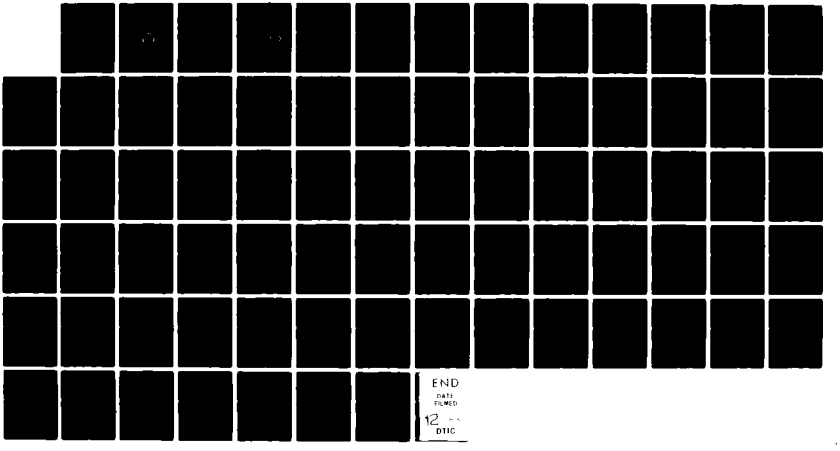
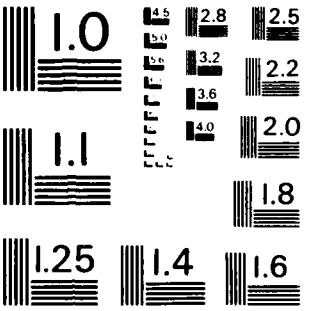
AD-A134 825 DRAFT SPECIFICATION OF THE COMMON APSE (ADA PROGRAMMING 1/1
SUPPORT ENVIRONMENTS) INTERFACE SET (CAIS) VERSION 11
(U) ADA JOINT PROGRAM OFFICE ARLINGTON VA 30 SEP 83
UNCLASSIFIED F/G 9/2 NL





NOTE: This draft, dated 30 September 1983, prepared for the Ada Joint Program Office, has not been approved and is subject to modification. DO NOT USE PRIOR TO APPROVAL.

Draft Specification

of the

Common APSE Interface Set (CAIS)

Version 1.1

30 September, 1983

Prepared by

KIT/KITIA
CAIS Working Group

for the
Ada® Joint Program Office

(® Ada is a Registered Trademark of the Department of Defense, Ada Joint Program Office)

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. A134823 — 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Draft Specification of the Common APSE Interface Set (CAIS) Version 1.1	5. TYPE OF REPORT & PERIOD COVERED Sep. 1982 - Sep. 1983 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) KIT/KITTIA CAIS Working Group for the Ada Joint Program Office KAPSE Interface Team/KAPSE Interface Team from Industry and Academia	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Deputy Undersecretary of Defense Research and Advanced Technology Washington, DC 20301	12. REPORT DATE 30 September 1983 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unclassified

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

CAIS, APSEs, KIT, KITTIA, Ada Programming Language, interface requirements, transportability, CAIS Node Model, Ada toolsets, environments, Ada packages, Structural Nodes, File Nodes, Process Nodes, Device Nodes, utilities, Ada Language Reference Manual.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number).

This version of the CAIS establishes interface requirements for the transportability of Ada toolsets software to be utilized in Department of Defense (DoD) Ada Programming Support Environments (APSEs) known as the Ada integrated Environment (AIE) and the Ada Language System (ALS). Strict adherence to this interface set will ensure that the Ada toolsets will possess the highest degree of portability across APSEs.

The scope of the CAIS includes interfaces to those services traditionally provided by an operating system that affects tool transportability. Ideally, all APSE tools

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

would be implementable using only the Ada language and the CAIS. This version of the CAIS is intended to provide most interfaces required by common tools. This version of the CAIS includes six interface areas: Node Model, Structural Nodes, File Nodes, Process Nodes, Device Nodes and Utilities.

Draft Report
DTIC should process report per Mr. Thomas, Ada Joint Program Office. Final version will not be published for six month or a year.

FORWORD

This document is a draft for public review. It will be revised in accordance with comments received during this public review cycle.

This document has been prepared in response to the Memorandum of Agreement signed by the Undersecretary of Defense and the Assistant Secretaries of the Air Force, Army, and Navy. The memorandum established agreement for defining a set of common interfaces for the Department of Defense (DoD) Ada Programming Support Environments (APSEs) to promote Ada tool transportability and interoperability. The initial phase of this effort is directed toward the interfaces of the Ada Integrated Environment (AIE) and the Ada Language System (ALS). This version derives a set of specific interfaces from these two APSEs, but the CAIS is intended to be implementable as part of a wide variety of intended APSEs. It is anticipated that the CAIS will evolve, changing to meet new needs. Ultimately it is the intention of the DoD to submit CAIS for standardization. Through the acceptance of such a standard it is anticipated that the source level compatability of Ada software tools will be enhanced for both the DoD and non-DoD users.

The authors of this document include technical representatives of the two DoD APSE contractors, representatives from the DoD's Kernel Ada Programming Support Environment (KAPSE) Interface Team (KIT), and volunteer representatives from the KAPSE Interface Team, Industry and Academia (KITIA).

The initial effort for definition of the CAIS was begun in September 1982 by the following members of the KAPSE Interface Team (KIT); J. Foidl (TRW), J. Kramer (Ada Joint Program Office), T. Oberndorf (Naval Ocean Systems Center), T. Taft (Intermetrics), R. Thall and W. Wilder (both of SofTech). In February 1983 the design team was expanded by Lcdr. B. Schaar (Ada Joint Program Office) to utilize the professional capabilities and experience of the KIT and KAPSE Interface Team from Industry and Academia (KITIA). These new members include: H. Fischer (Litton Data Systems), T. Harrison (Texas Instruments), E. Lamb (Bell Labs), T. Lyons (Software Sciences Ltd., U.K.), D. McGonagle (General Electric), H. Morse (Frey Federal Systems), E. Ploedereder (I.A.B.G., West Germany), H. Willman (Raytheon), and L. Yelowitz (Ford Aerospace). The Ada Joint Program Office is particularly grateful to those KITIA members and their companies for providing the time and resources that significantly contributed to this document. Additional constructive criticism and direction was provided by G. Myers (Naval Ocean Systems Center) and the general memberships of the KIT and KITIA.

This document was typeset by McMahon Engineering Services, San Diego, using a Compugraphics MCS20-8400 typesetting unit, with the Advanced Communication Interface used to transfer data already keystroked from a word processor to the Compugraphics typesetter and then inputting typesetting codes to format the document.

CONTENTS

1. INTRODUCTION

This document provides specifications for a set of Ada packages which together form a Common APSE Interface Set (CAIS) for Ada Programming Support Environments (APSEs). This interface set is designed to promote the source-level portability of Ada programs, particularly Ada software development tools. The initial phase of this effort is directed toward the interfaces of the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Version 1.1 of the CAIS, presented herein, is intended to provide the basis for evolution of the CAIS as APSEs are implemented, as tools are transported, and as tool interoperability issues are encountered.

Tools written in Ada, using only the packages described herein, should be transportable to other CAIS implimentations. However, where tools function as a set, the CAIS facilitates transportability of the set of tools as a whole, but individual tools may not be individually transportable.

1.1 SCOPE OF THE CAIS

This version of the CAIS establishes interface requirements for the transportability of Ada toolsets software to be utilized in Department of Defense (DoD) Ada Programming Support Environments (APSEs) known as the Ada Integrated Environment (AIE) and the Ada Language System (ALS). Strict adherence to this interface set will ensure that the Ada toolsets will possess the highest degree of portability across APSEs.

The scope of the CAIS includes interfaces to those services traditionally provided by an operating system that affect tool transportability. Ideally, all APSE tools would be implementable using only the Ada language and the CAIS. This version of the CAIS is intended to provide most interfaces required by common tools. This version of the CAIS includes six interface areas:

a. **Node Model.** This area presents a node model for the CAIS in which contents, relationships and attributes of nodes are defined. Also included are the foundations for access control and synchronization.

b. **Structural Nodes.** This area covers the creation of structural nodes.

c. **File Nodes.** This area covers file input/output.

d. **Process Nodes.** This area covers creation of processes for program invocation, control of processes, process attribute management, and inter-process communication.

e. **Device Nodes.** This area covers basic device input/output support, along with special device control facilities.

f. **Utilities.** This area covers text and list manipulation.

1.2 EXCLUDED AND DEFERRED TOPICS

During the design of the CAIS many aspects of environments have been considered. It has been determined that several aspects should be explicitly excluded from this version of the CAIS:

Interfaces for non-software development environments (target systems) are not a part of this version.

The acronyms KAPSE and MAPSE are not used in this document because there is disagreement on their meanings.

Multi-lingual environments are not addressed by the CAIS.

A number of interface issues remain unresolved in this version of the CAIS, even though they have been considered. These issues are important for a complete interface specification, but their resolution has been deferred until a later version. Deferred interface issues (in alphabetical order) include:

Access control — Access rights and privileges to system resources.

Asynchronous interfaces — Most interfaces in this document are task synchronous interfaces (i.e., the specified operation is completed before the calling task is allowed to proceed.)

Communications transformation — filtering of data before receipt by processes, mappings (lower case to upper case, break, key to escape sequence), terminator character for input.

Configuration management — configuration control including keeping versions, referencing the latest revision, identifying the state of an object, etc.

Device control — Controls for printers, tape drives, disk drives, graphics, windowing, etc.

Distributed environments — Explicit support for environments in which parts of Ada programs or data bases are distributed across multiple processors.

Interoperability — Inter-tool interfaces for tool sets; calling sequences and data formats used to invoke/interact with common APSE tools, including the compilation/program library system, the text editing systems, the command processor, and the mail system.

Predefined attributes/names — A full set of attributes and names that exist in all APSEs which implement the CAIS.

Predefined exceptions — A full set of exceptions that exist in all APSEs which implement the CAIS; identification of all situations where exceptions are raised by the CAIS.

Resource access and management — Resource control and allocation, such as for processor time, processor memory, and shared data pools.

Security — Mechanisms for handling discretionary and non-discretionary information based on classification of the data and system requirements.

Typed database — Typing of the objects in the database organization.

1.3 CONFORMANCE

Conformance of an implementation to the CAIS is established on a package-by-package basis. Each package must be available as a library unit, with the name specified in this document. From the package user's point of view, the package must have indistinguishable syntax and semantics from those stated herein. The following differences in CAIS package implementation from the specifications in this document are considered indistinguishable from a user's point of view:

a) The package may have additional WITH or USE clauses.

b) Parameter modes listed here as OUT may be IN OUT or those listed as IN OUT may be OUT.

c) Types specified as limited private may be simply limited types.

d) Packages may be instantiations of generic sub-packages of some other (private) library unit package.

Examples of differences which are NOT legal:

a) Additional or missing declarations, as these affect name visibility.

b) Parameter mode IN OUT, as this prevents passing of expressions.

c Limited private types being changed to sub-types or derived types, when this changes the semantics of "deriving" from the type.

d) Packages which are not available as specified library units, because this changes the means of reference to package components.

1.4 DOCUMENT ORGANIZATION

Each of the interface areas described in Section 1.1 is the subject of a subsequent section of this document. A discussion introduces the underlying model for that area. Ada package specifications describe the facilities provided. These are followed by a narrative of the intended semantics of the package. New terms introduced in the narrative sections of the CAIS have been highlighted with **boldface** type. **Boldface** type within the package specifications and package semantics sections indicate reserved words in accordance with the Ada Language Reference Manual.

2. REFERENCES

[LRM]: Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A; United States Department of Defense; January 1983.

[STONEMAN]: Requirements for Ada Programming Support Environments, "Stoneman"; Department of Defense; February 1980.

KERNEL Ada Programming Support Environment (KAPSE) Interface Team: *Public Report*; *Volume I*, Naval Ocean Systems Center TD509; April 1982.

APSE Interoperability and Transportability Implementation Strategy; Ada Joint Program Office; June 1983.

[ANSI 79]: American National Standards Institute, "American National Standard Additional Controls for Use with American National Standard Code for Information Interchange (ANSI Standard X3.64-1979)"; July 1979.

[ANSI 77]: American National Standards Institute, "American National Standard Code for Information Interchange (ANSI Standard X3.4-1977)"; June 1977.

ALS KAPSE — B5 Specification, SofTech; February 1982.

Computer Program Development Specification for Ada Integrated Environment: KAPSE/DATABASE TYPE B5, Intermetrics Inc.; 12 Nov 1982.

3. CAIS NODE MODEL

The CAIS implementation acts as a manager for a set of entities that may be files, processes, and devices. These entities have properties and may be interrelated in many ways.

The CAIS model uses the notion of a **node** as a carrier of information about an entity. It uses the notion of a **relationship** for representing an interrelation between entities and the notion of an **attribute** for representing a property of an entity or of an interrelation.

This version of the CAIS identifies four different kinds of nodes: structural nodes, file nodes, process nodes, and device nodes.

The structure provided by the CAIS node model is a directed graph of nodes, each of which may have content, relationships and attributes; relationships may also have attributes. The **content** varies with the kind of node. If a node is a structural node, there is no content and the node is used strictly as a holder of relationships and attributes. If a node is a file node, the content is an Ada external file. If a node is a process node, the content is the representation of the execution of an Ada program. If a node is a device node, its content is a representation of a logical or physical device.

3.1 RELATIONSHIPS AND RELATIONS

The relationships of CAIS nodes form the edges of a directed graph; they are used to build conventional hierarchical directory and process structures (see Section 4.1 CAIS_STRUCTURAL_NODES, Section 6.2 CAIS_PROCESS_CONTROL and Appendix B) as well as arbitrary directed-graph structures. Relationships are unidirectional and are said to emanate from a **source node** and to terminate at a **target node**.

Because any node may have many relationships representing many different classes of connections, the concept of a **relation** is introduced to categorize the relationships. These relations identify the nature of relationships, and **relationships** are instances of relations. There are several predefined relations provided by the CAIS. These are: PARENT, USER, JOB, CURRENT_JOB, CURRENT_USER, CURRENT_NODE, and DOT and are explained in the following sections.

Each relationship is designated by a relation name and a relationship key. The **relation name** identifies the relation and the **relationship key** distinguishes between multiple nodes each bearing the same relation with a given node. If a relationship is a unique instance of its relation (i.e., only one node bears the relation with a given node), the key may be omitted (i.e., its value is the null string). In this document, a relation name is often referred to simply as a relation and a relationship key is often referred to simply as a key. Nodes in the environment are accessible by navigating along the (named) relationships. Operations are provided to move from one node (along one of its relationships) to a connected node.

3.1.1 Kinds of Relationships

There are two kinds of relationships: primary and secondary. Primary relationships form a strict tree; secondary relationships may form an arbitrary directed graph. There is no requirement that all primary relationships have the same relation name.

When a node is created, a primary relationship must be initially established from some other node, called its **parent** node. This initial relationship is marked as the **primary** relationship for this new node. As a side effect of the creation, the new node will be connected back to this parent via the PARENT relation (which, because it is unique, has a null relationship key). To delete a node, the primary relationship is broken. RENAME (see Section 3.5) may be used to make the primary relationship emanate from a different parent. These operations maintain a state in which each non-root node has exactly one parent and a unique primary pathname (see Section 3.1.2).

Secondary relationships are arbitrary connections which may be established between two existing nodes. They are created with the LINK procedure (see Section 3.5) and broken with the UNLINK procedure. If a node is deleted (i.e., its primary relationship is broken), outstanding secondary relationships for which it is the target may remain, but attempts to access the node via these relationships will raise an exception.

3.1.2 Predefined Relations

The CAIS node model incorporates the notion of a user. Each user has one **top-level** node (often called the user directory). This top-level node is the root of the user's work-area tree, and from it he can access other structural, file, process and device nodes. Every node may be accessed by following a sequence of relationships; this sequence is called the **path** to the node. A path starting at a top-level node is called an **absolute** path. Every node can be traced back to its top-level node by recursively following PARENT relationships; the path obtained by inverting this chain is the unique primary path to the node.

A path can also start at a known (not necessarily top-level) node and follow a sequence of relationships to a desired node. This is a **relative** path and the known starting node is called the **base**.

Any user's top-level node can be accessed from a proces node using the relation USER and a relationship key which is interpreted as the user's name. User names may in fact be names of projects, services, people, or other organizational entities; each has a top level node associated with it. It is anticipated that certain special user names will be defined (as an eventual part of the CAIS) to provide uniform access to common tools, structures, etc.. Each implementation must identify such user names to be of special significance in the environment.

When a user enters the APSE, a root process node is created which often represents a command interpreter or other user-communication process; a process tree develops from this root node as other processes are invoked for the user. A particular user may have entered the APSE several times concurrently. Each corresponding process tree is referred to as a **job**. The JOB relation is provided for locating each of these root processes from the user's top-level node. Thus a JOB relation emanates from each user's top-level node to the root process node of each of the user's jobs. The JOB relation must always be used with a relationship-key which identifies the name of the particular job which is to be accessed.

Any process node in a job has associated with it at least three predefined relations. The CURRENT_JOB relationship always points to the root node for a process node's job. The CURRENT_USER relationship always points to the user's top-level node. The CURRENT_ NODE relationship always points to a node which represents the process node's current focus or context for its activities; the target node is often a structural node. The process node can thus use the CURRENT_NODE for a base node when specifying relative paths. All three of these relations (CURRENT_JOB, CURRENT_USER, and CURRENT_NODE) provide a convenient means for accessing other CAIS nodes.

Many CAIS operations allow the user to omit the relation name when referring to a relationship, defaulting it to "DOT". DOT is therefore referred to as the **default** relation.

The node model also uses the concept of **current process**. This is implicit in all calls to CAIS operations and refers to the currently executing process making the call. It defines the context in which the parameters are to be interpreted. In particular, paths are determined in the context of the current process.

3.1.3 Pathnames

Nodes are accessed by navigating along the relationships. These paths are specified using a pathname syntax. Starting from a given node, a path is followed by traversing a sequence of relationships until the desired node is reached. The **pathname** for this path is made up of the concatenation of the names of the traversed relationships in the same order in which they are encountered.

The syntax of a pathname is a sequence of path elements, each path element representing the traversal of a single relationship. A path element is an apostrophe (" ' ", pronounced "tick") followed by a relation name and a parenthesized relationship key (which may be omitted if the relationship is a unique instance of the relation for this node). If the relation is the default relation DOT, then the path element may be represented simply by a dot (" . ") followed by the key for the default relation DOT. Thus, "'DOT(CONTROLLER)" is the same as ".CONTROLLER".

Pathnames are interpreted relative to a known node. This node may be identified explicitly as an additional argument, the BASE, to many of the CAIS operations. Otherwise, the current process node is used as the starting point for interpretation of the path.

A pathname may begin simply with a relationship key, not prefixed by either " ' " or " . ". This is taken to mean interpretation following the DOT relation of the CURRENT_NODE. Thus "AIRPORT" is the same as " 'CURRENT_NODE.AIRPORT". By convention, the null pathname " " is interpreted as the CURRENT_NODE of the current process.

A pathname may also consist of just a single " . ". This is interpreted as referring to the current process node.

