my_node,purpose=>("","  add a string to the purpose attribute"))

-- add a string to the purpose attribute value.

CHATTR(my_node,read=>("thruster.roll.smith/","thruster.roll.jones/"))

-- replace "thruster.roll.smith/" with "thruster.roll.jones/".

CHATTR(my_node,no_access=>("ada.doc.smith/ada.soft.jones/",""))

-- delete a substring of an attribute value.

CHATTR(my_node,purpose=>"")

-- delete an existing attribute.
```

ALS COMMAND DESCRIPTION CHREF

NAME: CHREF - Change some references of an association

FUNCTION: Alter selected references within the association.

FORMAT: CHREF(node,assocname=> (val_val_list)
|(pos_val_list))

PARAMETER DESCRIPTION:

node	Name of the node possessing the association.
assocname	Name of the association whose references are to be changed.
val_val_list	List of (old_value,new_value) pairs which are interpreted in the following manner:
old_value	The reference to be changed in the association.
new_value	The reference to substitute in place of the old_value in the association. If the new_value is null (i.e., ""), the old_value is just deleted from the association. If the old_value is null (i.e., ""), the new_value is appended to the end of the association.
pos_val_list	List of (position,new_value) pairs which are interpreted in the following manner:
position	The position number of the reference to be changed. Position must conform to the Ada rules for integer literals.
new_value	The reference to substitute in place of the existing reference at the specified position. If the new_value is null (i.e., ""), the reference at the specified position is deleted from the association.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used

RSTATUS See Appendix 80

NOTES:

1. FAILS if node does not exist or if the association does not exist for the node.

2. FAILS if the user does not have attr_change permission.

3. FAILS if an attempt is made to alter a KAPSE controlled association.

4. FAILS if a reference in val_val_list, or pos_val_list is not a syntactically legal node name.

5. FAILS if a position in the pos_val_list is not a positive number.

6. When replacing by val_val_lists:

   a) No use of pos_val_lists is allowed.

   b) The substitutions of (old_value,new_value) pairs is performed in left to right order from the value_pair list.

   c) For each pair, the existing references in the association are scanned in ascending position number order starting at one until a match with old_value is found. The first matching reference is then changed to new_value. The change is effective for the next value pair in the val_val_list. If the old value is not found, a warning diagnostic is issued. Processing of remaining list entries continues.

   d) Each (old_value,new_value) pair can substitute for only one reference and does no substitution if the old_value is not found.

7. When replacing by pos_val_lists pairs:

   a) No use of value_lists or val_val_lists is allowed.

   b) The substitutions are done in numerically ascending order. The positions refer to the original positions of the references, which are numbered from one in ascending order with no holes in the sequence.

   c) If a position in a (position,new_value) pair is greater than any defined reference position, the new_value is not substituted. In this case, a WARNING diagnostic is issued.

d) If the pos_val_list contains multiple occurrences of the same position, the rightmost value will be the value set at that position. In this case, a WARNING diagnostic is issued.

8. This tool does not differentiate between absolute and relative association references. The value of the current working directory does not implicitly prefix old_value or new_value parameters; nor does it prefix values of existing association references. (For input parameters, this can be achieved explicitly by using the #CWD substitutor.)

EXAMPLES:

```
CHREF(my_node,assoc_x=>(path1,path2))
                      or
CHREF(my_node,assoc_x=>(3,path2))

-- Given the association assoc_x with the following values:
--        "trail road path1 street"
-- both replace "path1" with "path2". The first by value pair
-- replacement, the second by position value pair replacement.

CHREF(my_node,assoc_x=>((a,w),(d,x),(f,y),("",z))

-- Given that assoc_x contains the following values: "a,b,c,d,e,f,g".
-- The new assoc_x is:   ""w,b,c,x,e,y,g,z".
```

ALS COMMAND DESCRIPTION CHTEAM

NAME: CHTEAM - Change team id

FUNCTION: To change the user's team id without forcing a logoff-logon sequence

FORMAT: CHTEAM (team_id)

PARAMETER DESCRIPTION:

team_id	Name of the new team

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the user is not authorized to be a member of the new team
2. FAILS if the new team does not exist

EXAMPLE:

```
CHTEAM  (tel_switch.load_simul)

-- changes team_id to "tel_switch.load_simul"
```

ALS COMMAND DESCRIPTION CHWDIR

NAME: CHWDIR - Change the working directory

FUNCTION: Changes the working directory to the named node and updates the value of the CWD substitutor

FORMAT: CHWDIR (name)

PARAMETER DESCRIPTION:

name	Pathname of the new working directory

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the user does not have read access to the new directory
2. FAILS if the node named by the pathname does not exist or is not a directory

EXAMPLE:

```
CHWDIR  (another_directory)

-- changes the current working directory to another_directory
```

ALS COMMAND DESCRIPTION CMPFILE

NAME: CMPFILE - Compare file nodes

FUNCTION: Compare the contents of the data parts of two file nodes and reports whether they are identical.

FORMAT: CMPFILE (path_name_1,path_name_2)

PARAMETER DESCRIPTION:

path_name_1:	The name of the first of the two files to be compared
path_name_2:	The name of the second of the two files to be compared

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and message output
RSTRING	"EQUAL" if the data parts are identical bit-for-bit, "UNEQUAL" otherwise.
RSTATUS	See Appendix 80

NOTES:

1. FAILS if either node does not exist.
2. FAILS if the user does not have read access to both nodes.
3. FAILS if either node is not a file.

EXAMPLE:

```
CMPFILE  (my_file(+),my_file)

-- The data parts of myfile(+) and my_file are compared bit-for-bit,
-- independent of their structures.  If they are identical, "EQUAL" is
-- returned to RSTRING; otherwise "UNEQUAL" is returned.
```

ALS COMMAND DESCRIPTION CMPNODE

NAME: CMPNODE - Compare nodes except for file data

FUNCTION: Compare the contents of the attribute, association, and/or offspring lists of two nodes, or the names of the attributes and/or associations.

FORMAT: CMPNODE (path_name_1,path_name_2
[,ATTR=>attr_list][,ASSOC=>assoc_list]
[,OPT=>option_list])

PARAMETER DESCRIPTION:

path_name_1: The name of the first of the two nodes to be compared

path_name_2: The name of the second of the two nodes to be compared

attr_list: Attr_list is a list of attribute names. The values of the specified attributes are compared and a list of differences among these values is written to standard output.

assoc_list: Attr_list is a list of association names. The values of the specified associations are compared and a list of differences among these values is written to standard output.

option_list:

OFFSPRING If the OFFSPRING option is selected, the nodes being compared must be either both directories or both variation headers. OFFSPRING writes to standard output a list of differences between the lists of immediate offspring names of the first and second nodes.
Default: NO_OFFSPRING.

ATTR_NAMES The list of names of the attributes defined for the first node is compared to the list of names of the attributes defined for the second node. If one node has attribute names not defined for the other node, the differences are written to standard output.
Default: NO_ATTR_NAMES.

ASSOC_NAMES The list of names of the associations defined for the first node is compared to the list of names of the associations defined for the second node. If one node has association

	names not defined for the other node, the differences are written to standard output. Default: NO_ASSOC_NAMES.

DISPOSITION:

IN	Not used
OUT	Results of the node comparison
MSG	Confirmation and message output
RSTRING	"EQUAL" if the portions being compared are all equal, "UNEQUAL" otherwise.
RSTATUS	See Appendix 80

NOTES:

1. FAILS if either node does not exist.
2. FAILS if the user does not have proper access to both nodes.
3. FAILS if neither ATTR nor ASSOC parameter is present and none of the options is selected.
4. FAILS if the OFFSPRING option is selected and the two nodes are not both directories or not both variation headers.

EXAMPLE:

```
CMPNODE  (my_node,std_node,opt=>(attr_names,assoc_names))

-- Produces a list of any attribute or association name differences
-- between my_node and std_node.
```

ALS COMMAND DESCRIPTION CMPTEXT

NAME: CMPTEXT - Compare text files

FUNCTION: Compare the contents of the data parts of two text file nodes.

FORMAT: CMPTEXT (path_name_1,path_name_2
[,OPT=>option_list])

PARAMETER DESCRIPTION:

path_name_1:	The name of the first of the two text files to be compared.
path_name_2:	The name of the second of the two text files to be compared.
option_list:	
LIST	A report showing the correspondence between the two files is generated on standard output. Default: NO_LIST.
SCRIPT	Write to standard output a file which may be used as input to the EDIT tool to make the data part of the first file the same as the data part of the second file. A script is always produced, even when the files are equal. LIST and SCRIPT may not both be selected. Default: NO_SCRIPT.
VERIFY	Check the lines of the file with a string comparison to verify that all reported matches are really identical lines. Default: NO_VERIFY.

DISPOSITION:

IN	Not used
OUT	Results of the node comparison
MSG	Confirmation and message output
RSTRING	"EQUAL" if the data parts are identical. "UNEQUAL" otherwise.
RSTATUS	See Appendix 80

NOTES:

1. FAILS if either node is not a text file.

2. FAILS if the user does not have read access to both nodes.

3. FAILS if both LIST and SCRIPT are selected.

4. For increased performance, the file comparator CMPTEXT uses a hashing algorithm which can, very infrequently, consider two differing lines to be identical. In this sense, the algorithm is approximate. (The algorithm will never consider two identical lines to be different.) In those circumstances when an exact comparison is necessary, the VERIFY option should be used. Though this option will increase the execution time used by CMPTEXT, it will insure that all differences are detected. The probability of an undetected difference is zero with the VERIFY option and less than one in 100,000,000 lines without VERIFY.

EXAMPLES:

Suppose two text files, file_a and file_b, have the following contents:

file a

```
This is line one   of file A, and line one   of B.
This is line two   of file A, and line three of B.
This is line three of file A, and not in file   B.
This is line four  of file A, and line eight of B.
This is line five  of file A, and line four  of B.
This is line six   of file A, and line five  of B.
This is line seven of file A, and line six   of B.
This is line eight of file A, and not in file   B.
This is line nine  of file A, and not in file   B.
This is line ten   of file A, and not in file   B.
This is last line  of both, different in B.
```

file b

```
This is line one   of file A, and line one   of B.
This line is not   in file A, and line two   of B.
This is line two   of file A, and line three of B.
This is line five  of file A, and line four  of B.
This is line six   of file A, and line five  of B.
This is line seven of file A, and line six   of B.
This line is not   in file A, and line seven of B.
This is line four  of file A, and line eight of B.
This line is not   in file A, and line nine  of B.
This line is not   in file A, and line ten   of B.
This line is not   in file A, and line eleven of B.
This is last line of both, modified in B.
```

The following command:

```
CMPTEXT (file_a,file_b,opt=>list)
```

will cause a comparison listing to be generated on Standard Output. In these reports, the primary file is the file specified by path_name_1 and the secondary file is specified by path_name_2. The report is structured to show what changes would have to be applied to the primary file to yield the text of the secondary file. The first column shows the line numbers in the primary file, the second column shows the line numbers in the secondary file, and the third column shows the actual text of the lines. A line is the smallest unit of text compared. Where a contiguous block of lines is treated in the same way, only the first and last lines of the block are shown. The interior lines of the block are reduced to an ellipsis (...). If a line appears in the secondary file but not the primary, the primary line number is shown as "INS" meaning that the line has been inserted into the secondary file. In the opposite case, the secondary line number is shown is "DEL" indicating that the line has been deleted from the secondary file. Where a block of lines is involved, "END-INS" and "END-DEL" are used to show, respectively, the last lines of the inserted and deleted blocks. In the case where a line or block is moved, the primary and secondary line numbers will be shown in the respective columns. The above command will generate the following report:

```
PRIMARY:   file_a
SECONDARY: file_b

PRIMARY    SECONDARY      TEXT

1          1          This is line one   of file A, and line one   of B.
INS        2          This line is not   in file A, and line two   of B.
2          3          This is line two   of file A, and line three of B.
3          DEL        This is line three of file A, and not in file   B.
5          4          This is line five  of file A, and line four  of B.
..         ...        ...
7          6          This is line seven of file A, and line six   of B.
INS        7          This line is not   in file A, and line seven of B.
8          DEL        This is line eight of file A, and not in file   B.
..         ...        ...
11         END-DEL    This is last line of both, different in B.
4          8          This is line four  of file A, and line eight of B.
INS        9          This line is not   in file A, and line nine  of B.
..         ...        ...
END-INS    12         This is last line of both, modified in B.
```

Note that the block of lines 5 through 7 of the primary file have been moved to lines 4 through six of the secondary file. The sense of the comparison is entirely dependent upon the order in which the files are specified in the command line. If the files are specified in the reverse order, the following report will result:

```
PRIMARY:   file_b
SECONDARY: file_a

PRIMARY    SECONDARY      TEXT

1          1         This is line one   of file A, and line one   of B.
2          DEL       This line is not   in file A, and line two   of B.
3          2         This is line two   of file A, and line three of B.
INS        3         This is line three of file A, and not in file   B.
8          4         This is line four  of file A, and line eight of B.
9          DEL       This line is not   in file A, and line nine  of B.
...        ...       ...
12         END-DEL   This is last line of both, modified in B.
4          5         This is line five  of file A, and line four  of B.
...        ...       ...
6          7         This is line seven of file A, and line six   of B.
INS        8         This is line eight of file A, and not in file   B.
...        ...       ...
END-INS    11        This is last line of both, different in B.
7          DEL       This line is not  in file A, and line seven  of B.
```

CMPTEXT can be used to generate a script for EDT that will convert the primary file into the secondary file. The following command:

```
CMPTEXT (file_a,file_b,opt=>script) OUT=>SCRIPT.A
```

will yield the following script:

```
MOVE 1 TO 2
INSERT;THIS line is not in file A, and line two   of B.
MOVE 2 TO 4
DELETE 3
MOVE 5 thru 7 TO 7
INSERT;This line is not in file A, and line seven of B.
DELETE 8
DELETE 9
DELETE 10
DELETE 11
MOVE 4 TO 9
INSERT;This line is not in file A, and line nine  of B.
INSERT;This line is not in file A, and line ten   of B.
INSERT;This line is not in file A, and line eleven of B.
INSERT;This is last line of both, modified in B.
EXIT
```

If the following command is issued:

```
CMPTEXT (my_file,your_file)
```

the data parts of text files my_file and your_file are compared. If they are identical, RSTRING="EQUAL"; otherwise, RSTRING="UNEQUAL".

ALS COMMAND DESCRIPTION CONCAT

NAME: CONCAT - Concatenate files together

FUNCTION: Concatenate the data parts of text files to standard output.

FORMAT: CONCAT ([file_name {,file_name}] [,FILES => file_list])

PARAMETER DESCRIPTION:

file_name	A text file containing text to be concatenated.
file_list	A text file containing names of files to be concatenated, one file name per line.

DISPOSITION:

IN	Not used
OUT	Output of concatenated files
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. Output contains the files named in file_name first, then the files in the file_list.
2. FAILS if the files do not exist or access rights do not grant read permission.
3. FAILS if either file_name or file_list is not specified.
4. FAILS if the output file also appears in the input file list.

EXAMPLE:

```
CONCAT(this_file,that_file,another_file) out=> new_file;

-- concatenates this_file, that_file, and another_file together,
-- with output in new_file
```

ALS COMMAND DESCRIPTION CPYALL

NAME: CPYALL - Copy data, attributes and associations of a file

FUNCTION: Copy all information in an existing node to a new node. The new node is automatically created. Attributes initialized by revise are not copied.

FORMAT: CPYALL (from_file,to_file)

PARAMETER DESCRIPTION:

from_file	Name of file to be copied
to_file	Name of file to copy into

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. Both file names must be environment database file names (i.e., not VMS names)
2. FAILS if from_file does not exist, or issuer does not have read access
3. FAILS if to_file already exists
4. Fails if issuer does not have append or write access to the directory that will contain to_file

EXAMPLE:

```
CPYALL (.xyz_project.io.control, io)

-- copies file "io_control" from "xyz_project" directory
-- to file "io" in CWD; data, attributes and associations
-- are copied
```

ALS COMMAND DESCRIPTION CPYDATA

NAME: CPYDATA - Copy data part of a file

FUNCTION: Copy only data part of one file to another; destination file may exist, it will be created if it does not

FORMAT: CPYDATA (from_file,to_file)

PARAMETER DESCRIPTION:

from_file	Name of file to be copied
to_file	Name of file to copy into

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if issuer does not have read access to from_file
2. Fails if to_file must be created and issuer does not have append or write access to the directory that will contain it
3. FAILS if to_file exists and issuer does not have write access
4. If the to_file is the name of an existing VMS file but does not contain a revision number, then a file will be created with a revision number one higher than the highest existing revision of the file.

EXAMPLE:

```
CPYDATA (fft.source,temp)

-- copies data of "fft.source" file to "temp" file
```

ALS COMMAND DESCRIPTION — DATE

NAME: DATE - Display today's date

FUNCTION: Displays the current date in the format dd-mmm-yyyy, where:

dd	is the two-digit day of the month; leading zero is not suppressed
mmm	is the three-character month name, i.e., JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, or DEC.
yyyy	is the four digits of the year.

FORMAT: DATE ([OPT=>option_list])

PARAMETER DESCRIPTION:

option_list

DISPLAY	Causes the date to be displayed on the standard output. NO_DISPLAY causes the date to be put in RSTRING. Default: DISPLAY

DISPOSITION:

IN	Not used
OUT	Date string unless NO_DISPLAY is specified
MSG	Confirmation
RSTRING	Date if NO_DISPLAY is specified
RSTATUS	See Appendix 80

NOTES:
The date is obtained from the VAX/VMS operating system. There is no way to guarantee that the clock is correct.

EXAMPLE(S):

```
DATE                    -- typed by user
04-JAN-1982             -- typed by the DATE tool
DATE (OPT=>no_display)  -- typed by user
ECHO ("#RSTRING")       -- typed by user
04-JAN-1982             -- typed by the echo tool
```

ALS COMMAND DESCRIPTION DEBUGVMS

NAME: DEBUGVMS - Debug an Ada program

FUNCTION: Initiates the symbolic debugger for an Ada program that has been exported to the VAX/VMS target. The program will be executed on the host computer under the ALS.

FORMAT: DEBUGVMS

PARAMETER DESCRIPTION:

No parameters

DISPOSITION:

IN	Subcommands to the debugger
OUT	Normal part of the debugger-to-user dialogue
MSG	Confirmation messages and notification of any fatal errors in the debugger itself
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. Depending upon how the program under test is started, the program under test and the debugger may share IN, OUT, and MSG.

EXAMPLE:

```
DEBUGVMS
```

INTERACTIVE SUBCOMMANDS:

See Appendix 100.

ALS COMMAND DESCRIPTION DELNODE

NAME: DELNODE - Delete a node and all of its offspring

FUNCTION: Delete a node revision set, or entire subtree

FORMAT: DELNODE (node,[OPT=>option_list])

PARAMETER DESCRIPTION:

node	Name of node to be deleted
option_list	
CONFIRM	This will cause the subtree confirmation to be requested, i.e., confirmation will be requested to delete a subtree. NO_CONFIRM will cause the subtree confirmation to be supressed (this is intended for use within command procedures.). Default: CONFIRM.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if node does not exist.
2. FAILS if issuer does not have write access to containing directory (header) of node, or to containing directory of the revision set in the case of an individual deletion.
3. Requests confirmation if node has offspring; the entire subtree will be deleted if confirmation given. The prompt and reply are passed over master output and input, respectively. The NO_CONFIRM option causes the messages to be suppressed.
4. FAILS if parent tries to delete an offspring shared by other directories.

5. FAILS if any node being deleted has a non-zero derivation count.

6. If the node name includes the "*" index, every revision node in the revision set is deleted. Otherwise, a single revision is deleted.

7. FAILS if the node being deleted is a single revision and its revision set is shared.

EXAMPLES:

```
DELNODE (temp_file(2))
-- deletes the second revision of temp_file

DELNODE (temp_file)
-- deletes the latest revision of temp_file

DELNODE (temp_file(*))
-- deletes all revisions of temp_file

DELNODE (beta, opt=> no_confirm)
-- deletes the subtree beta without asking for
-- confirmation of the deletion.
```

ALS COMMAND DESCRIPTION DELREF

NAME: DELREF - Delete association reference(s).

FUNCTION: Delete reference(s) from an association of a node.

FORMAT: DELREF (node,assocname=> value_list
|position_list)

PARAMETER DESCRIPTION:

node	Name of node possessing the association
assocname	Name of the association from which the references are to be deleted.
value_list	List of references for which all occurrences are to be deleted from the association.
position_list	The list of positions at which references are to be deleted from the association.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if node does not exist or if association does not exist for the node.
2. FAILS if user does not have attr_change permission.
3. FAILS if an attempt is made to alter a KAPSE controlled association.
4. FAILS if a reference in the value_list is not a syntactically legal node name.
5. FAILS if a position in the position_list is not positive.
6. If a specified position in a position_list is not occupied, a WARNING diagnostic is issued. Similarly, a WARNING is issued if an element in a value_list cannot be found in the association.

7. When deleting by value_lists:

    a) No use of position_lists is allowed.

    b) Only the first occurrence of a reference is deleted from the association. Each value_list element results in the deletion of at most one reference.

8. When deleting by position_lists:

    a) No use of value_lists is allowed.

    b) Deletion is done in descending order, from highest to lowest. The position numbers apply to the association prior to deletion of any reference.

    c) The remaining references are shifted to lower numbered positions to fill any holes in the occupied positions.

9. This tool does not differentiate between absolute and relative association references. The value of the current working directory does not implicitly prefix old_value or new_value parameters; nor does it prefix values of existing association references. (For input parameters, this can be achieved explicitly by using the #CWD substitutor.)

EXAMPLES:

```
DELREF (netwk_comm.source,include_files=>incl_b)

-- remove first occurrence of the reference "incl_b" from the
-- "include_files" association of the "netwk_comm.source" node.