Relation names and relationship keys follow the syntax of Ada identifiers. Upper and lower case are treated as equivalent within such identifiers. For example, all of the following are legal node pathnames, and they would all refer to the same node if the CURRENT_NODE were " 'USER(JONES).TRACKER " and the CURRENT_USER were "JONES":

a. Landing_System'With_unit(Radar)

b. 'User(Jones).TRACKER.Landing_system'with_UNIT(RADAR)

c. 'CURRENT_USER.TRACKER.LANDING_SYSTEM'WITH_unit(radar)

By convention a relationship key of simply "#" is taken to represent the LATEST_KEY (lexicographically last). When creating a node or relationship, use of "#" as the final key of a pathname will cause a key to be automatically assigned, lexicographically following all previous keys for the same relation. This may be used to automatically assign revision identifiers or process keys (see Section 6.2).

The Backus-Naur Form (BNF) for pathnames is given in Table 3-1.

TABLE 3-1
PATHNAME BNF

```
PATHNAME :: = { PATHELEMENT } |
              RELATIONSHIP_KEY { PATHELEMENT } |
              " . "

PATHELEMENT :: = " ' " RELATION_NAME "(" RELATIONSHIP_KEY ")" |
                 " ' " RELATION_NAME |
                 " . " RELATIONSHIP_KEY

RELATION_NAME :: = IDENTIFIER
RELATIONSHIP_KEY :: = IDENTIFIER | " # "
```

3.2 ATTRIBUTES

Both nodes and relationships may have **attributes** which provide information about the node or relationship. Attributes are identified by an attribute name. Each attribute (see Section 3.6 CAIS_ATTRIBUTES) has a list of the values assigned to it, represented using the CAIS_LIST_UTILS (see Section 8.2.2) type called LIST.

Relation names and attribute names both have the same form (that is, the syntax of an Ada identifier), and they must be different from each other for a given node.

This version of the CAIS introduces two pre-defined node attributes: ACCESS_CONTROL and SECURITY_LEVEL.

3.3 GENERAL NODE MANAGEMENT

The operations defined in package CAIS_NODE_MANAGEMENT are applicable to all nodes except where explicitly stated otherwise in the package semantics section.

The creation of nodes for files is performed by the CREATE procedures of the Input/Output packages; the creation of nodes for processes is performed by INVOKE_PROCESS and SPAWN_PROCESS of CAIS_PROCESS_CONTROL (see Section 6.2); the creation of structural nodes is performed by CREATE_NODE (see Section 4.1); the creation of device nodes is performed by the CREATE procedures of CAIS_TERMINAL_SUPPORT (see Section 7.1.1).

To simplify manipulation by Ada programs, an Ada type NODE_TYPE is defined to represent an internal handle for a node. Most procedures either expect a NODE_TYPE parameter, or a pathname, or a combination of a BASE node (specified by a NODE_TYPE parameter) and a pathname relative to it.

3.4 PACKAGE CAIS_NODE_DEFS

This package defines the Ada type NODE_TYPE, which provides an internal (private) reference to CAIS nodes. This is referred to as a node **handle**. It also defines certain enumeration and record types and exceptions useful for node manipulations.

3.4.1 Package Specification

```
with IO_EXCEPTIONS;
package CAIS_NODE_DEFS is

    type NODE_TYPE is     limited private;
    type NODE_KIND is     (FILE, STRUCTURAL, PROCESS, DEVICE);

    subtype NAME_STRING is          STRING;

    subtype NAME_STRING is          STRING;
    subtype FORM_STRING is          STRING;
    subtype RELATIONSHIP_KEY is     STRING;
    subtype RELATION_NAME is        STRING;

    TOP_LEVEL          : constant STRING     := " 'CURRENT_USER";
    CURRENT_NODE       : constant STRING     := " ";
    CURRENT_PROCESS    : constant STRING     := " . ";
    LATEST_KEY         : constant STRING     := "#";

    -- Exceptions

    STATUS_ERROR       : exception renames IO_EXCEPTIONS.STATUS_ERROR;
    MODE_ERROR         : exception renames IO_EXCEPTIONS.MODE_ERROR;
    NAME_ERROR         : exception renames IO_EXCEPTIONS.NAME_ERROR;
    USE_ERROR          : exception renames IO_EXCEPTIONS.USE_ERROR;
    LAYOUT_ERROR       : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;

private
    -- implementation-dependent
end CAIS_NODE_DEFS;
```

3.4.2 Package Semantics

```
TOP_LEVEL          : constant STRING    : = " 'CURRENT_USER";
CURRENT_NODE       : constant STRING    : = " ";
CURRENT_PROCESS    : constant STRING    : = " . ";
LATEST_KEY         : constant STRING    : = " # ";
```

Define the standard pathnames for current user's top-level node, current node, current process, and latest key.

```
STATUS_ERROR : exception renames IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR   : exception renames IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR   : exception renames IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR    : exception renames IO_EXCEPTIONS.USE_ERROR;
LAYOUT_ERROR : exception renames IO_EXCEPTIONS.LAYOUT_ERROR;
```

Renames the corresponding exceptions for the LRM.

3.5 PACKAGE CAIS_NODE_MANAGEMENT

This package defines the general primitives for manipulating, copying, renaming, and deleting nodes and their relationships.

3.5.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_NODE_MANAGEMENT is

    subtype NODE_TYPE          is  CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING        is  CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY   is  CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME      is  CAIS_NODE_DEFS.RELATION_NAME;


    procedure OPEN (NODE:     in out  NODE_TYPE;
                    NAME:     in      NAME_STRING);
    procedure OPEN (NODE:     in out  NODE_TYPE;
                    BASE:     in      NODE_TYPE;
                    KEY:      in      RELATIONSHIP_KEY := "";
                    RELATION: in      RELATION_NAME := "DOT");

    procedure  CLOSE(NODE: in out NODE_TYPE);

    function  IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;

    function  KIND (NODE: in NODE_TYPE)
              return        CAIS_NODE_DEFS.NODE_KIND;

    function  PRIMARY_NAME(NODE: in NODE_TYPE) return NAME_STRING;

    function  PRIMARY_KEY (NODE: in NODE_TYPE)
              return RELATIONSHIP_KEY;

    function  PRIMARY_RELATION (NODE: in NODE_TYPE)
              return RELATION_NAME;
```

```
function  PATH_KEY(NODE: in NODE_TYPE) return RELATIONSHIP_KEY;
function  PATH_RELATION(NODE: in NODE_TYPE) return RELATION_NAME;

procedure GET_PARENT(NODE:   in      NODE_TYPE;
                     PARENT: in out  NODE_TYPE);

procedure COPY_NODE (FROM:      in  NODE_TYPE;
                     TO:        in  NAME_STRING);
procedure COPY_NODE (FROM:      in  NODE_TYPE;
                     TO_BASE:   in  NODE_TYPE;
                     TO_KEY:    in  RELATIONSHIP_KEY := " ";
                     TO_RELATION: in RELATION_NAME := "DOT");

procedure COPY_TREE (FROM:      in  NODE_TYPE;
                     TO:        in  NAME_STRING);
procedure COPY_TREE (FROM:      in  NODE_TYPE;
                     TO_BASE:   in  NODE_TYPE;
                     TO_KEY:    in  RELATIONSHIP_KEY := " ";
                     TO_RELATION: in RELATION_NAME := "DOT");

procedure RENAME(NODE:          in  NODE_TYPE;
                 NEW_NAME:      in  NAME_STRING);
procedure RENAME(NODE:             in     NODE_TYPE;
                 NEW_BASE:         in     NODE_TYPE;
                 NEW_KEY:          in     RELATIONSHIP_KEY := " ";
                 NEW_RELATION:     in     RELATION_NAME := "DOT");

procedure LINK(TO:          in  NAME_STRING;
               NEW_PATH:    in  NAME_STRING);
procedure LINK(TO_NODE:     in  NODE_TYPE;
               NEW_BASE:    in  NODE_TYPE;
               KEY:         in  RELATIONSHIP_KEY := " ";
               RELATION:    in  RELATION_NAME := "DOT");

procedure UNLINK(NAME:      in  NAME_STRING);
procedure UNLINK(BASE:      in  NODE_TYPE;
                 KEY:       in  RELATIONSHIP_KEY := " ";
                 RELATION:  in  RELATION_NAME := "DOT");

procedure DELETE_NODE(NAME:  in      NAME_STRING);
procedure DELETE_NODE(NODE:  in out  NODE_TYPE);

procedure DELETE_TREE(NODE:  in out  NODE_TYPE);

type NODE_ITERATOR is private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN is RELATION_NAME;
subtype NODE_KIND is CAIS_NODE_DEFS.NODE_KIND;

procedure ITERATE(ITERATOR:          out  NODE_ITERATOR;
                  NODE:              in   NODE_TYPE;
                  KIND:              in   NODE_KIND;
                  KEY:               in   RELATIONSHIP_KEY_PATTERN := "*";
                  RELATION:          in   RELATION_NAME_PATTERN := "DOT";
                  PRIMARY_ONLY:      in   BOOLEAN:= TRUE;
```

```
function MORE (ITERATOR:              in      NODE_ITERATOR)
        return  BOOLEAN;
procedure GET_NEXT(ITERATOR:          in out  NODE_ITERATOR;
                   NEXT_NODE:         in out  NODE_TYPE);

procedure SET_CURRENT_NODE(NAME: in       NAME_STRING);
procedure SET_CURRENT_NODE(NODE: in       NODE_TYPE);

procedure GET_CURRENT_NODE(NODE: out      NODE_TYPE);

function IS_SAME(NAME1: in      NAME_STRING;
                 NAME2: in      NAME_STRING)
        return BOOLEAN;

function IS_SAME(NODE1: in      NODE_TYPE;
                 NODE2: in      NODE_TYPE)
        return BOOLEAN;

-- Exceptions

NAME_ERROR :exception renames CAIS_NODE_DEFS.NAME_ERROR;
USE_ERROR  :exception renames CAIS_NODE_DEFS.USE_ERROR;


private
    -- implementation-dependent
end CAIS_NODE_MANAGEMENT;
```

3.5.2 Package Semantics

```
subtype NODE_TYPE          is   CAIS_NODE_DEFS.NODE_TYPE;
subtype NAME_STRING        is   CAIS_NODE_DEFS.NAME_STRING;
subtype RELATIONSHIP_KEY   is   CAIS_NODE_DEFS.RELATIONSHIP_KEY;
subtype RELATION_NAME      is   CAIS_NODE_DEFS.RELATION_NAME;
```

The key of a node is the relationship key of the last element of its pathname. Many operations are allowed to take either a pathname or a base-node/key/relation-name.

```
procedure OPEN (NODE:      in out  NODE_TYPE;
                NAME:      in      NAME_STRING);
procedure OPEN (NODE:      in out  NODE_TYPE;
                BASE:      in      NODE_TYPE;
                KEY:       in      RELATIONSHIP_KEY := " ";
                RELATION:  in      RELATION_NAME := "DOT");
```

Returns an open node handle on the designated node. The NAME_ERROR exception will be raised if the node does not exist.

An open node handle acts as if the handle forms a temporary secondary relationship to the node; this means that, if the opened node pointed to is renamed (potentially by another process), the operations on the opened node track the renaming. Tools which require that node relationships remain unchanged between node-level CAIS operations use have the features of the CAIS_NODE_CONTROL package (Section 3.7) to synchronize node usage.

```
procedure CLOSE(NODE: in out   NODE_TYPE);
```

Severs any association between the node handle and the node and releases any associated lock. This must be done before another OPEN can be done using the same NODE_TYPE variable by the same process.

```
function IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;
```

Returns TRUE or FALSE according to open status of the node handle.

```
function KIND (NODE:  in       NODE_TYPE)
        return CAIS_NODE_DEFS.NODE_KIND;
```

Returns the kind of a node, either FILE, PROCESS, STRUCTURAL, or DEVICE.

```
function PRIMARY_NAME(NODE: in NODE_TYPE) return NAME_STRING;
```

Returns the *full primary pathname* to the node.

```
function PRIMARY_KEY (NODE:        in      NODE_TYPE)
        return RELATIONSHIP_KEY;
function PRIMARY_RELATION (NODE: in        NODE_TYPE)
        return RELATION_NAME;
```

Returns the corresponding part of the last element of the primary path to the node. If the node is a top-level node, the key is the user name, and the relation name is USER.

```
function PATH_KEY(NODE: in NODE_TYPE) return RELATIONSHIP_KEY;
function PATH_RELATION(NODE: in NODE_TYPE) return RELATION_NAME;
```

Returns the corresponding part of the last element of the path used to access this node. If the path was an absolute path and this is a top-level node, the relationship key is the user name, and the relation name is USER.

```
procedure GET_PARENT(NODE:  in       NODE_TYPE;
                     PARENT: in out  NODE_TYPE);
```

Returns the parent node. Generate an exception if NODE is a top-level node.

```
procedure COPY_NODE (FROM:       in     NODE_TYPE;
                     TO:         in     NAME_STRING);
procedure COPY_NODE (FROM:       in     NODE_TYPE;
                     TO_BASE:    in     NODE_TYPE;
                     TO_KEY:     in     RELATIONSHIP_KEY := " ";
                     TO_RELATION: in    RELATION_NAME := "DOT");
```

Copies a node. Any secondary relationships emanating from the original node are recreated in the copy. Unless the target of the original node's relationship is the node itself, then the copied relationship still refers to the same target node. If the target is the node itself, then the copy will have an analogous relationship to itself. It is an error (USE_ERROR) if the node is a process or device node, or if any primary relationships emanate from the original node.

```
procedure COPY_TREE (FROM:       in     NODE_TYPE;
                     TO:         in     NAME_STRING);
procedure COPY_TREE (FROM:       in     NODE_TYPE;
                     TO_BASE:    in     NODE_TYPE;
                     TO_KEY:     in     RELATIONSHIP_KEY := " ";
                     TO_RELATION: in    RELATION_NAME := "DOT");
```

Copies a tree of nodes (formed by primary relationships), as well as their secondary relationships. Secondary relationships between two nodes which are both copied are recreated between the two copies. Secondary relationships emanating from a node which is copied, but which refer to nodes outside the tree being copied, are copied so that they emanate from the copy, but still refer to the old (uncopied) node. The exception USE_ERROR will be raised if any node in the tree is a process or device.

```
procedure RENAME(NODE:          in   NODE_TYPE;
                 NEW_NAME:      in   NAME_STRING);
procedure RENAME(NODE:          in   NODE_TYPE;
                 NEW_BASE:      in   NODE_TYPE;
                 NEW_KEY:       in   RELATIONSHIP_KEY := " ";
                 NEW_RELATION:  in   RELATION_NAME := "DOT");
```

Changes the primary connection to a node and adjusts the PARENT relationship appropriately.

Existing secondary relationships with the renamed node as target will track the renaming. An implementation may raise USE_ERROR if the renaming cannot be accomplished while still maintaining consistent secondary relationships and acircularity of primary relationships. RENAME raises the exception USE_ERROR if a node already exists with the new name.

Existing processes with open node handles track the renamed node; the node's handle acts as if the accessing process had a temporary secondary relationship to the node.

```
procedure LINK (TO:          in   NAME_STRING;
                NEW_PATH:    in   NAME_STRING);
procedure LINK (TO_NODE:     in   NODE_TYPE;
                NEW_BASE:    in   NODE_TYPE;
                KEY:         in   RELATIONSHIP_KEY := " ";
                RELATION:    in   RELATION_NAME := "DOT");
```

Creates a relationship from one existing node to another. This relationship will be identified as a secondary relationship.

The first LINK procedure takes the name of the target node as the TO argument and a NEW_PATH which should lead to it. The base/key/relation are implied by the NEW_PATH. The second LINK procedure takes a handle on the target node, a handle on the NEW_BASE, and an explicit key and relation to be established from NEW_BASE to TO_NODE.

```
procedure UNLINK (NAME:      in   NAME_STRING);
procedure UNLINK (BASE:      in   NODE_TYPE;
                  KEY:       in   RELATIONSHIP_KEY := " ";
                  RELATION:  in   RELATION_NAME := "DOT");
```

Deletes a secondary relationship. Raises USE_ERROR if the specified relationship is a primary relationship or does not exist.

```
procedure DELETE_NODE(NAME:  in       NAME_STRING);
procedure DELETE_NODE(NODE:  in out   NODE_TYPE);
```

Deletes the primary relationship to a node and the node itself. It is an error if any primary relationships emanate from this node.

This delete operation closes NODE, removes the appropriate relationship from the node's parent and updates the node's parent. If a process node is not TERMINATED (see Section 6.1), this action aborts its process. This delete operation cannot be used to delete more than one node in a single operation.

```
procedure DELETE_TREE(NODE:      in out NODE_TYPE);
```

DELETE_TREE deletes a node and recursively deletes all nodes with the designated node as their parent. This operation closes the NODE handle and removes the appropriate relationship from the node's parent. This operation can be used to delete more than one node in a single operation. If DELETE_TREE raises the USE_ERROR exception, no node may be deleted.

```
type NODE_ITERATOR is private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
subtype RELATION_NAME_PATTERN is RELATION_NAME;
subtype NODE_KIND is CAIS_NODE_DEFS.NODE_KIND;
```

RELATIONSHIP_KEY_PATTERN and RELATION_NAME_PATTERN follow the syntax of relationship keys/relation names, except that a "?" will match any single character and a "*" will match any string of characters.

```
procedure ITERATE(ITERATOR:          out  NODE_ITERATOR;
                  NODE:              in   NODE_TYPE;
                  KIND:              in   NODE_KIND;
                  KEY:               in   RELATIONSHIP_KEY_PATTERN := "*";
                  RELATION:          in   RELATION_NAME_PATTERN := "DOT";
                  PRIMARY_ONLY:      in   BOOLEAN := TRUE);

function MORE (ITERATOR:             in   NODE_ITERATOR)
        return  BOOLEAN;

procedure GET_NEXT(ITERATOR:         in out  NODE_ITERATOR;
                   NEXT_NODE:        in out  NODE_TYPE);
```

These three operations iterate through those nodes referred to from the given NODE, via primary or secondary relationships that have keys/relations satisfying the specified patterns.

The nodes are returned in ASCII lexicographical order by RELATION and then by relationship KEY. The key and relation are available by the functions PATH_KEY and PATH_RELATION (see above). Nodes that are of a different kind than the KIND specified are omitted.

If PRIMARY_ONLY is true, then only primary relationships are considered when creating the iterator. In this case, either PATH_KEY/PATH_RELATION or PRIMARY_KEY/PRIMARY_RELATION may be used to determine the relationship which caused the node to be included in the iteration.

Similarly, these operations iterate through the primary or secondary relationships from the given NODE which have keys/relations satisfying the specified patterns.

```
procedure SET_CURRENT_NODE(NAME: in   NAME_STRING);
procedure SET_CURRENT_NODE(NODE: in   NODE_TYPE);
```

Specifies NODE/NAME as the current node.

```
procedure GET_CURRENT_NODE(NODE: out NODE_TYPE);
```

Opens a handle on the current node. This is equivalent to OPEN(NODE, " 'CURRENT_NODE")

```
function IS_SAME(NAME1: in   NAME_STRING;
                 NAME2: in   NAME_STRING)
    return BOOLEAN;

function IS_SAME(NODE1: in   NODE_TYPE;
                 NODE2: in   NODE_TYPE)
    return BOOLEAN;
```

Returns TRUE if both names/node handles refer to the same CAIS node.

3.6 PACKAGE CAIS_ATTRIBUTES

This package supports the definition and manipulation of named attributes for nodes and relationships. Each attribute is a list of the format defined by the package CAIS_LIST_UTILS (see Section 8.2.2). The name of an attribute follows the syntax of an Ada identifier. Upper/lower case distinctions are significant within the value of attributes, but not within the attribute name.

It is anticipated that certain attribute names and their values will be included as part of the CAIS definition. In any case, each implementation must identify those attribute names and values which are reserved or which have special significance.

The operations in this package are overloaded to permit access to nodes and relationships by either the name strings or the node handles. Access by the node handle assures that the operation tracks the node (which may be renamed or locked once open).

3.6.1 Package Specification

```
with CAIS_LIST_UTILS;
with CAIS_NODE_DEFS;
package CAIS_ATTRIBUTES is

    subtype NAME_STRING   is    CAIS_NODE_DEFS.NAME_STRING;
    subtype NODE_TYPE     is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype LIST          is    CAIS_LIST_UTILS.LIST;
    subtype ATTRIB_NAME   is    STRING;
    type FLAG_ENUM        is    (READ_ONLY, INHERIT);

    procedure SET_NODE_ATTRIBUTE(NAME:   in out NAME_STRING;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);
    procedure SET_NODE_ATTRIBUTE(NODE:   in out NODE_TYPE;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);

    procedure SET_PATH_ATTRIBUTE(NAME:   in out NAME_STRING;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);
    procedure SET_PATH_ATTRIBUTE(NODE:   in     NODE_TYPE;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);

    procedure GET_NODE_ATTRIBUTE(NAME:   in     NAME_STRING;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);
    procedure GET_NODE_ATTRIBUTE(NODE:   in out NODE_TYPE;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);

    procedure GET_PATH_ATTRIBUTE(NAME:   in     NAME_STRING;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);
    procedure GET_PATH_ATTRIBUTE(NODE:   in     NODE_TYPE;
                                 ATTRIB: in     ATTRIB_NAME;
                                 VALUE:  in     LIST);

    type ATTRIB_ITERATOR is private;
    subtype ATTRIB_PATTERN is STRING;
```

```
    procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:   in out   ATTRIB_ITERATOR;
                                      NAME:       in       NAME_STRING;
                                      PATTERN:    in       ATTRIB_PATTERN := " * ");
    procedure  NODE_ATTRIBUTE_ITERATE (ITERATOR:  in out   ATTRIB_ITERATOR;
                                      NODE:       in       NODE_TYPE;
                                      PATTERN:    in       ATTRIB_PATTERN := " * ");

    procedure PATH_ATTRIBUTE_ITERATE (ITERATOR:   in out   ATTRIB_ITERATOR;
                                      NAME:       in       NAME_STRING;
                                      PATTERN:    in       ATTRIB_PATTERN := " * ");
    procedure PATH_ATTRIBUTE_ITERATE(ITERATOR:    in out   ATTRIB_ITERATOR;
                                      NODE:       in       NODE_TYPE;
                                      PATTERN:    in       ATTRIB_PATTERN := " * ");

    function MORE (ITERATOR:          in       ATTRIB_ITERATOR)
            return BOOLEAN;

    procedure GET_NEXT(ITERATOR:      in out   ATTRIB_ITERATOR;
                       ATTRIB:           out   ATTRIB_NAME;
                       VALUE:         in out   LIST);

    procedure SET_FLAG(NAME:          in       NAME_STRING;
                       ATTRIB:        in       ATTRIB_NAME;
                       WHICH:         in       FLAG_ENUM;
                       TO:            in       BOOLEAN := TRUE);
    procedure SET_FLAG(NODE:          in       NODE_TYPE;
                       ATTRIB:        in       ATTRIB_NAME;
                       WHICH:         in       FLAG_ENUM;
                       TO:            in       BOOLEAN := TRUE);

    function FLAG (NAME:              in       NAME_STRING;
                   ATTRIB:            in       ATTRIB_NAME;
                   WHICH:             in       FLAG_ENUM)
            return BOOLEAN;
    function FLAG (NODE:              in       NODE_TYPE;
                   ATTRIB:            in       ATTRIB_NAME;
                   WHICH:             in       FLAG_ENUM)
            return BOOLEAN;

    -- Exceptions

    USE_ERROR :exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_ATTRIBUTES;
```

3.6.2 Package Semantics

```
    subtype NAME_STRING   is    CAIS_NODE_DEFS.NAME_STRING;
    subtype NODE_TYPE     is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype LIST          is    CAIS_LIST_UTILS.LIST;
    subtype ATTRIB_NAME   is    STRING;
```

Each CAIS node or relationship may have list-valued attributes. They are associated with nodes referred to by a pathname or node handle and with relationships referred to by the last step in a pathname or by the last step associated by a pathname.

```
type FLAG_ENUM is (READ_ONLY, INHERIT);
```

The type FLAG_ENUM selects one of two flags associated with each attribute. Attributes with the READ_ONLY flag may not be written. Attributes with no READ_ONLY flag may be read or written. If a node has attributes with the INHERIT flag set, then nodes created with that node as their parent will have the initial values for these attributes copied from those of the parent node.

```
procedure SET_NODE_ATTRIBUTE(NAME:  in out NAME_STRING;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);
procedure SET_NODE_ATTRIBUTE(NODE:  in out NODE_TYPE;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);

procedure SET_PATH_ATTRIBUTE(NAME:  in out NAME_STRING;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);
procedure SET_PATH_ATTRIBUTE(NODE:  in     NODE_TYPE;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);
```

Sets the given node/relationship attribute. If an attribute with the given name already exists, then the existing value is over-written by the given value; if it does not exist, a new attribute is created and set to the given value. Setting the value of the attribute to an empty list deletes the attribute. This operation will fail with USE_ERROR if the attribute is READ_ONLY or if the current process does not have update access to the node.

```
procedure GET_NODE_ATTRIBUTE(NAME:  in     NAME_STRING;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);
procedure GET_NODE_ATTRIBUTE(NODE:  in out NODE_TYPE;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);

procedure GET_PATH_ATTRIBUTE(NAME:  in     NAME_STRING;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);
procedure GET_PATH_ATTRIBUTE(NODE:  in     NODE_TYPE;
                             ATTRIB: in     ATTRIB_NAME;
                             VALUE:  in     LIST);
```

Gets the current value of an attribute. If the attribute has never been set, then these operations return the empty list.

```
type ATTRIB_ITERATOR is private;
subtype ATTRIB_PATTERN is STRING;
```

An attribute iterator is used to sequence through the names of the attributes of a node or a relationship. An ATTRIB_PATTERN has the same syntax as an ATTRIB_NAME, except that "?" stands for any character and "*" stands for zero or more arbitrary characters.

By using simply the pattern "*" it is possible to iterate through the names of all of the non-null attributes of a node.

```
procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:   in out  ATTRIB_ITERATOR;
                                  NAME:       in      NAME_STRING;
                                  PATTERN:    in      ATTRIB_PATTERN := "*");
procedure NODE_ATTRIBUTE_ITERATE (ITERATOR:   in out  ATTRIB_ITERATOR;
                                  NODE:       in      NODE_TYPE;
                                  PATTERN:    in      ATTRIB_PATTERN := "*");

procedure PATH_ATTRIBUTE_ITERATE (ITERATOR:   in out  ATTRIB_ITERATOR;
                                  NAME:       in      NAME_STRING;
                                  PATTERN:    in      ATTRIB_PATTERN: = "*");
procedure PATH_ATTRIBUTE_ITERATE (ITERATOR:   in out  ATTRIB_ITERATOR;
                                  NODE:       in      NODE_TYPE;
                                  PATTERN:    in      ATTRIB_PATTERN: = "*");

function MORE (ITERATOR:  in   ATTRIB_ITERATOR)
        return BOOLEAN;

procedure GET_NEXT(ITERATOR:  in out   ATTRIB_ITERATOR;
                   ATTRIB:       out   ATTRIB_NAME
                   VALUE:     in out   LIST);
```

These operations iterate through the names of the attributes of a node or relationship which match the given pattern. The names are returned in ASCII lexicographical order.

```
procedure SET_FLAG(NAME:    in  NAME_STRING;
                   ATTRIB:  in  ATTRIB_NAME;
                   WHICH:   in  FLAG_ENUM;
                   TO:      in  BOOLEAN := TRUE);
procedure SET_FLAG(NODE:    in  NODE_TYPE;
                   ATTRIB:  in  ATTRIB_NAME;
                   WHICH:   in  FLAG_ENUM;
                   TO:      in  BOOLEAN := TRUE);

function FLAG (NAME:    in   NAME_STRING;
               ATTRIB:  in   ATTRIB_NAME;
               WHICH:   in   FLAG_ENUM)
        return BOOLEAN;

function FLAG (NODE:    in   NODE_TYPE;
               ATTRIB:  in   ATTRIB_NAME;
               WHICH:   in   FLAG_ENUM)
        return BOOLEAN;
```

These two operations provide access to the READ_ONLY and INHERIT flags for each attribute. SET_FLAG sets the specified flage. The function FLAG returns the current setting of the flag.

3.7 PACKAGE CAIS_NODE_CONTROL

This version of the CAIS defines only primitives for dynamic access synchronization. Each operation on a node is independent, and both access control and synchronization status are re-checked for each operation. This package defines access synchronization operations at the node levels. For file (and device) nodes, an implementation may define the FORM string to permit an OPEN operation (LRM chapter 14; see also Sections 5 and 7 of this document) which specifies exclusive access; in that case the sequence of file (and device) opening, reading and writing, and closing, is considered a single node-level "operation". Use of file (or device) level access synchronization thus provides for longer transactions at the node level without locking the node's attributes and relationships (only content may be locked by file level OPEN actions). Use of node level access synchronization is intended for control at the level of the node as a whole (content, relationships, and attributes).

3.7.1 Package Specification

```
with CAIS_ATTRIBUTES;
with CAIS_NODE_DEFS;
with CAIS_NODE_CONTROL is

    subtype NODE_TYPE   is CAIS_NODE_DEFS.NODE_TYPE;
    subtype ATTRIB_NAME is CAIS_ATTRIBUTES.ATTRIB_NAME;
    ACCESS_CONTROL  : constant ATTRIB_NAME  := "ACCESS_CONTROL";
    SECURITY_LEVEL  : constant ATTRIB_NAME  := "SECURITY_LEVEL";

    procedure LOCK (NODE:       in  NODE_TYPE;
                    TIME_LIMIT: in  DURATION := DURATION'LAST);

    procedure UNLOCK (NODE:     in  NODE_TYPE);

private
    -- implementation-dependent
end CAIS_NODE_CONTROL;
```

3.7.2 Package Semantics

```
    subtype NODE_TYPE   is CAIS_NODE_DEFS.NODE_TYPE;
    subtype ATTRIB_NAME is CAIS_ATTRIBUTES.ATTRIB_NAME;
    ACCESS_CONTROL  : constant ATTRIB_NAME  := "ACCESS_CONTROL";
    SECURITY_LEVEL  : constant ATTRIB_NAME  := "SECURITY_LEVEL";
```

The CAIS provides two predefined attribute names for acces control: ACCESS_CONTROL for discretionary access control and SECURITY_LEVEL for non-discretionary access control. These attributes may be set at node creation *(by inclusion in the FORM string — see Section 4.1) or later with* SET_NODE_ATTRIBUTE *(see* Section 3.6).

```
    procedure LOCK (NODE:       in  NODE_TYPE;
                    TIME_LIMIT: in  DURATION := DURATION'LAST);

    procedure UNLOCK (NODE:     in  NODE_TYPE);
```

Locks/unlocks the designated node for a series of updates. When a node is locked, any other process that attempts to modify any attribute, relationship, or content of the node will receive an exception. If the node is already locked, then LOCK will be delayed until the node is unlocked or until the time limit expires. In the later case an exception will be raised.

3.8 PRAGMATICS

Several private types are defined as part of the CAIS Node Model. The actual implementation of these types may vary from one CAIS implementation to the next. Nevertheless, it is important to establish certain minimums for each type to enhance portability.

a. NAME_STRING — At least 255 characters in a CAIS pathname.

b. RELATIONSHIP_KEY
   KEY_STRING — At least 20 characters must be significant in (relationship) key.

c.	ATTRIB_NAME RELATION_NAME	At least 20 characters must be significant in attribute/relation names.
d.	Tree-height	At least 10 levels of heirarchy must be supported for the primary relationships.
e.	Record size number	At least 32767 bits per record must be supported.
f.	Open node count	Each process must be able to have at least 15 nodes open simultaneously.

4. CAIS STRUCTURAL NODES

Structural nodes are special nodes in the sense that they do not contain contents as do the other nodes of the CAIS model. Their purpose is solely to be carriers of common information about other nodes related to the structural node. Structural nodes are typically used to create conventional directories, configuration objects, etc.

The package CAIS_STRUCTURAL_NODES defines the primitive operations for creating structural nodes. All other operations for structural nodes are defined in Section 3.

4.1 PACKAGE CAIS_STRUCTURAL_NODES

4.1.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_STRUCTURAL_NODES is

    subtype NODE_TYPE              is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING            is    CAIS_NODE_DEFS.NAME_STRING;
    subtype FORM_STRING            is    CAIS_NODE_DEFS.FORMS_STRING;
    subtype RELATIONSHIP_KEY       is    CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME          is    CAIS_NODE_DEFS.RELATION_NAME;

    procedure CREATE_NODE(NAME:        in    NAME_STRING;
                          FORM:        in    FORM_STRING := " ");
    procedure CREATE_NODE(BASE:        in    NODE_TYPE;
                          KEY:         in    RELATIONSHIP_KEY := " ";
                          RELATION:    in    RELATION_NAME := "DOT";
                          FORM:        in    FORM_STRING := " ");
    procedure CREATE_NODE(NODE:        in out NODE_TYPE;
                          NAME:        in    NAME_STRING;
                          FORM:        in    FORM_STRING := " ");
    procedure CREATE_NODE(NODE:        in out NODE_TYPE;
                          BASE:        in    NODE_TYPE;
                          KEY:         in    RELATIONSHIP_KEY := " ";
                          RELATION:    in    RELATION_NAME := "DOT";
                          FORM:        in    FORM_STRING := " ");

private
    -- implementation-dependent
end CAIS_STRUCTURAL_NODES;
```

4.1.2 Package Semantics

```
    subtype NODE_TYPE              is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING            is    CAIS_NODE_DEFS.NAME_STRING;
    subtype FORM_STRING            is    CAIS_NODE_DEFS.FORM_STRING;
    subtype RELATIONSHIP_KEY       is    CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME          is    CAIS_NODE_DEFS.RELATION_NAME;
```

```
procedure CREATE_NODE(NAME:     in     NAME_STRING;
                      FORM:     in     FORM_STRING := " ");
procedure CREATE_NODE(BASE:     in     NODE_TYPE;
                      KEY:      in     RELATIONSHIP_KEY := " ";
                      RELATION: in     RELATION_NAME := "DOT";
                      FORM:     in     FORM_STRING := " ");
procedure CREATE_NODE(NODE:     in out NODE_TYPE;
                      NAME:     in     NAME_STRING;
                      FORM:     in     FORM_STRING := " ");
procedure CREATE_NODE(NODE:     in out NODE_TYPE;
                      BASE:     in     NODE_TYPE;
                      KEY:      in     RELATIONSHIP_KEY := " ";
                      RELATION: in     RELATION_NAME := "DOT";
                      FORM:     in     FORM_STRING := " ");
```

Creates a structural node with its primary relationship and parent node implied by the NAME in the first and third procedures and given explicitly in the second and fourth procedures.

The last two procedures return a node handle allowing immediate access to attributes and relationships.

If non-null, the FORM parameter provides initial values for attributes of the node, using Ada aggregate syntax, with each attribute name followed by a right-arrow (" = > ") and the attribute value (see Section 8.2.2 CAIS_LIST_UTILS for the syntax of attribute value).

5. CAIS FILE NODES

CAIS file nodes are nodes that represent information about and contain external files. The underlying model for the content of such a node is that of a file of data items, accessible randomly by some index or indices or sequentially. The basic operations on such files are provided by the Ada packages for Input/Output specified in Chapter 14 of the Ada LRM. While the semantics of the packages as specified in the LRM are fully adhered to, the CAIS imposes additional requirements on those semantics that the LRM designates as being implementation-defined. These requirements ensure consistent cooperation between the file-related, node-related, and device-related operations.

The CAIS defines additional Input/Output packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, CAIS_TEXT_IO, and CAIS_INTERACTIVE_IO. The first three packages are identical to the Input/Output packages specified in the Ada LRM, except that additional subprograms are added supporting more convenient and efficient file management operations by exploiting the CAIS Node Model. The package CAIS_INTERACTIVE_IO defines additional Input/Output facilities appropriate for files which are assigned to terminals.

To insure the consistency of file- and node-related operations the CAIS imposes the following two constraints on all I/O packages:

> A file must first be made accessible to an Ada program by an OPEN or CREATE, specifying the external file by a NAME and a FORM, both character strings. The formats of these strings are not specified in the Ada LRM. The CAIS requires the formats and semantics for NAME and FORM to adhere to the specifications given in Sections 3 and 4, respectively. Thus file names have the syntax of node pathnames.
>
> The CREATE operations both establish a new external file (as described in Chapter 14 of the Ada LRM) and have the side effect of creating the node for the file. The file node's primary relationship and parent node are implied by the NAME parameter. The DELETE operations have the side effect of deleting the node itself. DELETE operations are not legal if a file's node has primary relationships emanting from it. I/O DELETE operations require that the file be open; CAIS_NODE_MANAGEMENT DELETE operations require only that the node be open (but they also delete the contents with the deletion of the node itself).

While an implementation may provide a mechanism for file creation and opening to specify access synchronization, via the FORM parameter, that access synchronization refers to the file contents level only. To utilize node level access synchronization, the user must open the node explicitly and specify node synchronization operations (see Section 3.7). Files may be opened with or without node handles being opened, and nodes may be open before or while associated file handles are open.

5.1 Ada LRM INPUT/OUTPUT

5.1.1 Package IO_EXCEPTIONS

This package is specified by Chapter 14 of the Ada LRM. The LRM-defined package provides the definition for all exceptions generated by the input/output packages.

5.1.2 Package SEQUENTIAL_IO

This package provides sequential access to files/devices. This package is specified by Chapter 14 of the Ada LRM; however, because of additional pragmatic requirements it may require a specialized implementation in order to be utilized in a CAIS implementation.

5.1.3 Package DIRECT_IO

This package provides for direct-access input/output to files/devices. This package is specified by Chapter 14 of the Ada LRM; however, because of pragmatic and additional implied semantic requirements, it may require a specialized implementation in order to be utilized in a CAIS implementation.

5.1.4 Package TEXT_IO

This package provides sequential formatted input/output to ASCII text files. This package is specified in Chapter 14 of the Ada LRM, however, because of pragmatics and additional implied semantics, it may require a specialized implementation in order to be utilized in a CAIS implementation.

5.2 CAIS INPUT/OUTPUT

5.2.1 CAIS File Management

Section 14.2.1 of the Ada LRM defines the file management operations CREATE and OPEN that are included in each of the Ada LRM Input/Output packages. These operations use a pathname as identification of the external file. In the CAIS model, this pathname implies a navigation along relationships to reach the node whose content represents the desired external file.

In the CAIS, the navigation operations of CAIS_NODE_MANAGEMENT allow the identification of the node associated with a file by means of a pathname and also by means of an opened node handle, or a base node and a relationship identification (i.e., relation name and relationship key) leading to the desired node.