DELREF(my_node,assoc_x=>(a,b,c))

-- Delete first occurrence of the references "a", "b", and "c" from
-- the association "assoc_x".

DELREF(my_node,assoc_x=>(1,3,5))

-- Delete references in positions "1", "3", and "5" from "assoc_x".
```

ALS COMMAND DESCRIPTION ECHO

NAME: ECHO - Display the arguments

FUNCTION: Arguments to the command are written to standard output

FORMAT: ECHO ({arg})

PARAMETER DESCRIPTION:

arg	One or more arguments

DISPOSITION:

IN	Not used
OUT	Arguments
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

A single space is inserted between pairs of arguments. If named parameters are specified, the name and value are separated by an arrow.

EXAMPLE(S):

```
ECHO ("the date is:")              -- typed by user
The date is:                       -- typed by ECHO

today :="24 August 1980"
ECHO ("the date is:", #today)      -- typed by user
the date is: 24 August 1980        -- typed by ECHO

ECHO (Today)                       -- typed by user
Today                              -- typed by ECHO
```

ALS COMMAND DESCRIPTION EDT

NAME: EDT - VAX-11 EDT Text Editor

FUNCTION: Invokes the VAX-11 EDT Text Editor.

FORMAT:
```
EDT (file [,output => output_file]
    [,command => command_file]
    [,opt => option_list] )
```

PARAMETER DESCRIPTION:

file	name of the file to edit
output_file	Name of the output file. If none is specified, the output file will be the same as the input file.
command_file	Name of a file containing editor commands which EDT will execute before prompting the user for input.
option_list:	
RECOVER	RECOVER will cause EDT to apply the previous session's edit commands to the input file. This allows the user to recover from aborted EDT sessions. Default: NO_RECOVER.
READ_ONLY	READ_ONLY will open the input file for read access. No output file will be created. This can be used to examine a file for which the user has no write priveleges. Default: NO_READ_ONLY.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages not generated directly by EDT.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. The use of EDT is described in the VAX-11 EDT Editor Reference Manual (AA-H944A-TE).

2. If file does not exist, it is created.

3. If the latest revision of the file is frozen, or has a non-zero derivation count, a new revision is created. Otherwise the latest revision is overwritten.

4. Communication with the user is via master input and master output.

5. As for any other ALS tool, file names specified in the EDT tool command line can refer to either ALS EDB nodes or to VMS files (indicated by the <<VMS>> construct). VAX-11 EDT, however, will interpret all file names given in EDT commands as VMS file names; therefore ALS EDB nodes cannot be referenced within EDT. In EDT commands, all file names must comply with VAX/VMS file name syntax. The <<VMS>> construct is not appropriate within the context of EDT.

6. FAILS if the input file is not a file node.

7. FAILS if the user does not have appropriate access to the files.

8. FAILS if the user specifies a command_file that does not exist.

EXAMPLE:

```
EDT  (my_prog)

-- creates an edit session for the EDB node "my_prog"

EDT  (my_prog, opt => recover)

-- creates an edit session for my_prog with recovery from the last
-- edit session of my_prog.  The edits from the last session of
-- editting my_prog will be applied to the file.


EDT  (my_prog, output => new_prog, command => editinit)

-- creates an edit session for my_prog, with the edited output
-- going to the EDB node "new_prog".  The editor commands in
-- the EDB node "edtinit" will be executed prior to
-- EDT prompting for input.


EDT  (<<VMS>>my_prog, command => <<VMS>>edtinit.edt)

-- creates an edit session for the VMS file "my_prog".  The
```

```
-- editor commands in the VMS file "edtinit.edt" will be executed
-- prior to EDT prompting for input.


EDT  (my_prog, opt => read_only)

-- creates a read-only edit session for my_prog.  The user
-- can examine the contents of my_prog, but cannot alter it
-- permanently.
```

ALS COMMAND DESCRIPTION EXPMCF

NAME: EXPMCF - Exporter to MCF target

FUNCTION: Transforms the input container to the MCF Target format necessary for execution, and writes the resulting output to the MCF target transport medium interface

FORMAT: EXPMCF (name,prog_lib, output_medium [,OP_SYS=>nucleus_name] [,OPT=>option_list])

PARAMETER DESCRIPTION:

name	Name of the linked Container to be exported
prog_lib	Name of the program library containing program to be exported
output_medium	The ALS name for the file where the load module is to be written
nucleus_name	Specifies the name of a Container that represents a runtime nucleus (other than the default runtime nucleus) to be written onto the output file. This option is only valid when the INITIAL option has been specified. If this option is not specified the Exporter will use the default runtime nucleus for the MCF Target.
option_list	
INITIAL	Commands the exporter to make this load module the first one on the output tape file or in the database file. If NO_INITIAL is specified, this load module is appended to other load modules already on the tape file or in the database file. Default: NO_INITIAL.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the program library does not exist or access rights are not appropriate

2. FAILS if linked Container to be exported does not exist

EXAMPLE:

```
EXPMCF  (my_node, my_lib, new_node)

-- generates MCF format for my_node from my_lib to new_node
```

ALS COMMAND DESCRIPTION EXPVMS

NAME: EXPVMS - Exporter to VAX/VMS target

FUNCTION: Transforms the input container to the VAX/VMS target format necessary for execution, and writes the resulting output to the VAX/VMS target transport medium interface

FORMAT: EXPVMS (name,prog_lib, output_medium [,OPT => option_list])

PARAMETER DESCRIPTION:

name	Name of the linked Container to be exported
prog_lib	Name of the program library containing program to be exported
output_medium	The ALS name for the file where the load module is to be written
option_list	
DEBUG	Directs the Exporter to activate the Debugger Kernel in the program image to allow debugging. Default: NO_DEBUG.
FREQUENCY	Activate the frequency kernel to monitor execution frequency. When NO_FREQUENCY is specified, or is in effect by default, no execution frequency is monitored, regardless of compile options. Default: NO_FREQUENCY.
STAT	Activate the timing kernel to monitor execution timing. When NO_STAT is specified, or is in effect by default, no execution timing is monitored. Default: NO_STAT.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the program library does not exist or access rights are not appropriate

2. FAILS if the input Container does not exist

EXAMPLE:

```
EXPVMS  (my_prog, my_lib, my_file)

-- generates target format for my_prog in my_lib
-- which is written to the ALS file my_file
```

ALS COMMAND DESCRIPTION FREEZE

NAME: FREEZE - Freeze latest version

FUNCTION: Freezes the latest revision of a file, making it unchangeabln

FORMAT: FREEZE (file_name)

PARAMETER DESCRIPTION:

file_name	Name of the file whose latest revision is to be frozen

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and diagnostics
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the named node is not a file, or user does not have READ access.

EXAMPLE:

```
FREEZE  (program.source)