The procedures and functions described in this section provide for the control of external files; their declarations are repeated in each of the three packages for CAIS sequential, direct, and text input/output. In order to provide for a smooth transition from a file node to the file itself, and to prevent unnecessary repetitions of navigations, the file management operations CREATE and OPEN included in the packages CAIS_SEQUENTIAL_IO, CAIS_DIRECT_IO, and CAIS_TEXT_IO are provided in overloaded versions:

```
subtype NODE_TYPE is CAIS_NODE_DEFS.NODE_TYPE;
procedure CREATE (FILE:     in out   FILE_TYPE;
                  MODE:     in       FILE_MODE;
                  BASE:     in       NODE_TYPE;
                  KEY:      in       RELATIONSHIP_KEY: = " ";
                  RELATION: in       RELATION_NAME: = "DOT";
                  FORM:     in       FORM_STRING: = " ");

procedure OPEN (FILE:     in out   FILE_TYPE;
                MODE:     in       FILE_MODE;
                BASE:     in       NODE_TYPE;
                KEY:      in       RELATIONSHIP_KEY: = " ";
                RELATION: in       RELATION_NAME: = "DOT";
                FORM:     in       FORM_STRING: = " ");
procedure OPEN (FILE:     in out   FILE_TYPE;
                MODE:     in       FILE_MODE;
                NODE:     in       NODE_TYPE;
                FORM:     in       FORM_STRING: = " ");
```

The semantics of the operations are the same as specified in the Ada LRM Section 14.2.1 and CAIS Section 5.0, except that the external file is identified by means of the associated node handle or BASE, KEY, RELATION.

In addition, the following operation is provided to obtain an opened node handle for the node associated with a file:

```
procedure OPEN_NODE(NODE:  in out   NODE_TYPE;
                    FILE:  in       FILE_TYPE);
```

The exception STATUS_ERROR is raised if either the actual parameter for FILE is a closed file handle or the actual parameter for NODE is an already open node handle.

5.2.2 Package CAIS_SEQUENTIAL IO

This package provides sequential access to files/devices. This package is specified by Chapter 14 of the Ada RML; however, because of additional pragmatic requirements it may require a specialized implementation in order to be utilized in a CAIS implementation. Furthermore, the declarations given in Section 5.2.1 are added to the package.

5.2.3 Package CAIS_DIRECT_IO

This package provides for direct-access input/output to files/devices. This package is specified by Chapter 14 of the Ada LRM; however, because of pragmatic and additional implied semantic requirements, it may require a specialized implementation in order to be utilized in a CAIS implementation. Furthermore, the declarations in Section 5.2.1 are added to the package.

A conforming implementation should support access with package CAIS_SEQUENTIAL_IO to an external file created and/or maintained with CAIS_DIRECT_IO. (This requires that the generic instantiations of both packages utilize the identical ELEMENT_TYPE.)

5.2.4 Package CAIS_TEXT_IO

This package provides sequential formatted input/output to ASCII text files. This package is specified in Chapter 14 of the Ada LRM; however, because of pragmatics and additional implied semantics, it may require a specialized implementation in order to be utilized in a CAIS implementation. Furthermore, the declarations given in Section 5.2.1 are added to the package.

A conforming implementation that supports CAIS_INTERACTIVE_IO provides additional semantics in the CAIS_TEXT_IO package for the CAIS_TEXT_IO procedures and functions which are used in reference to printer-type terminals and Video Display Terminal (VDT) type terminals associated with an object of type CAIS_INTERFACE_IO.INTERACTIVE_TERMINAL.

The line terminator, page terminator, and file terminator characters are implementation-dependent.

A VDT functions identically to a hardcopy terminal unless bounds are set for the line length and/or page length. For a cursor-addressable VDT, the current column number and current line number of the associated input file and output file indicate the column number and line number, respectively, on the VDT display. The character position in the upper left corner of the VDT display is the first column of the first line of the first page.

The following procedures have additional semantics when used in reference to a terminal.

```
procedure SET_LINE_LENGTH(FILE :  in   FILE_TYPE; TO : in COUNT);
procedure SET_LINE_LENGTH(TO :    in   COUNT);
```

The exception USE_ERROR is raised if the value of TO is greater than the number of character positions on a line of the display.

```
procedure SET_PAGE_LENGTH(FILE :  in   FILE_TYPE; TO : in COUNT);
procedure SET_PAGE_LENGTH(TO :    in   COUNT);
```

In reference to a VDT the exception USE_ERROR is raised if the value of TO is greater than the number of lines on the display.

```
procedure NEW_LINE(FILE :        in  FILE_TYPE;
                   SPACING :     in  POSITIVE_COUNT := 1);
procedure NEW_LINE(SPACING :     in  POSITIVE_COUNT := 1);
```

In reference to a VDT the active position is moved to the first column of the line below the current line. If the active position was on the last line of the page, NEW_LINE causes all lines of the display to be moved upward such that the top line(s) is lost and the last line of the page is blank.

SPACING acts as defined in the LRM.

```
procedure NEW_PAGE(FILE :  in  FILE_TYPE);
procedure NEW_PAGE;
```

In reference to a VDT the screen is cleared and the active position is moved to the first column of the first line of the display.

```
procedure GET(. . .);
```

In reference to a cursor-addressable VDT with a bounded line length the GET procedures clear a portion of the display starting at the active position and equal in length to the maximum possible length of the item to be read. The active position is not changed. The data to be read is buffered as the user enters it. Implementation defined editing operations are permitted. No characters other than the printable characters and horizontal tab (HT) may be returned.

```
procedure SET_ERROR (FILE :  in  FILE_TYPE);
```

Provides an open file handle to be used for current error output. The exception MODE_ERROR is raised if the mode of FILE is IN_FILE.

```
function STANDARD_ERROR return FILE_TYPE;
```

Returns error output set at start of program execution.

```
function CURRENT_ERROR return FILE_TYPE;
```

Returns current error output, set by SET_ERROR.

5.2.5 Package CAIS_INTERACTIVE_IO

This package defines input and output facilities appropriate to files which are assigned to terminals.

The package provides for association of input and output text files with an output logging file. It also provides for turning on and off local echoing of input, association of a prompt string with terminal input, and simplistic random access within a terminal display.

Finally, this package defines a standard error-output text file which is used for error messages which are generated during program execution, but which would be missed if they were output to a re-directed standard output.

5.2.5.1 Package Specification

```
with CAIS_TEXT_IO;
with CAIS_NODE_DEFS;
package CAIS_INTERACTIVE_IO is

    subtype FILE_TYPE is CAIS_TEXT_IO.FILE_TYPE;

    type INTERACTIVE_TERMINAL is limited private;
```

```
    procedure ASSOCIATE (TERMINAL :    in out   INTERACTIVE_TERMINAL;
                         INFILE :      in       FILE_TYPE;
                         OUTFILE :     in       FILE_TYPE);

    procedure SET_LOG (TERMINAL :      in out   INTERACTIVE_TERMINAL;
                       LOG_FILE :      in       FILE_TYPE);

    function LOG (TERMINAL : in INTERACTIVE_TERMINAL)
            return FILE_TYPE;

    type CURSOR_POSITION is
        record
            LINE : POSITIVE;
            COLUMN : POSITIVE;
        end record;

    procedure SET_CURSOR (TERMINAL :   in out   INTERACTIVE_TERMINAL;
                          POSITION :   in       CURSOR_POSITION);

    function CURSOR (TERMINAL : in out INTERACTIVE_TERMINAL)
        return CURSOR_POSITION;

    function SIZE (TERMINAL : in out INTERACTIVE_TERMINAL)
        return CURSOR_POSITION;

    procedure UPDATE (TERMINAL : in out INTERACTIVE_TERMINAL);

    procedure SET_ECHO (TERMINAL :     in out   INTERACTIVE_TERMINAL;
                        TO :           in       BOOLEAN := TRUE);

    function ECHO (TERMINAL : in INTERACTIVE_TERMINAL) return BOOLEAN;

    procedure SET_PROMPT (TERMINAL :   in       INTERACTIVE_TERMINAL;
                          TO :         in       STRING);

    function PROMPT (TERMINAL : in INTERACTIVE_TERMINAL) return STRING;

    -- Exceptions

    LAYOUT_ERROR    : exception renames CAIS_NODE_DEFS.LAYOUT_ERROR;
    MODE_ERROR      : exception renames CAIS_NODE_DEFS.MODE_ERROR;
    STATUS_ERROR    : exception renames CAIS_NODE_DEFS.STATUS_ERROR;
    USE_ERROR       : exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_INTERACTIVE_IO;
```

5.2.5.2 Package Semantics

```
    procedure ASSOCIATE (TERMINAL :    in out   INTERACTIVE_TERMINAL;
                         INFILE :      in       FILE_TYPE;
                         OUTFILE :     in       FILE_TYPE);
```

Associates the INFILE (a file of mode IN_FILE) and the file OUTFILE (a file of mode OUT_FILE) with the TERMINAL. The exception MODE_ERROR is raised if the mode of INFILE is OUT_FILE or the mode of OUTFILE is IN_FILE. The exception STATUS_ERROR is raised if either INFILE or OUTFILE is not open.

```
procedure SET_LOG (TERMINAL :      in out    INTERACTIVE_TERMINAL;
                   LOG_FILE :      in        FILE_TYPE);
```

Sets LOG_FILE as the file on which the output log is written. When logging is active, all output is simultaneously provided to both the output file and the log file. Logging associations on the standard input and standard output text files are required to be preserved across program invocations. The exception MODE_ERROR is raised if the mode of LOG_FILE is IN_FILE. The exception STATUS_ERROR is raised if CAIS_TEXT_IO.IS_OPEN(LOG_FILE) returns FALSE.

```
function LOG (TERMINAL :      in        INTERACTIVE_TERMINAL)
                              return    FILE_TYPE;
```

Returns the current logging file associated with TERMINAL. The file handle returned is not open if not logging.

```
type CURSOR_POSITION is
    record
        LINE : POSITIVE;
        COLUMN : POSITIVE;
    end record;
```

CURSOR_POSITION identifies the line and column numbers of a terminal.

```
procedure SET_CURSOR (TERMINAL :   in out   INTERACTIVE_TERMINAL;
                      POSITION :   in       CURSOR_POSITION);
```

Moves the active position on the display to that specified by POSITION. The exception LAYOUT_ERROR is raised if the LINE or COLUMN number exceeds PAGE_LENGTH or LINE_LENGTH, respectively, when bounded. For a hard-copy terminal the exception USE_ERROR is raised if the LINE or COLUMN number is less than the current line or column number, respectively.

```
function CURSOR (TERMINAL :   in out      INTERACTIVE_TERMINAL)
                              return      CURSOR_POSITION;
```

Returns the current CURSOR_POSITION.

```
function SIZE (TERMINAL :     in out      INTERACTIVE_TERMINAL)
                              return      CURSOR_POSITION;
```

Returns the number of lines and number of columns on the terminal.

```
procedure UPDATE (TERMINAL :  in out      INTERACTIVE_TERMINAL);
```

Forces all data that has not already been output to the physical terminal to be output immediately.

```
procedure SET_ECHO (TERMINAL :   in out   INTERACTIVE_TERMINAL;
                    TO :         in       BOOLEAN := TRUE);
```

Turns on (TRUE) or off (FALSE) echoing for input file.

```
function ECHO (TERMINAL : in INTERACTIVE_TERMINAL) return BOOLEAN;
```

Indicates current state of echoing.

```
procedure SET_PROMPT (TERMINAL in      INTERACTIVE_TERMINAL;
                      TO :     in      STRING);
```

Sets prompting string for TERMINAL. All future requests for a line of input from TERMINAL will output prompt string first. The prompting string and any echoed input are also copied to the log file, if any.

```
function PROMPT (TERMINAL : in INTERACTIVE_TERMINAL) return STRING;
```

Returns current prompt string for input file.

5.3 PRAGMATICS

a. DIRECT_IO
CAIS_DIRECT_IO

Each element of a direct-access file is selected by an integer index of type COUNT. A conforming implementation must at least support a range of indices from one to 32767 (2**15-1).

b. SEQUENTIAL_IO
CAIS_SEQUENTIAL_IO
DIRECT_IO
CAIS_DIRECT_IO

A conforming implementation must support generic instantiation of these packages with any (non-limited) constrained Ada type whose maximum size in bits (as defined by the attribute ELEMENT_TYPE'SIZE) is at least 32767. A conforming implementation must also support instantiation with unconstrained record types which have default constraints and a maximum size in bits of at least 32767, and may (but need not) use variable length elements to conserve space in the external file.

c. TEXT_IO
CAIS_TEXT_IO

A conforming implementation must support files with at least 32767 records/lines in total and at least 32767 lines per page. A conforming implementation must support at least 255 columns per line.

6. CAIS_PROCESS_NODES

Each time an Ada program is invoked, a process node is created to represent the execution of the program. Even where the Ada program uses tasking, the execution of the program and its tasks is treated as a single CAIS **process.** This use of the term process does not preclude the CAIS implementation from devoting more than one host process or one physical processor to the execution of the single process.

The mechanism by which a user enters the APSE (e.g., logs on) is not defined as part of the CAIS. The facility to verify access rights to a system via user ID and password, for example, and to establish privileges and resource rights and quotas may be supported either by the APSE or its underlying implementation.

Each time a user enters the APSE a root process node is created dynamically at the top-level node of the user. This root process node initiates a tree of dependent processes. The primary relationship for the node of the root process emanates from the top-level node of the user. It has relation name "JOB" and a relationship key assigned by the APSE or underlying implementation of the APSE. This key is unique for each process node created by the user. In other words, the format 'USER (XXX)' JOB (YYY) is the absolute pathname of a job.

The root process node exists for the duration of the job's existence in the APSE. When the user's job terminates, the root process is terminated and the root process node is deleted.

A process may create other processes by invocation. This act of invocation creates both the node representing the process and the process itself. The new process is a child of the invoking process. The primary relationship of the nodes of these processes emanates from the invoking process with relation name "DOT" and a relationship key that is unique among nodes bearing the DOT relation with the invoker. The relationship key is an identifier assigned by the invoking process. By default, the 'CURRENT_NODE relationship of the new process is established to be the 'CURRENT_NODE of the invoking process.

A process is identified by providing a pathname to its process node (see CAIS Node Model, Section 3). List-valued attributes and secondary relationships for a process are established using the general node manipulation routines (see CAIS Node Model, Section 3).

Processes may communicate with each other using the techniques and procedures described in CAIS_PROCESS COMMUNICATION (see Section 6.3). The basic capability provides for sending and receiving messages over channels between processes, using a queueing model.

Processes may interrupt each other using the techniques and procedures described in CAIS_PROCESS_INTERRUPTS (see Section 6.5). This basic capability allows for signalling and responding to "pseudo-interrupts," using an asynchronous model for the delivery of the signal. The response to any pseudo-interrupt is definable by the Ada program before the delivery of the signal.

6.1 PACKAGE CAIS_PROCESS_DEFS

This package defines types and exceptions associated with process nodes.

6.1.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_PROCESS_DEFS is

    type PROCESS_STATUS is
        (READY, SUSPENDED, ABORTING, TERMINATING);
```

```
type COMPLETION_STATUS is (ABORTED, TERMINATED);

ROOT_PROCESS:    constant STRING := " 'CURRENT_JOB";
CURRENT_PROCESS: constant STRING := ".";

-- Exceptions

NAME_ERROR:      exception renames CAIS_NODE_DEF.NAME_ERROR;
USE_ERROR:       exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PROCESS_DEFS;
```

6.1.2 Package Semantics

```
type PROCESS_STATUS is
    (READY, SUSPENDED, ABORTING, TERMINATING);
```

The PROCESS_STATUS is the state a process is in when viewed from another process. Table 6-1 indicates the states and the events which will cause transition from one state to another. In the READY state a process is actually running or is waiting for resources.

TABLE 6-1
PROCESS STATE TABLE

STATE / OPERATION	READY	SUSPENDED	ABORTING	TERMINATING
TERMINATE	TERMINATING	TERMINATING	——	——
ABORT	ABORTING	ABORTING	——	ABORTING
SUSPEND	SUSPENDED	——	N/A	N/A
RESUME	——	READY	N/A	N/A

N/A: marks events that are not applicable to the state specified.
——: marks events that have no effect on the state.

Transition to a state as the result of an event is instantaneous with the occurrence of the event. As the state-transition diagram indicates, there is no transition from an ABORTING or TERMINATING state into any running state.

```
type COMPLETION_STATUS is (ABORTED, TERMINATED);
```

COMPLETION_STATUS is made available to an invoking process upon completion of a descendant process. These are representative states of a process, since at the time of their receipt the process may have already ceased to exist, depending upon the mechanism provided in the implementation underlying the CAIS for handling completed processes.

```
ROOT_PROCESS:    constant STRING: = " 'CURRENT_JOB";
CURRENT_PROCESS: constant STRING: = " . ";
```

ROOT_PROCESS and CURRENT_PROCESS are two strings defined to represent respectively the root process of the current job and the current process.

6.2 PACKAGE CAIS_PROCESS_CONTROL

This package provides support for the invocation of a program. A program can be invoked using the synchronous model, in which the calling task is suspended during the life of the dependent process and is resumed when the dependent process terminates, either normally or abnormally. A program can also be spawned using an asynchronous model, in which the calling task continues execution after the call which creates a dependent process.

6.2.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_PROCESS_DEFS;
with CAIS_TEXT_IO;
with CAIS_TEXT_UTILS;
package CAIS_PROCESS_CONTROL is

    subtype PROGRAM_STRING      is    STRING;
    subtype RESULTS_STRING      is    CAIS_TEXT_UTILS.TEXT;
    subtype PARAMS_STRING       is    CAIS_TEXT_UTILS.TEXT;
    subtype NAME_STRING         is    CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY    is    CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype COMPLETION_STATUS   is    CAIS_PROCESS_DEFS.COMPLETION_STATUS;
    subtype FILE_TYPE           is    CAIS_TEXT_IO.FILE_TYPE;
    subtype NODE_TYPE           is    CAIS_NODE_DEFS.NODE_TYPE;
    subtype PROCESS_STATUS      is    CAIS_PROCESS_DEFS.PROCESS_STATUS;

    UNIQUE_CHILD_KEY: STRING renames CAIS_NODE_DEFS.LATEST_KEY;

    procedure INVOKE_PROCESS (PROGRAM:    in      PROGRAM_STRING;
                              PARAMS:     in      PARAMS_STRING;
                              RESULTS:    in out  RESULTS_STRING;
                              STATUS:       out   COMPLETION_STATUS;
                              KEY:        in      RELATIONSHIP_KEY := UNIQUE_CHILD_KEY;
                              STD_IN:     in      FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_INPUT;
                              STD_OUT:    in      FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_OUTPUT;
                              STD_ERR:    in      FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_ERROR;
                              CURR_NODE:  in      NAME_STRING :=
                                          " 'CURRENT_NODE");

    procedure SPAWN_PROCESS (PROGRAM:     in      PROGRAM_STRING;
                             PARAMS:      in      PARAMS_STRING;
                             NODE:        in out  NODE_TYPE;
                             KEY:         in      RELATIONSHIP_KEY :=
                                          UNIQUE_CHILD_KEY;
                             STD_IN:      in      FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_INPUT;
                             STD_OUT:     in      FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_OUTPUT;
                             STD_ERR:     in      FILE_TYPE :=
                                          CAIS_TEXT_IO.CURRENT_ERROR;
                             CURR_NODE:   in      NAME_STRING :=
                                          " 'CURRENT_NODE");
```

```
    procedure AWAIT_PROCESS (PROCESS:    in out  NODE_TYPE;
                             RESULTS:    in out  RESULTS_STRING;
                             STATUS:        out  COMPLETION_STATUS;
                             LIMIT:      in      DURATION := DURATION'LAST);

    procedure GET_PARAMS (PARAMS:        in out  PARAMS_STRING);

    procedure RETURN_TERMINATED(RESULTS: in      RESULTS_STRING;

    procedure RETURN_ABORTED(RESULTS:    in      RESULTS_STRING);

    procedure ABORT_PROCESS (PROCESS:    in      NAME_STRING);
    procedure ABORT_PROCESS (NODE:       in      NODE_TYPE);

    procedure SUSPEND_PROCESS(PROCESS:   in      NAME_STRING);
    procedure SUSPEND_PROCESS(NODE:      in      NODE_TYPE);

    procedure RESUME_PROCESS (PROCESS:   in      NAME_STRING);
    procedure RESUME_PROCESS (NODE:      in      NODE_TYPE);

    function STATE_OF_PROCESS (PROCESS:  in      NAME_STRING) return PROCESS_STATUS;
    function STATE_OF_PROCESS (NODE:     in      NODE_TYPE) return PROCESS_STATUS;

    function JOB_INPUT return FILE_TYPE;
    function JOB_OUTPUT return FILE_TYPE;

    -- Exceptions

    USE_ERROR: exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PROCESS_CONTROL;
```

6.2.2 Package Semantics

```
    subtype PROGRAM_STRING      is  STRING;
    subtype RESULTS_STRING      is  CAIS_TEXT_UTILS.TEXT;
    subtype PARAMS_STRING       is  CAIS_TEXT_UTILS.TEXT;
    subtype NAME_STRING         is  CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY    is  CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype COMPLETION_STATUS   is  CAIS_PROCESS_DEFS.COMPLETION_STATUS;
    subtype FILE_TYPE           is  CAIS_TEXT_IO.FILE_TYPE;
    subtype NODE_TYPE           is  CAIS_NODE_DEFS.NODE_TYPE;
    subtype PROCESS_STATUS      is  CAIS_PROCESS_DEFS.PROCESS_STATUS;

    UNIQUE_CHILD_KEY: STRING renames CAIS_NODE_DEFS.LATEST_KEY;

    procedure INVOKE_PROCESS (PROGRAM:   in      PROGRAM_STRING;
                              PARAMS:    in      PARAMS_STRING;
                              RESULTS:   in out  RESULTS_STRING;
                              STATUS:       out  COMPLETION_STATUS;
                              KEY:       in      RELATIONSHIP_KEY:=
                                                 UNIQUE_CHILD_KEY;
                              STD_IN:    in      FILE_TYPE :=
                                                 CAIS_TEXT_IO.CURRENT_INPUT;
                              STD_OUT:   in      FILE_TYPE :=
                                                 CAIS_TEXT_IO.CURRENT_OUTPUT;
```

```
                    STD_ERR:     in     FILE_TYPE := CAIS_TEXT_IO.CURRENT_ERROR;
                    CURR_NODE:   in     NAME_STRING := " 'CURRENT_NODE");
```

Creates a new node and a new process and passes a list of parameters to the new process. The calling task can either supply the KEY or the CAIS implementation will assign a unique key via UNIQUE_CHILD_KEY. The calling task is suspended until the new process terminates or aborts. The results are returned as a list, along with an enumeration specifying the process's completion status. The node of the terminated process is automatically deleted upon termination.

```
procedure SPAWN_PROCESS (PROGRAM:    in      PROGRAM_STRING;
                         PARAMS:     in      PARAMS_STRING;
                         NODE:       in out  NODE_TYPE;
                         KEY:        in      RELATIONSHIP_KEY := UNIQUE_CHILD_KEY;
                         STD_IN:     in      FILE_TYPE := CAIS_TEXT_IO.CURRENT_INPUT;
                         STD_OUT:    in      FILE_TYPE := CAIS_TEXT_IO.CURRENT_OUTPUT;
                         STD_ERR:    in      FILE_TYPE := CAIS_TEXT_IO.CURRENT_ERROR;
                         CURR_NODE:  in      NAME_STRING := " 'CURRENT_NODE");
```

Results in a new node and a new process being created to represent the execution of the specified program. Control returns to the invoking process. This invocation provides no technique for coordination of the new process with its parent, except that termination of the parent will not be completed until all children are terminated or aborted. Similarly, no technique is provided for returning a result string to the invoking process. Communication between parent and child can be provided using the techniques provided in CAIS_PROCESS_COMMUNICATION.

```
procedure AWAIT_PROCESS (PROCESS: in out  NODE_TYPE;
                         RESULTS: in out  RESULTS_STRING;
                         STATUS:     out  COMPLETION_STATUS;
                         LIMIT:      in   DURATION := DURATION'LAST);
```

Suspend the calling task and wait for the process created by SPAWN_PROCESS to complete. The USE_ERROR exception is generated if this is not the first attempt to wait for this descendant process. The result parameter and COMPLETION_STATUS are provided by spawned process's return, even if the process completes execution before the call is made. A time limit is provided in which the parameters must be received or a TIME_OUT exception is raised.

```
procedure GET_PARAMS(PARAMS: in out  PARAMS_STRING);
```

Retrieve the parameters passed to a process by its caller.

```
procedure RETURN_TERMINATED (RESULTS: in  RESULTS_STRING);
```

Await termination of all descendant processess, and then return the specified result parameter to the calling process. The COMPLETION_STATUS will be TERMINATED.

```
procedure RETURN_ABORTED (RESULTS: in  RESULTS_STRING);
```

Abort the current process (and all of its descendant processes) and then return the specified result parameter to the calling process. The COMPLETION_STATUS will be ABORTED.

```
procedure ABORT_PROCESS (PROCESS:  in  NAME_STRING);
procedure ABORT_PROCESS (NODE:     in  NODE_TYPE);
```

Aborts the specified process and recursively forces any descendants of the named process to be aborted. The sequencing of the process abortions is not specified. ABORT_PROCESS returns control to the issuing process immediately. At that time, if the state of the aborted process is examined, it will be either ABORTING or the process will be nonexistent. This node associated with the aborted process remains until explicitly deleted by the invoking process.

The COMPLETION_STATUS of the process will be ABORTED. ABORT_PROCESS can be used by a process to abort itself.

```
procedure SUSPEND_PROCESS (PROCESS:  in  NAME_STRING);
procedure SUSPEND_PROCESS (NODE:     in  NODE_TYPE);
procedure RESUME_PROCESS (PROCESS:   in  NAME_STRING);
procedure RESUME_PROCESS (NODE:      in  NODE_TYPE);
```

Suspends or resumes the designated process. SUSPEND_PROCESS can include suspension of the requesting process. While a process is suspended, the PROCESS_STATUS is SUSPENDED. RESUME causes an immediate change to the READY state. Similarly, the transition to SUSPENDED state takes place immediately.

```
function STATE_OF_PROCESS (PROCESS:  in  NAME_STRING) return PROCESS_STATUS;
function STATE_OF_PROCESS (NODE:     in  NODE_TYPE) return PROCESS_STATUS;
```

Returns the current state of the specified process. The PROCESS_STATUS of a process issuing that function will always be READY.

```
function JOB_INPUT return FILE_TYPE;

function JOB_OUTPUT return FILE_TYPE;
```

Returns the standard input or output defined at the initiation of the root process of the job. In general, these files will refer to the interactive terminal or batch input or output files, even if the current input or output file for this process has been re-directed to a different file.

6.3 PACKAGE CAIS_PROCESS_COMMUNICATION

CAIS_PROCESS_COMMUNICATION provides techniques for a process to communicate with another process or itself.

A process may send and receive inter-process messages on a number of named channels. The channels are identified by a character string with the syntax of an Ada identifier.

It is anticipated that certain channel names will eventually have standard meanings with CAIS. Each implementation must identify those channel names which have special significance.

6.3.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_PROCESS_DEFS;
with CAIS_TEXT_UTILS;
package CAIS_PROCESS_COMMUNICATION is

    subtype NODE_TYPE         is  CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING       is  CAIS_NODE_DEFS.NAME_STRING;
    subtype CHANNEL_STRING    is  STRING;
    subtype MESSAGE_TEXT      is  CAIS_TEXT_UTILS.TEXT;
```

```
    procedure SEND (PROCESS :  in      NAME_STRING;
                    CHANNEL :  in      CHANNEL_STRING;
                    MESSAGE :  in      MESSAGE_TEXT;
                    LIMIT :    in      DURATION := DURATION'LAST);
    procedure SEND(NODE :      in out  NODE_TYPE;
                    CHANNEL :  in      CHANNEL_STRING;
                    MESSAGE :  in      MESSAGE_TEXT;
                    LIMIT :    in      DURATION := DURATION'LAST);

    procedure RECEIVE(SENDER :   in out  NODE_TYPE;
                      CHANNEL :  in      STRING;
                      MESSAGE :  in out  MESSAGE_TEXT;
                      LIMIT :    in      DURATION := DURATION'LAST);

    -- Exceptions

    TIME_OUT: exception;

private
    -- implementation-dependent
end CAIS_PROCESS_COMMUNICATION;
```

6.3.2 Package Semantics

```
    subtype NODE_TYPE      is CAIS_NODE_DEFS.NODE_TYPE;
    subtype NAME_STRING    is CAIS_NODE_DEFS.NAME_STRING;
    subtype CHANNEL_STRING is STRING;
```

Provides logical name of a communication channel between communicating processes. The name is determined by mutual agreement.

```
    subtype MESSAGE_TEXT is CAIS_TEXT_UTILS.TEXT;
```

The message being sent.

```
    procedure SEND(PROCESS :  in      NAME_STRING;
                   CHANNEL :  in      CHANNEL_STRING;
                   MESSAGE :  in      MESSAGE_TEXT;
                   LIMIT :    in      DURATION := DURATION'LAST);
    procedure SEND(NODE :     in out  NODE_TYPE;
                   CHANNEL:   in      CHANNEL_STRING;
                   MESSAGE:   in      MESSAGE_TEXT;
                   LIMIT:     in      DURATION := DURATION'LAST);
```

Attempts to queue up the specified MESSAGE (text) for the designated process with the specified logical CHANNEL name. If the queue is full, the calling task will be suspended up to the time LIMIT specified, after which a TIME_OUT exception is raised in the calling process. As soon as there is room for the MESSAGE, it is queued and SEND returns. It is the responsibility of the two processes to insure that whatever additional coordination required is done.

```
    procedure RECEIVE(SENDER :   in out  NODE_TYPE;
                      CHANNEL :  in      CHANNEL_STRING;
                      MESSAGE :  in out  MESSAGE_TEXT;
                      LIMIT :    in      DURATION := DURATION'LAST);
```

Suspends the calling task until a message is available on the specified CHANNEL or the time LIMIT is reached. Multiple queued messages are received in a first-in first-out order. The capacity of the queue for a particular channel name is implementation dependent. However, before the first RECEIVE is done by a process on a particular channel name, the capacity of the queue is defined to be zero, and any SENDers will be delayed because the queue is by definition already "full." The sending process is identified by an open node handle on the process node.

6.4 PACKAGE CAIS_PROCESS_ANALYSIS

This package provides standardized debugging capabilities for processes within the CAIS implementation.

6.4.1 Package Specification

```
with CAIS_PROCESS_DEFS;
package CAIS_PROCESS_ANALYSIS is
{TBD}
end CAIS_PROCESS_ANALYSIS;
```

6.5 PACKAGE CAIS_PROCESS_INTERRUPTS

This package provides support for **pseudo-interrupts,** asynchronous signal sent between processes. Each interrupt is identified by a string with the syntax of an Ada identifier. When an interrupt is generated, the receiving process may respond by ignoring it, aborting execution, waking up a suspended task, or simply putting it on HOLD.

It is anticipated that the CAIS will define standard interrupt names, as well as standard default interrupt responses associated with each standard interrupt, in effect prior to an explicit SET_RESPONSE. The most likely default responses are ABORT for certain serious interrupts and IGNORE for all others.

Note that the predefined Ada language mechanism for associating interrupts with tasks is not being used here, so as to remain independent of any compiler implementation of this feature.

6.5.1 Package Specification

```
with CAIS_PROCESS_DEFS;
package CAIS_PROCESS_INTERRUPTS is

    subtype NODE_TYPE is CAIS_PROCESS_DEFS.NODE_TYPE;
    subtype NAME_STRING is CAIS_PROCESS_DEFS.NAME_STRING;

    subtype INTERRUPT_NAME is STRING;

    type INTERRUPT_RESPONSE is (IGNORE, ABORT, AWAKE, HOLD);

    procedure SIGNAL (PROCESS:   in   NAME_STRING,
                      INTERRUPT: in   INTERRUPT_NAME);
    procedure SIGNAL (PROCESS:   in   NODE_TYPE;
                      INTERRUPT: in   INTERRUPT_NAME);

    procedure SET_RESPONSE(INTERRUPT:  in   INTERRUPT_NAME;
                           RESPONSE:   in   INTERRUPT_RESPONSE;
                           TIME_LIMIT: in   DURATION := DURATION'LAST);

    function RESPONSE (INTERRUPT: in   INTERRUPT_NAME)
                       return INTERRUPT_RESPONSE;
```

```
    -- Exceptions

    USE_ERROR: exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PROCESS_INTERRUPTS;
```

6.5.2 Package Semantics

```
    subtype NODE_TYPE      is CAIS_PROCESS_DEFS.NODE_TYPE;
    subtype NAME_STRING    is CAIS_PROCESS_DEFS.NAME_STRING;
    subtype INTERRUPT_NAME is STRING;
```

Typical interrupt names would be "BREAK", "HANG_UP", etc.

```
    type INTERRUPT_RESPONSE is (IGNORE, ABORT, AWAKE, HOLD);
```

This enumeration specifies the possible responses associated with an interrupt. Each interrupt has exactly one of these responses associated with it at any one time. If the response is AWAKE, then some task has executed a SET_RESPONSE (INTERRUPT_NAME, AWAKE, TIME_LIMIT) and is still suspended awaiting the interrupt signal.

```
    procedure SIGNAL (PROCESS:    in    NAME_STRING;
                      INTERRUPT:  in    INTERRUPT_NAME);
    procedure SIGNAL (PROCESS:    in    NODE_TYPE;
                      INTERRRUPT  in    INTERRUPT_NAME);
```

Generates the designated pseudo-interrupt in the named process. This call always returns immediately, even if the associated response in the receiving process is HOLD.

```
    procedure SET_RESPONSE (INTERRUPT:  in    INTERRUPT_NAME;
                            RESPONSE:   in    INTERRUPT_RESPONSE;
                            TIME_LIMIT: in    DURATION := DURATION'LAST);
```

Handles a designated pseudo-interrupt according to the designated response. If the previously set response were HOLD, and the interrupt had already occurred at least once, then the newly specified response is immediately enacted. The USE_ERROR is raised if an attempt is made to SET_RESPONSE when some other task is still suspended with the response AWAKE. In all other cases, the new response supercedes any previous default or explicitly set response.

If the response is AWAKE, then the calling task is suspended until the interrupt is received or until the time limit expires (in which case the TIME_OUT exception is raised). When setting the response to AWAKE, the previously set response is remembered, and again becomes the current response after the task is awoken due either to an interrupt or to a time-out.

```
    function RESPONSE (INTERRUPT  in    INTERRUPT_NAME)
                       return INTERRUPT_RESPONSE;
```

Indicates the current response associated with the designated interrupt for the current process. If the response is AWAKE, then some other task of the current process is suspended awaiting the interrupt.

6.6 PRAGMATICS

a. Channels — A conforming implementation must support channel names of up to 20 characters. A conforming implementation must support up to 20 simultaneous accepting channels from the same process.

7. CAIS Device Nodes

This area provides basic device input/output support, along with special device control facilities. A device must first be made accessible to an Ada program by an OPEN, specifying the external device by a NAME and a FORM, both character strings. When opening device node handles, the NAME and FORM string formats are required to be the same and refer to the same external devices in both file node usage and in the device node packages. The collection of packages in this section are defined with careful consideration of standards established for information interchange by the American National Standards Institute [ANSI77] and [ANSI79]. The interfaces are also defined with consideration for existing interactive terminals that do not conform to the ANSI standards.

7.1 VIRTUAL TERMINALS

There are three primary classes of character-imaging terminals in use today: scroll, page, and form. Four packages are provided in this section, one package for the common terminal support functions and one package for each of the three classes of terminals supported.

7.1.1 Package CAIS_TERMINAL_SUPPORT

This package provides the routines that are common to scroll, page, and form terminals.

7.1.1.1 Package Specification

```
with CAIS_NODE_DEFS;
package CAIS_TERMINAL_SUPPORT is

    type TERMINAL_TYPE is limited private;

    subtype FORM_STRING          is CAIS_NODE_DEFS.FORM_STRING;
    subtype NAME_STRING          is CAIS_NODE_DEFS.NAME_STRING;
    subtype RELATIONSHIP_KEY     is CAIS_NODE_DEFS.RELATIONSHIP_KEY;
    subtype RELATION_NAME        is CAIS_NODE_DEFS.RELATION_NAME;

    type TERMINAL_CLASS is (SCROLL, PAGE, FORM);

    procedure CREATE (TERMINAL:  in out  TERMINAL_TYPE;
                      CLASS:     in      TERMINAL_CLASS := SCROLL;
                      NAME:      in      NAME_STRING;
                      FORM:      in      FORM_STRING := " ");
    procedure CREATE (TERMINAL:  in out  TERMINAL_TYPE;
                      CLASS:     in      TERMINAL_CLASS := SCROLL;
                      BASE:      in      NODE_TYPE;
                      KEY:       in      RELATIONSHIP_KEY := " ";
                      RELATION:  in      RELATION_NAME: = "DOT";
                      FORM:      in      FORM_STRING := " ");

    procedure OPEN (TERMINAL :   in out  TERMINAL_TYPE;
                    CLASS :      in      TERMINAL_CLASS := SCROLL;
                    NAME :       in      NAME_STRING;
                    FORM :       in      FORM_STRING := " ");
```

```
        procedure OPEN (TERMINAL:    in out   TERMINAL_TYPE;
                        CLASS:       in       TERMINAL_CLASS := SCROLL:
                        BASE:        in       NODE_TYPE;
                        KEY:         in       RELATIONSHIP_KEY := " ";
                        RELATION:    in       RELATION_NAME: = "DOT";
                        FORM:        in       FORM_STRING := " ");
        procedure OPEN (TERMINAL:    in out   TERMINAL_TYPE;
                        CLASS:       in       TERMINAL_CLASS := SCROLL;
                        NODE:        in       NODE_TYPE;
                        FORM:        in       FORM_STRING := " ");

        procedure CLOSE (TERMINAL :  in out   TERMINAL_TYPE);

        procedure DELETE (TERMINAL : in out   TERMINAL_TYPE);

        procedure RESET (TERMINAL :  in out   TERMINAL_TYPE;
                         CLASS :     in       TERMINAL._CLASS);
        procedure RESET (TERMINAL :  in out   TERMINAL_TYPE);

        function CLASS(TERMINAL : in TERMINAL_TYPE) return TERMINAL_CLASS;
        function NAME (TERMINAL : in TERMINAL_TYPE) return NAME_STRING;
        function FORM (TERMINAL : in TERMINAL_TYPE) return FORM_STRING;


        function IS_OPEN (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

        type ACTIVE_POSITION is
            record
                LINE : POSITIVE;
                COLUMN : POSITIVE;
            end record;

        procedure SET_POSITION (TERMINAL :  in out   TERMINAL_TYPE;
                                POSITION :  in       ACTIVE_POSITION);

        function POSITION (TERMINAL : in TERMINAL_TYPE)
            return ACTIVE_POSITION;

        function SIZE (TERMINAL : in TERMINAL_TYPE)
            return ACTIVE_POSITION;

        -- Exceptions

        CLASS_ERROR :    exception;
        NAME_ERROR :     exception renames CAIS _NODE_DEFS.NAME_ERROR;
        USE_ERROR :      exception renames CAIS_NODE_DEFS.USE_ERROR;
        STATUS_ERROR :   exception renames CAIS_NODE_DEFS.STATUS_ERROR;

    private
        -- implementation-dependent
    end CAIS_TERMINAL_SUPPORT;
```

7.1.1.2 Package Semantics

```
        type TERMINAL_CLASS is (SCROLL, PAGE, FORM);
```

Indicates the different classes of terminals that are supported.