-- make latest revision of the file "program.source" unchangeable
```

ALS COMMAND DESCRIPTION GENLISTMCF

NAME: GENLISTMCF - generate listing for MCF target

FUNCTION: generates listings from the specified Containers produced by the ADAMCF, ASMMCF, or LNKMCF tools.

FORMAT: GENLISTMCF(Ada_name, prog_lib [,OPT => option_list])

PARAMETER DESCRIPTION:

prog_lib	The name of the program library
Ada_name	The name of the Container from which the listing is to be extracted.
option_list	Genlist options

The following option is valid only if the creator of the Container was ADAMCF or ASMMCF.

SOURCE	Produce a source listing (Note: the compiler source listing will be reformatted only if the original compilation specified the REFORMAT option). Default:NO_SOURCE.

The following options are valid only if the creator of the Container was LNKMCF.

SYMBOLS	Provide a symbol definition listing. Default: NO_SYMBOLS.
LOCAL_SYMBOLS	If a symbol definition listing is produced, include names local to library package bodies. Default: NO_LOCAL_SYMBOLS, means include only names which are externally visible.
UNITS	Provide a units listing. Default: NO_UNITS.

The following options are valid only if the creator of the Container was ADAMCF

LIST_INCLUDE	If there is a source listing, text brought in with an INCLUDE pragma should be listed, subject to requirements of list pragmas. Default: NO_LIST_INCLUDE.
PRIVATE	If there is a source listing, text in the private part of a package specifier is to be listed, subject to specifications of LIST pragmas. Default: PRIVATE.

NOTES	Include diagnostics of severity NOTE in source listing, and in the Diagnostic Summary Listing. Default: NOTES.
ATTRIBUTE	Produce a symbol attribute listing. When both a Symbol Attribute Listing and a Cross Reference Listing are requested, a single listing, called the Attribute-Cross-Reference Listing, containing both types of information will be produced. Default: NO_ATTRIBUTE.
XREF	Produce a cross-reference listing. When both a Symbol Attribute Listing and a Cross Reference Listing are requested, a single listing, called the Attribute-Cross-Reference Listing, containing both types of information will be produced. Default: NO_XREF
STATISTICS	Produce a statistics listing. Default: NO_STATISTICS
MACHINE	If there is machine code, produce a machine code listing, otherwise produce a diagnostic of severity level WARNING. Default: NO_MACHINE
DIAGNOSTICS	Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS

DISPOSITION:

IN	Not used
OUT	All listings except for the diagnostic summary listing.
MSG	Diagnostic summary listing and other messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if program library cannot be found
2. FAILS if Container cannot be found
3. FAILS if the option is not appropriate for the creator.

ALS COMMAND DESCRIPTION GENLISTVAX

NAME: GENLISTVAX - generate listing for VAX-11/780 target

FUNCTION: generates listings from the specified Containers produced by the ADAVAX, ASMVAX, LNKVMS or LNKVAX tools.

FORMAT: GENLISTVAX(Ada_name, prog_lib [,OPT => option_list])

PARAMETER DESCRIPTION:

prog_lib	The name of the program library
Ada_name	The name of the Container from which the listing is to be extracted.
option_list	Genlist options

The following option is valid only if the creator of the Container was ADAVAX or ASMVAX.

SOURCE	Produce a source listing (Note: the compiler source listing will be reformatted only if the original compilation specified the REFORMAT option). Default: NO_SOURCE.

The following options are valid only if the creator of the Container was LNKVAX or LNKVMS.

SYMBOLS	Provide a symbol definition listing. Default: NO_SYMBOLS.
LOCAL_SYMBOLS	If a symbol definition listing is produced, include names local to library package bodies. Default: NO_LOCAL_SYMBOLS, means include only names which are externally visible.
UNITS	Provide a units listing. Default: NO_UNITS.

The following options are valid only if the creator of the Container was ADAVAX

LIST_INCLUDE	If there is a source listing, text brought in with an INCLUDE pragma should be listed, subject to requirements of list pragmas. Default: NO_LIST_INCLUDE.
PRIVATE	If there is a source listing, text in the private part of a package specifier is to be li ed, subject to specifications of LIST pragmas. Default: PRIVATE.

NOTES	Include diagnostics of severity NOTE in source listing, and in the Diagnostic Summary Listing. Default: NOTES.
ATTRIBUTE	Produce a symbol attribute listing. When both a Symbol Attribute Listing and a Cross Reference Listing are requested, a single listing, called the Attribute-Cross-Reference Listing, containing both types of information will be produced. Default: NO_ATTRIBUTE.
XREF	Produce a cross-reference listing. When both a Symbol Attribute Listing and a Cross Reference Listing are requested, a single listing, called the Attribute-Cross-Reference Listing, containing both types of information will be produced. Default: NO_XREF
STATISTICS	Produce a statistics listing. Default: NO_STATISTICS
MACHINE	If there is machine code, produce a machine code listing, otherwise produce a diagnostic of severity level WARNING. Default: NO_MACHINE
DIAGNOSTICS	Produce a diagnostic summary listing. Default: NO_DIAGNOSTICS

DISPOSITION:

IN	Not used
OUT	All listings except for the diagnostic summary listing.
MSG	Diagnostic summary listing and other messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if program library cannot be found
2. FAILS if Container cannot be found
3. FAILS if the option is not appropriate for the creator.

ALS COMMAND DESCRIPTION HELP

NAME: HELP - Interactive information tool

FUNCTION: Interactively supplies information from the HELP database.

FORMAT: HELP ([subject])

PARAMETER DESCRIPTION:

subject	Initial current subject for which to obtain information. If this parameter is ommitted, the initial current subject is the HELP root.

DISPOSITION:

IN	Subcommand input
OUT	Responses
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. HELP allows the user to interactively explore the HELP database. It can be used to obtain extensive information. The non-interactive tool QHELP also provides information from the HELP database. QHELP extracts only a single piece of information from the database, and is intended to be used for quick reference.

2. The HELP database is a tree of subjects. Direct offspring of the HELP root are main subjects. Each subject can be divided into subtopics which can also be further divided. There is no fixed limit to the number of levels in the HELP tree. Information can be obtained from each level in the tree.

3. HELP maintains a "current subject". Upon entry to HELP the current subject is either the subject named in the HELP command or the HELP database root when no subject is named. The user may change the current subject with the CHSUB subcommand.

4. The subject parameter is optional on all of HELP's subcommands. When it is omitted, the current subject is assumed.

5. The subject parameter can be either a main subject or a subtopic of a subject or another subtopic. Subject names are analogous to pathnames in the environment database where the root is the HELP database root and the current working directory is the current subject. Thus "." designates the HELP database root and "!" designates the parent subject of the current subject. Subject names are either absolute starting at the root or relative to the current subject.

   Example(s):

   ```
   .LIB.ACQUIRE

   -- The subject is ACQUIRE which is a subtopic of the subject
   -- LIB.  This is an absolute pathname indicating that LIB is
   -- a direct offspring of the root and thus is a main subject
   -- under HELP.

   ACQUIRE

   -- This subject is the subtopic ACQUIRE under the current
   -- subject.

   .

   -- The HELP database root

   !

   -- Parent subject of the current subject.
   ```

6. Fails if the subject does not exist.

EXAMPLE(S):

```
HELP (LIB)

-- Initiates HELP with the current subject, LIB.

HELP

-- Initiates HELP with the current subject being the HELP database
-- root.
```

INTERACTIVE SUBCOMMANDS:

ALS SUBCOMMAND DESCRIPTION HELP.CHSUB

SUBCOMMAND: CHSUB - Change the current subject

FUNCTION: Change the current subject. Allows movement through the HELP database.

FORMAT: CHSUB ([subject])

PARAMETER DESCRIPTION:

subject	Name of the new current subject.

NOTES:

1. Displays a diagnostic if the subject does not exist.

EXAMPLES:

```
CHSUB (.LIB)
-- Changes the current subject to LIB which is a main subject
-- under HELP.

CHSUB (ACQUIRE)
-- Changes the current subject to ACQUIRE which is a subtopic
-- under the current subject.

CHSUB (.)
-- Changes the current subject to the root.

CHSUB (!)
-- The parent of the current subject becomes the new current
-- subject.
```

ALS SUBCOMMAND DESCRIPTION HELP.EXIT

SUBCOMMAND: EXIT - terminates a HELP session

FUNCTION: terminates a help session

FORMAT: EXIT

EXAMPLE:

```
EXIT
-- terminates HELP
```

ALS SUBCOMMAND DESCRIPTION HELP.FORMAT

SUBCOMMAND: FORMAT

FUNCTION: Display the command format of a subject

FORMAT: FORMAT ([subject])

PARAMETER DESCRIPTION:

subject	Subject whose command format should be displayed.

NOTES:

1. Displays a diagnostic if the subject does not exist.
2. Displays a diagnostic if there is no command format for the subject.

EXAMPLE(s):

```
FORMAT (LST)
-- The format of the LST command is displayed.
```

ALS SUBCOMMAND DESCRIPTION HELP.INFO

SUBCOMMAND: INFO - supply information on a subject

FUNCTION: Displays information

FORMAT: INFO ([subject])

PARAMETER DESCRIPTION:

subject	Subject for which information should be displayed

NOTE:

1. Displays a diagnostic if the subject does not exist.

EXAMPLE(s):

```
INFO (.LSTATTR)
-- Gives information on the tool, LSTATTR.

INFO
-- Displays information on the current subject.

INFO (.)
-- Displays information on the HELP tool itself.
```

ALS SUBCOMMAND DESCRIPTION HELP.LSTSUB

SUBCOMMAND: LSTSUB - Display a directory of subjects

FUNCTION: Displays a directory of subjects.

FORMAT: LSTSUB ([subject] [,opt=> option_list])

PARAMETER DESCRIPTION:

subject — Subject for which the subtopics will be listed.

option_list:

TREE — Causes all of the subtopics in the subject's subtree to be displayed in indented form. NO_TREE means to list only the immediate subtopics. Default NO_TREE.

NOTES:

1. Displays a diagnostic if subject does not exist.

2. Displays a diagnostic if no subtopics exist for the subject.

EXAMPLE(s):

```
LSTSUB (.)
-- Lists all the main subjects for which help is available.

LSTSUB (LIB)
-- Lists the subtopics of LIB.

LSTSUB (.LIB.ACQUIRE)
-- Lists the subtopics of ACQUIRE which is a subtopic of LIB.
```

ALS COMMAND DESCRIPTION LIB

NAME: LIB - Program Library Manager

FUNCTION: Interactive program library manager; has subcommands for creating, examining and manipulating program libraries.

FORMAT: LIB ([prog_lib])

PARAMETER DESCRIPTION:

prog_lib	Name of program library to be used

DISPOSITION:

IN	Command input
OUT	Responses
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. LIB temporarily changes the CWD to be the PL directory node.
2. All interactive subcommands operate on the current program library as specified by the user either via the prog_lib parameter on the LIB command or via a MKLIB interactive subcommand. The user must supply a current program library before he can use any LIB interactive subcommand other than MKLIB.
3. FAILS if the program library does not exist.
4. FAILS if the user does not have read access to the program library.

EXAMPLES:

```
LIB (my_pl)

-- Initiates a LIB session using the program library "my_pl".
```

INTERACTIVE SUBCOMMANDS:

ALS SUBCOMMAND DESCRIPTION LIB.ACQUIRE

SUBCOMMAND: ACQUIRE - Obtain Container from another PL.

FUNCTION: Obtain Containers from another PL.

FORMAT: ACQUIRE (prog_lib, Adaname)

PARAMETER DESCRIPTION:

prog_lib	Name of the program library which has the Containers to be acquired.
Adaname	Container or subtree to be acquired. If it is a subtree, the latest revision of each Container in the subtree will be acquired.

NOTES:

1. Displays a diagnostic if a current program library has not been specified.
2. Displays a diagnostic if the program library does not exist.
3. Displays a diagnostic if the Adaname does not exist.
4. Displays a diagnostic if the Ada compilation order rules are violated; that is, if the Containers being acquired depend on Containers which have not yet been acquired.
5. Displays a diagnostic if a subtree being acquired is internally incompatible; that is, if a Container in the subtree does not depend on the latest revision of another Container in the subtree.
6. Displays a diagnostic if a linked Container is being acquired and the Containers used to create the linked Container have not yet been acquired.
7. Displays a diagnostic if the target of the user's PL is not listed in the compatible_targets attribute of the Container to be acquired.

EXAMPLES:

```
ACQUIRE (from_prog_lib, my_prog.spec)
-- The latest revisic of the Container "my_prog.spec"
-- which resides in the program library "from_prog_lib"
-- is acquired into the user's program library.

ACQUIRE (from_prog_lib, my_prog.all)
```

```
-- The latest revision of each Container in the library
-- unit subtree, "my_prog" in the program library
-- "from_prog_lib" is acquired into the user's program
-- library

ACQUIRE (from_prog_lib, my_prog.body(2))
-- The second revision of the Container for the body of
-- the library unit "my_prog" in the program library,
-- "from_prog_lib" is acquired into the user's program
-- library.  (The specification for this revision of the
-- body must already have been acquired into the user's
-- program library.)

ACQUIRE (from_prog_lib, .all)
-- The latest revision of every Contianer in the program
-- library "from_prog_lib" is acquired.
```

ALS SUBCOMMAND DESCRIPTION LIB.ARCHIVE

SUBCOMMAND: ARCHIVE - archive Containers

FUNCTION: Send a list of Containers to be archived to a protected system file which will subsequently be used by the ALS.

FORMAT: ARCHIVE ([ADANAME => list_1] [,file=> list_2] [,OPT=>option_list])

PARAMETER DESCRIPTION:

list_1: list of Adanames of Containers.

list_2: List of pathnames of ALS text files which contain Adanames of Containers, one per line. At least one ADANAME or FILE parameter must be present.

option_list:

LIST Default: NO_LIST. LIST writes to standard output every Adaname appended to the system file.

NOTES:

1. Displays a diagnostic if a current program library has not been specified.
2. Displays a diagnostic if the Adaname does not include a revision number.
3. Displays a diagnostic if the Container has already been archived.
4. Displays a diagnostic if any node named by list_2 is not a text file.

EXAMPLES:

```
ARCHIVE (ADANAME=>my_prog.spec(2), my_prog.body(1))
-- The names of the Containers for the second revision
-- of the specification and the first revision of the
-- body of the library unit "my_prog" are sent to the
-- ALS for subsequent archiving.