```
procedure CREATE (TERMINAL:  in out   TERMINAL_TYPE;
                  CLASS:     in       TERMINAL_CLASS := SCROLL;
                  NAME:      in       NAME_STRING;
                  FORM:      in       FORM_STRING := " ");
```

Creates an extenal termnal (and its device node) that is associated with the given terminal. The given terminal is left open. A null string for the FORM specifies default options of the implementation.

The exception STATUS_ERROR is raised if the given terminal is already open. The exception NAME_ERROR is raised if the NAME does not identify an external logical terminal.

```
procedure CREATE (TERMINAL:  in out   TERMINAL_TYPE;
                  CLASS:     in       TERMINAL_CLASS := SCROLL;
                  BASE:      in       NODE_TYPE;
                  KEY:       in       RELATIONSHIP_KEY := " ";
                  RELATION:  in       RELATION_NAME: = "DOT";
                  FORM:      in       FORM_STRING := " ");
```

The semantics are the same as above except that the terminal is identified by means of BASE/KEY/RELATION.

```
procedure OPEN (TERMINAL :  in out  TERMINAL_TYPE;
                CLASS :     in      TERMINAL_CLASS := SCROLL;
                NAME :      in      NAME_STRING;
                FORM :      in      FORM_STRING := " ");
```

Associates the given terminal handle with a terminal having the given name and form and sets the current class of the terminal handle to the given class.

The exception NAME_ERROR is raised if the string given as NAME does not identify a terminal. The exception USE_ERROR is raised if the terminal identified by NAME cannot be opened in the given class or form.

```
procedure OPEN (TERMINAL:  in out   TERMINAL_TYPE;
                CLASS:     in       TERMINAL_CLASS := SCROLL;
                BASE:      in       NODE_TYPE;
                KEY:       in       RELATIONSHIP_KEY := " ";
                RELATION:  in       RELATION_NAME: = "DOT";
                FORM:      in       FORM_STRING := " ");
procedure OPEN (TERMINAL:  in out   TERMINAL_TYPE;
                CLASS:     in       TERMINAL_CLASS := SCROLL;
                NODE:      in       NODE_TYPE;
                FORM:      in       FORM_STRING := " ");
```

The semantics are the same as above except that the terminal is identified by means of the associated node or BASE/KEY/RELATION.

```
procedure CLOSE (TERMINAL :  in out   TERMINAL_TYPE);
```

Severs the association between the terminal handle and its associated terminal.

```
procedure DELETE (TERMINAL:  in out   TERMINAL_TYPE);
```

Deletes the external terminal (and its device node) associated with the given terminal. The given terminal is closed, and the external logical terminal ceases to exist.

The exception STATUS_ERROR is raised if the given terminal is not open. The exception USE_ERROR is raised if deletion of the external logical terminal is not allowed by the caller.

```
procedure RESET (TERMINAL :  in out   TERMINAL_TYPE;
                 CLASS :     in       TERMINAL_CLASS);
procedure RESET (TERMINAL :  in out   TERMINAL_TYPE);
```

Changes the terminal handle to the given class and/or resets the terminal handle to its initial state.

```
function CLASS(TERMINAL : in TERMINAL_TYPE) return TERMINAL_CLASS;
```

Returns the class of the node associated with the given terminal handle.

```
function NAME (TERMINAL : in TERMINAL_TYPE) return NAME_STRING;
```

Returns the name of the node associated with the given terminal handle.

```
function FORM (TERMINAL : in TERMINAL_TYPE) return FORM_STRING;
```

Returns the form associated with the given terminal handle.

```
function IS_OPEN (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns TRUE if the given terminal handle is associated with a logical terminal, otherwise returns FALSE.

```
type ACTIVE_POSITION is
    record
        LINE : POSITIVE;
        COLUMN : POSITIVE;
    end record;
```

The ACTIVE_POSITION indicates the row and column position on the display of a terminal at which the next operation may occur.

```
procedure SET_POSITION (TERMINAL :  in out   TERMINAL_TYPE;
                        POSITION :  in       ACTIVE_POSITION);
```

Moves the active position to the specified POSITION on the display of the given terminal.

```
function POSITION (TERMINAL : in TERMINAL_TYPE)
    return ACTIVE_POSITION;
```

Returns the POSITION of the active position on the given terminal.

```
function SIZE (TERMINAL : in TERMINAL_TYPE)
    return ACTIVE_POSITION;
```

Returns the maximum line and maximum column of the given terminal.

7.1.2 Package CAIS_SCROLL_TERMINAL

This package provides the functionality of a common "teleprinter" type terminal. It is capable of a minimal set of operations. Characters are transmitted between a program and the terminal a character or a line at a time. This type of terminal is typically configured to echo each character as it is entered at the keyboard (before transmission to the computer or intervening communications equipment).

7.1.2.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_TERMINAL_SUPPORT;
package CAIS_SCROLL_TERMINAL is

    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    procedure SET_TAB (TERMINAL :       in out    TERMINAL_TYPE);

    procedure CLEAR_TAB (TERMINAL :     in out    TERMINAL_TYPE);

    procedure TAB (TERMINAL :           in out    TERMINAL_TYPE;
                   COUNT :              in        POSITIVE);

    procedure NEW_LINE (TERMINAL :      in out    TERMINAL_TYPE);

    procedure NEW_PAGE (TERMINAL :      in out    TERMINAL_TYPE);

    procedure PUT (TERMINAL :           in out    TERMINAL_TYPE;
                   ITEM :               in        CHARACTER);
    procedure PUT (TERMINAL :           in out    TERMINAL_TYPE;
                   ITEM :               in        STRING);

    procedure UPDATE (TERMINAL :        in out    TERMINAL_TYPE);

    procedure GET (TERMINAL :           in out    TERMINAL_TYPE;
                   ITEM :                  out    CHARACTER);
    procedure GET (TERMINAL :           in out    TERMINAL_TYPE;
                   ITEM :                  out    STRING);
    procedure GET (TERMINAL :           in out    TERMINAL_TYPE;
                   ITEM :                  out    STRING;
                   LAST :                  out    NATURAL);

    procedure SET_ECHO (TERMINAL :      in        TERMINAL_TYPE;
                        TO :            in        BOOLEAN := TRUE);

    function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

    procedure BELL (TERMINAL :          in out    TERMINAL_TYPE);

    -- Exceptions

    CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_SCROLL_TERMINAL;
```

7.1.2.2 Package Semantics

```
    procedure SET_TAB (TERMINAL :   In out      TERMINAL_TYPE);
```

Creates a horizontal tab stop at the active position (used by TAB).

```
    procedure CLEAR_TAB (TERMINAL :     in out    TERMINAL_TYPE);
```

Deletes a horizontal tab stop at the active position. The exception USE_ERROR is raised if a horizontal tab stop does not exist at the active position.

```
procedure TAB (TERMINAL :    in out    TERMINAL_TYPE;
               COUNT :       in        POSITIVE);
```

Moves the active position the specified number of horizontal tab stops. The exception USE_ERROR is raised if there are fewer than COUNT tab stops on the active line.

```
procedure NEW_LINE (TERMINAL :    in out    TERMINAL_TYPE);
```

Moves the active position to the first column of the next line. The display scrolls upward if entered on the last line of the display.

```
procedure NEW_PAGE (TERMINAL :    in out    TERMINAL_TYPE);
```

Moves the active position to the first column of the first line of a new page.

```
procedure PUT (TERMINAL :    in out    TERMINAL_TYPE;
               ITEM :        in        CHARACTER);
```

Writes a single character to the display and advances the active position. If the active position is at the last column on a line, a NEW_LINE operation is performed after writing the character.

```
procedure PUT (TERMINAL :    in out    TERMINAL_TYPE;
               ITEM :        in        STRING);
```

Writes a character at a time in the same manner as PUT of a character, writing each character in the given string successively.

```
procedure UPDATE (TERMINAL :    in out    TERMINAL_TYPE);
```

Forces all data that has not already been transmitted to the terminal to be transferred.

```
procedure GET (TERMINAL :    in out    TERMINAL_TYPE;
               ITEM :        out       CHARACTER);
```

Reads a single (unedited) character from the terminal keyboard.

```
procedure GET (TERMINAL :    in out    TERMINAL_TYPE;
               ITEM :        out       STRING);
```

Reads ITEM'LENGTH (unedited) characters from the terminal keyboard into ITEM.

```
procedure GET (TERMINAL :    in out    TERMINAL_TYPE;
               ITEM :        out       STRING;
               LAST :        out       NATURAL);
```

Successively reads (unedited) characters from the terminal keyboard into ITEM, until either all positions of ITEM are filled or there are no more characters buffered for the terminal. Upon completion LAST contains the index of the last position in ITEM to contain a character that has been read.

```
procedure SET_ECHO (TERMINAL :    in        TERMINAL_TYPE;
                    TO :          in        BOOLEAN := TRUE);
```

When TO is given as TRUE, each character entered at the keyboard is echoed to the display.

```
function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns whether echo is enabled (TRUE) or disabled (FALSE).

```
procedure BELL (TERMINAL :  in out      TERMINAL_TYPE);
```

Activates the bell (beeper) on the terminal.

```
-- Exceptions

CLASS_ERROR :   exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
USE_ERROR :     exception renames CAIS_NODE_DEFS.USE_ERROR;
```

The exception CLASS_ERROR is raised if any of the operations in the package CAIS_SCROLL_TERMINAL are invoked with a TERMINAL which is not OPENed or RESET with class SCROLL.

7.1.3 Package CAIS_PAGE_TERMINAL

This package provides the functionality of a page terminal. A page terminal is commonly referred to as a character-oriented or interactive terminal. This terminal may have many types of format effectors, cursor controls, and local (built-in) editing functions. Typical controls for page terminals are to position the cursor, to erase within a line or display area, to insert into or delete from a line, to insert or delete entire lines, to scroll up, and to select graphic rendition for subsequent output characters (intensity, reverse-image, blink, underscore, etc.). The terminal may be configured to echo before transmission to the computer (or intervening equipment) or not to echo at all. Each character is transmitted to the computer as it is entered at the keyboard. Except when locally echoed, the control action implied by the character keyed is deferred until (and if) the computer (or communications equipment) echoes the character. (This allows some programs, operating with non-echoing terminals, to reinterpret the meanings of control characters keyed by not directly echoing these characters. A number of popular text editors operate this way.)

7.1.3.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_TERMINAL_SUPPORT;
package CAIS_PAGE_TERMINAL is

    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    procedure SET_TAB (TERMINAL :                   in out      TERMINAL_TYPE);

    procedure CLEAR_TAB (TERMINAL :                 in out      TERMINAL_TYPE);

    procedure TAB (TERMINAL :                       in out      TERMINAL_TYPE;
                   COUNT :                          in          POSITIVE);

    procedure BELL (TERMINAL :                      in out      TERMINAL_TYPE);

    procedure DELETE_CHARACTER (TERMINAL :          in out      TERMINAL_TYPE;
                                COUNT :             in          POSITIVE);

    procedure DELETE_LINE (TERMINAL :               in out      TERMINAL_TYPE;
                           COUNT :                  in          POSITIVE);

    function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

    procedure ERASE_CHARACTER (TERMINAL :           in out      TERMINAL_TYPE;
                               COUNT :              in          POSITIVE);

    type SELECT_ENUM is
        (FROM_ACTIVE_POSITION_TO_END,
```

```
        FROM_START_TO_ACTIVE_POSITION,
        ALL_POSITIONS);

    procedure ERASE_IN_DISPLAY (TERMINAL :    in out    TERMINAL_TYPE;
                                SELECTION :   in        SELECT_ENUM);

    procedure ERASE_IN_LINE (TERMINAL :    in out    TERMINAL_TYPE;
                             SELECTION :   in        SELECT_ENUM);

    procedure GET (TERMINAL : in out    TERMINAL_TYPE;
                   ITEM :     out       CHARACTER);
    procedure GET (TERMINAL : in out    TERMINAL_TYPE;
                   ITEM :     out       STRING;
    procedure GET (TERMINAL : in out    TERMINAL_TYPE;
                   ITEM :     out       STRING;
                   LAST :     out       NATURAL);

    procedure INSERT_CHARACTER (TERMINAL : in out    TERMINAL_TYPE;
                                COUNT :    in        POSITIVE);

    procedure INSERT_LINE (TERMINAL :   in out    TERMINAL_TYPE;
                           COUNT :      in        POSITIVE);

    procedure PUT (TERMINAL :   in out    TERMINAL_TYPE;
                   ITEM :       in        CHARACTER);
    procedure PUT (TERMINAL :   in out    TERMINAL_TYPE;
                   ITEM :       in        STRING);

    type GRAPHIC_RENDITION_ENUM is
        (PRIMARY_RENDITION,
         BOLD,
         FAINT,
         UNDERSCORE,
         SLOW_BLINK,
         RAPID_BLINK,
         REVERSE_IMAGE)

    procedure SELECT_GRAPHIC_RENDITION (TERMINAL : in out  TERMINAL_TYPE;
                                        SELECTION : in     GRAPHIC_RENDITION_ENUM);

    procedure SET_ECHO (TERMINAL :          in out  TERMINAL_TYPE;
                        TO :                in      BOOLEAN := TRUE);

    procedure UPDATE (TERMINAL : in out    TERMINAL_TYPE);

    -- Exceptions

    CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_PAGE_TERMINAL;
```

7.1.3.2 Package Semantics

```
    procedure SET_TAB (TERMINAL :    in out    TERMINAL_TYPE);
```

Creates a horizontal tab stop at the active position.

```
procedure CLEAR_TAB (TERMINAL : in out        TERMINAL_TYPE);
```

Deletes a horizontal tab stop at the active position. The exception USE_ERROR is raised if a horizontal tab stop does not exist at the active position.

```
procedure TAB (TERMINAL : in out        TERMINAL_TYPE;
               COUNT :    in            POSITIVE);
```

Moves the active position the specified number of horizontal tab stops. The exception USE_ERROR is raised if there are fewer than COUNT tab stops on the active line.

```
procedure BELL (TERMINAL : in out        TERMINAL_TYPE);
```

Activates the bell (beeper) on the terminal.

```
procedure DELETE_CHARACTER (TERMINAL : in out        TERMINAL_TYPE;
                            COUNT :    in            POSITIVE);
```

Deletes the given number of characters on the active line starting at the active position. Adjacent characters to the right of the active position are shifted left. Open space on the right is filled with SPACE characters. The active position is not changed.

```
procedure DELETE_LINE (TERMINAL : in out        TERMINAL_TYPE;
                       COUNT :    in            POSITIVE);
```

Deletes the given number of lines starting at the active line. Adjacent lines are shifted from the bottom toward the active line. COUNT lines from the bottom of the display are cleared. The active position is not changed.

```
function ECHO (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns whether echo is enabled (TRUE) or disabled (FALSE).

```
procedure ERASE_CHARACTER (TERMINAL : in out        TERMINAL_TYPE;
                           COUNT :    in            POSITIVE);
```

Replaces the given number of characters on the active line with SPACE characters starting at the active position. The active position is not changed. The exception USE_ERROR is raised if COUNT is greater than SIZE(TERMINAL).COLUMN_POSITION(TERMINAL).COLUMN.

```
type SELECT_ENUM is
     (FROM_ACTIVE_POSITION_TO_END,
      FROM_START_TO_ACTIVE_POSITION,
      ALL_POSITIONS);

procedure ERASE_IN_DISPLAY (TERMINAL : in out        TERMINAL_TYPE;
                            SELECTION : in           SELECT_ENUM);
```

Erases the characters in the entire display as determined by the active position and the given SELECTION (include the active position). The active position is not changed.

```
procedure ERASE_IN_LINE (TERMINAL : in out        TERMINAL_TYPE;
                         SELECTION : in           SELECT_ENUM);
```

Erases the characters in the active line as determined by the active position and the given SELECTION (include the active position). The active position is not changed.

```
procedure GET (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :      out     CHARACTER);
```

Reads a single (unedited) character from the terminal keyboard.

```
procedure GET (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :      out     STRING);
```

Reads ITEM'LENGTH (unedited) characters from the terminal keyboard into ITEM.

```
procedure GET (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :      out     STRING;
               LAST :      out     NATURAL);
```

Successively reads (unedited) characters from the terminal keyboard into ITEM, until either all positions of ITEM are filled or there are no more characters buffered for the terminal. Upon completion LAST contains the index of the last position in ITEM to contain a character that has been read.

```
procedure INSERT_CHARACTER (TERMINAL :  in out  TERMINAL_TYPE;
                            COUNT :     in      POSITIVE);
```

Inserts COUNT SPACE characters into the active line at the active position. Adjacent characters are shifted to the right. The rightmost characters on the line may be lost. The active position is advanced to the right one character position.

```
procedure INSERT_LINE (TERMINAL :  in out  TERMINAL_TYPE;
                       COUNT :     in      POSITIVE);
```

Inserts COUNT blank lines into the display at the active line. The lines at and below the top of the display are lost. The active position remains unchanged.

```
procedure PUT (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :      in      CHARACTER);
```

Writes a single character at the active position. Advances the active position to the next column. If the character is written to the last character position on a line, advances the active position to the first column of the next line. If the character is written to the last character position of the last line, inserts a line at the bottom of the display and moves the active position to the first column of the last line.

```
procedure PUT (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :      in      STRING);
```

Writes each character of the given string according to the semantics for PUT with ITEM as a single character.

```
type GRAPHIC_RENDITION_ENUM is
     (PRIMARY_RENDITION,
      BOLD,
      FAINT,
      UNDERSCORE,
      SLOW_BLINK,
      RAPID_BLINK,
      REVERSE_IMAGE);

procedure SELECT_GRAPHIC_RENDITION (TERMINAL:  in out  TERMINAL_TYPE;
                                    SELECTION: in      GRAPHIC_RENDITION_ENUM);
```

Sets the graphic rendition for subsequent characters to be PUT. If the graphic rendition specified is not supported by the terminal, the primary rendition is used. The exception USE_ERROR is raised if the specified graphic rendition is not supported.

```
procedure SET_ECHO(TERMINAL :    in out      TERMINAL_TYPE;
                   TO :          in          BOOLEAN := TRUE);
```

Turns on (TRUE) or off (FALSE) echoing for input file.

```
procedure UPDATE (TERMINAL : in out      TERMINAL_TYPE);
```

Forces all data that has not already been transmitted to the terminal to be transmitted.

```
-- Exceptions

CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;
```

The exception CLASS_ERROR is raised if any of the routines in the package CAIS_PAGE_TERMINAL are invoked with a terminal handle which is not OPENed or RESET with class PAGE.

7.1.4 Package CAIS_FORM_TERMINAL

This package provides functionality for manipulating a form terminal. A form terminal controls much of the display modification itself (or within local "cluster" controllers). Typically a form is built by writing control and prompting characters to desired positions on the display, setting specific character positions to be guarded (protected, as for prompts) or unguarded (unprotected, as for ill-in qualified area), and designating the attributes of the characters (legal entries, color, and intensity. The display is divided into areas of contiguous character positions (qualified area space) that have the same attributes (e.g., unprotected, high intensity). Once the form is built, the form is transmitted to the terminal. At this point, the terminal is in "local" control of the display. The user may move the cursor about on the display, insert, delete, and replace characters in any unprotected area of the display (all under local control, without use of the computer or communications circuitry). When the user has finished all the modifications/entries that are desired, the user presses a special key (function key or enter key) which causes the modified portions of the display to be accessible to the program.

7.1.4.1 Package Specification

```
with CAIS_NODE_DEFS;
with CAIS_TERMINAL_SUPPORT;
package CAIS_FORM_TERMINAL is

    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    type TERMINATION_KEY_ RANGE is INTEGER
        range 0 . . implementation_defined;


    type AREA_INTENSITY is
          (NONE,
           NORMAL,
           HIGH);

    type AREA_PROTECTION is
          (UNPROTECTED,
           PROTECTED);

    type AREA_INPUT is
          (GRAPHIC_CHARACTERS,
          NUMERICS
          ALPHABETICS);
```

```
    type AREA_VALUE is
        (NO_FILL,
         FILL_WITH_ZEROES,
         FILL_WITH_SPACES);

    procedure DEFINE_QUALIFIED_AREA (TERMINAL :     in out  TERMINAL_TYPE;
                                     INTENSITY :    in      AREA_INTENSITY := NORMAL;
                                     PROTECTION :   in      AREA_PROTECTION :=
                                                            PROTECTED;
                                     INPUT :        in      AREA_INPUT :=
                                                            GRAPHIC_CHARACTER_INPUT;
                                     VALUE :        in      AREA_VALUE := NO_FILL);

    procedure CLEAR_QUALIFIED_AREA (TERMINAL :      in out  TERMINAL_TYPE);

    procedure TAB (TERMINAL :   in out  TERMINAL_TYPE;
                   COUNT:       in      POSITIVE);

    procedure PUT (TERMINAL:    in out  TERMINAL_TYPE;
                   ITEM:        in      CHARACTER);
    procedure PUT (TERMINAL :   in out  TERMINAL_TYPE;
                   ITEM :       in      STRING);

    procedure ERASE_AREA (TERMINAL :        in out  TERMINAL_TYPE);

    procedure ERASE_DISPLAY (TERMINAL :     in out  TERMINAL_TYPE);

    procedure ACTIVATE_FORM (TERMINAL :     in out  TERMINAL_TYPE);

    procedure GET (TERMINAL :   in out  TERMINAL_TYPE;
                   ITEM :       out     CHARACTER);
    procedure GET (TERMINAL :   in out  TERMINAL_TYPE;
                   ITEM :       out     STRING);

    function IS_FORM_UPDATED (TERMINAL :    in  TERMINAL_TYPE return BOOLEAN;

    function TERMINATION_KEY (TERMINAL : in TERMINAL_TYPE) return TERMINATION_KEY_RANGE;

    function AREA_QUALIFIER_REQUIRES_SPACE (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;

    -- Exceptions

    CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
    USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;

private
    -- implementation-dependent
end CAIS_FORM_TERMINAL;
```

7.1.4.2 Package Semantics

```
    subtype TERMINAL_TYPE is CAIS_TERMINAL_SUPPORT.TERMINAL_TYPE;

    type TERMINATION_KEY_RANGE is INTEGER range 0 . . implementation_defined;
```

```
type AREA_INTENSITY is
      (NONE,
       NORMAL,
       HIGH);

type AREA_PROTECTION is
      (UNPROTECTED,
       PROTECTED);

type AREA_INPUT is
      (GRAPHIC_CHARACTERS,
       NUMERICS,
       ALPHABETICS);

type AREA_VALUE is
      (NO_FILL,
       FILL_WITH_ZEROES,
       FILL_WITH_SPACES);
```

These types define the attributes for a qualified area of a form. AREA_INTENSITY indicates the intensity at which the characters in the area should be displayed (NONE indicates that characters are not displayed). AREA_PROTECTION specifies whether the user can modify the contents of the area when the form has been activated. AREA_INPUT specifies the valid characters that may be entered by the user (GRAPHIC_CHARACTERS indicates that any printable character may be entered). AREA_VALUE indicates the initial value that the area should have when activated (NO_FILL indicates that the value has been specified by a previous PUT statement).

```
procedure DEFINE_QUALIFIED_AREA (TERMINAL:   in out   TERMINAL_TYPE;
                                 INTENSITY:  in       AREA_INTENSITY: = NORMAL;
                                 PROTECTION: in       AREA_PROTECTION: = PROTECTED;
                                 INPUT:      in       AREA_INPUT : =
                                                      GRAPHIC_CHARACTER_INPUT;
                                 VALUE:      in       AREA_VALUE : = NO_FILL);
```

Indicates that the active position is the first character position of a qualified area. The end of the qualified area is indicated by the beginning of the following qualified area.

```
procedure CLEAR_QUALIFIED_AREA (TERMINAL :   in out   TERMINAL_TYPE);
```

Removes an area qualifier from the active position.

```
procedure TAB (TERMINAL :   in out   TERMINAL_TYPE;
               COURT :      in       POSITIVE);
```

Moves the active position the specified number of qualified areas toward the end of the display. The active position is the first character position of the designated qualified area. The exception USE_ERROR is raised if there are fewer than COUNT qualified areas after the active position.

```
procedure PUT (TERMINAL :   in out   TERMINAL_TYPE;
               ITEM :       in       CHARACTER);
```

Writes a character to the display in the active position. The column of the active position is incremented by one. If the character is written in the last column of a line, the active position is advanced to the first column of the following line. If the character is written to the last column of the last line, the active position is moved to the first column of the first line. If the area qualifier takes space on the display, writing to the position containing an area qualifier removes the area qualifier. Only characters in the range SPACE through STANDARD.ASCII.TILDE may be written. An attempt to write any other character raises the USE_ERROR exception.

```
procedure PUT (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :      in      STRING);
```

Writes each character of the ITEM according to the semantics for writing an individual character.

```
procedure ERASE_AREA (TERMINAL :  in out  TERMINAL_TYPE);
```

Clears the area in which the active position is located.

```
procedure ERASE_DISPLAY (TERMINAL : in out  TERMINAL_TYPE);
```

Clears the display and removes all area qualifiers.

```
procedure ACTIVATE_FORM (TERMINAL :in out  TERMINAL_TYPE);
```

Activates the form that has been created enabling the user to edit the form. Returns control to the calling task when user enters a termination key.

```
procedure GET (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :         out  CHARACTER);
```

Reads a character from the display at the active position. Advances the active position forward one position. An area qualifier (on a display on which the area qualifier requires space) is read as the SPACE character.

```
procedure GET (TERMINAL :  in out  TERMINAL_TYPE;
               ITEM :         out  STRING);
```

Reads ITEM'LENGTH characters from the display one at a time filling the ITEM from ITEM'FIRST through ITEM'LAST.

```
function IS_FORM_UPDATED (TERMINAL : in TERMINAL_TYPE) return BOOLEAN;
```

Returns whether the form was modified by the user during the previous ACTIVATE_FORM operation.

```
function TERMINATION_KEY (TERMINAL : in TERMINAL_TYPE) return TERMINATION_KEY_RANGE;
```

Returns a number that indicates which (implementation dependent) key terminated the ACTIVATE_FORM procedure. A value of zero indicates the normal termination key (i.e., the ENTER key).

```
function AREA_QUALIFIER_REQUIRES_SPACE (TERMINAL : in TERMINAL_TYPE)
    return BOOLEAN;
```

Returns TRUE if the area qualifier requires space on the display.

```
-- Exceptions

CLASS_ERROR : exception renames CAIS_TERMINAL_SUPPORT.CLASS_ERROR;
USE_ERROR :   exception renames CAIS_NODE_DEFS.USE_ERROR;
```

The exception CLASS_ERROR is raised if any of the routines in the package CAIS_FORM_TERMINAL are invoked with a terminal handle which is not OPENed or RESET with class FORM.

7.2 PACKAGE CAIS_DEVICE_CONTROL

This package provides physical device control interfaces. For each device type, there is a set of operations defined to manipulate the device.

Certain generic device-oriented status information is available outside of the specific packages.

7.2.1 Package Specification

```
package CAIS_DEVICE_CONTROL is
    {TBD}
end CAIS_DEVICE_CONTROL;
```

8. CAIS UTILITIES

This area provides packages for manipulating strings and parameter lists. It also defines additional pragmatic requirements for a conforming implementation of the predefined Ada LRM packages.

8.1 PREDEFINED LANGUAGE ENVIRONMENT

The facilities described in the Ada LRM that are used directly by the CAIS include the packages STANDARD and SYSTEM, as discussed in the following subsections. See the Pragmatics Section 8.3.

8.1.1 Package STANDARD

Package STANDARD forms the outermost scope of all Ada compilation units.

Package STANDARD is not replaceable by implementors of the CAIS, and hence the "CAIS_" prefix is not used.

8.1.2 Package SYSTEM

The package SYSTEM is provided as a language-defined package which defines certain parameters of the language implementation.

Package SYSTEM is not replaceable by implementors of the CAIS, and hence the "CAIS_" prefix is not used.

8.2 PREDEFINED UTILITY PACKAGES

The utilities necessary for the support of other CAIS interfaces include the packages CAIS_TEXT_UTILS and CAIS_LIST_UTILS, as discussed in the following sections.

8.2.1 Package CAIS_TEXT_UTILS

This package implements basic operations on a string type which is of dynamic length. It defines the type used to implement lists and is used for MESSAGE_TEXT, PROCESS_STRING, and RESULTS_STRING.

8.2.1.1 Package Specification

```
package CAIS_TEXT_UTILS is
    MAXIMUM : constant : = implementation_defined;
    subtype INDEX is INTEGER range 0..MAXIMUM;

    type TEXT is limited private;

    function LENGTH  (T: TEXT) return INDEX;
    function VALUE   (T: TEXT) return STRING;
    function EMPTY   (T: TEXT) return BOOLEAN;
```

```
    procedure INIT_TEXT(T:   in out TEXT);
    procedure FREE_TEXT(T:  in out TEXT);

    function TO_TEXT (S: STRING)          return TEXT;
    function TO_TEXT (C: CHARACTER)    return TEXT;

    function "&" (LEFT: TEXT;          RIGHT: TEXT)          return TEXT;
    function "&" (LEFT: TEXT;          RIGHT: STRING)       return TEXT;
    function "&" (LEFT: STRING;       RIGHT: TEXT)          return TEXT;
    function "&" (LEFT: TEXT;          RIGHT: CHARACTER) return TEXT;
    function "&" (LEFT: CHARACTER; RIGHT: TEXT)          return TEXT;

    function "="     (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function "<="    (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function "<"     (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function ">="    (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;
    function ">"     (LEFT: TEXT;          RIGHT: TEXT)          return BOOLEAN;

    procedure SET   (OBJECT: in out TEXT;  VALUE: in TEXT);
    procedure SET   (OBJECT: in out TEXT;  VALUE: in STRING);
    procedure SET   (OBJECT: in out TEXT;  VALUE: in CHARACTER);

    procedure APPEND (TAIL: in TEXT;          TO: in out TEXT);
    procedure APPEND (TAIL: in STRING;       TO: in out TEXT);
    procedure APPEND (TAIL: in CHARACTER;  TO: in out TEXT);

    procedure AMEND  (OBJECT:    in out TEXT;
                      BY:          in     TEXT;
                      POSITION:  in     INDEX);
    procedure AMEND  (OBJECT:    in out TEXT;
                      BY:          in     STRING;
                      POSITION:  in     INDEX);
    procedure AMEND  (OBJECT:    in out TEXT;
                      BY:          in     CHARACTER;
                      POSITION:  in     INDEX);

    function LOCATE  (FRAGMENT: TEXT;          WITHIN: TEXT) return INDEX;
    function LOCATE  (FRAGMENT: STRING;       WITHIN: TEXT) return INDEX;
    function LOCATE  (FRAGMENT: CHARACTER; WITHIN: TEXT) return INDEX;

private
    -- implementation-dependent
end CAIS_TEXT_UTILS;
```

8.2.1.2 Package Semantics

```
    type TEXT is limited private;
```

The type is made limited private because it may be reference counted and automatically freed at last use.

```
    function LENGTH  (T: TEXT) return INDEX;
    function VALUE   (T: TEXT) return STRING;
    function EMPTY   (T: TEXT) return BOOLEAN;
```

Provides text string functions.

```
    procedure INIT_TEXT(T: in out TEXT);
```

Creates a null string.

```
procedure FREE__TEXT(T: in out TEXT);
```

Frees a string.

```
function TO__TEXT (S: STRING)          return TEXT;
function TO__TEXT (C: CHARACTER)       return TEXT;
```

Converts the given string or characters to text.

```
function "&" (LEFT: TEXT;      RIGHT: TEXT)      return TEXT;
function "&" (LEFT: TEXT;      RIGHT: STRING)    return TEXT;
function "&" (LEFT: STRING;    RIGHT: TEXT)      return TEXT;
function "&" (LEFT: TEXT;      RIGHT: CHARACTER) return TEXT;
function "&" (LEFT: CHARACTER; RIGHT: TEXT)      return TEXT;
```

Concatenates to text.

```
function "="  (LEFT: TEXT; RIGHT: TEXT) return BOOLEAN;
function "<=" (LEFT: TEXT; RIGHT: TEXT) return BOOLEAN;
function "<"  (LEFT: TEXT; RIGHT: TEXT) return BOOLEAN;
function ">=" (LEFT: TEXT; RIGHT: TEXT) return BOOLEAN;
function ">"  (LEFT: TEXT; RIGHT: TEXT) return BOOLEAN;
```

Provides indicated comparison functions.

```
procedure SET (OBJECT: in out TEXT; VALUE: in TEXT);
procedure SET (OBJECT: in out TEXT; VALUE: in STRING);
procedure SET (OBJECT: in out TEXT; VALUE: in CHARACTER);
```

Sets the object to the given value.

```
procedure APPEND (TAIL: in TEXT;      TO: in out TEXT);
procedure APPEND (TAIL: in STRING;    TO: in out TEXT);
procedure APPEND (TAIL: in CHARACTER; TO: in out TEXT);
```

Appends the given TAIL to the TO TEXT.

```
procedure AMEND (OBJECT:   in out TEXT;
                 BY:       in     TEXT;
                 POSITION: in     INDEX);
procedure AMEND (OBJECT:   in out TEXT;
                 BY:       in     STRING;
                 POSITION: in     INDEX);
procedure AMEND (OBJECT:   in out TEXT;
                 BY:       in     CHARACTER;
                 POSITION: in     INDEX);
```

Replaces part of the OBJECT by the given TEXT, STRING, or CHARACTER starting at the given position in the OBJECT.

```
function LOCATE (FRAGMENT: TEXT;      WITHIN: TEXT) return INDEX;
function LOCATE (FRAGMENT: STRING;    WITHIN: TEXT) return INDEX;
function LOCATE (FRAGMENT: CHARACTER; WITHIN: TEXT) return INDEX;
```

Returns the INDEX of the FRAGMENT within the given TEXT.

8.2.2 Package CAIS_LIST_UTILS

This package is generally useful for the manipulation of all lists built following the CAIS parameter list conventions: a parenthesized, comma-separated list of items, each item in the form of a list, a string without embedded spaces or separators, or a quoted string following the Ada syntax rules, optionally preceded by a keyword identifier and a "right arrow." This syntax roughly corresponds to the Ada syntax for aggregates or for subprogram calling sequences. An approximate BNF for the CAIS list is as follows:

```
LIST            ::= '(' [ KEYWORD '=  ' ] ITEM { ' , ' [ KEYWORD '=  ' ] ITEM } ')'
ITEM            ::= IDENTIFIER | NUMBER | LIST | QUOTED_STRING
KEYWORD         ::= IDENTIFIER
QUOTED_STRING   ::= ' '' ' { NON_QUOTE_CHARACTER | ' '' '' ' } ' '' '
```

The package CAIS_LIST_UTILS uses the TEXT type defined within *CAIS_TEXT_UTILS and defines additional* operations. It defines the type list which is used to represent CAIS_ATTRIBUTE values.

8.2.2.1 Package Specification

```
with CAIS_TEXT_UTILS;
package CAIS_LIST_UTILS is

    type COUNT is range 0 . . implementation_defined;
    subtype POSITIVE_COUNT is COUNT range 1 . . COUNT'LAST;
    subtype LIST is CAIS_TEXT_UTILS.TEXT;
    subtype KEY_STRING is STRING;
    type ITEM_KIND is (LIST, IDENTIFIER, NUMBER, QUOTED_STRING);

    procedure INIT_LIST(L: in out LIST);

    procedure FREE_LIST(L: in out LIST);
    function  IS_EMPTY  (L: in LIST)  return BOOLEAN;
    function KIND   (L: in LIST)   return ITEM_KIND;

    function QUOTED_STRING  (L: in LIST)   return STRING;
    function IDENTIFIER     (L: in LIST)   return STRING;
    function NUMBER         (L: in LIST)   return INTEGER;

    procedure TO_LIST_QUOTED      (L: in out LIST; FROM: STRING);

    procedure TO_LIST             (L: in out LIST; FROM: STRING);
    procedure TO_LIST             (L: in out LIST; FROM: INTEGER);

    procedure SET (L: in out LIST; VALUE in LIST);
    function NUM_POSITIONAL  (L: LIST) return COUNT;

    procedure ADD_POSITIONAL  (L:    in out LIST;
                               ITEM: in     LIST);
    procedure ADD_POSITIONAL  (L:    in out LIST;
                               ITEM: in     STRING);

    procedure GET_POSITIONAL  (L:    in     LIST;
                               ITEM: in out LIST;
                               AT:   in     POSITIVE_COUNT);

    procedure SET_POSITIONAL (L:    in out LIST;
                              ITEM: in     LIST;
                              AT:   in     POSITIVE_COUNT);
    function NUM_NAMED (L: LIST)    return COUNT;
```

```
    procedure ADD_NAMED   (L:          in out LIST;
                           KEYWORD:    in     KEY_STRING;
                           ITEM:       in     LIST);
    procedure ADD_NAMED   (L:          in out LIST;
                           KEYWORD:    in     KEY_STRING;
                           ITEM:       in     STRING);

    procedure GET_NAMED   (L:          in     LIST;
                           ITEM:       in out LIST;
                           AT:         in     KEY_STRING);
    procedure GET_NAMED   (L:          in     LIST;
                           ITEM:       in out LIST;
                           AT:         in     POSITIVE_COUNT);

    procedure  SET_NAMED  (L:          in out     LIST;
                           ITEM:       in         LIST;
                           AT:         in         KEY_STRING);
    procedure  SET_NAMED  (L:          in out     LIST;
                           ITEM:       in         LIST;
                           AT:         out        POSITIVE_COUNT);

    function KEYWORD  (L:          in     LIST;
                       AT:         in     POSITIVE_COUNT)
                      return KEY_STRING;

private
    -- implementation-dependent
end CAIS_LIST_UTILS;
```

8.2.2.2 Package Semantics

```
    type ITEM_KIND is (LIST, IDENTIFIER, NUMBER, QUOTED_STRING);
```

Each item is recognizable as a list, identifier, number, or quoted-string.

```
    procedure INIT_LIST   (L: in out LIST);
```

Creates a null LIST.

```
    procedure FREE_LIST   (L: in out LIST);
```

Frees a LIST.

```
    function IS_EMPTY   (L: in     LIST) return BOOLEAN;
```

Returns TRUE if the list is an empty LIST.

```
    function KIND   (L: in     LIST) return ITEM_KIND;
```