ARCHIVE (FILE=>archive)
-- The names of the Contianers listed in the text file
-- "archive" in the user's directory are sent to the
-- ALS for subsequent archiving.
```

ALS SUBCOMMAND DESCRIPTION LIB.CHATTR

SUBCOMMAND: CHATTR - Change the value of an access attribute of a program library root node.

FUNCTION: Changes the users who can access a program library by changing an access attribute of the program library root node.

FORMAT: CHATTR (attrname => new_value
|(old_substring,new_substring))

PARAMETER DESCRIPTION:

attrname	Name of the access attribute whose value is to be changed. Any access attribute except via can be specified.
new_value	Value to be given to the attribute.
old_substring	The substring to be changed in the attribute value. The string must be enclosed in quotes (") if the string contains any delimiter characters.
new_substring	The substring to substitute in place of the old_substring. If old_substring is the null string, then new substring is appended to the end of the attribute. If new_substring is the null string, then the old_substring is just deleted from the attribute. The string must be enclosed in quotes (") if it contains any delimiter characters.

NOTES:

1. The current program library root node is implicitly the subject of the CHATTR.
2. The access attributes of the container nodes within the program library will be changed as appropriate to be consistent with the program library root node.
3. Displays a diagnostic if a current program library has not been specified.
4. Displays a diagnostic if the attrname names any attribute other than no_access, read, append, write, attr_change or execute.

5. Displays a diagnostic if the user does not have attr_change permission for the current program library.

EXAMPLES:

```
CHATTR (read => "thruster.roll.smith")

--  Changes the value of the read attribute of the current
--  program library to "thruster.roll.smith".

CHATTR(read=>("thruster.roll.smith/","thruster.roll.jones/"))

--  replace "thruster.roll.smith/" with "thruster.roll.jones/".

CHATTR(no_access=>("ada.doc.smith/ada.soft.jones/",""))

--  delete smith and jones from no_access.
```

ALS SUBCOMMAND DESCRIPTION LIB.DELETE

SUBCOMMAND: DELETE - delete a container

FUNCTION: Delete Containers or subtrees from a program library

FORMAT: DELETE (Adaname)

PARAMETER DESCRIPTION:

Adaname Container or subtree to be deleted

NOTES:

1. The user will be asked to reconfirm the deletion of an entire program library.
2. Displays a diagnostic if a current program library has not been specifed.
3. Displays a diagnostic if any Container in any program library references the Container(s) being deleted.
4. If a Container Adaname includes the revision index "*", every revision of the Adaname is deleted; otherwise a single revision of the Adaname is deleted.
5. Displays a diagnostic if the Adaname does no exist.
6. Displays a diagnostic if the user does not have write access to the program library.

EXAMPLES:

```
DELETE (my_prog.all)
-- Every Contianer in the library unit "my_prog" is deleted.

DELETE (my_prog.apsec(2))
-- The Container for the second revision of the specification of the
-- library unit "my_prog" is deleted

DELETE (.all)
-- Delete the entire program library.

DELETE (my_prog.body)
-- Delete the latest revision of the body for the library unit "my_prog".

DELETE (my_prog.body(*))
-- Delete every revision of the body for the library unit "my_prog".
```

ALS SUBCOMMAND DESCRIPTION LIB.EXIT

SUBCOMMAND: EXIT - terminates a LIB session

FUNCTION: Terminates a LIB session and returns the user to the ALS

FORMAT: EXIT

EXAMPLE:

```
EXIT
-- Terminates the LIB session
```

ALS SUBCOMMAND DESCRIPTION LIB.LST

SUBCOMMAND: LST - lists the contents of the program library

FUNCTION: Lists a directory of Adanames in a program library.

FORMAT: LST (Adaname)

PARAMETER DESCRIPTION:

Adaname	Adaname where the directory display should begin. If this parameter is omitted, a directory for the entire program library is given.

NOTES:

1. Displays a diagnostic if a current program library has not been specified.

2. Displays a diagnostic if the Adaname is non-existent.

EXAMPLES:

```
LST (my_prog)
-- Displays the Adanames in the library unit, "my_prog".

LST (my_prog.B))
-- Displays the Adanames in the subunit, "my_prog.B".

LST (my_prog.spec)
-- Displays the Adaname for the specification of the
-- library unit, "my_prog".

LST (.all)
-- Display a directory for the entire program library.
-- (same as LST with no parameter).
```

ALS SUBCOMMAND DESCRIPTION LIB.LSTASS

SUBCOMMAND: LSTASS - List association contents

FUNCTION: Lists the contents of an association

FORMAT: LSTASS (Adaname [,assocname])

PARAMETER DESCRIPTION:

Adaname	Container or subtree for which association is to be listed
assocname	Association to be listed

NOTES:

1. If the Adaname is the only parameter, then just the names of all associations possessed by the Container or subtree are listed.
2. If a subtree is named, only the Containers in the subtree possessing the specified association are listed.
3. Displays a diagnostic if a current program library has not been specified.
4. Displays a diagnostic if the Adaname does not exist.
5. Displays a diagnostic if the association does not exist.

EXAMPLES:

```
LSTASS (my_prog.body, DEPENDS_ON)
-- Lists the contents of the association, "DEPENDS_ON",
-- in the Container "my_prog.body".

LSTASS (my_prog.spec)
-- Lists all the associations possessed by the Container
-- "my_prog.spec".
```

ALS SUBCOMMAND DESCRIPTION LIB.LSTATTR

SUBCOMMAND: LSTATTR - List the value of an attribute

FUNCTION: Lists the value of an attribute.

FORMAT: LSTATTR (Adaname [,attrname])

PARAMETER DESCRIPTION:

Adaname	Container or subtree for which the attribute value is to be listed
attrname	Attribute to be listed

NOTES:

1. If the Adaname is the only parameter, then just the names of all attributes possessed by the Container or subtree are listed.
2. If a subtree is named, only the Containers in the subtree possessing the specified attribute are listed.
3. Displays a diagnostic if a current program library has not been specified.
4. Displays a diagnostic if the Adaname does not exist.
5. Displays a diagnostic if the attribute does not exist.

EXAMPLES:

```
LSTATTR (.all, TARGET)
-- Lists the value of the target attribute of the program library
-- root node.  No Containers are listed since Containers do not
-- possess the target attribute.

LSTATTR (my_prog.spec, creation_date)
-- Lists the value of the attribute, "creation_date" for
-- the Container "my_prog.spec".

LSTATTR (my_prog.body)
-- Lists all the attributes possessed by the Container
-- "my_prog.body".

LSTATTR (my_prog.all, creation_date)
-- Lists the values of the attribute "creation_date"
-- for every Container in the library unit "myprog".
```

ALS SUBCOMMAND DESCRIPTION LIB.MKLIB

SUBCOMMAND: MKLIB - Make a new program library

FUNCTION: Creates a new program library.

FORMAT: MKLIB (prog_lib [,TARGET=>target_name])

PARAMETER DESCRIPTION:

prog_lib	Name of the new program library.
target_name	Name of the intended target environment for all Containers in the program library. If this parameter is omitted, the default target is specified by the default_variation attribute of the system program library variation set. The default is initially the host, but can be changed by the system administrator. Target_name is a value of enumeration type SYSTEM.SYSTEM_NAME and must specify a single target. (SYSTEM.SYSTEM_NAME is defined in package SYSTEM which is a part of package STANDARD. See Appendix 10.1.3)

NOTES:

1. Resets LIB's current program library to the one just created.
2. Displays a diagnostic if a program library by this name already exists.
3. Displays a diagnostic if the target_name is unrecognized.
4. Displays a diagnostic if the target_name does not name a single target.
5. Displays a diagnostic if the user has neither write nor append access to the directory.

EXAMPLES:

```
MKLIB (new_prog_lib, target=>VAX780_VMS)
-- Creates a new program library "new_prog_lib" in the user's
-- current working directory.  All the Containers in this program
-- library will be targeted for the VAX780_VMS.  It will
-- automatically acquire STANDARD.PACKAGE and the runtime support
-- library for the VAX780_VMS target.

MKLIB (my_dir.new_prog_lib, target=>ROLM1602B)
-- Creates a new program library "new_prog_lib" in the directory
```

```
-- "my_dir".  All the Containers in this program library will be
-- targeted for the ROLM1602B.  It will automatically acquire
-- STANDARD.PACKAGE and the runtime support library for the
-- ROLM1602B target.

MKLIB (new_pl)
-- Creates a new program library "new_pl" in the user's current
-- working directory.  All of the Containers in this Program
-- library will be targeted for the host.  It will automatically
-- acquire STANDARD.PACKAGE and the runtime support library for
-- the host.
```

ALS SUBCOMMAND DESCRIPTION LIB.UNARCHIVE

SUBCOMMAND: UNARCHIVE - unarchives Containers

FUNCTION: Send a list of Containers to be unarchived to a protected system file which will subsequently be used by the ALS.

FORMAT: UNARCHIVE ([ADANAME => list_1] [.FILE=> list_2] [,OPT=>option_list])

PARAMETER DESCRIPTION:

list_1: List of Adanames of Containers.

list_2: List of pathnames of ALS text files which contain Adanames of Containers, one per line.

At least one ADANAME or FILE parameter must be present.

option_list:

LIST Default: NO_LIST. LIST writes to standard output every Adaname appended to the system file.

NOTES:

1. Displays a diagnostic if a current program library has not been specified.
2. Displays a diagnostic if the Adaname does not include a revision number.
3. Displays a diagnostic if the Container has not been archived.
4. Displays a diagnostic if any node named in list_2 is not a text file.

EXAMPLES:

```
UNARCHIVE (ADANAME=>my_prog.body(3))
-- The name of the Container "my_prog.body(3)
-- is sent to the ALS for subsequent unarchiving.

UNARCHIVE (FILE=>UNARCHIVE)
-- The names of the Containers listed in the text file
-- "unarchive" in the user's directory are sent to the
-- ALS for subsequent unarchiving.
```

ALS COMMAND DESCRIPTION LNKMCF

NAME: LNKMCF - ALS MCF linker

FUNCTION: Bind multiple Containers representing machine text into a single Container by partial or full linking

FORMAT: LNKMCF (main_name,prog_lib,output_name[,UNITLIST=>file_name] [,OPT=>option_list])

PARAMETER DESCRIPTION:

main_name	Either the name of the main subprogram or the keyword NULL indicating that there is no main subprogram. If a subprogram name is given, the linker will normally link that subprogram and any other units in the same program library that are directly or indirectly referenced in one of its WITH or SEPARATE clauses. This can be altered through the use of the NO_SEARCH option.
prog_lib	Name of the program library where all of the input Containers will come from and where the new linked output Container will be placed.
output_name	Name of the new linked Container. This name must follow the Ada naming rules for compilation units.
file_name	Name of a file containing a list of Containers to be linked in addition to the main subprogram. If main_name is NULL, file_name is required.
option_list	
SYMBOLS	Provide a symbol definition listing, if a Container is produced. Default: NO_SYMBOLS
LOCAL_SYMBOLS	If a symbol definition listing is produced, include names local to library package bodies. Default: NO_LOCAL_SYMBOLS, means to include only names which are externally visible.
UNITS	Provide a units listing. Default: UNITS.
SEARCH	Linker will automatically follow the WITH and SEPARATE clauses to find all of the units in the program library that are referenced from the designated starting point units. All of the referenced units will be linked

into the output Container. If NO_SEARCH is specified only the starting point Containers and the runtime routines referenced by the Containers are included in the linked output. The SEARCH procedure will take the first occurrence of a unit. Default: SEARCH

DISPOSITION:

IN	Not used
OUT	Listing except for the summary and diagnostics
MSG	Summary listing and diagnostics
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the program library does not exist
2. FAILS if any of the containers to be linked do not exist

EXAMPLES:

```
LNKMCF  (main_prog ,my_lib, new_prog, OPT=> NO_UNITS)

-- links main_prog and any other units in my_lib that it
-- references into the new Container - new_prog

-- additionally - no units listing is produced

LNKMCF  (NULL, my_lib, partial_prog, UNITLIST=>units_file)

-- links those units named in units_file and any other units
-- in my_lib that are referenced by any of those units
-- the result goes into the Container - partial_prog

LNKMCF (main_prog, my_lib, new_prog, UNITLIST=>units_file,
            OPT=> NO_SEARCH)

-- links main_prog and those units named in unit_file exactly
-- the result is placed into Container new_prog
```

ALS COMMAND DESCRIPTION LNKVAX

NAME: LNKVAX - ALS VAX 11/780 linker

FUNCTION: Bind multiple Containers representing machine text into a single Container by partial or full linking

FORMAT: LNKVAX (main_name,prog_lib,output_name[,UNITLIST=>file_name] [,OPT=>option_list])

PARAMETER DESCRIPTION:

main_name — Either the name of the main subprogram or the keyword NULL indicating that there is no main subprogram. If a subprogram name is given, the linker will normally link that subprogram and any other units in the same program library that are directly or indirectly referenced in one of its WITH or SEPARATE clauses. This can be altered through the use of the NO_SEARCH option.

prog_lib — Name of the program library where all of the input Containers will come from and where the new linked output Container will be placed.

output_name — Name of the new linked Container. This name must follow the Ada naming rules for compilation units.

file_name — Name of a file containing a list of Containers to be linked in addition to the main subprogram. If main_name is NULL, file_name is required.

option_list

SYMBOLS — Provide a symbol definition listing, if a Container is produced. Default: NO_SYMBOLS

LOCAL_SYMBOLS — If a symbol definition listing is produced, include names local to library package bodies. Default: NO_LOCAL_SYMBOLS, means to include only names which are externally visible.

UNITS — Provide a units listing. Default: UNITS.

SEARCH — Linker will automatically follow the WITH and SEPARATE clauses to find all of the units in the program library that are referenced from the designated starting point units. All of the referenced units will be linked

into the output Container. If NO_SEARCH is specified only the starting point Containers and the runtime routines referenced by the Containers are included in the linked output. The SEARCH procedure will take the first occurrence of a unit. Default: SEARCH

DISPOSITION:

IN	Not used
OUT	Listing except for the summary and diagnostics
MSG	Summary listing and diagnostics
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if the program library does not exist.
2. FAILS if any of the Containers to be linked do not exist.
3. FAILS if any of the Containers to be linked is marked as unusable or recompilation needed.
4. FAILS if the designated main subprogram does not exist or is not legal.
5. FAILS if the designated main subprogram is "NULL" and no UNITLIST parameter was provided.
6. FAILS if a designated UNITLIST file does not exist.
7. FAILS if the given output Container name is the same as an existing library unit in the designated program library.

EXAMPLES:

```
LNKVAX  (main_prog ,my_lib, new_prog, OPT=> NO_UNITS)

-- links main_prog and any other units in my_lib that it
-- references into the new Container - new_prog

-- additionally - no units listing is produced

LNKVAX  ("NULL", my_lib, partial_prog, UNITLIST=>units_file)

-- links those units named in units_file and any other units
```

```
-- in my_lib that are referenced by any of those units
-- the result goes into the Container - partial_prog

LNKVAX (main_prog, my_lib, new_prog, UNITLIST=>units_file,
            OPT=> NO_SEARCH)

-- links main_prog and those units named in unit_file exactly
-- the result is placed into Container new_prog
```

ALS COMMAND DESCRIPTION LST

NAME: LST - List contents or offspring of a node

FUNCTION: List data portion of a file or list names of offspring of a directory or variation header on the standard output

FORMAT: LST ([node_name][,ATTR=>attr_name][,ASSOC=>assoc_name][,OPT=>option_list])

PARAMETER DESCRIPTION:

node_name	Name of node to be listed. If omitted, the current working directory is listed.
attr_name	Name of one attribute whose value is to be listed.
assoc_name	Name of one association whose references are to be listed.
option_list	
TREE	Causes the names of all the nodes in the subtree to be displayed in indented form. (The root of the subtree is the node given by node_name.) The data portions of files in the subtree are not displayed. NO_TREE means to list the names of only the immediate offspring. Default: NO_TREE

DISPOSITIONA

IN	Not used
OUT	Output listing
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If node_name specifies a file, the data portion is listed. If node_name specifies a directory or variation header, the names of offspring are listed. The revision number is also listed for those offspring that are files.

2. FAILS if node_name does not exist or access rights forbid read access.

3. If the TREE option is specified for listing of a file, a WARNING will be issued. In other respects, the operation will continue normally.

EXAMPLES:

```
LST (my_directory)
-- Lists offspring of directory "my_directory".

LST (my_file,ATTR=>read)
-- Lists the contents of the file "my_file" and the "read" attribute.

LST
-- Displays contents of the current working directory.

LST ATTR=>purpose
-- Displays contents of the current working directory with
-- the values of the purpose attribute.
```

ALS COMMAND DESCRIPTION LSTASS

NAME: LSTASS - List association contents

FUNCTION: Lists the contents of an association on the standard output

FORMAT: LSTASS (node [,assoc_name])

PARAMETER DESCRIPTION:

node	Node for which the association is to be listed
assoc_name	Association to be listed

DISPOSITION:

IN	Not used
OUT	Output list of path names (one per line)
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If assoc_name is omitted, the names of all associations possessed by the node are listed. Otherwise, the contents of the specified association are listed.
2. FAILS if the node does not exist or access is not granted.
3. The references in the association are printed exactly as stored. Relative pathnames must be interpreted as relative to the node possessing the association, not as relative to the CWD.

EXAMPLE:

```
-- continuing the example from ADDREF, with the CWD being

      .landsat.analysis:

LSTASS (signal_analyzer.doc,cross_ref)

-- will list at least

      .landsat.analysis.fft.doc
      .landsat.analysis.spectrum.doc
```

ALS COMMAND DESCRIPTION LSTATTR

NAME: LSTATTR - List attribute value

FUNCTION: Lists the value of an attribute on the standard output

FORMAT: LSTATTR (node [,attr_name])

PARAMETER DESCRIPTION:

node	Node for which attribute value is to be listed
attr_name	Attribute to be listed

DISPOSITION:

IN	Not used
OUT	Output list (a single line)
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If attr_name is omitted, then the names of all attributes of the node are listed. Otherwise, the value of the specified attribute is listed.

2. FAILS if the node does not exist or access is not granted.

EXAMPLE:

```
LSTATTR (my_node)

-- lists names of all attributes of my_node
```

ALS COMMAND DESCRIPTION MKDIR

NAME: MKDIR - Make a directory

FUNCTION: Makes an empty directory

FORMAT: MKDIR (name {,attr_name=>value})

PARAMETER DESCRIPTION:

name	Name of directory to be created
attr_name	Name of attribute to give to directory
value	Value for attribute

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if a node with this name already exists.
2. FAILS if the user does not have either append or write access to the parent node.
3. If the attr_name is KAPSE-controlled, an error diagnostic will be generated. The node will be created.

EXAMPLE:

```
MKDIR (my_dir)
-- creates the empty directory named my_dir
```

ALS COMMAND DESCRIPTION MKFILE

NAME: MKFILE - Make a file

FUNCTION: Makes a file with empty data portion

FORMAT: MKFILE (name {,attr_name=>value})

PARAMETER DESCRIPTION:

name	Name of file to be created
attr_name	Name of attribute to give to file
value	Value for attribute

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. If no category attribute is given, defaults to "text".
2. FAILS if a node by this name already exists.
3. FAILS if user has neither append nor write access to parent node.
4. If the specified attr_name is KAPSE-controlled, an error diagnostic will be generated. The node will be created.
5. FAILS if a revison nubmer is specified in the file name.

EXAMPLE:

```
MKFILE  (my_dir.new_file,category=>source,purpose=>"simple example
file")

-- make a file called "new_file" in "my_dir",
-- give it a category attribute of "source", and
-- a purpose attribute of "simple example file"
```

ALS COMMAND DESCRIPTION MKVAR

NAME: MKVAR - Make a variation set

FUNCTION: Makes an empty variation set (header with no offspring)

FORMAT: MKVAR (name {,attr_name=>value})

PARAMETER DESCRIPTION:

name	Name of the new variation set.
attr_name	Name of attribute to give to variation set
value	Value for attribute

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and message output
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if a node by this name already exists.
2. FAILS if the user does not have either append or write access to the parent node.
3. If the specified attr_name is KAPSE-controlled, an error diagnostic will be generated. The node will be created.

EXAMPLE:

```
MKVAR (my_node)
-- creates my_node as an empty variation set
```

ALS COMMAND DESCRIPTION PRINT

NAME: PRINT - print file contents

FUNCTION: Print data parts of text files on the system line printer

FORMAT: PRINT ([file_name {,file_name}] {,FILES => file_list})

PARAMETER DESCRIPTION:

file_name	A text file containing text to be printed
file_list	A text file containing names of files to be printed, one file name per line.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if access rights do not grant read permission.
2. FAILS if at least one file_name or one file_list is not specified.

EXAMPLE(S):

```
PRINT (this_file, that_file, another_file)

-- print this_file, that_file, and another_file.
```

ALS COMMAND DESCRIPTION PROFILEVMS

NAME: PROFILEVMS - display statistical and frequency data

FUNCTION: Display the recorded statistical and frequency analyzer data.

FORMAT: PROFILEVMS (data_list,[,NAME=>name_list]
[,FILE=>file_list][,OPT=>option_list])

PARAMETER DESCRIPTION:

data_list — A file or an aggregate of files where each file contains either the recorded Statistical and Frequency Analyzer data to be displayed, or a list of files, one file name per line.

name_list — List of the subprogram names for which Statistical and Frequency Analyzer data should be displayed. If both name_list and file_list are empty, data are displayed for all subprograms.

file_list — List of files each of which contain subprogram names (one name per line) for which Statistical and Frequency Analyzer data should be displayed. If both name_list and file_list are empty, then data for all subprograms are displayed.

DISPOSITION:

IN — Not used

OUT — Profile listings

MSG — Diagnostic messages

RSTRING — Not used

RSTATUS — See Appendix 80

NOTES:

1. In order to create statistical data, an Ada program must be exported with the STAT option (see Section 40.1.1), and executed with the statistical_data_file name specified (see Section 40.1.2).

2. In order to create frequency data, an Ada program must be compiled with the FREQUENCY option (see Section 3.7.1.1.1.3), exported with the FREQUENCY option (see Section 40.1.1), and executed with the frequency_data_file_name specified (see Section 40.1.2).

3. Data are displayed on a subprogram basis, i.e. all counts within each subprogram are summed.

EXAMPLE(S):

```
PROFILEVMS (data, NAME=>my_prog, OPT=>BLOCKS)

-- displays a profile analysis of the
-- data in data for the programs in my_prog
```

ALS COMMAND DESCRIPTION QHELP

NAME: QHELP - Quick Help

FUNCTION: Supplies information from the HELP database.

FORMAT: QHELP (subject)

PARAMETER DESCRIPTION:

subject	subject for which information should be supplied.

DISPOSITION:

IN	Not used
OUT	Explanatory text
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. QHELP extract a single piece of information from the HELP database. It is intended for quick reference. The HELP tool also supplies information from the HELP database. HELP allows the user to interactively explore the database and is intended to provide more extensive information.

2. The subject parameter can be either a main subject or a subtopic of a subject or another subtopic. The subject name is simply the pathname of the subject or subtopic relative to the HELP database root. for example, the subject names LIB and LIB.ACQUIRE would reference the main subject LIB and its subtopic ACQUIRE respectively.

3. If the subject is a tool, the command format of the tool is provided. Information about the subject is always supplied.

4. FAILS if the subject does not exist.

EXAMPLES:

```
QHELP (LIB)
-- Prints information about LIB to standard output.

QHELP (LIB.ACQUIRE)
-- Prints information about the subtopic LIB.ACQUIRE.
```

ALS COMMAND DESCRIPTION RECEIVE

NAME: RECEIVE - Receive external subtree from tape

FUNCTION: Read a new subtree from tape as written by the TRANSMIT tool.

FORMAT: RECEIVE (volume_name,path_name[,NAME=>search_name] [,OPT=>option_list])

PARAMETER DESCRIPTION:

volume_name: An identifier which must match the tape volume label before reading can take place. It has the form of an Ada identifier limited in length to six characters, and containing only digits and upper-case alphabetic characters.

path_name: If this parameter names a directory or a variation header, the node must not exist in the database. It is created by the tool and is the root of the new subtree. If this parameter names a file without an explicit revision number, then the latest revision on the tape will be written into the ALS database, if that revision does not already exist in the database. If this parameter names a file with an explicit revision number, that single revision will be written into the ALS database if it exists on the tape and does not exist in the database. If this parameter names a file with a wildcard revision, then all revisions which exist on the tape and do not exist in the database will be written into the database.

search_name: This is the external identifier assigned to the subtree by the NAME parameter of the TRANSMIT tool. If the NAME parameter is present in the RECEIVE parameter list, the tape is searched for a subtree with the specified external name. If the NAME parameter is absent, the subtree to be read is the first one on the tape.

option_list:

LIST Default: NO_LIST. LIST writes to standard output the name of every node read from tape.

WARNINGS Default: WARNINGS. WARNINGS produces a diagnostic message for each received node whose availability attribute does not have the value on_line. NO_WARNINGS suppresses these diagnostic messages.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if path_name exists.
2. FAILS if NAME parameter is present and specified search_name is not found on the tape.
3. FAILS if user does not have write or append access to directory containing path_name.
4. FAILS if volume_name does not match the tape volume label.
5. If a revision already exists in the ALS database, it will not be overwritten from the tape.

EXAMPLES:

```
RECEIVE  (MYTAPE,interpreter,NAME=>interp_781205)

-- Searches tape volume named MYTAPE for a subtree
-- with the external name interp_781205.
-- Creates the node INTERPRETER in the user's CWD,
-- which becomes the root of the subtree being received.
```

ALS COMMAND DESCRIPTION RENAME

NAME: RENAME - Rename a node

FUNCTION: Changes the name of a node, possibly moving it to a new directory

FORMAT: RENAME (old_name,new_name)

PARAMETER DESCRIPTION:

old_name	Pathame of the node to be RENAMEd
new_name	New name of the node

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if new_name already exists, and either the old_name or the new_name do not refer to files.
2. FAILS if the old_name does not exist
3. FAILS if user lacks WRITE access to source directory.
4. FAILS if user lacks WRITE or APPEND access to destination directory.
5. FAILS if new_name includes a revison number.
6. FAILS if old_name includes the "*" index, and new_name already exists.
7. FAILS if oldname is a single revision and its revision set is shared.

EXAMPLE:

```
RENAME (my_node, my_new_node)
```

```
-- changes the name of the my_node to my_new_node

RENAME  (my_node(2),my_new_node)

-- changes the name of the second revision of my_node to be
-- the latest revision of my_new_node.  A new revision of
-- my_new_node is created if either the revision set did not
-- previously exist, or the latest revision is frozen, or
-- has a non-zero derivation count.

RENAME  (my_node(*),my_new_node)

-- Moves the entire revision set my_node to the
-- new revision set my_new_node.
```

ALS COMMAND DESCRIPTION REVISE

NAME: REVISE - Create a new revision of a file

FUNCTION: Create a new revision of a file with the data portion of the new revision being initialized to a copy of the data portion of the specified revision.

FORMAT: REVISE (file_name)

PARAMETER DESCRIPTION:

file_name	Name of the file to be revised. If no revision is specified, the data from the latest revision is copied to make the new revision. Otherwise, the specified revision is used.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if name does not refer to an ALS file.
2. FAILS if user does not have write access to the file.
3. FAILS if the specified revision does not exist.
4. This command causes the previous revision to be frozen.

EXAMPLES:

```
REVISE (proj_dir.my_prog)

-- given that there are five existing revisions of my_prog, this
-- command adds a 6th revision of the file, proj_dir.my_prog,
-- having the same data portion as the 5th revision.

REVISE  (proj_dir.my_prog(1))

-- given that there are five existing revisions of my_prog, this
-- command creates Revision 6 with the data from Revision 1.
```

ALS COMMAND DESCRIPTION RUNOFF

NAME: RUNOFF - DEC* Standard Runoff Text Formatter

FUNCTION: Documentation aid that does justification, page formatting and other text manipulations.

FORMAT: RUNOFF (in_file,out_file [,QUAL => qual_string]
[,CONTENTS=>binary_toc]
[,INDEX=>binary_index])

PARAMETER DESCRIPTION:

in_file	Name of the input file.
out_file	Name of the output file containing the reformatted text. A file will be created if it does not exist.
qual_string	String of command qualifiers (switches) to append to the VMS RUNOFF invocation.
binary_toc	Name of the VAX/VMS file into which the binary table of contents is to be written. This construct is equivalent to the /CONTENTS:binary_toc qualifier. The <<VMS>>... notation must be used.
binary_index	Name of the VAX/VMS file into which the binary index is to be written. This construct is equivalent to the /INDEX:binary_index qualifier. The <<VMS>>... notation must be used.

DISPOSITION:

IN	Not used.
OUT	Not used.
MSG	Confirmation and diagnostics not generated directly by RUNOFF.
RSTRING	Not used.
RSTATUS	See Appendix 80

*Digital Equipment Corporation, Maynard, Mass.

NOTES:

1. FAILS if in_file does not exist, cannot be read, or is not a file.

2. If any output file does not exist, it is created. To do this, the parent directory must exist and be writable.

3. FAILS if any output exists and cannot be written.

4. File specifications appearing in the qual_string are VMS file specifications, not ALS node names. The use of <<VMS>> is not appropriate in this context.

5. The input to RUNOFF is described in the DEC Standard Runoff (DSR) User's Guide (2.2). Input to the ALS RUNOFF is the same, except for the .REQUIRES command which will be replaced by a set of lines as follows:

```
.!SYMBOL <user_defined_name> <pathname>
.require "<user_defined_name>:"
```

   where <user_defined_name> must be an alphanumeric string of up to sixty-three, case-insensitive characters which is unique within the entire text being runoff. The <pathname> is an ALS pathname referring to a text file.

6. FAILS if an ALS file referenced indirectly in a RUNOFF .REQUIRES command using the conventions specified in 5 cannot be found or cannot be read.

7. The table of contents and indexing functions of RUNOFF are supported in conjunction with the ALS TOC and TCX commands which correspond to the VAX/VMS utilities of the same names. The CONTENTS and INDEX optional parameters should be used instead of the /CONTENTS and /INDEX qualifiers.

8. FAILS if the binary_TOC file specified is NOT a VAX/VMS file specification.

9. FAILS if the binary_index file specified is NOT a VAX/VMS file specification.

EXAMPLE(S):

```
RUNOFF (my_file, new_file)
-- Invokes DEC* Standard Runoff Text Formatter to take the file
-- my_file and produce the formatted file new_file.

RUNOFF (my_file, new_file, CONTENTS=>"<<VMS>>tocfile.toc")
-- Invokes DEC* Standard Runoff Text Formatter to take the file
-- my_file and produce the formatted file new_file.
```

```
-- In addition, the VAX/VMS binary table of contents file
-- tocfile.toc is generated.