Returns ITEM_KIND of LIST.ITEM_KIND is LIST for empty LIST.

```
    function QUOTED_STRING   (L: in LIST)   return STRING;
    function IDENTIFIER      (L: in LIST)   return STRING;
    function NUMBER          (L: in LIST)   return INTEGER;
```

Converts from a LIST according to the ITEM_KIND.

```
procedure TO_LIST_QUOTED  (L: in out LIST; FROM: STRING);

procedure TO_LIST         (L: in out LIST; FROM: STRING);
procedure TO_LIST         (L: in out LIST; FROM: INTEGER);
```

Converts to a LIST according to the ITEM_KIND.

```
procedure SET  (L: in out LIST; VALUE: in LIST);
```

Sets the LIST L to the given VALUE.

```
function NUM_POSITIONAL(L: LIST) return COUNT;
```

Returns COUNT of positional components (i.e., those without the "KEYWORD => " part).

```
procedure ADD_POSITIONAL  (L:    in out LIST;
                           ITEM: in     LIST);
procedure ADD_POSITIONAL  (L:    in out LIST;
                           ITEM: in     STRING);
```

Adds another ITEM to the end of the LIST of positional components.

```
procedure GET_POSITIONAL  (L:    in     LIST;
                           ITEM: in out LIST;
                           AT:   in     POSITIVE_COUNT);
```

Retrieves ITEM at specified position of LIST. Returns empty LIST if AT > NUM_POSITONAL(L).

```
procedure SET_POSITIONAL  (L:    in out LIST;
                           ITEM in     LIST
                           AT:   in     POSITIVE_COUNT);
```

Sets VALUE at specified position of LIST to the given ITEM.

```
function NUM_NAMED  (L: LIST) return COUNT;
```

Returns count of named components (i.e., those with the "KEYWORD =>" part).

```
procedure ADD_NAMED (L:           in out LIST;
                     KEYWORD:     in     KEY_STRING;
                     ITEM:        in     LIST);
procedure ADD_NAMED (L:           in out LIST;
                     KEYWORD:     in     KEY_STRING;
                     ITEM:        in     STRING);
```

Adds another named ITEM to LIST. An exception is generated if an ITEM with the given KEYWORD already exists within LIST.

```
procedure GET_NAMED (L:           in     LIST;
                     ITEM:        in out LIST;
                     AT:          in     KEY_STRING);
procedure GET_NAMED (L:           in     LIST;
                     ITEM:        in out LIST;
                     AT:          in     POSITIVE_COUNT);
```

Gets the named ITEM at the given KEYWORD or POSITIVE_COUNT; returns empty LIST if the ITEM is not found.

```
procedure SET__NAMED (L:      in out LIST;
                      ITEM:   in     LIST;
                      AT:     in     KEY__STRING);
procedure SET__NAMED (L:      in out LIST;
                      ITEM:   in     LIST;
                      AT:     in     POSITIVE__COUNT
```

Sets the named component at the given KEY__STRING or POSITIVE__COUNT to the given ITEM.

```
function KEYWORD (L: in     LIST;
                  AT: in    POSITIVE__COUNT)
                  return KEY__STRING;
```

Returns the KEYWORD of the specified named item.

8.2.3 Package CAIS__HELP__UTILS

This package provides standard support for help facilities.

{TBD}

8.3 PRAGMATICS

a.	STANDARD	The CAIS places certain requirements on the pre-defined types available. In particular, a conforming implementation must support some integer type with at least the range -32767 to 32767.
b.	SYSTEM	The CAIS places certain requirements on the machine parameters. In particular, a conforming implementation must have MIN__INT = -32767 and MAX__INT = 32767.
c.	CAIS__TEXT__UTILS	A conforming implementation must support strings of at least 32767 characters in length.

APPENDIX A
NOTES AND EXPLANATIONS

A.1 INTRODUCTION

This appendix is provided to give the reader a perspective of the context in which the CAIS is expected to function and some of the design considerations included during the CAIS generation process. While Version 1.1 of the CAIS is directed toward the DoD AIE and ALS developments, the goal of future versions of the CAIS is to provide a standard for DoD APSEs.

A.1.1 BACKGROUND

Version 1.1 of the CAIS is predicated on four premises:

1) the CAIS will be implementable on the AIE
2) the CAIS will be implementable on the ALS
3) the CAIS will be implementable on a bare machine
4) the CAIS will be compatible with modern operating systems

The CAIS as described in Version 1.1 has strived to retain these perspectives while establishing a sufficiently flexible structure that can be evolved into a Version 2.0 document. This structure is believed to be flexible enough to provide CAIS implementors considerable amplitude in selecting specific approaches for actual implementations. Interference with implementation strategies has been avoided.

A.2 CONTEXT

The CAIS applies to Ada Programming Support Environments [STONEMAN] which are to become the basic software development environments for DoD development programs. Those Ada programs that are used in support of software development are defined as **tools**. This includes the spectrum of support software from project management through code development, configuration management and life-cycle support. Tools are not restricted to only those software items normally associated with program generation such as editors, compilers, debuggers, and linker-loaders. Those tools that are composed of a number of independent but inter-related programs (such as a debugger which is related to a specific compiler) are classed as **toolsets**. In this document the terms tool and toolset are used interchangeability.

Since the goal of the CAIS is to promote interoperability and transportability of Ada software across DoD APSEs, the following definitions of these terms are provided. **Interoperability** is defined as "the ability of APSEs to exchange data base objects and their relationships in forms usable by tools and user programs without conversion." **Transportability** of an APSE tool is defined as "the ability of the tool to be installed on a different KAPSE; the tool must perform with the same functionality in both APSEs. Transportability is measured in the degree to which this installation can be accomplished without reprogramming. Portability and transferability are commonly used synonymously." [Reference: KAPSE Interface Team: Public Report, Volume I, 1 April 1982; p. C1].

APPENDIX B
PROVIDING DIRECTORY STRUCTURES USING A TRANSITIONAL SUBSET OF THE CAIS

B.1 INTRODUCTION

While conformance with the CAIS will be measured on a package-by-package basis, it is sometimes not possible to implement one package without the provision of another. This is especially true for packages depending on the package CAIS_NODE_MANAGEMENT. In the interest of the availability of CAIS implementations within a very short time frame, a transitional subset of the node-related packages are defined in this appendix. They include the most important interfaces that are vital for the majority of simple tools. This subset restricts the model of the file organization to the equivalent of a hierarchical tree-oriented file-system. Leaves in the tree are file nodes; all other nodes are structural nodes representing directories or they are process nodes.

In order to prevent incompatibilities with more sophisticated CAIS implementations, the syntactic appearance and semantic meaning of calls on CAIS interfaces have been kept upward compatible, rather than providing more appropriate mnemonic names for the subprograms. (The latter is left to a trivial renaming package outside the CAIS subset.) Hence, any program executing properly on an implementation of the CAIS subset will also execute properly on any implementation of the CAIS (but obviously not vice-versa).

An implementator of these transitional subset packages may choose to use different implementation strategies than required for the provision of the full functionality of these packages as defined in the CAIS.

The subset is obtained by imposing restrictions and adjusting package specifications as follows:

1. Pathnames are allowed to contain only path elements referring to the "DOT"-relation using the abbreviated form " . " or to " 'USER" and " 'CURRENT_USER" as predefined optional prefixes to pathnames.

2. In all subprograms of the node-related packages CAIS_NODE_MANAGEMENT and CAIS_STRUCTURAL_NODES any occurrence of a formal parameter of type RELATION_NAME is deleted. The implementation of these subprograms must default the RELATION_NAME to "DOT".

3. The formal parameters RELATION and PRIMARY_ONLY of the subprograms CAIS_NODE_MANAGEMENT.ITERATE are deleted. The implementation of the subprograms must default the RELATION to "DOT".

4. The following subprograms of the package CAIS _NODE_MANAGEMENT are defined to raise the USE_ERROR exception:

   PRIMARY_RELATION
   PATH_KEY
   PATH_RELATION
   LINK
   UNLINK

5. The following subprograms of the package CAIS_STRUCTURAL_NODE are defined to raise the USE_ERROR exception:

   CREATE_NODE with formal parameter "RELATION" (two instances)

Bearing these restrictions in mind, the specified semantics for all subprograms of the packages involved describe those operations useful in particular for handling directories (structural nodes) of a conventional tree-structured file system and files contained in such directories. Pathnames have the conventional form of identifiers separated by dots, except for the optional prefix path elements " 'USER " and " 'CURRENT_USER".

B.1.1 Package Semantics

NOTE: These semantics do not include the procedures and functions which are defined to raise USE_ERROR in the above list.

a) CAIS_STRUCTURAL_NODES

```
procedure CREATE_NODE(NAME:    in      NAME_STRING;
                      FORM:    in      FORM_STRING := " ");
procedure CREATE_NODE(NODE:    in out  NODE_TYPE;
                      NAME:    in      NAME_STRING;
                      FORM:    in      FORM_STRING := " ");
```

Creates a directory (structural node) with its "DOT" relationship and parent node implied by the NAME argument.

b) CAIS_NODE_MANAGEMENT

The **key** of a file or directory is the relationship key of the last element of its pathname. Many operations are allowed to take either a pathname, or a parent node (i.e., a directory) and a key. The keys of process nodes, file nodes or subdirectories in a directory must be mutually distinct.

```
procedure OPEN (NODE:    in out  NODE_TYPE;
                NAME:    in      NAME_STRING;
procedure OPEN (NODE:    in out  NODE_TYPE;
                BASE:    in      NODE_TYPE;
                KEY:     in      RELATIONSHIP_KEY := " ");
```

Opens the designated file node, process node or directory and returns an open handle on the designated file node, process node or directory node. The NAME_ERROR exception will be raised if the file, process or directory does not exist.

```
procedure CLOSE(NODE: in out NODE_TYPE);
```

Severs any association between the internal node handle and an external node and releases any associated lock. This must be done before another OPEN can be done using the same NODE_TYPE variable.

```
function IS_OPEN (NODE: in NODE_TYPE) return BOOLEAN;
```

Returns TRUE if the NODE is open.

```
function KIND (NODE: in NODE_TYPE) return CAIS_NODE_DEFS.FILE_KIND;
```

Returns the "kind" of a node, either FILE, PROCESS, STRUCTURAL or DEVICE. Structural nodes are directories.

```
function PRIMARY_NAME(NODE: in NODE_TYPE) return NAME_STRING;
```

Returns the full path name to the file node, process node, or directory.

```
function PRIMARY_KEY (NODE: in NODE_TYPE) return RELATIONSHIP_KEY;
```

Return the last relationship key of the pathname to the file node, process node or directory. If the NODE is a top-level directory, the key is the user name.

```
procedure GET_PARENT(NODE:    in      NODE_TYPE;
                     PARENT:  in out  NODE_TYPE);
```

Returns the parent process or directory. Generates an exception if NODE is a top-level directory.

```
procedure COPY_NODE (FROM:      in      NODE_TYPE;
                     TO:        in      NAME_STRING);
procedure COPY_NODE (FROM:      in      NODE_TYPE;
                     TO_BASE:   in      NODE_TYPE;
                     TO_KEY:    in      RELATIONSHIP_KEY := " ");
```

Copies a file. It is an error (KIND_ERROR) if the node referenced is a process node or a device node or directory node(structural node).

```
procedure COPY_TREE (FROM:      in      NODE_TYPE;
                     TO:        in      NAME_STRING);
procedure COPY_TREE (FROM:      in      NODE_TYPE;
                     TO_BASE:   in      NODE_TYPE;
                     TO_KEY:    in      RELATIONSHIP_KEY := " ");
```

Copies a directory including its files. It is an error (KIND_ERROR) if any node referenced is a process node or a device node.

```
procedure RENAME(NODE:          in      NODE_TYPE;
                 NEW_NAME:      in      NAME_STRING);
procedure RENAME(NODE:          in      NODE_TYPE;
                 NEW_BASE:      in      NODE_TYPE;
                 NEW_KEY:       in      RELATIONSHIP_KEY := " "
                 NEW_RELATION:  in      RELATION_NAME :  = " . ";
```

Allows the renaming of file nodes process nodes, or directories using a node handle for the renamed node and, in the second case, a node handle on the parent directory or process node. RENAME raises the exception USE_ERROR if a node already exists with the new_name.

```
procedure DELETE_NODE (NODE: in out NODE_TYPE);
procedure DELETE_NODE (NAME: in     NAME_STRING);
```

Deletes the relationships between a file or process node and its parent and deletes the node itself. This is only legal if the node has no children. Deletes a file, empty directory or a process with no descendants as well as the associated node.

```
procedure DELETE_TREE (NODE: in out NODE_TYPE);
```

DELETE_TREE deletes a node and recursively deletes all its descendants.

```
type NODE_ITERATOR is private;
subtype RELATIONSHIP_KEY_PATTERN is RELATIONSHIP_KEY;
```

RELATIONSHIP_KEY_PATTERNs follow the syntax of relationship keys, except that a "?" will match any single character and a "*" will match any string of characters.

```
procedure ITERATE(ITERATOR:         out     NODE_ITERATOR;
                  NODE:             in      NODE_TYPE;
                  KIND:             in      NODE_KIND;
                  KEY:              in      RELATIONSHIP_KEY_PATTERN := " * ");

function MORE (ITERATOR: in NODE_ITERATOR) return BOOLEAN;

procedure GET_NEXT(ITERATOR:        in out  NODE_ITERATOR;
                   NEXT_NODE:       in out  NODE_TYPE);
```

These three routines iterate through those nodes referred to from the given NODE, via "DOT"-relationships, that have keys satisfying the specified patterns and are of the KIND specified.

The nodes are returned in ASCII lexicographical order by relationship KEY. The key is available from the function PRIMARY_KEY (see above).

```
procedure SET_CURRENT_NODE(NAME: in NAME_STRING);
procedure SET_CURRENT_NODE(NODE: in NODE_TYPE);
```

Specifies NODE/NAME as the current directory.

```
procedure GET_CURRENT_NODE(NODE: in out NODE_TYPE);
```

Associates NODE with the current directory.

```
function IS_SAME(NAME1:          in    NAME_STRING;
                 NAME2:          in    NAME_STRING)
    return BOOLEAN;
function IS_SAME(NODE1:          in    NODE_TYPE
                 NODE2:          in    NOTE_TYPE)
    return BOOLEAN;
```

APPENDIX C
CAIS IMPLEMENTABILITY

C.1 INTRODUCTION

The specification of the CAIS has been separated into multiple packages to simplify initial or partial implementations. The rules for Ada limited private types can interfere with this kind of separation. This appendix outlines several implementation approaches which are consistent with both the rules of the Ada language and the rules for CAIS conformance. This appendix will ultimately be superceded by a CAIS implemator's guide.

(a) NESTED GENERIC SUBPACKAGES IMPLEMENTATION

This implementation strategy seeks to minimize visibility of the limited private types of CAIS_NODE_DEFS by using these private types strictly as intended by Ada. All operations on the private types are encapsulated within the package defining CAIS_NODE_DEFS. A sketch of this is as follows:

```
package CAIS is
    -- type definitions of CAIS_NODE_DEFS
        generic
        package NODE_DEFS is
        -- subtype Declarations
        end NODE_DEFS;

        generic
        package NODE_MANAGEMENT is
        --specifications of Section 3.5
        end NODE_MANAGEMENT;

        generic
        package STRUCTURAL_NODES is
        --specifications of Section 4.1
        end STRUCTURAL_NODES;
        -- and so forth for all of the CAIS packages

end CAIS;

with CAIS;

package CAIS_NODE_DEFS is new CAIS.NODE_DEFS;

with CAIS;
package CAIS_NODE_MANAGEMENT is new CAIS.NODE_MANAGEMENT;

. . . for each of the CAIS packages
```

This organization, while unwieldy, allows the CAIS packages specified in this document to be utilized in the organization provided in earlier document sections.

(b) LIMITED RECORD TYPE IMPLEMENTATION

This sketch shows how an implementor might separate the limited private definitions and operations on the limited private types into a separate isolated package. The user-visible package structure remains the same, except that NODE_TYPE is defined as a limited record type, rather than limited private.

```
package CAIS_PRIVATE is
    type NODE_TYPE is limited private;
    . . . and other types with limited private visibility needs

    -- The remainder of this package specification is
    -- implementation specific, and not specified as part
    -- of the CAIS. No tool or APSE application should
    -- make use of this package; it is solely for the
    -- use for implementation of other CAIS packages.

end CAIS_PRIVATE;

with CAIS_PRIVATE;
package CAIS_NODE_DEFS is
    type NODE_TYPE is
      record
        INTERNALS: CAIS_PRIVATE.NODE_TYPE;
      end record;
    . . . and the rest of CAIS_NODE_DEFS from 3.1
```

The implementation of the other CAIS packages (i.e., the package bodies) may now use the underlying subprograms of CAIS_PRIVATE to manipulate the INTERNALS of NODE_TYPE. This provides an implementation which is safe, so long as no tool or applications program "withs" CAIS_PRIVATE.

A typical CAIS implementation package body may have the following appearance:

```
with CAIS_PRIVATE;
package body CAIS_NODE_MANAGEMENT is
. . .
procedure OPEN(NODE:                    in out   NODE_TYPE;
               NAME:                    in       NAME_STRING) is
    begin
      CAIS_PRIVATE.OPEN(NODE.INTERNALS, NAME);
    end OPEN;


. . .

end CAIS_NODE_MANAGEMENT;
```

(c) NON-ADA IMPLEMENTATION

If the package bodies are implemented in a language other than Ada, then the problems of limited private types may be absent. The implementation may have a structure dictated by the facilities of an underlying operating system, by the facilities of a microcoded system and by the processor architecture itself.

Postscript : Submission of Comments

For submission of comments on this CAIS Version 1.1, we would appreciate them being sent by Arpanet to the address

CAIS-COMMENT at ECLB

If you do not have Arpanet access, please send the comments by mail

Mr. Jack Foidl
TRW SYSTEMS
3420 Kenyon St.
Suite 202
San Diego, CA 92110

For mail comments, it will assist us if you are able to send them on 8-inch single-sided single-density DEC format diskette—but even if you can manage this, please also send us a paper copy, in case of problems with reading the diskette.

All comments are sorted and processed mechanically in order to simplify their analysis and to facilitate giving them proper consideration. To aid this process you are kindly requested to precede each comment with a three line header

!section . . .
!version 1983
!topic . . .
!rationale

The section line includes the section number, your name or affiliation (or both), and the date in ISO standard form (year-month-day). As an example, here is the section line of comment 1194 on a previous version:

!section 03.02.01(12)D.Taffs 82-04-26

The version line, for comments on the current document, should only contain "!version 1983". Its purpose is to distinguish comments that refer to different versions.

The topic line should contain a one line summary of the comment. This line is essential, and you are kindly asked to avoid topics such as "Typo" or "Editorial comment" which will not convey any information when printed in a table of contents. As an example of an informative topic line, consider:

!topic FILE NODE MANAGEMENT

Note also that nothing prevents the topic line from including all the information of a comment, as in the following topic line:

!topic Insert: "...are {implicitly} defined by a subtype declaration"

The rationale line should contain some reasoning for your comment.

As a final example here is a complete comment:

!section 03.02.01(12)D.Taffs 82-04-26
!version 1983
!topic FILE NODE MANAGEMENT

Change component to subcomponent in the last sentence.

!rationale

Otherwise the statement is inconsistent with the defined use of subcomponent in 3.3, which says that subcomponents are excluded when the term component is used instead of subcomponent.