RUNOFF (my_file, <<VMS>>newfile.mem, QUAL=>"/right:20")
-- Invokes DEC* Standard Runoff Text Formatter to take the file
-- my_file and produce the VAX/VMS file newfile.mem.  In
-- addition, the qualifier /right:20 is to be used in the
-- invocation.
```

ALS COMMAND DESCRIPTION SHARE

NAME: SHARE - share a node

FUNCTION: Adds an existing node as an offspring of some other node, thereby sharing it (see Appendix 50 for a full description of node sharing)

FORMAT: SHARE (old_name,new_name)

PARAMETER DESCRIPTION:

old_name	Existing name of the node to be SHAREd
new_name	New name of the node to be SHAREd

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if old_name does not exist.
2. FAILS if name given by new_name already exists
3. FAILS if user lacks READ access for the directory containing the file named by <old_name>.
4. FAILS if user lacks WRITE or APPEND access for the directory containing the file named by <new_name>.
5. FAILS if oldname or new name specify a revision index other than "*". Since files are always shared as entire revision sets, the "*" is optional.

EXAMPLE:

```
SHARE  (engine_control.revolution_sensor.source, rev_src)
```

— creates an alias in CWD for a long file name

ALS COMMAND DESCRIPTION SHOW SUBS

NAME: SHOW_SUBS - Show currently defined substitutors and values

FUNCTION: Display the currently defined substitutors and their values on the standard output

FORMAT: SHOW_SUBS ([OPT=>option_list])

PARAMETER DESCRIPTION:

option_list

LOCALS - Display the local substitutors.

GLOBALS - Display the global substitutors.

The default will show both local and global substitutors.

DISPOSITION:

IN	Not used
OUT	Table of names and values
MSG	Confirmation and error messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:
Substitutors are typed in alphabetical order.

EXAMPLE:

```
SHOW_SUBS                -- typed by user

ARGS      0              -- typed by SHOW_SUBS
CONFIRM   off
CSTATUS   ok
DOC       manual         -- a user-defined substitutor
```

ALS COMMAND DESCRIPTION STUBGEN

NAME: STUBGEN - Stub Generator

FUNCTION: Generate Ada source code for a library package body or subprogram body, given a unit which is its declaration; or for a subunit package or subprogram body, given a unit which contains its body stub.

FORMAT: STUBGEN (name, prog_lib, output_name [, unit_name]
[, OPT=>option_list])

PARAMETER DESCRIPTION:

name	Name of the compilation unit which is the declaration, or which contains its body stub.
prog_lib	Name of the program library where the Container for the compilation unit is stored.
output_name	Pathname of the output text file.
unit_name	The name of the unit to be stubbed. This argument is supplied if the unit to be stubbed is a subunit.
option_list	
PRINT	The generated stub contains calls to Text_IO to print out the name of the subprogram. Default: PRINT.

DISPOSITION:

IN	Not used
OUT	Not used
MSG	Confirmation and diagnostics messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. Fails if program library does not exist.
2. Fails if name is not found in program library.

3. Fails if declaration or specification of unit_name is not found in the Container.

4. Fails if access controls prohibit output of text file.

5. Fails if any subprogram being stubbed is a function which returns a task or private value, or a record or array containing such values, a return statement is made without a return value. If a function returns a record type with discriminants, an initial value is given to the discriminants.

6. The stub contains initialization of all output parameters. In the case of task, limited, or private types, or record or array components of task, private, or limited types, an assignment is made without a value. If a parameter is a record type with discriminants, no assignment is made.

7. FAILS if the unit to be stubbed contains errors.

ALS COMMAND DESCRIPTION TCX

NAME: TCX - Index Generator

FUNCTION: Index generator to be used in conjunction with RUNOFF. This is equivalent to the VAX/VMS TCX utility.

FORMAT: TCX (in_file,out_file)

PARAMETER DESCRIPTION:

in_file	Name of the input file. This should be a binary index file produced by RUNOFF and stored in VMS. The <<VMS>> notation must be used.
out_file	Name of the output file. A file will be created if it does not exist. This file is equivalent to the .RNX file. It can be input to RUNOFF, or inserted into other RUNOFF input by use of .REQUIRES.

DISPOSITION:

IN	Not used.
OUT	Not used.
MSG	Confirmation and diagnostics not generated directly by TCX.
RSTRING	Not used.
RSTATUS	See Appendix 80

NOTES:

1. FAILS if in_file does not exist, cannot be read, is not a file, or is not a VAX/VMS file specification.
2. If out_file does not exist, it is created. To do this, the parent directory must exist and be writable.
3. FAILS if out_file exists and cannot be written.

ALS COMMAND DESCRIPTION TIME

NAME: TIME - Display the TIME

FUNCTION: Displays the current time in the format hh:mm:ss.fff with zeros not suppressed, where:
hh is the hour number using a 24-hour clock
mm is the minute reading
ss is the second reading
fff is the fractional seconds

FORMAT: TIME (OPT=>option_list)

PARAMETER DESCRIPTION:

option_list

DISPLAY	Causes the time to be displayed on the standard output; NO_DISPLAY causes the time to be returned via RSTRING; Default: DISPLAY

DISPOSITION:

IN	Not used
OUT	Time string unless NO_DISPLAY is specified
MSG	Confirmation
RSTRING	Time if NO_DISPLAY is specified
RSTATUS	See Appendix 80

NOTES:

1. The TIME is obtained from the host operating system. There is no way to guarantee that the clock is correct.

2. Note that the precision of the fractional portion of the current time is host dependent.

EXAMPLE(S):

```
TIME                        -- typed by user
14:27:33.457                -- typed by the TIME tool

TIME (OPT=>no_display)      -- typed by user
echo "#RSTRING"             -- typed by user
14:27:56.890                -- typed by the echo tool
```

ALS COMMAND DESCRIPTION TOC

NAME: TOC - Table of Contents Generator

FUNCTION: Table of contents generator to be used in conjunction with RUNOFF. This is equivalent to the VAX/VMS TOC utility.

FORMAT: TOC (in_file,out_file)

PARAMETER DESCRIPTION:

in_file	Name of the input file. This should be a binary table of contents file produced by RUNOFF and stored in VMS. The <<VMS>> notation must be used.
out_file	Name of the output file. A file will be created if it does not exist. This file is equivalent to the .RNT file. It can be input to RUNOFF, or inserted into other RUNOFF input by use of .REQUIRES.

DISPOSITION:

IN	Not used.
OUT	Not used.
MSG	Confirmation and diagnostics not generated directly by TOC.
RSTRING	Not used.
RSTATUS	See Appendix 80

NOTES:

1. FAILS if in_file does not exist, cannot be read, is not a file, or is not a VAX/VMS file specification.
2. If out_file does not exist, it is created. To do this, the parent directory must exist and be writable.
3. FAILS if out_file exists and cannot be written.

ALS COMMAND DESCRIPTION TRANSMIT

NAME: TRANSMIT - Write subtree to tape

FUNCTION: Write to tape the indicated subtree in a format receivable by the RECEIVE tool on an ALS host.

FORMAT: TRANSMIT (volume_name,path_name[,NAME=>search_name] [,OPT=>option_list])

PARAMETER DESCRIPTION:

volume_name — An identifier which must match the tape volume label before writing can take place. It has the form of an Ada identifier limited in length to six characters, and containing only digits and upper-case alphabetic characters.

path_name — The root node of the subtree to be written. Both owned and shared offspring nodes are written. Shared nodes within the subtree are written only once. If the availability attribute of any transmitted node does not have the value on_line, the node is written as it is on disk. If this parameter names a file without an explicit revision number, only the latest revision will be transmitted. If an explicit revision number is given, only that revision will be transmitted. If a wildcard revision is given, all existing revisions will be transmitted.

search_name — The external identifier which will be given to the subtree on tape if this parameter is present. It is employed by the NAME parameter of the RECEIVE tool and bears no necessary relationship to any name in the existing ALS database.

option_list

APPEND — Default: APPEND. NO_APPEND rewinds the tape before writing. APPEND appends information to end of tape.

LIST — Default: NO_LIST. LIST writes to standard output the name of every node transmitted.

WARNINGS — Default: WARNINGS. WARNINGS produces a diagnostic message for each transmitted node whose availability attribute does not have the value on_line. NO_WARNINGS suppresses these

diagnostic messages.

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. FAILS if subtree does not exist.
2. FAILS if the user does not have read access to any node of the subtree.
3. FAILS if volume_name does not match tape volume label.

EXAMPLE:

```
TRANSMIT  (TWB,synthesizer,NAME=>prog_8012)

-- Prompts the user to mount tape TWB and positions to the end of
-- recorded information.  Writes ALS subtree named synthesizer under
-- the user's CWD, giving it the external identifier proj_8012.
```

DISPOSITION:

IN	Not used
OUT	Optional LIST output
MSG	Confirmation and diagnostic messages.
RSTRING	Not used
RSTATUS	See Appendix 80

NOTES:

1. A diagnostic message is produced for any pathname of list_1 or list_2 or for any node named in list_2 which is not a file to which the user has read access.

2. A diagnostic message is produced for each node specified for rollin which is in a directory to which the user does not have read access.

EXAMPLE:

```
UNARCHIVE  (FILE=>pre_config_c)

-- File pre_config_c is a file of pathnames, one per line,
-- of rolled-out file revisions.  These names are appended
-- to the to-be-rolled-in file.
```

NAME: UNARCHIVE - Unarchive a set of file revisions

FUNCTION: Send a list of names of nodes to be rolled in from archive tape(s) to a protected file which will subsequently be used as input to a rollin operation.

FORMAT: UNARCHIVE ([NODE=>list_1][,FILE=>list_2][,OPT=>option_list])

PARAMETER DESCRIPTION:

list_1

list_2: The syntax of both list_1 and list_2 is a list of pathnames. At least one NODE or FILE parameter must be present. Execution of the UNARCHIVE tool is a request by the user to the system operator(s) that the node(s) specified in the parameter list be rolled in. Each node specified for rollin whether by the NODE or FILE form, is a specific revision of a frozen file node which has been rolled out. The NODE form specifies directly, in the parameter, one or more nodes for rollin. The FILE form names one or more ALS text files each of which contains a list of pathnames for rollin, one per line.

The set of all nodes to be rolled in is appended to a system file of to-be-rolled-in pathnames which is intended to be subsequently referenced by an operator in a rollin operation. In addition to transmitting the list, the UNARCHIVE tool checks that each node specified for rollin is a file node whose availability attribute has the value off_line. Pathnames found to be invalid by these checks are not sent to the to-be-rolled-in file; instead, appropriate diagnostic messages are written to the message file.

option_list

LIST Default: NO_LIST. LIST writes to standard output every pathname appended to the to-be-rolled-in file.